Join the discussion@ p2p.wrox.com

Wrox Programmer to Programmer™

.NET开发经典名著

Professional C# 5.0 and .NET 4.5.1

C#高级编程（第9版）

——C# 5.0 & .NET 4.5.1

2011年度
2009年度 全行业优秀畅销品种

Christian Nagel
[美] Jay Glynn 著
Morgan Skinner
李铭 译
黄静 审校

清华大学出版社

提升C#技能的必备参考资源

本书由.NET专家的梦幻组合编写，包含开发人员使用C#所需的所有内容。C#是编写.NET应用程序的一种语言，本书适合于希望提高编程技巧的、有经验的C#程序员，也适用于刚开始使用C#的专业开发人员。本书探讨了Visual Studio 2013和.NET Framework 4.5.1、新的测试驱动开发和并发编程功能。所有示例的源代码都可以下载，读者可以立即开始编写Windows桌面应用程序、Windows Store应用程序和ASP.NET Web应用程序。

主要内容

- 涵盖Visual Studio 2013的主要更新和改进，重新讨论了C#开发人员与VS的交互方式
- 提供了专业开发人员必须了解和掌握的所有C#知识
- 研究了.NET Framework 4.5.1 GC的更新、Visual Studio 2013新的UI和用于Windows 8.1的Windows Store应用程序
- 包含大量有益的示例和用于实践的代码，以及处理常见问题的灵活方法

作者简介

Christian Nagel是Microsoft RD、Microsoft MVP、thinktecture的合作伙伴、CN革新技术的奠基人，他还是一位软件架构师和开发人员，为开发Microsoft .NET解决方案提供培训和咨询服务。他具备超过25年的软件开发经验。Christian从PDP 11和VAX/VMS系统开始其计算机生涯，熟悉各种语言和平台。他具备Microsoft技术的深厚功底，编写了大量图书，并获得了Microsoft认证培训师和专业开发人员证书。

Jay Glynn开发软件的时间超过20年，使用PICK Basic为PICK操作系统编写应用程序。到目前为止，他使用过Delphi、VBA、Visual Basic、C、Java和C#编写软件。他目前是VGT的高级软件工程师，编写基于Web的应用程序。

Morgan Skinner是一位自由顾问，他在开始自己的顾问生涯之前，在Microsoft工作了将近10年。

源代码下载及技术支持

http://www.wrox.com

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

wrox.com

Programmer Forums

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

Code Downloads

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

Read More

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

清华大学出版社数字出版网站

www.wqbook.com

定价：148.00元

.NET开发经典名著

C#高级编程(第9版)

——C# 5.0 & .NET 4.5.1

[美] Christian Nagel
Jay Glynn 著
Morgan Skinner
李 铭 译
黄 静 审校

清华大学出版社
北 京

Christian Nagel, Jay Glynn, Morgan Skinner
Professional C# 5.0 and .NET 4.5.1
EISBN：978-1-118-83303-2
Copyright © 2014 by John Wiley & Sons, Inc.
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2014-2797

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

C#高级编程(第 9 版)——C# 5.0 & .NET 4.5.1 / (美)内格尔 (Nagel,C.) 等著；李铭 译. —北京：清华大学出版社，2014

(.NET 开发经典名著)

书名原文：Professional C# 5.0 and .NET 4.5.1

ISBN 978-7-302-38002-3

Ⅰ.C… Ⅱ.①内… ②李… Ⅲ.①C 语言—程序设计 Ⅳ.①TP312

中国版本图书馆 CIP 数据核字(2014)第 210595 号

责任编辑：王　军　于　平
装帧设计：孔祥峰
责任校对：成凤进
责任印制：刘海龙

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
　　社 总 机：010-62770175　　邮　　购：010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　印　张：97　　字　　数：2676 千字
版　　次：2014 年 10 月第 1 版　　印　　次：2014 年 10 月第 1 次印刷
印　　数：1～5000
定　　价：148.00 元

产品编号：058141-01

译 者 序

C#是一种简洁、类型安全的面向对象的语言，开发人员可以使用它来构建在.NET Framework 上运行的各种安全、可靠的应用程序。常用于开发 Windows 客户端应用程序、XML Web Services、分布式组件、客户端/服务器应用程序、数据库应用程序等。

C#简单易学。它的大括号语法使任何熟悉 C、C++或 Java 的人都可以立即上手。C#语法简化了 C++的诸多复杂性，并提供了很多强大的功能，例如，可为 null 的值类型、枚举、委托、lambda 表达式和直接内存存取，这些都是新版本的 Java 想要尽快添加的。

C#的生成过程比 C 和 C++简单，比 Java 更为灵活。没有单独的头文件，也不要求按照特定顺序声明方法和类型。"互操作"使 C#程序能够完成本机 C++应用程序可以完成的几乎任何任务。在直接内存存取必不可少的情况下，C# 甚至支持指针和"不安全"代码的概念。

本书由.NET 开发的梦幻专家团队编写，包含开发人员使用 C#语言所需的所有内容。新版本涵盖了 Visual Studio 2013 的主要更新和改进，研究了.NET Framework 4.5.1、测试驱动的新开发和并发编程功能。既适合于希望提高编程技巧且有经验的 C#程序员，也适合于刚开始使用 C#的专业开发人员。

作为一本"资历深厚"的经典著作，本书历经十余年多次改版，一直保持了内容合理、讲解清晰的优良传统。开篇的 1-16 章对 C#语言进行了非常全面的阐述，强烈建议读者首先阅读这一部分内容，根据理解的层次以决定是否继续研读下去，这一部分内容是本书非常有价值的部分。第 17-18 章对 Visual Studio 开发工具进行了简述，以便读者对 Visual Studio 2013 的新特性有些了解！第 19-31 章对.NET 框架基本功能进行了阐述，是全书中比较乏味的部分(这是译者自己的一家之言)，在翻译的过程中也异常痛苦，毕竟 PE 文件头和 CLR 头的概念并不常见。第 32-34 章对.NET 操作各种不同类型数据进行全面的阐述。这些内容非常有用，工作中也经常用到，尤其是 ADO.NET Entity Framework。第 35-42 章对.NET 框架提供的各种不同类型界面进行简单的阐述，了解一下这些内容犹如如虎添翼。第 43-47 章对.NET 框架提供的各种不同类型的通信方式进行全面的阐述，这也是全书的精华所在，尤其是在复杂的环境中进行数据交互，这部分内容也是非常有用的。

随着新版本的发布，肯定会有越来越多的开发人员想尝试使用.NET Framework 4.5.1 中的新特性，毕竟开发者社区对这个新版本期待了很长的时间。在.NET 程序中使用这些新特性，可以提高代码质量，提升工作效率，提供解决方案。但是，问题也在所难免。在将新技术真正应用到新项目时，许多开发人员往往会认为使用新技术存在未知的风险，而且要学的东西很多，未必就比使用旧技术更有效，因而一次次地失去了采用新技术的机会，从而止步不前。译者也算是一个走在时代前端的人了，有新技术肯定会首先去学习和了解，但当真的要将新技术应用到项目的时候，往往会打退堂鼓。不过，只要使用新技术做一些演示，往往会有效地消除心魔。例如，我在 MVC 出现之前一直使用 Web 窗体来创建网站，一直不敢尝试使

用 MVC，后来有一个项目，先用 MVC 做了一个演示，尝试之后发现开发效率比使用熟悉的 Web 窗体实在高太多，等项目完成就已经对 MVC 技术相当娴熟了。以上笔墨是译者多年来的工作心得，仅供参考。

在这里要感谢清华大学出版社的编辑们，他们为本书的翻译投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。本书全部章节由李铭翻译，参与本书翻译活动的还有孔祥亮、陈跃华、杜思明、熊晓磊、曹汉鸣、陶晓云、王通、方峻、李小凤、曹晓松、蒋晓冬、邱培强、洪妍、李亮辉、高娟妮、曹小震、陈笑。在此，一并表示感谢！

对于这本经典之作，译者在翻译过程中力求忠于原文，做到“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。感激不尽！

译　者

作 者 简 介

Christian Nagel 是 Microsoft RD、Microsoft MVP、thinktecture 的合作伙伴、CN 革新技术的奠基人，他还是一位软件架构师和开发人员，为开发 Microsoft .NET 解决方案提供培训和咨询服务。他具备超过 25 年的软件开发经验。Christian 从 PDP 11 和 VAX/VMS 系统开始其计算机生涯，熟悉各种语言和平台。自从 2000 年以来(那时.NET 还只是一个技术框架)，他就开始使用各种.NET 技术构建大量.NET 解决方案。目前，他主要开发 Windows Store 应用程序来访问 Windows Azure 服务。他具备 Microsoft 技术的深厚功底，编写了大量图书，并获得了 Microsoft 认证培训师和专业开发人员证书。Christian 在国际会议上发表演讲(如 TechEd 和 Tech Days)并创立 INETA Europe，以支持.NET 用户组。通过 Web 站点 www.cninnovation.com 和 www.thinktecture.com 可以联系 Christian，其推持是@christiannagel。

Jay Glynn 开发软件的时间超过 20 年，使用 PICK Basic 为 PICK 操作系统编写应用程序。到目前为止，他使用过 Paradox PAL and Object PAL、Delphi、VBA、Visual Basic、C、Java 和 C#编写软件。他目前是 VGT 的高级软件工程师，编写基于 Web 的应用程序。

Morgan Skinner 年轻时对 Sinclair ZX80 很感兴趣，在校期间就开始了计算机生涯，当时他对教师编写的一些代码不感兴趣，便开始用汇编语言编程。从此以后他使用各种语言和平台，包括 VAX 宏汇编程序、Pascal、Modula2、Smalltalk、X86 汇编语言、PowerBuilder、C/C++、VB 和目前的 C#，自从 2000 年发布 PDC 以来，他就用.NET 编程，而且非常喜欢.NET，于是在 2001 年加入 Microsoft。他现在是一位独立顾问。

技术编辑简介

Don Reamey 是 TIBCO Software 的架构师/首席工程师，负责 TIBCO Sportfire 商务智能分析软件。在加入 TIBCO 之前，Don 在 Microsoft 做了 12 年的软件开发工程师，其主要工作是开发 SharePoint、SharePoint Online 和 InfoPath Forms Service。Don 还用 10 年的时间为资本市场编写财务服务软件。

George Evjen 是 ArchitectNow 的开发总监，ArchitectNow 是位于圣路易斯的一家咨询公司，专注于定制客户应用程序的结构、设计和开发，其客户既有小型技术初创企业，也有全球大公司。George 在涉足软件业之前，有十余年时间在各种级别的高等院校担任篮球教练的工作，由于能在几乎所有的情况下用富有感染力的积极态度激励队员，因此 George 非常适合直接领导 ArchitectNow 的许多大项目和客户。他不仅是一位顶尖的开发人员，还负责 ArchitectNow 与公司外部承包商和资源之间的协调工作。

George 在 Microsoft 所有基于 Web 和 XAML 的技术和最新的 Web 框架方面都有丰富的经验和专业知识，他擅长企业级 WPF、Silverlight、Windows 8 项目，以及 ASP.NET MVC 商业应用程序的开发。他在用户组和会议上发言，其主题包括激励领导、项目管理和组织。George 和 ArchitectNow 的更多信息可访问 http://www.architectnow.net。

致　　谢

非常感谢 Adaobi Obi Tulton、Maureen Spears 和 Luann Rouff 提高了本书的可读性，非常感谢 Mary James、Jim Minatel 和 Wiley 中为出版本书提供过帮助的每个人，我还要感谢妻子和孩子支持我的写作。你们都是我的力量源泉。

——Christian Nagel

我想感谢妻儿支持我完成本书，我还要感谢 Wiley 为出版本书付出努力的所有人。

——Jay Glynn

前　　言

对于开发人员，把 C#语言及其相关联的.NET Framework 环境描述为最重要的新技术一点都不夸张。.NET 提供了一种环境。在这个环境中，可以开发在 Windows 上运行的几乎所有应用程序，而 C#是专门用于.NET Framework 的编程语言。例如，使用 C#可以编写动态 Web 页面、Windows Presentation Foundation 应用程序、XML Web 服务、分布式应用程序的组件、数据库访问组件、传统的 Windows 桌面应用程序，甚或可以联机/脱机运行的新型智能客户端应用程序。本书介绍.NET Framework 4.5.1。如果读者使用以前的版本编码，本书的一些章节就不适用。本书将标注出专用于.NET Framework 4.5 和 4.5.1 的新增内容。

不要被这个架构名称中的.NET 所迷惑，认为这是一个只关注 Internet 的架构。这个名称中的.NET 仅强调 Microsoft 相信分布式应用程序是未来的趋势，即处理过程分布在客户端和服务器上。理解 C#不仅仅是编写 Internet 或网络识别应用程序的一种语言也很重要。它还提供了一种编写 Windows 平台上几乎任何类型的软件或组件的方式。另外，C#和.NET 都对开发人员编写程序的方式进行了革新，更易于实现在 Windows 上的编程。

那么，.NET 和 C#有什么优点？

.NET 和 C#的重要性

为了理解.NET 的重要性，就一定要了解过去 20 年来出现的许多 Windows 技术的本质。尽管所有 Windows 操作系统在表面上看来完全不同，但从 Windows 3.1(1993 年引入)到 Windows 8.1 和 Windows Server 2012 R2，在内核上都有相同的 Windows API 用于 Windows 桌面和服务器应用程序。在我们转而使用 Windows 的新版本时，虽然 API 中增加了非常多的新功能，但这是一个演化和扩展 API 的过程，并非替换它。

在 Windows 8 中，操作系统的主 API 被 Windows 运行库替代。但这个运行库仍部分基于 Windows API。

开发 Windows 软件所使用的许多技术和架构也是这样。例如，组件对象模型(Component Object Model，COM)源自对象链接和嵌入(Object Linking and Embedding，OLE)。最初，因为它在很大程度上仅把不同类型的 Office 文档链接在一起，所以利用它，例如，可以把一个小型 Excel 电子表格放在 Word 文档中。之后，它逐步演化为 COM、DCOM(Distributed COM，分布式组件对象模型)和最终的 COM+。COM+是一种复杂的技术，它是几乎所有组件通信方式的基础，实现了事务处理、消息传输服务和对象池。

Microsoft 选择这种软件革新方法的原因非常明显：它关注后向兼容性。在过去的这些年中，

人们编写了大量 Windows 第三方软件，如果 Microsoft 每次都引入一项不遵循现有基本代码的新技术，Windows 就不会获得今天的成功。

后向兼容性是 Windows 技术的极其重要的功能，也是 Windows 平台的一个长处。但它有一个很大的缺点：每次某项技术更新换代，增加了新功能后，它都会比以前更复杂。

很明显，对此必须改进。Microsoft 不可能一直扩展相同的开发工具和语言，总是使它们越来越复杂，既要保证能跟上最新硬件的发展步伐，又要与 20 世纪 90 年代初开始流行的 Windows 产品向后兼容。如果要得到一系列简单而专业的语言、环境和开发工具，让开发人员轻松地编写一流的软件，就需要一个新的开端。

这就是 C#和.NET 的作用。粗略地说，.NET 是一种在 Windows 平台上编程的架构——一种 API。C#是一种从头开始设计的用于.NET 的语言，它可以利用.NET Framework 及其开发环境中的所有新增功能，以及在最近 25 年来出现的面向对象的编程方法。

在继续介绍前，必须先说明，后向兼容性并没有在这个演化进程中丧失。现有的程序仍可以使用，.NET 也兼容现有的软件。现在，在 Windows 上软件组件之间的通信几乎都使用 COM 实现。因此，.NET 能够提供现有 COM 组件的包装器(wrapper)，以便.NET 组件与之通信。

我们不需要学习了 C#才能给.NET 编写代码，因为 Microsoft 已经扩展了 C++，还对 Visual Basic 进行了很多改进，把它转变成了功能更强大的语言，并允许把用这些语言编写的代码用于.NET 环境。但其他这些语言都因有多年演化的遗留痕迹，并非一开始就用现在的技术来编写，导致它们不能用于.NET 环境。

本书将介绍 C#编程技术，同时提供.NET 体系结构工作原理的必要背景知识。我们不仅会介绍 C#语言的基础，还会给出使用各种相关技术的应用程序对应的示例，包括数据库访问、动态的 Web 页面、高级的图形和目录访问等。

Windows API 自从 1993 年发布的 Windows NT 以来一直在演化和扩展，但自从 2002 年以来，.NET Framework 对程序编写方式进行了重大的修改，2012 年又进行了一次很大的改动。每 10 年就会发生这种改变吗？Windows 8 现在提供了一种新的 API：用于 Windows Store 应用程序的 Windows 运行库(WinRT)。这个运行库是一个本机 API(类似于 Windows API)，它没有把.NET 运行库作为其核心，但提供了基于.NET 理念的非常好的新功能。Windows 8 包含这个 API 的第一个版本，可用于现代模式的应用程序。尽管它不基于.NET，但仍可以将.NET 的一个子集应用于 Windows Store 应用程序，用 C#编写该应用程序。这个新的运行库正在演化，在 Windows 8.1 中包含它的版本 2。本书也讨论了如何使用 C#和 WinRT 编写 Windows Store 应用程序。

.NET 的优点

前面阐述了.NET 的优点，但并没有说它会使开发人员的工作更易完成。本节将简要讨论.NET 的一些功能。

- **面向对象编程**：.NET Framework 和 C#从一开始就完全基于面向对象的原则。
- **优秀的设计**：一个基类库，它以一种非常直观的方式设计出来。
- **语言无关性**：在.NET 中，Visual Basic、C#和托管 C++等语言都可以编译为通用的中间语言(Intermediate Language)。这说明，语言可以用以前没有的方式交互操作。

- **对动态 Web 页面的更好支持**：虽然经典 ASP 具有很大的灵活性，但效率不是很高，这是因为它使用了解释性的脚本语言，且缺乏面向对象的设计，从而导致 ASP 代码比较混乱。.NET 使用 ASP.NET，为 Web 页面提供了一种集成支持。使用 ASP.NET，可以编译页面中的代码，这些代码还可以使用.NET 能识别的高级语言来编写，如 C#或 Visual Basic 2013。.NET 现在还添加了对最新 Web 技术的重要支持，如 Ajax 和 jQuery。
- **高效的数据访问**：一组.NET 组件，统称为 ADO.NET，提供了对关系数据库和各种数据源的高效访问。这些组件也可用于访问文件系统和目录。尤其是，.NET 内置了 XML 支持，可以处理从非 Windows 平台导入或导出的数据。
- **代码共享**：.NET 引入了程序集的概念，替代了传统的 DLL，可以完美无瑕地改进代码在应用程序之间的共享方式。程序集是解决版本冲突的正式设备，程序集的不同版本可以并存。
- **增强的安全性**：每个程序集还可以包含内置的安全信息，这些信息可以准确地指出哪种类型的用户或进程可以调用什么类的哪些方法。这样就可以非常准确地控制用户部署的程序集的使用方式。
- **对安装没有任何影响**：有两种类型的程序集，分别是共享程序集和私有程序集。共享程序集是可用于所有软件的公共库，而私有程序集只用于特殊软件。由于私有程序集完全自包含，因此安装过程非常简单。没有注册表项，只需要把相应的文件放在文件系统的相应文件夹中即可。
- **Web 服务的支持**：.NET 完全集成了对开发 Web 服务的支持，用户可以轻松地开发任何类型的应用程序。
- **Visual Studio 2013**：.NET 附带了一个 Visual Studio 2013 开发环境，它同样可以很好地利用 C++、C#、Visual Basic 2013 和 ASP.NET 或 XML 进行编码。Visual Studio 2013 集成了这个 IDE 所有以前版本中的各种语言专用环境中的所有最佳功能。
- **C#**：是使用.NET 的一种面向对象的强大且流行的语言。

第 1 章将详细讨论.NET 体系结构的优点。

.NET Framework 4.5 和 4.5.1 中的新增特性

.NET Framework 的第 1 版(1.0 版)在 2002 年发布，赢得了许多人的喝彩。.NET Framework 2.0 在 2005 年发布，是该架构的一个主要版本。2.0 版本的主要新特性是 C#和运行库中对泛型的支持(为泛型修改了 IL 代码)、新类和接口。.NET 3.0 以 2.0 运行库为基础，引入了创建 UI 的新方式(WPF 和 XAML，基于矢量的图形替代了基于像素的图形)和一个新的通信技术(WCF)。.NET 3.5 和 C# 3.0 引入了 LINQ，这是可用于所有数据源的查询语法。.NET Framework 4 是该产品的另一个重要的版本，也引入了运行库的一个新版本 4.0 和 C#的新版本 4.0，提供了动态语言集成和大量用于并行编程的新库。.NET Framework 4.5 基于 4.0 运行库的更新版本，包含了许多重要的新功能。.NET Framework 4.5.1 增加了一些新功能。但是，随着.NET Framework 中越来越多的库发布为 NuGet 包，越来越多的功能超出了.NET Framework 的界限。例如，Entity Framework、ASP.NET Web API 和其他.NET 库都获得了巨大的改进。

对于.NET Framework 的每个版本，Microsoft 总是试图确保对已开发出的代码进行尽可能少的不兼容的更改。到目前为止，Microsoft 在这方面做得很成功。

下面将详细描述 C# 5.0 和.NET Framework 4.5.1 中的一些新变化。

异步编程

阻塞 UI 对用户并不友好，如果 UI 不响应，用户就会不耐烦。也许读者在 Visual Studio 中也有这种经历。而 Visual Studio 在这方面好了许多，在许多情形下响应得更快。

.NET Framework 总是提供方法的异步调用。但是，使用同步方法比调用其异步变体容易得多。这在 C# 5.0 版本中有了改变。异步编程与编写同步程序一样容易。新的 C#关键字基于自从.NET 4.0 以来就有的.NET 并行库，现在该语言提供了高效功能。

Windows Store 应用程序和 Windows 运行库

Windows Store 应用程序可以使用 C#、Windows 运行库和.NET Framework 的一个子集编写，Windows 运行库是一个新的本机 API，提供了类似于.NET 的类、方法、属性和事件。使用语言投射功能，改善了.NET 运行库。在.NET 4.5 中，.NET 4.0 运行库进行了就地更新。

数据访问的改善

ADO.NET Entity Framework 提供了重要的新功能。其版本从.NET 4.0 的 4.0 改为.NET 4.5 的 5.0，再改为.NET 4.5.1 的 6.0。.NET 4.0 发布后，Entity Framework 已经在 4.1、4.2 和 4.3 中接受了更新。现在新功能，例如 Code First、空间类型、使用枚举、表值功能等，都可以使用了。

WPF 的改善

WPF 进行了改进，以编写 Windows 桌面应用程序。现在可以在非 UI 线程中填充集合，功能区控件现在是架构的一部分，通过事件的弱引用也更容易实现，数据验证可以用 INotifyDataErrorInfo 接口异步完成；实时绘图功能可以方便地动态排序、分组修改了的数据。

ASP.NET MVC

Visual Studio 2010 包含 ASP.NET MVC 2.0。Visual Studio 2013 发布后，就可以使用 ASP.NET MVC 5.0 了。ASP.NET MVC 提供了许多开发人员期待的、使用模型-视图-控制器来创建 ASP.NET 应用程序的方式。ASP.NET MVC 在开发人员构建的应用程序中提供了可测试性、灵活性和可维护性。ASP.NET MVC 不是 ASP.NET Web 窗体的替代品，而只是构建应用程序的另一种方式。

C#的优点

C#在某种程度上可以看作.NET 面向 Windows 环境的一种编程语言。在过去的 15 年中，Microsoft 给 Windows 和 Windows API 添加了许多功能，Visual Basic 2013 和 C++也进行了许多

扩展。虽然 Visual Basic 和 C++最终已成为非常强大的语言，但这两种语言也存在问题，因为它们保留了原来的一些遗留内容。

对于 Visual Basic 6 及其早期版本，它的主要优点是很容易理解，许多编程工作都很容易完成，从很大程度上对开发人员隐藏了 Windows API 和 COM 组件结构的详细信息。其缺点是因为 Visual Basic 从来没有实现真正意义上的面向对象，所以大型应用程序很难分解和维护。另外，因为 Visual Basic 的语法继承自 BASIC 的早期版本(BASIC 主要是为了让刚入门的程序员更容易理解，而不是为了编写大型商业应用程序)，所以不能真正成为结构良好或面向对象的编程语言。

另一方面，C++基于 ANSI C++语言定义。它与 ANSI 不完全兼容，因为 Microsoft 在 ANSI 定义标准化之前编写其 C++编译器，但它已经相当接近。但是，这导致了两个问题。首先，ANSI C++是在十几年前的技术条件下开发的，因此它不支持现在的概念(如 Unicode 字符串和生成 XML 文档)，某些古老的语法结构是为以前的编译器设计的(如成员函数的声明和定义是分开的)。其次，Microsoft 同时还试图把 C++演变为一种用于在 Windows 上执行高性能任务的语言，为此不得不在语言中添加大量 Microsoft 专用的关键字和各种库。其结果是在 Windows 上，该语言非常杂乱。让 C++开发人员描述字符串有多少种定义就可以证明这一点：char*、LPTSTR、string、CString(MFC 版本)、CString(WTL 版本)、wchar_t*、OLECHAR*等。

现在进入.NET 时代——一种全新的环境，它对这两种语言都进行了新的扩展。Microsoft 给 C++添加了许多 Microsoft 专用的关键字，并把 Visual Basic 演变为 Visual Basic 2013，保留了一些基本的 Visual Basic 语法，但在设计上完全不同于原始 Visual Basic，从实际应用的角度来看，Visual Basic 2013 是一种新语言。

在这里，Microsoft 决定给开发人员提供另一个选择——专门用于.NET、具有新起点的一种语言，即 C#。Microsoft 在正式场合将 C#描述为一种简单、现代、面向对象、类型非常安全、派生自 C 和 C++的编程语言。大多数独立的评论员对 C#的描述改为"派生自 C、C++和 Java"。这种描述在技术上非常准确，但没有表达出该语言的真正优点。从语法上看，C#非常类似于 C++和 Java，许多关键字都相同，C#也使用类似于 C++和 Java 的块结构，并用花括号({})来标记代码块，用分号分隔各行语句。对 C#代码的第一印象是它非常类似于 C++或 Java 代码。但在这些表面的类似性后面，C#学习起来要比 C++容易得多，与 Java 的难度相当。其设计比其他语言更适合现代开发工具，它同时具有 Visual Basic 的易用性，以及 C++的高性能、低级内存访问。C#包括以下一些功能：

- 完全支持类和面向对象编程，包括接口和实现继承、虚函数和运算符重载。
- 一致且定义完善的基本类型集。
- 对自动生成 XML 文档的内置支持。
- 自动清理动态分配的内存。
- 可以用用户定义的属性来标记类或方法。这可以用于文档，对编译有一定的影响(例如，把方法标记为只在调试版本中编译)。
- 可以完全访问.NET 基类库，并易于访问 Windows API(如果实际需要它，这就不常见)。
- 可以使用指针和直接访问内存，但 C#语言可以在没有它们的条件下访问内存。
- 以 Visual Basic 的风格支持属性和事件。

- 改变编译器选项，可以把程序编译为可执行文件或.NET 组件库，该组件库可以用与 ActiveX 控件(COM 组件)相同的方式由其他代码调用。
- C#可以用于编写 ASP.NET 动态 Web 页面和 XML Web 服务。

应该指出，对于上述大多数功能，Visual Basic 2013 和 Managed C++也具备。事实上，虽然 C#从一开始就使用.NET，但对.NET 功能的支持不仅更完整，而且在比其他语言更合适的语法环境中提供了这些功能。C#语言本身非常类似于 Java，但其中有一些改进，尤其是，Java 并不应用于.NET 环境。

在结束这个主题前，还要指出 C#的两个局限性。一方面是该语言不适用于编写时间急迫或性能非常高的代码，例如一个要占用 1000 或 1050 个机器周期的循环，并在不需要这些资源时，立即清理它们。在这方面，C++可能仍是所有低级语言中的佼佼者。另一方面是 C#缺乏性能极高的应用程序所需要的关键功能，包括能够指定那些保证在代码的特定地方运行的内联函数和析构函数。但这类应用程序非常少。

编写和运行 C#代码的环境

.NET Framework 4.5.1 运行在 Windows Vista/7/8/8.1 和服务器操作系统 Windows Server 2008、 2008 R2、2012 和 2012 R2 上。要使用.NET 编写代码，需要安装.NET 4.5.1 SDK。

此外，除非要使用文本编辑器或其他第三方开发环境来编写 C#代码，否则用户几乎肯定也希望使用 Visual Studio 2013。运行托管代码不需要安装完整的 SDK，但需要.NET 运行库。需要把.NET 运行库和代码分布到还没有安装它的客户端上。

本书内容

本书首先在第 1 章介绍.NET 的整体体系结构，给出编写托管代码所需要的背景知识，此后本书分几部分介绍 C#语言及其在各个领域中的应用。

第 I 部分——C#语言

本部分给出 C#语言的背景知识。尽管这一部分假定读者是有经验的编程人员，但它没有假设读者拥有任何特殊语言的知识。首先介绍 C#的基本语法和数据类型，再介绍 C#的面向对象功能，之后是 C#中的一些高级编程主题。

第 II 部分——Visual Studio

本部分介绍全世界 C#开发人员都使用的主要 IDE：Visual Studio 2013。本部分的两章探讨使用工具构建基于.NET Framework 4.5.1 的应用程序的最佳方式，另外，本部分还讨论项目的部署。

第III部分——基础

本部分介绍在.NET 环境中编程的规则。特别是安全性、线程、本地化、事务、构建 Windows 服务的方式，以及将自己的库生成为程序集的方式等主题。其中一部分介绍如何使用平台调用和 COM 交互操作功能，与本地代码和程序集进行交互操作。本部分还讨论了 Windows 运行库

与.NET 的区别，以及如何编写 Windows 8 模式的程序。

第Ⅳ部分——数据

本部分介绍如何使用 ADO.NET 访问数据库，学习 ADO.NET Entity Framework。我们可以使用核心 ADO.NET 获得最佳性能，而使用 ADO.NET Entity Framework 可以方便地把对象映射到关系上。还讨论了现在可以使用的 Model First、Database First 和 Code First 编程模型。我们还详细说明.NET 对 XML 的支持，以及如何使用 LINQ 查询 XML 数据源。

第Ⅴ部分——显示

本部分首先阐述如何编写基于 Windows Presentation Foundation 的应用程序，介绍不同的控件类型、样式、资源和数据绑定，以及如何创建固定的和流畅的文档并打印出来。本部分还会介绍如何创建 Windows Store 应用程序，使用图片生成更漂亮的 UI、网格，以及与其他应用程序交互操作的协定。最后讨论 ASP.NET 提供的许多新功能，用 ASP.NET Web 窗体创建 Web 站点、ASP.NET MVC 和动态数据。

第Ⅵ部分——通信

这一部分介绍通信，主要论述独立于平台使用 Windows Communication Foundation(WCF)和 ASP.NET Web API 进行通信的服务。通过消息队列，揭示了断开连接的异步通信。本部分还介绍如何利用 Windows Workflow Foundation(WF)和对等网络。

如何下载本书的示例代码

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/go/procsharp 上下载。登录到站点 http://www.wrox.com/上，使用 Search 工具或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：

许多图书的书名都很相似，所以通过 ISBN 查找本书是最简单的，本书英文版的 ISBN 是 978-1-118-83303-2。

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。

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，但错误总是难免的，如果你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。

要在网站上找到本书的勘误表，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 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。

如果在 Book Errata 页面上没有看到你找出的错误，请进入 www.worx.com/contact/techsupport.shtml，填写表单，发电子邮件，我们就会检查你的信息，如果是正确的，就在本书的勘误表中粘贴一条消息，我们将在本书的后续版本中采用。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于传送与 Wrox 图书相关的信息和相关技术，与其他读者和技术用户交流。该论坛提供了订阅功能，当论坛上有新帖子时，会给你发送你选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 http://p2p.wrox.com 上有许多不同的论坛，帮助读者阅读本书，在读者开发自己的应用程序时，也可以从这个论坛中获益。要加入这个论坛，必须执行下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读其内容，单击 Agree 按钮。

(3) 提供加入论坛所需的信息及愿意提供的可选信息，单击 Submit 按钮。

(4) 然后就可以收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

提示：

不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，才能发送自己的信息。

加入论坛后，就可以发送新信息，回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作原理，以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

目　录

第I部分　C# 语 言

第1章　.NET体系结构 ······ 2
1.1　C#与.NET的关系 ······ 2
1.2　公共语言运行库 ······ 3
1.2.1　平台无关性 ······ 3
1.2.2　提高性能 ······ 3
1.2.3　语言的互操作性 ······ 4
1.3　中间语言 ······ 6
1.3.1　面向对象和接口的支持 ······ 6
1.3.2　不同的值类型和引用类型 ······ 7
1.3.3　强数据类型化 ······ 7
1.3.4　通过异常处理错误 ······ 12
1.3.5　特性的使用 ······ 12
1.4　程序集 ······ 12
1.4.1　私有程序集 ······ 13
1.4.2　共享程序集 ······ 13
1.4.3　反射 ······ 14
1.4.4　并行编程 ······ 14
1.4.5　异步编程 ······ 14
1.5　.NET Framework类 ······ 15
1.6　名称空间 ······ 15
1.7　用C#创建.NET应用程序 ······ 16
1.7.1　创建ASP.NET应用程序 ······ 16
1.7.2　使用WPF ······ 17
1.7.3　Windows Store应用程序 ······ 18
1.7.4　Windows服务 ······ 18
1.7.5　WCF ······ 18
1.7.6　Windows WF ······ 19
1.8　C#在.NET企业体系结构中的作用 ······ 19
1.9　小结 ······ 20

第2章　核心C# ······ 22
2.1　C#基础 ······ 23
2.2　第一个C#程序 ······ 23
2.2.1　代码 ······ 23
2.2.2　编译并运行程序 ······ 23
2.2.3　详细介绍 ······ 24
2.3　变量 ······ 26
2.3.1　变量的初始化 ······ 26
2.3.2　类型推断 ······ 27
2.3.3　变量的作用域 ······ 28
2.3.4　常量 ······ 30
2.4　预定义数据类型 ······ 31
2.4.1　值类型和引用类型 ······ 31
2.4.2　CTS类型 ······ 32
2.4.3　预定义的值类型 ······ 32
2.4.4　预定义的引用类型 ······ 35
2.5　流控制 ······ 37
2.5.1　条件语句 ······ 37
2.5.2　循环 ······ 40
2.5.3　跳转语句 ······ 43
2.6　枚举 ······ 44
2.7　名称空间 ······ 46
2.7.1　using语句 ······ 47
2.7.2　名称空间的别名 ······ 48
2.8　Main()方法 ······ 48
2.8.1　多个Main()方法 ······ 49

2.8.2 给 Main()方法传递参数……50
2.9 有关编译 C#文件的更多内容……50
2.10 控制台 I/O……52
2.11 使用注释……54
2.11.1 源文件中的内部注释……54
2.11.2 XML 文档……54
2.12 C#预处理器指令……56
2.12.1 #define 和#undef……57
2.12.2 #if、#elif、#else 和#endif……57
2.12.3 #warning 和#error……58
2.12.4 #region 和#endregion……58
2.12.5 #line……59
2.12.6 #pragma……59
2.13 C#编程规则……59
2.13.1 关于标识符的规则……59
2.13.2 用法约定……60
2.14 小结……66
第 3 章 对象和类型……67
3.1 创建及使用类……67
3.2 类和结构……68
3.3 类……69
3.3.1 数据成员……69
3.3.2 函数成员……69
3.3.3 只读字段……81
3.4 匿名类型……82
3.5 结构……82
3.5.1 结构是值类型……84
3.5.2 结构和继承……84
3.5.3 结构的构造函数……85
3.6 弱引用……85
3.7 部分类……86
3.8 静态类……87
3.9 Object 类……88
3.9.1 System.Object()方法……88
3.9.2 ToString()方法……89
3.10 扩展方法……90
3.11 小结……91
第 4 章 继承……92
4.1 继承……92
4.2 继承的类型……92
4.2.1 实现继承和接口继承……92
4.2.2 多重继承……93
4.2.3 结构和类……93
4.3 实现继承……93
4.3.1 虚方法……94
4.3.2 隐藏方法……95
4.3.3 调用函数的基类版本……96
4.3.4 抽象类和抽象函数……97
4.3.5 密封类和密封方法……97
4.3.6 派生类的构造函数……98
4.4 修饰符……102
4.4.1 可见性修饰符……103
4.4.2 其他修饰符……103
4.5 接口……104
4.5.1 定义和实现接口……105
4.5.2 派生的接口……108
4.6 小结……109
第 5 章 泛型……110
5.1 泛型概述……110
5.1.1 性能……111
5.1.2 类型安全……112
5.1.3 二进制代码的重用……112
5.1.4 代码的扩展……113
5.1.5 命名约定……113
5.2 创建泛型类……113
5.3 泛型类的功能……117
5.3.1 默认值……118
5.3.2 约束……118
5.3.3 继承……120
5.3.4 静态成员……121

5.4 泛型接口 ……122
5.4.1 协变和抗变……122
5.4.2 泛型接口的协变 ……123
5.4.3 泛型接口的抗变 ……125
5.5 泛型结构 ……125
5.6 泛型方法 ……128
5.6.1 泛型方法示例 ……128
5.6.2 带约束的泛型方法 ……129
5.6.3 带委托的泛型方法 ……130
5.6.4 泛型方法规范 ……131
5.7 小结 ……132
第6章 数组……133
6.1 同一类型和不同类型的多个对象……133
6.2 简单数组 ……134
6.2.1 数组的声明……134
6.2.2 数组的初始化 ……134
6.2.3 访问数组元素 ……135
6.2.4 使用引用类型 ……136
6.3 多维数组 ……137
6.4 锯齿数组 ……138
6.5 Array类……139
6.5.1 创建数组……139
6.5.2 复制数组……140
6.5.3 排序……141
6.6 数组作为参数 ……144
6.6.1 数组协变……144
6.6.2 ArraySegment<T> ……144
6.7 枚举 ……145
6.7.1 IEnumerator接口 ……146
6.7.2 foreach语句……146
6.7.3 yield语句……147
6.8 元组 ……152
6.9 结构比较 ……152
6.10 小结 ……155
第7章 运算符和类型强制转换……156
7.1 运算符和类型转换……156
7.2 运算符……156
7.2.1 运算符的简化操作 ……158
7.2.2 运算符的优先级 ……162
7.3 类型的安全性……163
7.3.1 类型转换……163
7.3.2 装箱和拆箱……167
7.4 比较对象的相等性……168
7.4.1 比较引用类型的相等性……168
7.4.2 比较值类型的相等性……169
7.5 运算符重载……169
7.5.1 运算符的工作方式 ……170
7.5.2 运算符重载的示例：Vector结构 ……171
7.6 用户定义的类型强制转换……178
7.6.1 实现用户定义的类型强制转换 …179
7.6.2 多重类型强制转换 ……185
7.7 小结……188
第8章 委托、lambda表达式和事件……189
8.1 引用方法……189
8.2 委托……190
8.2.1 声明委托……190
8.2.2 使用委托……191
8.2.3 简单的委托示例 ……194
8.2.4 Action<T>和Func<T>委托 ……196
8.2.5 BubbleSorter示例……197
8.2.6 多播委托……199
8.2.7 匿名方法……203
8.3 lambda表达式……204
8.3.1 参数 ……204
8.3.2 多行代码……205
8.3.3 闭包 ……205
8.3.4 使用foreach语句的闭包 ……206
8.4 事件……207

8.4.1 事件发布程序 ……207
8.4.2 事件侦听器……209
8.4.3 弱事件……210
8.5 小结 ……214

第 9 章 字符串和正则表达式……215
9.1 System.String 类……216
9.1.1 创建字符串……217
9.1.2 StringBuilder 成员 ……220
9.1.3 格式字符串……221
9.2 正则表达式 ……227
9.2.1 正则表达式概述 ……227
9.2.2 RegularExpressionsPlayaround 示例 ……228
9.2.3 显示结果……230
9.2.4 匹配、组合和捕获 ……232
9.3 小结 ……233

第 10 章 集合 ……234
10.1 概述 ……235
10.2 集合接口和类型 ……235
10.3 列表 ……236
10.3.1 创建列表 ……238
10.3.2 只读集合 ……247
10.4 队列 ……247
10.5 栈 ……251
10.6 链表 ……252
10.7 有序列表 ……258
10.8 字典 ……259
10.8.1 键的类型 ……260
10.8.2 字典示例 ……261
10.8.3 Lookup 类 ……265
10.8.4 有序字典 ……265
10.9 集 ……266
10.10 可观察的集合 ……268
10.11 位数组 ……269
10.11.1 BitArray 类 ……270
10.11.2 BitVector32 结构 ……272
10.12 不变的集合……274
10.13 并发集合……276
10.13.1 创建管道 ……277
10.13.2 使用BlockingCollection …… 279
10.13.3 使用ConcurrentDictionary … 281
10.13.4 完成管道 ……282
10.14 性能……284
10.15 小结……285

第 11 章 LINQ ……286
11.1 LINQ 概述 ……286
11.1.1 列表和实体……287
11.1.2 LINQ 查询……290
11.1.3 扩展方法……291
11.1.4 推迟查询的执行 ……292
11.2 标准的查询操作符……294
11.2.1 筛选 ……296
11.2.2 用索引筛选……296
11.2.3 类型筛选……297
11.2.4 复合的 from 子句……297
11.2.5 排序 ……298
11.2.6 分组……299
11.2.7 对嵌套的对象分组……300
11.2.8 内连接……301
11.2.9 左外连接……303
11.2.10 组连接 ……303
11.2.11 集合操作 ……306
11.2.12 合并 ……308
11.2.13 分区 ……309
11.2.14 聚合操作符……310
11.2.15 转换操作符……311
11.2.16 生成操作符……312
11.3 并行 LINQ……313
11.3.1 并行查询……313
11.3.2 分区器……314
11.3.3 取消 ……314

11.4 表达式树……315
11.5 LINQ 提供程序……318
11.6 小结……319

第 12 章 动态语言扩展……320
12.1 DLR……320
12.2 dynamic 类型……321
12.3 包含 DLR ScriptRuntime……325
12.4 DynamicObject 和 ExpandoObject……328
12.4.1 DynamicObject……328
12.4.2 ExpandoObject……330
12.5 小结……331

第 13 章 异步编程……332
13.1 异步编程的重要性……332
13.2 异步模式……333
13.2.1 同步调用……340
13.2.2 异步模式……341
13.2.3 基于事件的异步模式……342
13.2.4 基于任务的异步模式……343
13.3 异步编程的基础……345
13.3.1 创建任务……345
13.3.2 调用异步方法……346
13.3.3 延续任务……346
13.3.4 同步上下文……347
13.3.5 使用多个异步方法……347
13.3.6 转换异步模式……348
13.4 错误处理……349
13.4.1 异步方法的异常处理……350
13.4.2 多个异步方法的异常处理……350
13.4.3 使用 AggregateException 信息……351
13.5 取消……352
13.5.1 开始取消任务……352
13.5.2 使用框架特性取消任务……352
13.5.3 取消自定义任务……353
13.6 小结……353

第 14 章 内存管理和指针……354
14.1 内存管理……354
14.2 后台内存管理……354
14.2.1 值数据类型……355
14.2.2 引用数据类型……356
14.2.3 垃圾回收……358
14.3 释放非托管的资源……360
14.3.1 析构函数……360
14.3.2 IDisposable 接口……361
14.3.3 实现 IDisposable 接口和析构函数……362
14.4 不安全的代码……364
14.4.1 用指针直接访问内存……364
14.4.2 指针示例：PointerPlayground……373
14.4.3 使用指针优化性能……377
14.5 小结……380

第 15 章 反射……381
15.1 在运行期间处理和检查代码……381
15.2 自定义特性……382
15.2.1 编写自定义特性……382
15.2.2 自定义特性示例：WhatsNewAttributes……386
15.3 反射……388
15.3.1 System.Type 类……388
15.3.2 TypeView 示例……391
15.3.3 Assembly 类……393
15.3.4 完成 WhatsNewAttributes 示例……394
15.4 小结……397

第 16 章 错误和异常……398
16.1 简介……398
16.2 异常类……399

16.3 捕获异常……400
16.3.1 实现多个 catch 块……402
16.3.2 在其他代码中捕获异常……406
16.3.3 System.Exception 属性……406
16.3.4 没有处理异常时所发生的情况……406
16.3.5 嵌套的 try 块……407
16.4 用户定义的异常类……409
16.4.1 捕获用户定义的异常……410
16.4.2 抛出用户定义的异常……411
16.4.3 定义用户定义的异常类……414
16.5 调用者信息……416
16.6 小结……417

第II部分 Visual Studio

第17章 Visual Studio 2013……419
17.1 使用 Visual Studio 2013……419
17.1.1 项目文件的改进……421
17.1.2 Visual Studio 的版本……422
17.1.3 Visual Studio 设置……423
17.2 创建项目……423
17.2.1 面向多个版本的.NET Framework……424
17.2.2 选择项目类型……426
17.3 浏览并编写项目……429
17.3.1 Solution Explorer……429
17.3.2 使用代码编辑器……435
17.3.3 学习和理解其他窗口……437
17.3.4 排列窗口……441
17.4 构建项目……441
17.4.1 构建、编译和生成……441
17.4.2 调试版本和发布版本……442
17.4.3 选择配置……443
17.4.4 编辑配置……444
17.5 调试代码……445
17.5.1 设置断点……445
17.5.2 使用数据提示和调试器可视化工具……446
17.5.3 监视和修改变量……447
17.5.4 异常……448
17.5.5 多线程……449
17.5.6 IntelliTrace……449
17.6 重构工具……450
17.7 体系结构工具……451
17.7.1 依赖项关系图……452
17.7.2 层关系图……453
17.8 分析应用程序……454
17.8.1 代码地图……454
17.8.2 序列图……454
17.8.3 探查器……455
17.8.4 Concurrency Visualizer……457
17.8.5 Code Analysis……458
17.8.6 Code Metrics……459
17.9 单元测试……459
17.9.1 创建单元测试……459
17.9.2 运行单元测试……460
17.9.3 预期异常……462
17.9.4 测试全部代码路径……462
17.9.5 外部依赖……463
17.9.6 Fakes Framework……466
17.10 Windows Store 应用程序、WCF、WF 等……467
17.10.1 使用Visual Studio 生成WCF应用程序……467
17.10.2 使用Visual Studio 生成WF应用程序……468
17.10.3 使用Visual Studio 2013生成Windows Store应用程序……469
17.11 小结……470

第18章 部署……471
18.1 部署是应用程序生命周期的一部分……471

18.2 部署的规划……472
18.2.1 部署选项……472
18.2.2 部署要求……472
18.2.3 部署.NET 运行库……473
18.3 传统的部署选项……473
18.3.1 xcopy 部署……474
18.3.2 xcopy 和 Web 应用程序……475
18.3.3 Windows Installer……475
18.4 ClickOnce……475
18.4.1 ClickOnce 操作……476
18.4.2 发布 ClickOnce 应用程序……476
18.4.3 ClickOnce 设置……477
18.4.4 ClickOnce 文件的应用程序缓存……479
18.4.5 应用程序的安装……479
18.4.6 ClickOnce 部署 API……480
18.5 Web 部署……481
18.5.1 Web 应用程序……481
18.5.2 配置文件……482
18.5.3 创建 Web Deploy 包……482
18.6 Windows Store 应用程序……483
18.6.1 创建应用程序包……484
18.6.2 Windows App Certification Kit……485
18.6.3 旁加载……486
18.6.4 Windows 部署 API……486
18.7 小结……488

第Ⅲ部分 基 础

第 19 章 程序集……490
19.1 程序集的含义……490
19.1.1 程序集的功能……491
19.1.2 程序集的结构……492
19.1.3 程序集清单……492
19.1.4 名称空间、程序集和组件……493
19.1.5 私有程序集和共享程序集……493
19.1.6 附属程序集……493
19.1.7 查看程序集……493
19.2 构建程序集……494
19.2.1 创建模块和程序集……494
19.2.2 程序集的特性……495
19.2.3 创建和动态加载程序集……497
19.3 应用程序域……500
19.4 共享程序集……504
19.4.1 强名……504
19.4.2 使用强名获得完整性……505
19.4.3 全局程序集缓存……506
19.4.4 创建共享程序集……506
19.4.5 创建强名……507
19.4.6 安装共享程序集……508
19.4.7 使用共享程序集……508
19.4.8 程序集的延迟签名……509
19.4.9 引用……510
19.4.10 本机映像生成器……511
19.5 配置.NET 应用程序……512
19.5.1 配置类别……512
19.5.2 绑定程序集……513
19.6 版本问题……514
19.6.1 版本号……515
19.6.2 通过编程方式获取版本……515
19.6.3 绑定到程序集版本……516
19.6.4 发行者策略文件……517
19.6.5 运行库的版本……518
19.7 在不同的技术之间共享程序集……519
19.7.1 共享源代码……519
19.7.2 可移植类库……520
19.8 小结……521
第 20 章 诊断……522
20.1 诊断概述……522
20.2 代码协定……523
20.2.1 前提条件……524

20.2.2 后置条件 ······525
20.2.3 不变量 ······526
20.2.4 纯粹性 ······527
20.2.5 接口的协定 ······527
20.2.6 简写 ······528
20.2.7 协定和遗留代码 ······529
20.3 跟踪 ······529
20.3.1 跟踪源 ······530
20.3.2 跟踪开关 ······532
20.3.3 跟踪侦听器 ······532
20.3.4 筛选器 ······534
20.3.5 相关性 ······535
20.3.6 使用 ETW 进行跟踪 ······539
20.3.7 使用 EventSource ······539
20.3.8 使用 EventSource 进行高级跟踪 ······541
20.4 事件日志 ······543
20.4.1 事件日志体系结构 ······543
20.4.2 事件日志类 ······544
20.4.3 创建事件源 ······546
20.4.4 写入事件日志 ······546
20.4.5 资源文件 ······547
20.5 性能监视 ······551
20.5.1 性能监视类 ······551
20.5.2 性能计数器生成器 ······551
20.5.3 添加 PerformanceCounter 组件 ······554
20.5.4 perfmon.exe ······556
20.6 小结 ······557
第 21 章 任务、线程和同步 ······558
21.1 概述 ······558
21.2 Parallel 类 ······560
21.2.1 用 Parallel.For()方法循环 ······560
21.2.2 使用 Parallel.ForEach()方法循环 ······563
21.2.3 通过 Parallel.Invoke()方法调用多个方法 ······564
21.3 任务 ······564
21.3.1 启动任务 ······565
21.3.2 Future——任务的结果 ······567
21.3.3 连续的任务 ······568
21.3.4 任务层次结构 ······569
21.4 取消架构 ······570
21.4.1 Parallel.For()方法的取消 ······570
21.4.2 任务的取消 ······571
21.5 线程池 ······572
21.6 Thread 类 ······574
21.6.1 给线程传递数据 ······575
21.6.2 后台线程 ······576
21.6.3 线程的优先级 ······577
21.6.4 控制线程 ······578
21.7 线程问题 ······578
21.7.1 争用条件 ······578
21.7.2 死锁 ······581
21.8 同步 ······583
21.8.1 lock 语句和线程安全 ······583
21.8.2 Interlocked 类 ······588
21.8.3 Monitor 类 ······589
21.8.4 SpinLock 结构 ······590
21.8.5 WaitHandle 基类 ······591
21.8.6 Mutex 类 ······591
21.8.7 Semaphore 类 ······593
21.8.8 Events 类 ······595
21.8.9 Barrier 类 ······598
21.8.10 ReaderWriterLockSlim 类 ······600
21.9 Timer 类 ······602
21.10 数据流 ······604
21.10.1 使用动作块 ······604
21.10.2 源和目标数据块 ······605
21.10.3 连接块 ······606
21.11 小结 ······608

第 22 章 安全性 ······ 609
22.1 概述 ······ 609
22.2 身份验证和授权 ······ 610
22.2.1 标识和 Principal ······ 610
22.2.2 角色 ······ 611
22.2.3 声明基于角色的安全性 ······ 612
22.2.4 声称 ······ 613
22.2.5 客户端应用程序服务 ······ 614
22.3 加密 ······ 619
22.3.1 签名 ······ 621
22.3.2 交换密钥和安全传输 ······ 622
22.4 资源的访问控制 ······ 625
22.5 代码访问安全性 ······ 628
22.5.1 第 2 级安全透明性 ······ 628
22.5.2 权限 ······ 629
22.6 使用证书发布代码 ······ 634
22.7 小结 ······ 635

第 23 章 互操作 ······ 636
23.1 .NET 和 COM 技术 ······ 636
23.1.1 元数据 ······ 637
23.1.2 释放内存 ······ 638
23.1.3 接口 ······ 638
23.1.4 方法绑定 ······ 639
23.1.5 数据类型 ······ 640
23.1.6 注册 ······ 640
23.1.7 线程 ······ 640
23.1.8 错误处理 ······ 641
23.1.9 事件 ······ 642
23.1.10 封送 ······ 642
23.2 在.NET 客户端中使用 COM 组件 ······ 643
23.2.1 创建 COM 组件 ······ 643
23.2.2 创建运行库可调用包装 ······ 649
23.2.3 使用 RCW ······ 650
23.2.4 通过动态语言扩展使用 COM 服务 ······ 651
23.2.5 线程问题 ······ 652
23.2.6 添加连接点 ······ 652
23.3 在 COM 客户端中使用 .NET 组件 ······ 654
23.3.1 COM 可调用包装 ······ 655
23.3.2 创建.NET 组件 ······ 655
23.3.3 创建类型库 ······ 656
23.3.4 COM 互操作特性 ······ 658
23.3.5 COM 注册 ······ 660
23.3.6 创建COM客户端应用程序 ··· 661
23.3.7 添加连接点 ······ 662
23.3.8 使用 sink 对象创建客户端 ··· 663
23.4 平台调用 ······ 665
23.5 小结 ······ 669

第 24 章 文件和注册表操作 ······ 670
24.1 文件和注册表 ······ 670
24.2 管理文件系统 ······ 671
24.2.1 表示文件和文件夹的 .NET 类 ······ 672
24.2.2 Path 类 ······ 674
24.2.3 FileProperties 示例 ······ 674
24.3 移动、复制和删除文件 ······ 679
24.3.1 FilePropertiesAndMovement 示例 ······ 679
24.3.2 FilePropertiesAndMovement 示例的代码 ······ 680
24.4 读写文件 ······ 682
24.4.1 读取文件 ······ 683
24.4.2 写入文件 ······ 684
24.4.3 流 ······ 685
24.4.4 缓存的流 ······ 687
24.4.5 使用 FileStream 类读写二进制文件 ······ 687
24.4.6 读写文本文件 ······ 692
24.5 映射内存的文件 ······ 698
24.6 读取驱动器信息 ······ 699

24.7 文件的安全性 ……701
24.7.1 从文件中读取 ACL ……701
24.7.2 从目录中读取 ACL ……702
24.7.3 添加和删除文件中的 ACL 项 ……704
24.8 读写注册表 ……705
24.8.1 注册表 ……706
24.8.2 .NET 注册表类 ……708
24.9 读写独立存储器 ……710
24.10 小结 ……714
第 25 章 事务处理 ……715
25.1 简介 ……715
25.2 概述 ……716
25.2.1 事务处理阶段 ……716
25.2.2 ACID 属性 ……717
25.3 数据库和实体类 ……717
25.4 传统的事务 ……719
25.4.1 ADO.NET 事务 ……719
25.4.2 System.EnterpriseServices ……721
25.5 System.Transactions ……722
25.5.1 可提交的事务 ……723
25.5.2 事务处理的升级 ……725
25.5.3 依赖事务 ……727
25.5.4 环境事务 ……729
25.6 隔离级别 ……736
25.7 自定义资源管理器 ……737
25.8 文件系统事务 ……743
25.9 小结 ……747
第 26 章 网络 ……748
26.1 网络 ……748
26.2 HttpClient 类 ……749
26.2.1 异步调用 Web 服务 ……749
26.2.2 标题 ……750
26.2.3 HttpContent ……752
26.2.4 HttpMessageHandler ……752
26.3 把输出结果显示为 HTML 页面 ……753
26.3.1 从应用程序中进行简单的 Web 浏览 ……754
26.3.2 启动 Internet Explorer 实例 ……755
26.3.3 给应用程序提供更多 IE 类型的功能 ……756
26.3.4 使用WebBrowser控件打印 ……761
26.3.5 显示所请求页面的代码 ……761
26.3.6 WebRequest类和WebResponse 类的层次结构 ……763
26.4 实用工具类 ……763
26.4.1 URI ……763
26.4.2 IP 地址和 DNS 名称 ……764
26.5 较低层的协议 ……766
26.5.1 使用 SmtpClient ……767
26.5.2 使用 TCP 类 ……769
26.5.3 TcpSend和TcpReceive示例 ……769
26.5.4 TCP 和 UDP ……771
26.5.5 UDP 类 ……771
26.5.6 Socket 类 ……772
26.5.7 WebSocket ……776
26.6 小结 ……779
第 27 章 Windows 服务 ……780
27.1 Windows 服务 ……780
27.2 Windows 服务的体系结构 ……781
27.2.1 服务程序 ……782
27.2.2 服务控制程序 ……783
27.2.3 服务配置程序 ……783
27.2.4 Windows 服务的类 ……783
27.3 创建 Windows 服务程序 ……784
27.3.1 创建服务的核心功能 ……784
27.3.2 QuoteClient 示例 ……787
27.3.3 Windows 服务程序 ……791
27.3.4 线程化和服务 ……795

27.3.5 服务的安装 ……795
27.3.6 安装程序 ……796
27.4 Windows服务的监控和控制 ……800
27.4.1 MMC管理单元 ……800
27.4.2 net.exe 实用程序 ……801
27.4.3 sc.exe 实用程序 ……801
27.4.4 Visual Studio Server Explorer ……801
27.4.5 编写自定义 ServiceController 类 ……802
27.5 故障排除和事件日志 ……809
27.6 小结 ……810
第28章 本地化 ……811
28.1 全球市场 ……811
28.2 System.Globalization 名称空间 ……812
28.2.1 Unicode 问题 ……812
28.2.2 区域性和区域 ……813
28.2.3 使用区域性 ……817
28.2.4 排序 ……823
28.3 资源 ……824
28.3.1 创建资源文件 ……824
28.3.2 资源文件生成器 ……824
28.3.3 ResourceWriter ……825
28.3.4 使用资源文件 ……826
28.3.5 System.Resources名称空间 ……830
28.4 使用 Visual Studio 的 Windows Forms 本地化 ……830
28.4.1 通过编程方式修改区域性 ……835
28.4.2 使用自定义资源消息 ……836
28.4.3 资源的自动回退 ……837
28.4.4 外包翻译 ……837
28.5 ASP.NET Web Forms 的本地化 ……838
28.6 用 WPF 本地化 ……839
28.6.1 用于 WPF 的.NET 资源 ……840
28.6.2 XAML 资源字典 ……841
28.7 自定义资源读取器 ……845
28.7.1 创建DatabaseResourceReader 类 ……845
28.7.2 创建DatabaseResourceSet 类 ……847
28.7.3 创建DatabaseResourceManager 类 ……847
28.7.4 DatabaseResourceReader 的客户端应用程序 ……848
28.8 创建自定义区域性 ……848
28.9 用 Windows Store 应用程序进行本地化 ……850
28.9.1 使用资源 ……851
28.9.2 使用多语言应用程序工具集进行本地化 ……851
28.10 小结 ……852
第29章 核心 XAML ……853
29.1 XAML 的作用 ……853
29.2 XAML 概述 ……854
29.2.1 元素如何映射到.NET 对象上 ……854
29.2.2 使用自定义.NET 类 ……856
29.2.3 把属性用作特性 ……857
29.2.4 把属性用作元素 ……857
29.2.5 基本的.NET 类型 ……858
29.2.6 使用集合和 XAML ……858
29.2.7 用 XAML 代码调用构造函数 ……859
29.3 依赖属性 ……859
29.3.1 创建依赖属性 ……860
29.3.2 强制值回调 ……861
29.3.3 值变更回调和事件 ……862
29.3.4 事件的冒泡和隧道 ……863
29.4 附加属性 ……866
29.5 标记扩展 ……868

29.6 创建自定义标记扩展 ……869
29.7 XAML 定义的标记扩展 ……870
29.8 读写 XAML ……871
29.9 小结 ……872

第 30 章 Managed Extensibility Framework ……873
30.1 概述 ……873
30.2 MEF 的体系结构 ……874
30.2.1 使用属性的 MEF ……875
30.2.2 基于约定的部件注册 ……880
30.3 定义协定 ……882
30.4 导出部件 ……883
30.4.1 创建部件 ……883
30.4.2 导出属性和方法 ……888
30.4.3 导出元数据 ……890
30.4.4 使用元数据进行惰性加载 ……892
30.5 导入部件 ……893
30.5.1 导入连接 ……895
30.5.2 部件的惰性加载 ……897
30.5.3 用惰性实例化的部件读取元数据 ……898
30.6 容器和出口提供程序 ……900
30.7 类别 ……902
30.8 小结 ……904

第 31 章 Windows 运行库 ……905
31.1 概述 ……905
31.1.1 .NET 与 Windows 运行库的比较 ……906
31.1.2 名称空间 ……906
31.1.3 元数据 ……908
31.1.4 语言投射 ……909
31.1.5 Windows运行库中的类型 ……911
31.2 Windows 运行库组件 ……912
31.2.1 集合 ……912
31.2.2 流 ……913
31.2.3 委托与事件 ……914
31.2.4 异步操作 ……914
31.3 Windows Store 应用程序 ……915
31.4 应用程序的生命周期 ……917
31.4.1 应用程序的执行状态 ……918
31.4.2 Suspension Manager ……919
31.4.3 导航状态 ……921
31.4.4 测试暂停 ……922
31.4.5 页面状态 ……922
31.5 应用程序的设置 ……924
31.6 小结 ……927

第Ⅳ部分 数 据

第 32 章 核心 ADO.NET ……929
32.1 ADO.NET 概述 ……929
32.1.1 名称空间 ……930
32.1.2 共享类 ……931
32.1.3 数据库专用类 ……931
32.2 使用数据库连接 ……932
32.2.1 管理连接字符串 ……933
32.2.2 高效地使用连接 ……934
32.2.3 事务 ……936
32.3 命令 ……938
32.3.1 执行命令 ……938
32.3.2 调用存储过程 ……941
32.4 快速数据访问：数据读取器 ……944
32.5 异步数据访问：使用 Task 和 await ……946
32.6 管理数据和关系：DataSet 类 ……948
32.6.1 数据表 ……949
32.6.2 数据列 ……950
32.6.3 数据关系 ……955
32.6.4 数据约束 ……956
32.7 XML 架构：用 XSD 生成代码 ……958
32.8 填充 DataSet 类 ……965

32.8.1 用数据适配器填充DataSet……965
32.8.2 从XML中填充DataSet类……966
32.9 持久化 DataSet 类的修改……966
32.9.1 通过数据适配器进行更新……966
32.9.2 写入 XML 输出结果……969
32.10 使用 ADO.NET……970
32.10.1 分层开发……970
32.10.2 生成 SQL Server 的键……972
32.10.3 命名约定……974
32.11 小结……975
第 33 章 ADO.NET Entity Framework……976
33.1 用 Entity Framework 编程……976
33.2 Entity Framework 映射……978
33.2.1 逻辑层……978
33.2.2 概念层……980
33.2.3 映射层……982
33.2.4 连接字符串……983
33.3 实体……983
33.4 对象上下文……984
33.5 关系……986
33.5.1 一个层次结构一个表……986
33.5.2 一种类型一个表……988
33.5.3 懒惰加载、延迟加载和预先加载……989
33.6 查询数据……991
33.6.1 Entity SQL……991
33.6.2 使用 DbSqlQuery……992
33.6.3 LINQ to Entities……993
33.7 把数据写入数据库……994
33.7.1 对象跟踪……994
33.7.2 改变信息……995
33.7.3 附加和分离实体……996
33.7.4 使用最后一个更改操作写入实体的更改……997
33.7.5 使用第一个更改操作写入实体的更改……997
33.7.6 写入实体的更改并处理冲突……998
33.8 使用 Code First 编程模型……999
33.8.1 定义实体类型……999
33.8.2 创建数据上下文……1000
33.8.3 创建数据库，存储实体……1000
33.8.4 数据库……1001
33.8.5 查询数据……1001
33.8.6 定制数据库的生成……1002
33.8.7 数据库的自动填充……1003
33.8.8 连接的弹性……1004
33.8.9 架构的迁移……1005
33.9 小结……1007
第 34 章 处理 XML……1008
34.1 XML……1008
34.2 .NET 支持的 XML 标准……1009
34.3 System.Xml 名称空间……1009
34.4 使用 System.Xml 类……1010
34.5 读写流格式的 XML……1011
34.5.1 使用 XmlReader 类……1011
34.5.2 使用 XmlReader 类进行验证……1015
34.5.3 使用 XmlWriter 类……1017
34.6 在.NET 中使用 DOM……1018
34.7 使用 XPathNavigator 类……1023
34.7.1 System.Xml.XPath 名称空间……1023
34.7.2 System.Xml.Xsl 名称空间……1028
34.7.3 调试 XSLT……1032
34.8 XML 和 ADO.NET……1034
34.8.1 将 ADO.NET 数据转换为 XML 文档……1034
34.8.2 把 XML 文档转换为 ADO.NET 数据……1040

34.9　在 XML 中序列化对象……1041
34.10　LINQ to XML 和.NET……1051
34.11　使用不同的 XML 对象……1051
34.11.1　XDocument 对象……1051
34.11.2　XElement 对象……1052
34.11.3　XNamespace 对象……1053
34.11.4　XComment 对象……1055
34.11.5　XAttribute 对象……1056
34.12　使用 LINQ 查询 XML 文档……1057
34.12.1　查询静态的XML文档……1057
34.12.2　查询动态的XML文档……1058
34.13　XML 文档的更多查询技术……1060
34.13.1　读取 XML 文档……1060
34.13.2　写入 XML 文档……1061
34.14　小结……1063

第Ⅴ部分　显　示

第 35 章　核心WPF……1065
35.1　理解 WPF……1066
35.1.1　名称空间……1066
35.1.2　类层次结构……1067
35.2　形状……1069
35.3　几何图形……1070
35.4　变换……1072
35.5　画笔……1073
35.5.1　SolidColorBrush……1073
35.5.2　LinearGradientBrush……1074
35.5.3　RadialGradientBrush……1074
35.5.4　DrawingBrush……1075
35.5.5　ImageBrush……1076
35.5.6　VisualBrush……1076
35.6　控件……1077
35.6.1　简单控件……1077
35.6.2　内容控件……1078
35.6.3　带标题的内容控件……1079
35.6.4　项控件……1081
35.6.5　带标题的项控件……1081
35.6.6　修饰……1081
35.7　布局……1082
35.7.1　StackPanel……1082
35.7.2　WrapPanel……1083
35.7.3　Canvas……1084
35.7.4　DockPanel……1084
35.7.5　Grid……1085
35.8　样式和资源……1086
35.8.1　样式……1087
35.8.2　资源……1088
35.8.3　系统资源……1090
35.8.4　从代码中访问资源……1090
35.8.5　动态资源……1090
35.8.6　资源字典……1091
35.9　触发器……1092
35.9.1　属性触发器……1093
35.9.2　多触发器……1094
35.9.3　数据触发器……1095
35.10　模板……1096
35.10.1　控件模板……1097
35.10.2　数据模板……1099
35.10.3　样式化列表框……1101
35.10.4　ItemTemplate……1102
35.10.5　列表框元素的控件模板……1103
35.11　动画……1105
35.11.1　时间轴……1105
35.11.2　非线性动画……1108
35.11.3　事件触发器……1109
35.11.4　关键帧动画……1111
35.12　可见状态管理器……1112
35.12.1　可见的状态……1113
35.12.2　变换……1115
35.13　3-D……1116

35.13.1 模型 ……1116
35.13.2 照相机 ……1118
35.13.3 光线 ……1118
35.13.4 旋转 ……1118
35.14 小结 ……1119

第36章 用WPF编写业务应用程序 ……1120
36.1 概述 ……1120
36.2 菜单和功能区控件 ……1121
36.2.1 菜单控件 ……1121
36.2.2 功能区控件 ……1122
36.3 Commanding ……1124
36.3.1 定义命令 ……1125
36.3.2 定义命令源 ……1126
36.3.3 命令绑定 ……1126
36.4 数据绑定 ……1127
36.4.1 BooksDemo应用程序内容 ……1128
36.4.2 用XAML绑定 ……1129
36.4.3 简单对象的绑定 ……1131
36.4.4 更改通知 ……1134
36.4.5 对象数据提供程序 ……1136
36.4.6 列表绑定 ……1138
36.4.7 主从绑定 ……1141
36.4.8 多绑定 ……1142
36.4.9 优先绑定 ……1144
36.4.10 值的转换 ……1145
36.4.11 动态添加列表项 ……1147
36.4.12 动态添加选项卡中的项 ……1148
36.4.13 数据模板选择器 ……1149
36.4.14 绑定到XML上 ……1151
36.4.15 绑定的验证和错误处理 ……1153
36.5 TreeView ……1161
36.6 DataGrid ……1165
36.6.1 自定义列 ……1167
36.6.2 行的细节 ……1168
36.6.3 用DataGrid进行分组 ……1168
36.6.4 实时成型 ……1171
36.7 小结 ……1177

第37章 用WPF创建文档 ……1178
37.1 简介 ……1178
37.2 文本元素 ……1179
37.2.1 字体 ……1179
37.2.2 TextEffect ……1180
37.2.3 内联 ……1181
37.2.4 块 ……1183
37.2.5 列表 ……1185
37.2.6 表 ……1185
37.2.7 块的锚定 ……1186
37.3 流文档 ……1188
37.4 固定文档 ……1192
37.5 XPS文档 ……1196
37.6 打印 ……1197
37.6.1 用PrintDialog打印 ……1198
37.6.2 打印可见元素 ……1198
37.7 小结 ……1200

第38章 Windows Store应用程序：用户界面 ……1201
38.1 概述 ……1201
38.2 Microsoft的现代设计 ……1202
38.2.1 内容，不是边框 ……1202
38.2.2 快速流畅 ……1203
38.2.3 可读性 ……1204
38.3 示例应用程序的核心功能 ……1204
38.3.1 文件和目录 ……1204
38.3.2 应用程序页面 ……1205
38.4 应用程序工具栏 ……1210
38.5 启动与导航 ……1213
38.6 布局的变化 ……1215
应用程序数据 ……1215
38.7 存储 ……1220
38.7.1 定义数据协定 ……1220
38.7.2 写入移动数据 ……1222

38.7.3 读取数据 ……1223
38.7.4 写入图像 ……1224
38.7.5 读取图像 ……1226
38.8 选择器 ……1227
38.9 活动的磁贴 ……1228
38.10 小结 ……1230
第39章 Windows Store 应用程序：协定和设备 ……1231
39.1 概述 ……1231
39.2 搜索 ……1232
39.3 共享协定 ……1234
39.3.1 共享源 ……1234
39.3.2 共享目标 ……1237
39.4 相机 ……1239
39.5 定位 ……1240
39.6 感应器 ……1243
39.6.1 光线 ……1244
39.6.2 罗盘 ……1244
39.6.3 加速计 ……1245
39.6.4 倾斜计 ……1246
39.6.5 陀螺仪 ……1246
39.6.6 方向 ……1246
39.6.7 Rolling Marble 示例 ……1247
39.7 小结 ……1249
第40章 核心 ASP.NET ……1250
40.1 用于 Web 应用程序的.NET Framework ……1250
40.1.1 ASP.NET Web Forms ……1251
40.1.2 ASP.NET Web Pages ……1251
40.1.3 ASP.NET MVC ……1252
40.2 Web 技术 ……1252
40.2.1 HTML ……1253
40.2.2 CSS ……1253
40.2.3 JavaScript 和 jQuery ……1253
40.3 托管和配置 ……1254
40.4 处理程序和模块 ……1256
40.4.1 创建自定义处理程序 ……1257
40.4.2 ASP.NET 处理程序 ……1258
40.4.3 创建自定义模块 ……1259
40.4.4 通用模块 ……1260
40.5 全局的应用程序类 ……1261
40.6 请求和响应 ……1262
40.6.1 使用 HttpRequest 对象 ……1262
40.6.2 使用 HttpResponse 对象 ……1264
40.7 状态管理 ……1264
40.7.1 视图状态 ……1265
40.7.2 cookie ……1266
40.7.3 会话 ……1267
40.7.4 应用程序状态 ……1270
40.7.5 缓存 ……1270
40.7.6 配置文件 ……1271
40.8 ASP.NET 身份系统 ……1275
40.8.1 基础知识 ……1276
40.8.2 存储和检索用户信息 ……1277
40.8.3 安全启动 ……1278
40.8.4 使用注册和身份验证 ……1278
40.9 小结 ……1279
第41章 ASP.NET Web Forms ……1281
41.1 概述 ……1281
41.2 ASPX 页面模型 ……1282
41.2.1 添加控件 ……1283
41.2.2 使用事件 ……1284
41.2.3 使用回送 ……1284
41.2.4 使用自动回送 ……1285
41.2.5 回送到其他页面 ……1285
41.2.6 定义强类型化的跨页面回送 ……1286
41.2.7 使用页面事件 ……1287
41.2.8 ASPX 代码 ……1288
41.2.9 服务器端控件 ……1290
41.3 母版页 ……1291
41.3.1 创建母版页 ……1291

41.3.2 使用母版页 ……1293
41.3.3 在内容页中定义母版页内容 ……1294
41.4 导航 ……1296
41.4.1 站点地图 ……1296
41.4.2 Menu 控件……1296
41.4.3 菜单路径 ……1297
41.5 验证用户输入 ……1297
41.5.1 使用验证控件 ……1298
41.5.2 使用验证摘要 ……1299
41.5.3 验证组……1300
41.6 访问数据 ……1300
41.6.1 使用 Object Framework ……1301
41.6.2 创建库……1302
41.6.3 使用 Object Data Source ……1303
41.6.4 编辑……1306
41.6.5 定制列……1307
41.6.6 在网格中使用模板……1308
41.7 安全性 ……1309
41.7.1 建立 ASP.NET 身份 ……1309
41.7.2 用户注册 ……1310
41.7.3 用户的身份验证……1311
41.7.4 用户授权 ……1312
41.8 Ajax……1313
41.8.1 ASP.NET AJAX 的概念……1314
41.8.2 ASP.NET AJAX 网站示例……1316
41.8.3 支持 ASP.NET AJAX 的网站配置……1319
41.8.4 添加 ASP.NET AJAX 功能……1320
41.9 小结 ……1326
第 42 章 ASP.NET MVC ……1328
42.1 ASP.NET MVC 概述……1328
42.2 定义路由 ……1330
42.2.1 添加路由 ……1330
42.2.2 路由约束……1331
42.3 创建控制器……1331
42.3.1 动作方法……1332
42.3.2 参数……1332
42.3.3 返回数据……1333
42.4 创建视图……1335
42.4.1 向视图传递数据 ……1336
42.4.2 Razor 语法……1337
42.4.3 强类型视图……1338
42.4.4 布局……1339
42.4.5 部分视图……1342
42.5 从客户端提交数据……1346
42.5.1 模型绑定器……1347
42.5.2 注释和验证……1348
42.6 HTML Helper……1350
42.6.1 简单的 Helper ……1350
42.6.2 使用模型数据 ……1351
42.6.3 定义 HTML 特性 ……1352
42.6.4 创建列表……1352
42.6.5 强类型化的 Helper……1353
42.6.6 编辑器扩展……1354
42.6.7 创建自定义 Helper……1355
42.6.8 模板……1355
42.7 创建数据驱动的应用程序……1356
42.7.1 定义模型……1357
42.7.2 创建控制器和视图……1358
42.8 动作过滤器……1364
42.9 身份验证和授权……1366
42.9.1 登录模型……1366
42.9.2 登录控制器……1366
42.9.3 登录视图……1368
42.10 小结……1370
第VI部分 通 信
第 43 章 WCF ……1372
43.1 WCF 概述……1372

43.1.1 SOAP……1374
43.1.2 WSDL……1374
43.1.3 REST……1375
43.1.4 JSON……1375
43.2 创建简单的服务和客户端……1375
43.2.1 定义服务和数据协定……1376
43.2.2 数据访问……1378
43.2.3 服务的实现……1379
43.2.4 WCF 服务宿主和 WCF 测试客户端……1380
43.2.5 自定义服务宿主……1382
43.2.6 WCF 客户端……1384
43.2.7 诊断……1386
43.2.8 与客户端共享协定程序集……1388
43.3 协定……1389
43.3.1 数据协定……1390
43.3.2 版本问题……1390
43.3.3 服务协定……1391
43.3.4 消息协定……1392
43.3.5 错误协定……1392
43.4 服务的行为……1394
43.5 绑定……1397
43.5.1 标准的绑定……1397
43.5.2 标准绑定的特性……1399
43.5.3 Web 套接字……1400
43.6 宿主……1403
43.6.1 自定义宿主……1403
43.6.2 WAS 宿主……1404
43.6.3 预配置的宿主类……1405
43.7 客户端……1406
43.7.1 使用元数据……1406
43.7.2 共享类型……1407
43.8 双工通信……1408
43.8.1 双工通信的协定……1408
43.8.2 双工通信的服务……1408
43.8.3 双工通信的客户端应用程序……1409
43.9 路由……1410
43.9.1 示例应用程序……1411
43.9.2 路由接口……1412
43.9.3 WCF 路由服务……1412
43.9.4 为失败使用路由器……1413
43.9.5 改变协定的桥梁……1415
43.9.6 过滤器的类型……1415
43.10 小结……1415
第 44 章 ASP.NET Web API……1416
44.1 概述……1416
44.2 创建服务……1417
44.2.1 定义模型……1418
44.2.2 创建控制器……1418
44.2.3 错误处理……1420
44.3 创建.NET 客户程序……1420
44.3.1 发送 GET 请求……1421
44.3.2 发送 POST 请求……1422
44.3.3 发送 PUT 请求……1423
44.3.4 发送 DELETE 请求……1423
44.4 Web API 路由和操作……1424
44.4.1 给操作添加 HTTP 方法……1424
44.4.2 基于特性的路由……1425
44.5 使用 OData……1427
44.5.1 创建数据模型……1427
44.5.2 创建服务……1428
44.5.3 OData 查询……1430
44.5.4 WCF Data Services 客户程序……1431
44.6 保护 Web API……1436
44.6.1 创建账户……1437
44.6.2 创建验证令牌……1439
44.6.3 发送验证过的调用……1440
44.6.4 获取用户信息……1440
44.7 自驻留……1441

44.8 小结 ······ 1443

第 45 章 Windows Workflow Foundation ······ 1444

45.1 工作流概述 ······ 1444
45.2 Hello World 示例 ······ 1445
45.3 活动 ······ 1446
45.3.1 If 活动 ······ 1447
45.3.2 InvokeMethod 活动 ······ 1448
45.3.3 Parallel 活动 ······ 1448
45.3.4 Delay 活动 ······ 1449
45.3.5 Pick 活动 ······ 1449
45.4 自定义活动 ······ 1450
45.4.1 活动的验证 ······ 1451
45.4.2 设计器 ······ 1452
45.4.3 自定义复合活动 ······ 1454
45.5 工作流 ······ 1456
45.5.1 实参和变量 ······ 1457
45.5.2 WorkflowApplication ······ 1458
45.5.3 存放 WCF 工作流 ······ 1461
45.5.4 工作流的版本 ······ 1465
45.5.5 驻留设计器 ······ 1466
45.6 小结 ······ 1471

第 46 章 对等网络 ······ 1472

46.1 P2P 网络概述 ······ 1472
46.1.1 客户端-服务器体系结构 ······ 1473
46.1.2 P2P 体系结构 ······ 1473
46.1.3 P2P 体系结构的挑战 ······ 1474
46.1.4 P2P 术语 ······ 1475
46.1.5 P2P 解决方案 ······ 1475
46.2 PNRP ······ 1475
46.3 构建 P2P 应用程序 ······ 1478
46.4 小结 ······ 1484

第 47 章 消息队列 ······ 1485

47.1 概述 ······ 1486
47.1.1 使用消息队列的场合 ······ 1486
47.1.2 消息队列功能 ······ 1487
47.2 Message Queuing 产品 ······ 1488
47.3 消息队列体系结构 ······ 1489
47.3.1 消息 ······ 1489
47.3.2 消息队列 ······ 1489
47.4 Message Queuing 管理工具 ······ 1490
47.4.1 创建消息队列 ······ 1490
47.4.2 消息队列属性 ······ 1491
47.5 消息队列的编程实现 ······ 1492
47.5.1 创建消息队列 ······ 1492
47.5.2 查找队列 ······ 1493
47.5.3 打开已知队列 ······ 1493
47.5.4 发送消息 ······ 1495
47.5.5 接收消息 ······ 1497
47.6 课程订单应用程序 ······ 1499
47.6.1 课程订单类库 ······ 1499
47.6.2 课程订单消息发送程序 ······ 1502
47.6.3 发送优先级和可恢复的消息 ······ 1504
47.6.4 课程订单消息接收应用程序 ······ 1505
47.7 接收结果 ······ 1511
47.7.1 确认队列 ······ 1511
47.7.2 响应队列 ······ 1512
47.8 事务队列 ······ 1512
47.9 消息队列和 WCF ······ 1514
47.9.1 带数据协定的实体类 ······ 1514
47.9.2 WCF 服务协定 ······ 1515
47.9.3 WCF 消息接收应用程序 ······ 1516
47.9.4 WCF 消息发送应用程序 ······ 1519
47.10 消息队列的安装 ······ 1520
47.11 小结 ······ 1520

第 I 部分

C# 语 言

- 第 1 章 .NET 体系结构
- 第 2 章 核心 C#
- 第 3 章 对象和类型
- 第 4 章 继承
- 第 5 章 泛型
- 第 6 章 数组
- 第 7 章 运算符和类型强制转换
- 第 8 章 委托、lambda 表达式和事件
- 第 9 章 字符串和正则表达式
- 第 10 章 集合
- 第 11 章 LINQ
- 第 12 章 动态语言扩展
- 第 13 章 异步编程
- 第 14 章 内存管理和指针
- 第 15 章 反射
- 第 16 章 错误和异常

第 1 章

.NET 体系结构

本章要点

- 编译和运行面向.NET 的代码
- Microsoft 中间语言(Microsoft Intermediate Language，MSIL)的优点
- 值类型和引用类型
- 数据类型化
- 理解错误处理和特性
- 程序集、.NET 基类和名称空间

1.1 C#与.NET 的关系

整本书都将强调，C#语言不能孤立地使用，而必须和.NET Framework 一起考虑。C#编译器专门用于.NET，这表示用 C#编写的所有代码总是使用.NET Framework 运行。对于 C#语言来说，可以得出两个重要的结论：

(1) C#的体系结构和方法论反映了.NET 基础方法论。

(2) 在许多情况下，C#的特定语言功能取决于.NET 的功能，或依赖于.NET 基类。

由于这种依赖性，在开始编写 C#程序前，了解.NET 的体系结构和方法论就非常重要，这就是本章的目标。

C#是一种相当新的编程语言，C#的重要性体现在以下两个方面：

- 它是专门为与 Microsoft 的.NET Framework 一起使用而设计的(.NET Framework 是一个功能非常丰富的平台，可开发、部署和执行分布式应用程序)。
- 它是一种基于现代面向对象设计方法的语言，在设计它时，Microsoft 还吸取了其他所有类似语言的经验，这些语言是近 20 年来面向对象规则得到广泛应用后才开发出来的。

C#就其本身而言只是一种语言，尽管它是用于生成面向.NET 环境的代码，但它本身不是.NET 的一部分。.NET 支持的一些特性，C#并不支持。而 C#语言支持的另一些特性，.NET 却不支持(如运算符重载)！

但是，因为 C#语言和.NET 一起使用，所以如果要使用 C#高效地开发应用程序，理解 Framework 就非常重要，所以本章将介绍.NET 的内涵。

1.2 公共语言运行库

.NET Framework 的核心是其运行库执行环境，称为公共语言运行库(CLR)或.NET 运行库。通常将在 CLR 控制下运行的代码称为托管代码(managed code)。

但是，在 CLR 执行编写好的源代码(使用 C#或其他语言编写的代码)之前，需要编译它们。在.NET 中，编译分为两个阶段：

(1) 将源代码编译为 Microsoft 中间语言(IL)。

(2) CLR 把 IL 编译为平台专用的代码。

这个两阶段的编译过程非常重要，因为 Microsoft 中间语言是提供.NET 的许多优点的关键。

Microsoft 中间语言与 Java 字节码共享一种理念：它们都是低级语言，语法很简单(使用数字代码，而不是文本代码)，可以非常快速地转换为本地机器码。对于代码，这种精心设计的通用语法有很重要的优点：平台无关性、提高性能和语言的互操作性。

1.2.1 平台无关性

首先，这意味着包含字节码指令的同一文件可以放在任一平台中，运行时，编译过程的最后阶段可以很轻松地完成，这样代码就可以运行在特定的平台上。换言之，编译为中间语言就可以获得.NET 平台无关性，这与编译为 Java 字节码就会得到 Java 平台无关性是一样的。

注意.NET 的平台无关性目前只是停留在理论范畴，因为在编写本书时，.NET 的完整实现只能用于 Windows 平台。不过，现在已经有了.NET 的一个部分跨平台实现(参见 Mono 项目，它用于实现.NET 的开放源代码，参见 http://www.go-mono.com/)。通过 Xamarin(www.xamarin.com)中的工具和库也可以在 iPhone 和 Android 设备上使用 C#。

1.2.2 提高性能

前面对 IL 和 Java 做了比较，实际上，IL 比 Java 字节码的作用还要大。IL 总是即时编译的(称为 JIT 编译)，而 Java 字节码常常是解释性的。Java 的一个缺点是，在运行应用程序时，把 Java 字节码转换为内部可执行代码的过程会导致性能的损失(但在最近，Java 在某些平台上能进行 JIT 编译)。

JIT 编译器并不是把整个应用程序一次编译完(这样会有很长的启动时间)，而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后，得到的本地可执行程序就存储起来，直到退出该应用程序为止，这样在下次运行这部分代码时，就不需要重新编译了。Microsoft 认为这个过程要比一开始就编译整个应用程序代码的效率高得多，因为任何应用程序的大部分代码实际上并不是在每次运行期间都执行。使用 JIT 编译器，从来都不会编译这种代码。

这解释了为什么托管 IL 代码几乎和本地机器代码的执行速度一样快，但是并没有说明为什么 Microsoft 认为这会提高性能。其原因是编译过程的最后一部分是在运行时进行的，JIT 编译器确切地知道程序运行在什么类型的处理器上，可以利用该处理器提供的任何特性或特定的机器代码指令来优化最后的可执行代码。

传统的编译器会优化代码，但它们的优化过程是独立于运行代码的特定处理器的。这是因为传

统的编译器是在发布软件之前编译为本地机器可执行的代码。即编译器不知道运行代码的处理器的类型，例如该处理器是兼容 x86 的处理器还是 Alpha 处理器，这超出了基本操作的范围。

1.2.3 语言的互操作性

使用 IL 不仅支持平台无关性，还支持语言的互操作性。简而言之，就是能将任何一种语言编译为中间语言，编译为中间语言的代码可以与从其他语言编译过来的代码进行交互操作。

那么除了 C#之外，还有什么语言可以通过.NET 进行交互操作呢？下面就简要讨论其他常见语言如何与.NET 交互操作。

1. Visual Basic 2013

Visual Basic 6 在升级到 Visual Basic .NET 2002 时，经历了一番脱胎换骨的变化，才集成到.NET Framework 的第 1 版中。Visual Basic 语言对 Visual Basic 6 进行了很大的演化，也就是说，Visual Basic 6 并不适合运行.NET 程序。例如，它与 COM(Component Object Model，组件对象模型)的高度集成，并且只把事件处理程序作为源代码显示给开发人员，大多数代码隐藏不能用作源代码。另外，它不支持继承的实现，Visual Basic 6 使用的标准数据类型也与.NET 不兼容。

Visual Basic 6 在 2002 年升级为 Visual Basic .NET，对 Visual Basic 进行的改变非常大，完全可以把 Visual Basic .NET 当成一种新语言。

从那之后，Visual Basic 经历了许多语言改进，与 C#一样多。Visual Basic 和 C#在功能上非常类似，因为它们都由 Microsoft 中同一个产品团队开发。随着时间的推移，Visual Basic 具备了曾经只能用于 C#的功能，C#也具备了曾经只能用于 Visual Basic 的功能。目前 Visual Basic 和 C#之间的区别很小，主要是使用花括号还是 END 语句的个人喜好问题。

2. Visual C++ 2013

Visual C++ 6 有许多 Microsoft 对 Windows 的特定扩展。Visual C++ .NET 又新增了更多的扩展内容来支持.NET Framework。现有的 C++源代码会继续编译为本地可执行代码，而不会有修改，但它会独立于.NET 运行库运行。如果让 C++代码在.NET Framework 中运行，就可以在代码的开头添加下述命令：

```
#using <mscorlib.dll>
```

还可以把标记/clr 传递给编译器，这样编译器假定要编译托管代码，因此会生成中间语言，而不是本地机器码。C++的一个有趣的问题是在编译成托管代码时，编译器可以生成包含内嵌本地可执行程序的 IL。这表示在 C++代码中可以把托管类型和非托管类型合并起来，因此托管 C++代码：

```
classMyClass
{
```

定义了一个普通的 C++类，而代码：

```
ref class MyClass
{
```

生成了一个托管类，就好像使用 C#或 Visual Basic 2013 编写类一样。实际上，托管 C++代码比

C#代码更优越的一点是可以在托管 C++代码中调用非托管 C++类，而不必采用 COM 互操作功能。

如果在托管类型上试图使用.NET 不支持的特性(例如，模板或类的多继承)，编译器就会出现一个错误。另外，在使用托管类时，还需要使用非标准 C++功能。

编写使用.NET 的 C++程序会得到几种不同的互操作场景。使用编译器设置/clr 启用公共语言运行库支持时，就可以完全混合所有的本地和托管 C++功能。其他选项(如/clr:safe 和/clr:pure)可以限制 C++指针的使用，从而像使用 C#和 Visual Basic 那样编写安全的代码。

Visual C++ 2013 允许为 Windows 8.1 的 Windows Runtime(WinRT)创建程序。在这样的程序中，C++不使用托管代码，而是本地访问 WinRT。

3. Visual F#

F#是一个强类型化的函数编程语言，这种语言在 Visual Studio 中得到了强力支持。

作为一种函数编程语言，F#表面上与 C#完全不同。例如，声明一个带成员 FirstName 和 LastName 的 Person 类型，如下所示：

```
module PersonSample

type Person(firstName : string, lastName : string) =
   member this.FirstName = firstName
   member this.LastName = lastName
```

使用 Person 类型，就要利用 let 关键字。printfn 会把结果写入控制台：

```
open PersonSample

[<EntryPoint>]
let main argv =
   let p = Person("Sebastian", "Vettel")
   let first = p.firstName
   let last = p.lastName
   printfn "%s %s" first last
   0 // return an integer exit code
```

F#可以使用 C#创建的所有类型，反之亦然。F#的优点是，这是一种函数语言，而不是面向对象的，这有助于编写复杂的算法，例如财务和科学应用程序。

4. COM 和 COM+

从技术上讲，COM 和 COM+并不是面向.NET 的技术，因为基于它们的组件不能编译为 IL(但如果原来的 COM 组件是用 C++编写的，那么使用托管 C++在某种程度上可以这么做)。但是，COM+仍然是一个重要工具，因为它包含一些.NET 不具备的特性。另外，COM 组件仍可以使用——.NET 集成了 COM 的互操作性，从而使托管代码可以调用 COM 组件，COM 组件也可以调用托管代码(见第 23 章)。一般情况下，把新组件编写为.NET 组件，大多是为了方便，因为这样可以利用.NET 基类和托管代码的其他优点。

5. Windows 运行库

Windows 8 提供了一种 Windows Store 应用可以使用的新运行库，这个运行库的某些部分也可以

由桌面应用程序使用。这个运行库可在 Visual Basic、C#、C++和 JavaScript 中使用。用在不同的环境中时，它会发生相应的变化。例如，在 C#中使用时，它看起来就像.NET Framework 中的类；在 JavaScript 中使用时，它看起来就像 JavaScript 开发人员所惯用的 JavaScript 库；而在 C++中使用时，它又像是一个 C++标准库。这种多样性是通过使用语言投影实现的。第 31 章将讨论 Windows 运行库以及在 C#中如何使用它。

1.3 中间语言

如前所述，Microsoft 中间语言显然在.NET Framework 中起着非常重要的作用。现在应详细讨论一下 IL 的主要特征，因为面向.NET 的所有语言在逻辑上都需要支持 IL 的主要特征。

下面就是中间语言的主要特征：

- 面向对象和使用接口
- 值类型和引用类型之间的显著差异
- 强数据类型化
- 使用异常来处理错误
- 使用特性(attribute)

下面详细讨论这些特征。

1.3.1 面向对象和接口的支持

.NET 的语言无关性还有一些实际的限制。中间语言在设计时就打算实现某些特殊的编程方法，这表示面向它的语言必须与该编程方法兼容，Microsoft 为 IL 选择的特定道路是传统的面向对象的编程，带有类的单一继承性。

除了传统的面向对象编程外，中间语言还引入了接口的概念，在带有 COM 的 Windows 下第一次实现了接口。用.NET 建立的接口与 COM 接口不同，它们不需要支持任何 COM 基础设施，例如，它们不是派生自 IUnknown，也没有对应的 GUID。但它们与 COM 接口共享下述理念：提供一个契约，实现给定接口的类必须提供该接口指定的方法和属性的实现。

前面介绍了使用.NET 意味着要编译为中间语言，即需要使用传统的面向对象的方法来编程。但这并不能提供语言的互操作性。毕竟，C++和 Java 都使用相同的面向对象的范例，但它们仍不是可交互操作的语言。下面需要详细探讨一下语言互操作性的概念。

首先，需要了解一下语言互操作性的准确含义。

毕竟，COM 允许以不同语言编写的组件一起工作，即可以调用彼此的方法。这就足够了吗？COM 是一个二进制标准，允许组件实例化其他组件，调用它们的方法或属性，而无须考虑编写相关组件的语言。但为了实现这个功能，每个对象都必须通过 COM 运行库来实例化，通过接口来访问。根据相关组件的线程模型，需要在不同线程的内存空间和运行组件之间编组数据，这可能造成很大的性能损失。在极端情况下，组件保存为可执行文件，而不是 DLL 文件，还必须创建单独的进程来运行它们。重要的是组件仅能通过 COM 运行库与其他组件通信。使用不同语言的组件无法通过 COM 直接彼此通信，或者创建彼此的实例——系统总将 COM 作为中间件。不仅如此，COM 体系结构还不允许利用继承实现，即它丧失了面向对象编程的许多优势。

一个相关的问题是，在调试时，仍必须单独调试使用不同语言编写的组件。不可能在调试器上交替调试不同语言的代码。语言互操作性的真正含义是用一种语言编写的类应能直接与用另一种语言编写的类通信。特别是：

- 用一种语言编写的类应能继承用另一种语言编写的类。
- 一个类应能包含另一个类的实例，而不管两个类是使用什么语言编写的。
- 一个对象应能直接调用用其他语言编写的另一个对象的方法。
- 对象(或对象的引用)应能在方法之间传递。
- 在不同的语言之间调用方法时，应能在调试器中交替调试这些方法调用，即调试不同语言编写的源代码。

这是一个雄心勃勃的目标，但令人惊讶的是，.NET 和中间语言已经实现了这个目标。在调试器上交替调试方法时，Visual Studio IDE(Intergrated Development Environment，集成开发环境)提供了这样的工具(不是 CLR 提供的)。

1.3.2 不同的值类型和引用类型

与其他编程语言一样，中间语言提供了许多预定义的基本数据类型。它的一个特性是值类型和引用类型之间有明显的区别。对于值类型(value type)，变量直接存储其数据，而对于引用类型(reference type)，变量仅存储地址，对应的数据可以在该地址中找到。

在 C++中，使用引用类型类似于通过指针来访问变量，而在 Visual Basic 中，与引用类型最相似的是对象，Visual Basic 6 总是通过引用来访问对象。中间语言也有数据存储的规范：引用类型的实例总是存储在一个名为“托管堆”的内存区域中，值类型一般存储在堆栈中(但如果值类型在引用类型中声明为字段，它们就内联存储在堆中)。第 2 章讨论堆栈和托管堆，及其工作原理。

1.3.3 强数据类型化

中间语言的一个重要方面是它基于强数据类型化。所有的变量都清晰地标记为属于某个特定数据类型(在中间语言中没有 Visual Basic 和脚本语言中的 Variant 数据类型)。特别是中间语言一般不允许对模糊的数据类型执行任何操作。

例如，Visual Basic 6 开发人员习惯于传递变量，而无须考虑它们的类型，因为 Visual Basic 6 会自动进行所需的类型转换。C++开发人员习惯于在不同类型之间转换指针类型。执行这类操作将极大地提高性能，但破坏了类型的安全性。因此，在某些编译为托管代码的语言中，这类操作只能在特殊情况下进行。确实，指针(相对于引用)只能在标记了的 C#代码块中使用，但在 Visual Basic 中不能使用(但一般在托管 C++中允许使用)。在代码中使用指针会立即导致 CLR 执行的内存类型安全性检查失败。注意，一些与.NET 兼容的语言，例如 Visual Basic 2010，在类型化方面的要求仍比较宽松，但这是可以的，因为编译器在后台确保在生成的 IL 上强制类型安全。

尽管强迫实现类型的安全性似乎会降低性能，但在许多情况下，我们从.NET 提供的、依赖于类型安全的服务中获得的好处更多。这些服务包括：

- 语言的互操作性
- 垃圾收集
- 安全性
- 应用程序域

下面讨论强数据类型化对.NET 的这些功能非常重要的原因。

1. 语言互操作性中强数据类型化的重要性

如果类派生自其他类，或包含其他类的实例，它就需要知道其他类使用的所有数据类型，这就是强数据类型化非常重要的原因。实际上，过去由于缺少用于指定这类信息的一致的系统，从而成为语言继承和交互操作的真正障碍。这类信息并未在标准的可执行文件或 DLL 中出现。

假定将 Visual Basic 2013 类中的一个方法定义为返回一个 Integer——Visual Basic 2013 可以使用的标准数据类型之一。但 C#没有该名称的数据类型。显然，只有编译器知道如何把 Visual Basic 2013 的 Integer 类型映射为 C#定义的某种已知类型，才可以从该类派生，使用这个方法，并在 C#代码中使用返回的类型。这个问题在.NET 中是如何解决的？

通用类型系统(CTS)

此类数据类型问题在.NET 中使用通用类型系统(CTS)得到了解决。CTS 定义了可以在中间语言中使用的预定义数据类型，所有面向.NET Framework 的语言都可以生成最终基于这些类型的编译代码。

对于上面的例子，Visual Basic 2013 的 Integer 实际上是一个 32 位有符号的整数，它实际映射为中间语言类型 Int32。因此在中间语言代码中就指定这种数据类型。C#编译器可以使用这种类型，所以就不会有问题了。在源代码中，C#用关键字 int 来表示 Int32，所以编译器就认为 Visual Basic 2013 方法返回一个 int 类型的值。

CTS 不仅指定了基本数据类型，还定义了一个内容丰富的类型层次结构，其中包含设计合理的位置，在这些位置上，代码允许定义它自己的类型。CTS 的层次结构反映了中间语言的单一继承的面向对象方法，如图 1-1 所示。

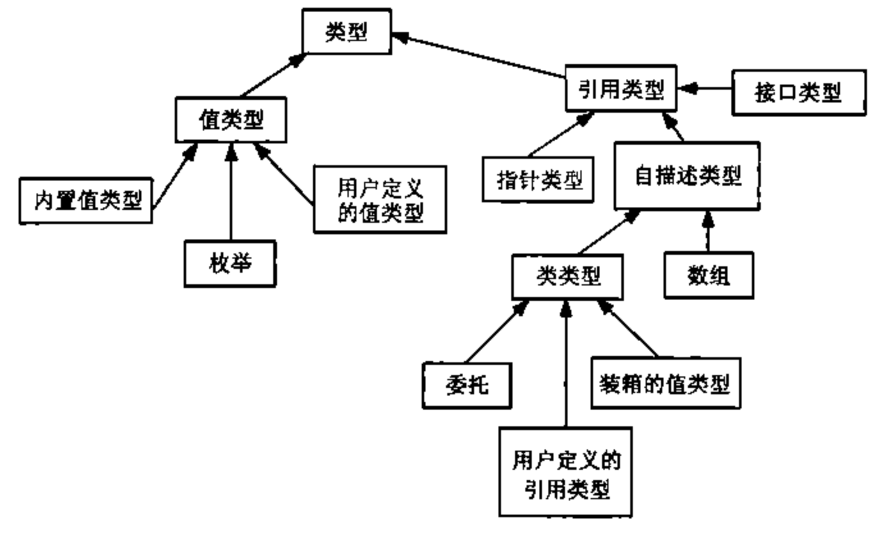


图 1-1

这里没有列出内置的所有值类型，因为第 3 章将详细介绍它们。在 C#中，编译器识别的每个预定义类型都映射为一个 IL 内置类型。这与 Visual Basic 2013 一样。

公共语言规范(CLS)

公共语言规范(Common Language Specification，CLS)和通用类型系统一起确保语言的互操作性。

CLS 是一个最低标准集，所有面向.NET 的编译器都必须支持它。因为 IL 是一种内涵非常丰富的语言，大多数编译器的编写人员有可能把给定编译器的功能限制为只支持 IL 和 CTS 提供的一部分功能。只要编译器支持已在 CLS 中定义的内容，这就很不错。

下面的一个例子是有关区分大小写字母的。IL 是区分大小写的语言。使用这些语言的开发人员常常利用区分大小写所提供的灵活性来选择变量名。但 Visual Basic 2013 是不区分大小写的语言。CLS 通过指定 CLS 兼容代码不使用任何只根据大小写来区分的名称，解决了这个问题。因此，Visual Basic 2013 代码可以与 CLS 兼容代码一起使用。

这个例子说明了 CLS 的两种工作方式。

(1) 各个编译器的功能不必强大到支持.NET 的所有功能，这将鼓励人们为其他面向.NET 的编程语言开发编译器。

(2) 如果限制类只能使用 CLS 兼容的特性，就要保证用其他兼容语言编写的代码可以使用这个类。

这种方法的优点是使用 CLS 兼容特性的限制只适用于公共和受保护的类成员和公共类。在类的私有实现方式中，可以编写非 CLS 代码，因为其他程序集(托管代码的单元，参见本章后面的内容)中的代码不能访问这部分代码。

这里不深入讨论 CLS 规范。一般情况下，CLS 对 C#代码的影响不会太大，因为 C#中的非 CLS 兼容特性非常少。

编写非 CLS 兼容代码是完全可以接受的。只是在编写了这种代码后，就不能保证编译好的 IL 代码完全支持语言的互操作性。

2. 垃圾回收

垃圾回收器(garbage collector)用来在.NET 中进行内存管理，特别是它可以恢复正在运行的应用程序需要的内存。到目前为止，Windows 平台已经使用了两种技术来释放进程向系统动态请求的内存：

- 完全以手工方式使应用程序代码完成这些工作。
- 让对象维护引用计数。

让应用程序代码负责释放内存是低级高性能的语言使用的技术，例如 C++。这种技术很有效，并且一般情况下可以让资源在不需要时就释放，但其最大的缺点是频繁出现错误。请求内存的代码还必须显式通知系统它什么时候不再需要该内存。但这是很容易被遗漏的，从而导致内存泄漏。

尽管现代的开发环境提供了帮助检测内存泄漏的工具，但它们很难跟踪错误，因为直到内存已大量泄漏，从而使 Windows 拒绝为进程提供资源时，它们才会发挥作用。到那个时候，由于对内存的需求很大，会使整个计算机变得相当慢。

维护引用计数是 COM 对象采用的一种技术，其方法是每个 COM 组件都保留一个计数，记录客户端目前对它的引用数。当这个计数下降到 0 时，组件就会删除自己，并释放相关的内存和资源。它带来的问题是仍需要客户端通知组件它们已经完成了内存的使用。只要有一个客户端没有这么做，对象就仍驻留在内存中。在某些方面，这是比 C++内存泄漏更为严重的问题，因为 COM 对象可能存在于它自己的进程中，从来不会被系统删除(在 C++内存泄漏问题上，系统至少可以在进程中断时

释放所有的内存)。

.NET 运行库采用的方法是垃圾回收器，这是一个程序，其目的是清理内存。方法是所有动态请求的内存都分配到堆上(所有的语言都是这样处理的，但在.NET 中，CLR 维护它自己的托管堆，供.NET 应用程序使用)。每隔一段时间，当.NET 检测到给定进程的托管堆已满，需要清理时，就调用垃圾回收器。垃圾回收器处理目前代码中的所有变量，检查对存储在托管堆上的对象的引用，确定哪些对象可以从代码中访问——即哪些对象有引用。没有引用的对象就不再认为可以从代码中访问，因而被删除。Java 就使用与此类似的垃圾回收系统。

之所以在.NET 中使用垃圾回收器，是因为中间语言已用来处理进程。其规则要求，第一，不能引用已有的对象，除非复制已有的引用。第二，中间语言是类型安全的语言。在这里，其含义是如果存在对对象的任何引用，该引用中就有足够的信息来确定对象的类型。

垃圾回收机制不能和诸如非托管 C++的语言一起使用，因为 C++允许指针自由地转换数据类型。

垃圾回收的一个重要方面是它的不确定性。换言之，不能保证什么时候会调用垃圾回收器：CLR 决定需要它时，就可以调用它。但可以重写这个过程，在代码中调用垃圾回收器。这在测试时很有帮助，但是在正常的程序中不应该这么做。

垃圾回收过程的详细信息可参见第 14 章。

3. 安全性

.NET 很好地弥补了 Windows 提供的安全机制，因为它提供的安全机制是代码访问安全性(Code Access Security)，而 Windows 仅提供了基于角色的安全性。

基于角色的安全性建立在运行进程的账户的身份基础上，换言之，就是谁拥有和运行进程。另一方面，代码访问安全性建立在代码实际执行的任务和代码的可信程度上。由于中间语言提供了强大的类型安全性，因此 CLR 可以在运行代码前检查它，以确定是否有需要的安全权限。.NET 还提供了一种机制，使代码可以在运行前，预先指定需要什么安全权限。

基于代码的安全性非常重要，原因是它降低了与运行来历不明的代码有关的风险(如代码是从 Internet 上下载的)。即使代码运行在管理员账户下，也可以使用基于代码的安全性，指定这段代码不能执行管理员账户一般可以执行的某些类型的操作，例如读写环境变量、读写注册表或访问.NET 反射特性。

安全问题详见第 22 章。

4. 应用程序域

应用程序域(application domain)是.NET 中的一个重要技术改进，它用于减少运行应用程序的系统开销，这些应用程序需要与其他程序分离开来，但仍需要彼此通信。典型的例子是 Web 服务器应用程序，它需要同时响应许多浏览器请求。因此，要有许多组件实例同时响应这些同时运行的请求。

在.NET 问世之前，可以让这些实例共享同一个进程，但此时一个运行的实例就有可能导致整个网站的崩溃；也可以把这些实例孤立在不同的进程中，但这样做会增加相关性能的系统开销。到现在为止，孤立代码的唯一方式是通过进程来实现的。在启动一个新的应用程序时，它会在一个进程

环境内运行。Windows 通过地址空间把进程分隔开来。这样，每个进程有 4GB 的虚拟内存来存储其数据和可执行代码(4GB 对应于 32 位系统，64 位系统要用更多的内存)。Windows 利用额外的间接方式把这些虚拟内存映射到物理内存或磁盘空间的一个特殊区域中。每个进程都会有不同的映射，虚拟地址空间块映射的物理内存之间不重叠，如图 1-2 所示。

一般情况下，任何进程都只能通过指定虚拟内存中的一个地址来访问内存——即进程不能直接访问物理内存，因此一个进程不可能访问分配给另一个进程的内存。这样就可以确保任何执行出错的代码不会损害其地址空间以外的数据。

进程不仅是运行代码的实例相互隔离的一种方式，它们还可以构成分配了安全权限和许可的单元。每个进程都有自己的安全标识，明确地表示 Windows 允许该进程可以执行的操作。

进程对确保安全有很大的帮助，而它们的一大缺点是性能。许多进程常常在一起工作，因此需要相互通信。一个常见的例子是进程调用一个 COM 组件，而该 COM 组件是可执行的，因此需要在它自己的进程上运行。在 COM 中使用代理时也会发生类似的情况。因为进程不能共享任何内存，所以必须使用一个复杂的编组过程在进程之间复制数据。这对性能有非常大的影响。如果需要使组件一起工作，但不希望性能有损失，唯一的方法是使用基于 DLL 的组件，让所有的组件在同一个地址空间中运行——其风险是执行出错的组件会影响其他组件。

应用程序域是分离组件的一种方式，它不会导致因在进程之间传送数据而产生的性能问题。其方法是把任何一个进程分解到多个应用程序域中。每个应用程序域大致对应一个应用程序，执行的每个线程都运行在一个具体的应用程序域中，如图 1-3 所示。

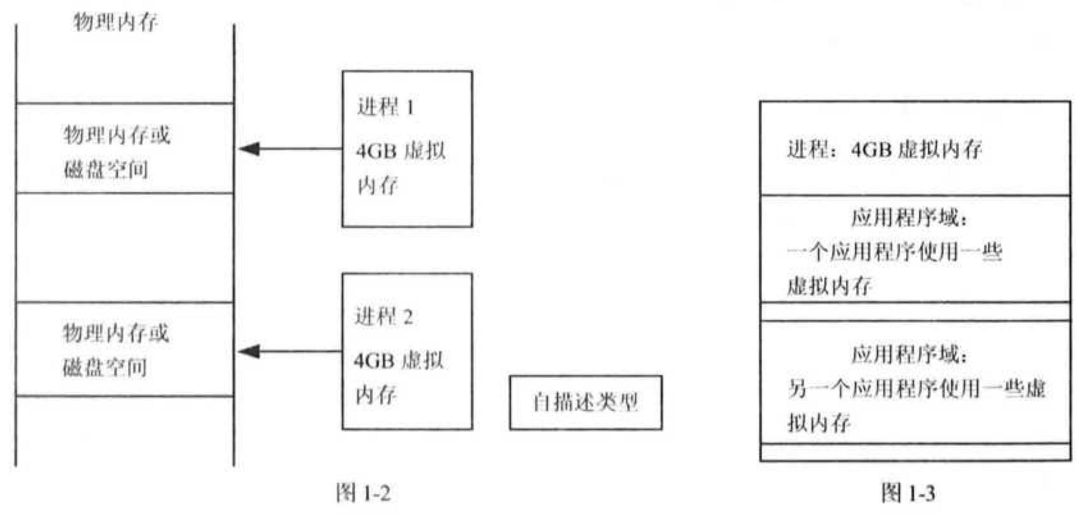


图 1-2　　图 1-3

如果不同的可执行文件都运行在同一个进程空间中，显然它们就能轻松地共享数据，因为理论上它们可以直接访问彼此的数据。虽然在理论上这是可以实现的，但是 CLR 会检查每个正在运行的应用程序的代码，以确保这些代码不脱离它自己的数据区域，保证不发生直接访问其他进程的数据的情况。这初看起来是不可能的，不真正运行程序，如何告诉程序要做什么工作？

实际上，这么做通常是可能的，因为中间语言拥有强大的类型安全功能。在大多数情况下，除非代码明确使用不安全的特性，例如指针，否则它使用的数据类型可以确保内存不会被错误地访问。例如，.NET 数组类型执行边界检查，以确保禁止执行超出边界的数组操作。如果运行的应用程序的确需要与运行在不同应用程序域中的其他应用程序通信或共享数据，就必须调用.NET 的远程服务。

被验证不能访问超出其应用程序域的数据(除非通过明确的远程处理机制)的代码就是内存类型安

全的代码。这种代码与运行在同一个进程中但应用程序域不同的类型安全代码一起运行是安全的。

1.3.4 通过异常处理错误

.NET Framework 可以与 Java 和 C++使用相同的基于异常的机制处理错误情况。C++开发人员应注意到，由于 IL 有更严格的强类型系统，因此在 IL 中不像 C++那样存在因使用异常带来的相关性能问题。另外，.NET 和 C#支持 finally 块，这是许多 C++开发人员长久以来一直希望 C++也能够提供的一种功能。

第 16 章会详细讨论异常。简要地说，代码的某些部分被看成异常处理例程，每个例程都能处理某种特殊的错误情况(例如，找不到文件，或拒绝执行某些操作)。这些条件可以定义得很宽或很窄。异常结构确保在发生错误情况时，执行进程立即跳到最有针对性的异常处理例程上，来处理错误情况。

异常处理的结构还提供了一种简便的方式，可以将包含异常情况的准确信息的对象传递给错误处理例程。这个对象包括给用户提供的相应消息和在代码的什么地方检测到错误的确切消息。

大多数异常处理结构，包括异常发生时的程序流控制，都是由高级语言处理的，例如 C#、Visual Basic 2013 和 C++，任何中间语言中的命令都不支持它。例如，C#使用 try{}、catch{}和 finally{}代码块来处理它，详见第 16 章。

但.NET 提供了一种基础设施，让面向.NET 的编译器支持异常处理。特别是它提供了一组.NET 类来表示异常，语言的互操作性则允许异常处理代码解释被抛出的异常对象，无论异常处理代码使用什么语言编写，都是这样。语言的无关性没有体现在 C++和 Java 的异常处理中，但在 COM 的错误处理机制中有一定限度的体现。COM 的错误处理机制包括从方法中返回错误代码以及传递错误对象。在不同的语言中，异常的处理是一致的，这是促进多语言开发的重要一环。

1.3.5 特性的使用

特性(attribute)是使用 C++编写 COM 组件的开发人员很熟悉的一个功能(在 Microsoft 的 COM 接口定义语言(Interface Definition Language，IDL)中使用特性)。特性最初是为了在程序中提供与某些项相关的额外信息，以供编译器使用。

.NET 支持特性，因此现在 C++、C#和 Visual Basic 2013 也支持特性。但在.NET 中，对特性的革新是可以在源代码中定义自己的自定义特性。这些用户定义的特性将和对应数据类型或方法的元数据放在一起，这对于文档记录十分有用，它们和反射技术一起使用，以根据特性执行编程任务。另外，与.NET 的语言无关性的基本原理一样，特性也可以在一种语言的源代码中定义，而被用另一种语言编写的代码读取。

第 15 章将详细介绍特性。

1.4 程序集

程序集(assembly)是包含编译好的、面向.NET Framework 的代码的逻辑单元。本章不详细论述程序集，而在第 19 章中论述，下面概述其中的要点。

程序集是完全自描述性的，它是一个逻辑单元而不是物理单元，可以存储在多个文件中(动态程序集存储在内存中，而不是存储在文件中)。如果一个程序集存储在多个文件中，其中就会有一个包含入口点的主文件，该文件描述了程序集中的其他文件。

可执行代码和库代码使用相同的程序集结构。唯一的区别是可执行的程序集包含一个主程序入口点，而库程序集不包含。

程序集的一个重要特征是它们包含的元数据描述了对应代码中定义的类型和方法。程序集也包含描述程序集本身的元数据，这种程序集元数据包含在一个称为“清单(manifest)”的区域中，可以检查程序集的版本及其完整性。

ildasm 是一个基于 Windows 的实用程序，可以用于检查程序集的内容，包括程序集清单和元数据。第 19 章将介绍 ildasm。

程序集包含程序的元数据，表示调用给定程序集中的代码的应用程序或其他程序集不需要引用注册表或其他数据源，就能确定如何使用该程序集。这与以前的 COM 有很大的区别，以前，组件和接口的 GUID 必须从注册表中获取，在某些情况下，方法和属性的详细信息也需要从类型库中读取。

把数据分散在 3 个以上的不同位置上，可能会出现信息不同步的情况，从而妨碍其他软件成功地使用该组件。有了程序集后，就不会发生这种情况，因为所有的元数据都与程序的可执行指令存储在一起。即使程序集存储在几个文件中，数据也不会出现不同步的问题。这是因为包含程序集入口的文件也存储了其他文件的细节、散列和内容，如果一个文件被替换，或者被篡改，系统肯定会检测出来，并拒绝加载程序集。

程序集有两种类型：私有程序集和共享程序集。

1.4.1 私有程序集

私有程序集是最简单的一种程序集类型。私有程序集一般附带在某个软件上，且只能用于该软件。附带私有程序集的常见情况是，以可执行文件或许多库的方式提供应用程序，这些库包含的代码只能用于该应用程序。

系统可以保证私有程序集不被其他软件使用，因为应用程序只能加载位于主执行文件所在文件夹或其子文件夹中的私有程序集。

用户一般会希望把商用软件安装在它自己的目录下，这样软件包不存在覆盖、修改或在无意间加载另一个软件包的私有程序集的风险。私有程序集只能用于自己的软件包，这样，用户对什么软件使用它们就有了更大的控制权。因此，不需要采取安全措施，因为这没有其他商用软件用某个新版本的程序集覆盖原来的私有程序集的风险(但软件是专门执行怀有恶意的损害性操作的情况除外)。名称也不会有冲突。如果私有程序集中的类正巧与另一个人的私有程序集中的类同名，是不会有问题的，因为给定的应用程序只能使用它自己的一组私有程序集。

因为私有程序集是完全自包含的，所以部署它的过程就很简单。只需要把相应的文件放在文件系统的对应文件夹中即可(不需要注册表项)，这个过程称为“0 影响(xcopy)安装”。

1.4.2 共享程序集

共享程序集是其他应用程序可以使用的公共库。因为其他软件可以访问共享程序集，所以需要

采取一定的保护措施来防止以下风险：

- 名称冲突，另一个公司的共享程序集执行的类型与自己的共享程序集中的类型同名。因为客户端代码理论上可以同时访问这些程序集，所以这是一个严重的问题。
- 程序集被同一个程序集的不同版本覆盖——新版本与某些已有的客户端代码不兼容。

这些问题的解决方法是把共享程序集放在文件系统的一个特定的子目录树中，称为全局程序集缓存(GAC)。与私有程序集不同，不能简单地把共享程序集复制到对应的文件夹中，而需要专门安装到缓存中，可以用许多.NET 工具完成这个过程，其中包含对程序集的检查、在程序集缓存中设置一个小的文件夹层次结构，以确保程序集的完整性。

为了避免名称冲突，应根据私钥加密法为共享程序集指定一个名称(而对于私有程序集，只需要指定与其主文件名相同的名称即可)。该名称称为强名(strong name)，并保证其唯一性，它必须由要引用共享程序集的应用程序来引用。

与覆盖程序集的风险相关的问题，可以通过在程序集清单中指定版本信息来解决，也可以通过同时安装来解决。

1.4.3 反射

因为程序集存储了元数据，包括在程序集中定义的所有类型和这些类型的成员的细节，所以可以编程访问这些元数据。这个技术称为反射，第 15 章详细介绍了它们。该技术很有趣，因为它表示托管代码实际上可以检查其他托管代码，甚至检查它自己，以确定该代码的信息。它们常常用于获取特性的详细信息，也可以把反射用于其他目的，例如作为实例化类或调用方法的一种间接方式(前提是将这些类或方法的名称指定为字符串)。这样，就可以选择类来实例化方法，以便在运行时调用，而不是在编译时调用，例如根据用户的输入来调用(动态绑定)。

1.4.4 并行编程

.NET Framework 允许利用目前出现的所有多核处理器。并行计算能力提供了分隔工作活动、并在多个处理器上运行这些活动的方式。现在可用的、新的并行编程 API 使得编写安全的多线程代码变得十分简单，但要注意，仍需要考虑竞态条件和死锁。

新的并行编程功能提供了一个新的 Task Parallel Library 和 PLINQ Execution Engine，并行编程的详细内容请参见第 21 章。

1.4.5 异步编程

C# 5.0 以 Task Parallel Library 中的 Task 为基础，提供了新的异步功能。自从.NET 1.0 以来，.NET Framework 中的许多类都同时提供了同步和异步版本。当用户界面线程在执行需要花费一些时间的任务时，是不应该被阻塞的。如果看到过不响应的程序，就会知道那是很烦人的。但是，异步方法的缺点是很难使用。对应的同步版本在编写程序时很方便，所以更加常用。

使用了很多年鼠标的用户会习惯延迟。移动对象或者使用滚动条时，延迟是很常见的。但是，对于触摸界面，延迟会造成很糟糕的用户体验。这可以通过调用异步方法来解决。如果 WinRT 中的某个方法需要超过 50ms 才能完成，那么 WinRT 会只提供异步方法调用。

在 C# 5.0 中调用新的异步方法是很简单的。C# 5.0 定义了两个新的关键字：async 和 await。第 13 章将介绍这两个关键字和它们的用法。

1.5 .NET Framework 类

至少从开发人员的角度来看，编写托管代码的最大好处是可以使用.NET 基类库。.NET 基类是一个内容丰富的托管代码类集合，它可以完成以前要通过 Windows API 来完成的绝大多数任务。这些类沿用中间语言使用的对象模型，也基于单一继承性。可以从任何适用的.NET 基类实例化对象，也可以从它们派生自己的类。

.NET 基类的一个主要优点是它们非常直观和易用。例如，要启动一个线程，可以调用 Thread 类的 Start()方法。要禁用 TextBox，应把 TextBox 对象的 Enabled 属性设置为 false。Visual Basic 和 Java 开发人员非常熟悉这种方式，它们的库也都很容易使用，但对于 C++开发人员这是极大的解脱，因为他们多年来一直在使用诸如 GetDIBits()、RegisterWndClassEx()和 IsEqualIID()这样的 API 函数，以及大量需要传递 Windows 句柄的函数。

另一方面，C++开发人员总是很容易访问整个 Windows API，而 Visual Basic 6 和 Java 开发人员只能访问其语言所能访问的基本操作系统功能。.NET 基类的新增内容就是把 Visual Basic 和 Java 库的易用性和 Windows API 函数较为丰富的功能结合起来。但 Windows 仍有许多功能不能通过基类来使用，而需要调用 API 函数。但一般情况下，这只限于比较复杂的特性。基类库足以应付日常工作的使用。如果需要调用 API 函数，.NET 提供了所谓的"平台调用"，来确保对数据类型进行正确的转换，这样无论是使用 C#、C++或 Visual Basic 2013 进行编码，该任务都不会比直接从已有的 C++代码中调用函数更困难。

第 3 章主要介绍基类。概述了 C#语言语法后，本书的其余内容将主要说明如何使用.NET Framework 4.5 的.NET 基类库中的各种类，即各种基类是如何工作的。.NET 4.5 基类大致包括以下范围：

- IL 提供的核心功能(例如，通用类型系统中的基本数据类型，详见第 2 章)
- Windows UI 支持和控件(参见第 35 章~第 39 章)
- 在 ASP.NET 中使用 Web 窗体和 MVC(参见第 30 章~第 42 章)
- 使用 ADO.NET 和 XML 进行数据访问(参见第 32 章~第 34 章)
- 文件系统和注册表访问(参见第 24 章)
- 网络和 Web 浏览(参见第 26 章)
- .NET 特性和反射(参见第 15 章)
- COM 互操作性(参见第 23 章)

附带说一下，根据 Microsoft 源文件，大部分.NET 基类实际上都是用 C#编写的！

1.6 名称空间

名称空间是.NET 避免类名冲突的一种方式。例如，名称空间可以避免下述情况：定义一个类来表示一个顾客，称此类为 Customer，同时其他人也在做相同的事(很可能出现这种情况，拥有客户的企业所占的比例很高)。

名称空间不过是数据类型的一种组合方式，但名称空间中所有数据类型的名称都会自动加上该名称空间的名字作为其前缀。名称空间还可以相互嵌套。例如，大多数用于一般目的的.NET 基类位

于名称空间 System 中，基类 Array 在这个名称空间中，所以其全名是 System.Array。

.NET 需要在名称空间中定义所有的类型，例如，可以把 Customer 类放在名称空间 YourCompanyName.ProjectName 中，则这个类的全名就是 YourCompanyName.ProjectName.Customer。

如果没有显式提供名称空间，类型就添加到一个没有名称的全局名称空间中。

在大多数情况下，Microsoft 建议都至少要提供两个嵌套的名称空间名，第一个是公司名，第二个是技术名称或软件包的名称，而类是其中的一个成员，例如 YourCompanyName.SalesServices.Customer。大多数情况下，这么做可以保证类名不会与其他组织编写的类名冲突。

第 2 章将详细介绍名称空间。

1.7 用 C#创建.NET 应用程序

C#可以用于创建控制台应用程序：仅使用文本、运行在 DOS 窗口中的应用程序。在对类库进行单元测试、创建 UNIX/Linux 守护进程时，就要使用控制台应用程序。但是，我们常使用 C#创建利用许多与.NET 相关的技术的应用程序，下面简要论述可以用 C#创建的不同类型的应用程序。

1.7.1 创建 ASP.NET 应用程序

最初引入的 ASP.NET 1.0 基本改变了 Web 编程模型。ASP.NET 4.5 是该产品的一个主要版本，它建立在以前改进的基础之上。ASP.NET 4.5 采取了一系列重要的革新步骤来提高效率。ASP.NET 的主要目标是使用最少的代码建立强大、安全、动态的应用程序。由于本书是关于 C#的，所以有许多章节介绍了如何使用这种语言建立最新的 Web 应用程序。

下面讨论 ASP.NET 的重要功能，详细信息参见第 40～第 42 章。

1. ASP.NET 的功能

ASP.NET 最初被设计出来时，只提供了 ASP.NET Web 窗体，其目标是按照 Windows 应用程序开发人员编写应用程序的方式轻松地创建 Web 应用程序，是不必编写 HTML 和 JavaScript 的。

现在情况发生了变化。HTML 和 JavaScript 重新焕发了生机，再次变得重要起来。相应地，ASP.NET 中有了一个新的框架，不只方便了编写 HTML 和 JavaScript，还基于流行的 MVC 模式提供了代码的分离，从而更便于进行单元测试。这个框架就是 ASP.NET MVC。

重构后的 ASP.NET 为 ASP.NET Web 窗体和 ASP.NET MVC 打下了基础，而且 UI 框架也构建在这个基础之上。

第 40 章将介绍 ASP.NET 打下的基础。

2. ASP.NET Web 窗体

为了简化 Web 页面的结构，Visual Studio 2013 提供了 Web 窗体。它们允许以图形化方式建立 ASP.NET 页面；换言之，就是把控件从工具箱拖放到窗体上，再考虑窗体的代码，为控件编写事件处理程序。在使用 C#创建 Web 窗体时，就是创建一个继承自 Page 基类的 C#类，并把这个类看成代码隐藏的 ASP.NET 页面。当然不是必须使用 C#创建 Web 窗体，也可以使用 Visual Basic 2013 或另一种.NET 兼容语言来创建。

ASP.NET Web 窗体提供了丰富的功能，使用它的控件不只可以创建简单的代码，还能够利用 JavaScript 和服务器端验证逻辑进行输入验证，以及使用网格和数据源来访问数据库。ASP.NET 的 Web 窗体还提供了 Ajax 功能，允许在客户端动态渲染页面的某个部分。

第 41 章将详细讨论 ASP.NET Web 窗体。

3. Web 服务器控件

用于添加到 Web 窗体上的控件与 ActiveX 控件并不是同一种控件，它们是 ASP.NET 名称空间中的 XML 标记。当请求一个页面时，Web 浏览器会动态地把它们转换为 HTML 和客户端脚本。Web 服务器能以不同的方式显示相同的服务器端控件，产生一个对应于请求者特定 Web 浏览器的转换。这意味着现在很容易为 Web 页面编写相当复杂的用户界面，而不必担心如何确保页面运行在可用的任何浏览器上，因为 Web 窗体会完成这些任务。

可以使用 C#或 Visual Basic 2013 扩展 Web Form 工具箱。创建一个新服务器端控件只需要实现.NET 的 System.Web.UI.WebControls.WebControl 类而已。

4. ASP.NET MVC

Visual Studio 自带了 ASP.NET MVC 4。这种技术已经发展到第 4 个版本了。Web 窗体采取的做法是为开发人员抽象掉了 HTML 和 JavaScript，但是随着 HTML5 和 jQuery 的出现，使用这些技术再次变得重要起来。ASP.NET MVC 将重点放到了在模型和控制器中单独编写服务器端代码，而在使用视图时只用少量服务器端代码从控制器中获取信息。这种功能分离使得编写单元测试变得简单，并且让开发人员能够充分利用 HTML5 和 JavaScript 库。

第 42 章介绍了 ASP.NET MVC。

1.7.2 使用 WPF

有两种技术可以用于创建 Windows 桌面应用程序：Windows 窗体和 Windows Presentation Foundation(WPF)。Windows 窗体包含的类只是封装了原生 Windows 控件，所以是基于像素图形的。WPF 则是基于矢量图的一种新技术。

WPF 在建立应用程序时使用 XAML。XAML 表示可扩展的应用程序标记语言(eXtensible

Application Markup Language)。这种在 Microsoft 环境下创建应用程序的新方式在 2006 年引入，是.NET Framework 3.0 的一部分。要运行 WPF 应用程序，需要在客户机上至少安装.NET Framework 3.0。当然，更新版本的.NET Framework 会提供新的 WPF 功能。例如，.NET 4.5 中新增了功能区控件和实时造形等功能。

XAML 是用于创建窗体的 XML 声明，它代表 WPF 应用程序的所有可视化部分和操作。虽然可以编程利用 WPF 应用程序，但 WPF 是迈向声明性编程的一步，而声明性编程是编程业的趋势。声明性编程是指，不是利用编译语言，如 C#、VB 或 Java，通过编程来创建对象，而是通过 XML 类型的编程来声明所有元素。第 29 章介绍了 XAML(XML Paper Specification、Windows Workflow Founding 和 Windows Communication Foundation 也使用了 XAML)。

第 35 章详细介绍了如何使用 XAML 和 C#构建 WPF 应用程序。第 36 章详细介绍了如何使用 WPF 和 XAML 创建数据驱动的业务应用程序。打印和创建文档是 WPF 的另外一个重要方面，第 37 章将进行讨论。

1.7.3 Windows Store 应用程序

Windows 8 用“触摸为先”的 Windows Store 应用程序开启了一种新的范式。桌面应用程序通常会提供一个菜单和一个工具栏，用户在应用程序的一个框架中查看下一步可以做什么。Windows Store 应用程序则将重点放到了内容。应用程序的框架应该缩减到最低，只提供用户与内容交互所需的任务，而不是提供他们可以使用的不同选项。关注点应是当前的任务，而不是用户下一步可能要执行的操作。这样一来，用户就会根据内容记住应用程序的用途。“有内容、无框架”是这种技术的口号。

可以使用 C#和 XAML，结合 Windows Runtime 和.NET Framework 的一个子集编写 Windows Store 应用程序。Windows Store 应用程序为开发人员提供了广阔的新世界。其主要缺点是只能运行在 Windows 8 或更高版本的操作系统上。

第 31、38、39 章将详细介绍创建 Windows Store 应用程序。

1.7.4 Windows 服务

Windows 服务(最初称为 NT 服务)是一个在基于 Windows NT 内核的操作系统上后台运行的程序。当希望程序连续运行，并在用户没有明确启动操作时响应事件，就应使用 Windows 服务。例如 Web 服务器上的 World Wide Web 服务，它们监听来自客户端的 Web 请求。

用 C#编写服务非常简单。System.ServiceProcess 名称空间中的.NET Framework 基类可以处理许多与服务相关的样本任务。另外，Visual Studio .NET 允许创建 C# Windows Service 项目，为基本 Windows 服务编写 C#源代码。第 27 章将详细介绍如何编写 C# Windows 服务。

1.7.5 WCF

ASP.NET Web API 可以实现客户端和服务器之间的通信，它使用起来十分简单，但是功能不如 SOAP 协议丰富。

WCF 是一种功能丰富的技术，提供了多种通信选项。使用 WCF 时，既可以使用基于 REST 的

通信，也可以使用基于 SOAP 的通信，都能获得标准 Web 服务(如安全性、事务、双向和单向通信、路由、发现等)提供的所有功能。WCF 允许建立好服务后，只要修改配置文件，就可以用多种方式提供该服务(甚至在不同的协议下)。WCF 是一种连接各种系统的强大的新方式。第 43 章将详细介绍 WCF。第 47 章也会讲到一些基于 WCF 的技术，例如 Message Queueing with WCF。

ASP.NET Web API

ASP.NET Web API 是在客户端和服务器之间进行简单通信的一种新方式(REST 风格)。这种新框架基于 ASP.NET MVC，并使用了控制器和路由。客户端可以收到符合开放数据(Open Data)规范的 JSON 或 Atom 数据。

这个新 API 具备的一些特征使得不只 Web 客户端很容易在 JavaScript 中使用它，而且在 Windows Store 应用程序中也很容易使用。

ASP.NET Web API 在第 44 章中介绍。

1.7.6 Windows WF

Windows Workflow Foundation(WF)实际上是在.NET Framework 3.0 中引入的，但经过全面修订，自从.NET 4 以外许多人都发现它更容易使用了。.NET 4.5 也对它做了一点小改进。Visual Studio 2013 在使用 WF 方面有了长足的进步，并使得使用 C#(原来的版本使用 VB)构建工作流和编写表达式变得更简单。WF 有一个新的状态机设计器和一些新活动。

第 45 章将详细讨论 WF。

1.8 C#在.NET 企业体系结构中的作用

新技术总在快速出现。应该为企业应用程序使用哪种技术呢？有许多因素影响着所要做出的决定。例如，如果现有应用程序是开发人员根据现有知识开发的，应该怎么办？可以在遗留应用程序中集成新功能吗？根据需要的维护量，可能重新构建一些现有的应用程序来使用新功能更加合理。通常，遗留应用程序和新应用程序是可以共存很长一段时间的。应用程序对客户端有什么需求？需要把.NET Framework 升级到 4.5 版本吗？还是 2.0 版本就够了？或者，客户端能够使用.NET 吗？

要做的决定很多，.NET 也提供了很多选项。可以在客户端的 Windows 窗体、WPF 或 Windows 8 应用程序中使用.NET，也可以在使用 IIS 和 ASP.NET 运行库托管的 Web 服务器上的 ASP.NET Web 窗体或 ASP.NET MVC 中使用.NET。服务可以运行在 IIS 内，也可以托管在 Windows 服务内。C# 为希望建立稳健的 n 层客户机/服务器应用程序的公司提供了一个最佳的机会。

C#与 ADO.NET 合并后，就可以快速而经常地访问数据存储库了，如 SQL Server 和其他带有数据提供程序的数据库。使用 ADO.NET Entity Framework 很容易将数据库关系映射为对象层次结构。这不只适用于 SQL Server，也适用于许多提供了 Entity Framework 提供程序的不同数据库。返回的

数据集很容易通过ADO.NET对象模型或LINQ来处理，并自动显示为XML或JSON，以便通过办公室内联网来传输。

一旦为新项目建立了数据库模式，C#就会为执行一层数据访问对象提供一个极好的媒介，每个对象都能提供对不同数据库表的插入、更新和删除访问。

因为C#是第一个基于组件的C语言，所以非常适合于执行业务对象层。它为组件之间的通信封装了杂乱的信息，让开发人员把注意力集中在如何把数据访问对象组合在一起，在方法中精确地强制执行公司的业务规则。

要使用C#创建企业应用程序，可以为数据访问对象创建一个类库项目，为业务对象创建另一个类库项目。在开发时，可以使用Console项目测试类上的方法。喜欢编程的人可以建立能自动从批处理文件中执行的Console项目，对工作代码进行单元测试，以便确定代码是否中断。

注意，C#和.NET都会影响物理封装可重用类的方式。过去，许多开发人员把许多类放在一个物理组件中，因为这样安排会使部署容易得多；如果有版本冲突问题，就知道在何处进行检查。因为部署.NET企业组件仅是把文件复制到目录中，所以现在开发人员可以把他们的类封装到逻辑性更高的离散组件中，而不会遇到“DLL Hell”。

最后，用C#编写的ASP.NET页面构成了用户界面的绝妙媒介。ASP.NET页面是编译过的，所以执行得比较快。它们可以在Visual Studio 2013 IDE中调试，所以十分健壮。它们支持所有的语言功能，例如早期绑定、继承和模块化，所以用C#编写的ASP.NET页面是很整洁的，很容易维护。

在SOA和基于服务的编程热潮过后，现在使用服务已经成为了一种业界常规。新的趋势是基于云的编程，Microsoft为此提供了Windows Azure。可以在本地服务器或者云中的ASP.NET Web窗体、ASP.NET Web API或WCF中运行.NET应用程序。客户端则可以使用HTML 5来接触更多的受众，或者使用WPF或Windows Store应用程序来实现丰富的功能。.NET仍然能够跟得上新技术，利用新选项，所以生机是无限的。

1.9 小结

本章介绍了许多基础知识，简要回顾了.NET Framework的重要方面以及它与C#的关系。首先讨论了所有面向.NET的语言如何编译为中间语言(之后由公共语言运行库进行编译和执行)，接着讨论了.NET的下述特性在编译和执行过程中的作用：

- 程序集和.NET基类
- COM组件
- JIT编译
- 应用程序域
- 垃圾回收

图1-4简要说明了这些特性在编译和执行过程中如何发挥作用。

本章还讨论了IL的特征，特别是其强数据类型化和面向对象的特征。探讨了这些特征如何影响面向.NET(包括C#)的语言，并阐述了IL的强类型本质如何支持语言的互操作性，以及CLR服务，如垃圾回收和安全性。还讨论了用于帮助处理语言互操作性的CLS和CTS。

本章最后讨论了C#如何用作基于几种.NET技术(包括ASP.NET和WPF)的应用程序的基础。

第 2 章将介绍如何用 C#语言编写代码。

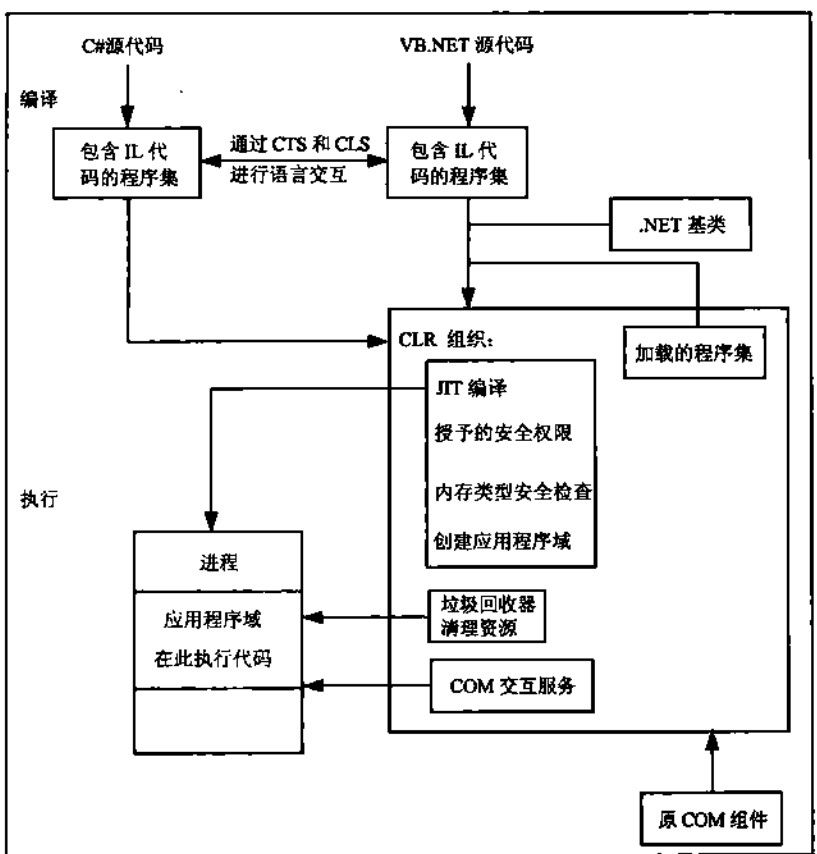


图 1-4

第 2 章

核心 C#

本章要点

- 声明变量
- 变量的初始化和作用域
- C#的预定义数据类型
- 在 C#程序中使用条件语句、循环和跳转语句指定执行流
- 枚举
- 名称空间
- Main()方法
- 基本的命令行 C#编译器选项
- 使用 System.Console 执行控制台 I/O
- 使用内部注释和文档编制功能
- 预处理器指令
- C#编程的推荐规则和约定

本章源代码下载地址(wrox.com):

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- ArgsExample.cs
- DoubleMain.cs
- ElseIf.cs
- First.cs
- MathClient.cs
- MathLibrary.cs
- NestedFor.cs
- Scope.cs

- ScopeBad.cs
- ScopeTest2.cs
- StringExample.cs
- Var.c.s

2.1 C#基础

理解了 C#的用途后，就可以学习如何使用它了。本章将介绍 C#的基础知识，本章的内容也是后续章节的基础，好的开端等于成功的一半。阅读完本章后，读者就有足够的 C#知识编写简单的程序了，但还不能使用继承或其他面向对象的特征。这些内容将在后面的几章中讨论。

2.2 第一个 C#程序

下面编译并运行最简单的 C#程序，这是一个简单的控制台应用程序，它由把某条消息写到屏幕上的一个类组成。

后面几章会介绍许多代码示例。编写 C#程序最常用的技巧是使用 Visual Studio 2013 生成一个基本项目，再添加自己的代码。但是，第 I 部分的目的是讲授 C#语言，为了简单起见，在第 17 章之前避免涉及 Visual Studio 2013。我们使代码显示为简单的文件，这样就可以使用任何文本编辑器输入它们，并在命令行上编译。

2.2.1 代码

在文本编辑器(如 Notepad)中输入下面的代码，把它保存为后缀名为.cs 的文件(如 First.cs)。Main()方法如下所示(更多信息参见 2.7 节)：

```
using System;

namespace Wrox
{
  public class MyFirstClass
  {
    static void Main()
    {
      Console.WriteLine("Hello from Wrox.");
      Console.ReadLine();
      return;
    }
  }
}
```

2.2.2 编译并运行程序

对源文件运行 C#命令行编译器(csc.exe)，编译这个程序：

```
csc First.cs
```

如果使用csc命令在命令行上编译代码，就应注意.NET命令行工具(包括csc)只有在设置了某些环境变量后才能使用。根据安装.NET(和Visual Studio)的方式，这里显示的结果可能与你计算机上的结果不同。

如果没有设置环境变量，有两种解决方法。第1种方法是在运行csc之前，从命令提示符窗口上运行批处理文件%Microsoft Visual Studio 2013%\Common7\Tools\vsvars32.bat。其中%Microsoft Visual Studio 2013%是Visual Studio 2013的安装文件夹。第2种方法(更简单)是使用Visual Studio 2013命令提示符代替通常的命令提示符窗口。Visual Studio 2013命令提示符在菜单"开始" | "程序" | Microsoft Visual Studio 2013|Visual Studio Tools子菜单下。它只是一个命令提示符窗口，打开时会自动运行vsvars32.bat。

编译代码，会生成一个可执行文件First.exe。在命令行或Windows Explorer上，像运行任何可执行文件那样运行该文件，得到如下结果：

```
csc First.cs
Microsoft (R) Visual C# Compiler version 12.021005.1
For C# 5.0
Copyright (C) Microsoft Corporation. All rights reserved.

First.exe
Hello from Wrox.
```

2.2.3 详细介绍

首先对C#语法做几个一般性的解释。在C#中，与其他C风格的语言一样，大多数语句都以分号(;)结尾，语句可以写在多个代码行上，不需要使用续行字符。用花括号({})把语句组合为块。单行注释以两个斜杠字符开头(//)，多行注释以一条斜杠和一个星号(/*)开头，以一个星号和一条斜杠(*/)结尾。在这些方面，C#与C++和Java一样，但与Visual Basic不同。分号和花括号使C#代码与Visual Basic代码有差异很大的外观。如果你以前使用的是Visual Basic，就应特别注意每条语句结尾的分号。对于新接触C风格语言的用户，忽略分号常常是导致编译错误的一个最主要的原因。另一个方面是，C#区分大小写，也就是说，变量myVar与MyVar是两个不同的变量。

在上面的代码示例中，前几行代码与名称空间有关(如本章后面所述)，名称空间是把相关类组合在一起的方式。namespace关键字声明了应与类相关的名称空间。其后花括号中的所有代码都被认为是在这个名称空间中。编译器在using语句指定的名称空间中查找没有在当前名称空间中定义但在代码中引用的类。这非常类似于Java中的import语句和C++中的using namespace语句。

```
using System;

namespace Wrox
{
```

在First.cs文件中使用using指令的原因是下面要使用一个库类System.Console。using System语句允许把这个类简写为Console(System名称空间中的其他类也与此类似)。如果没有using，就必

须完全限定对 Console.WriteLine()方法的调用，如下所示：

```
System.Console.WriteLine("Hello from Wrox.");
```

标准的 System 名称空间包含了最常用的.NET 类型。在 C#中做的所有工作都依赖于.NET 基类，认识到这一点非常重要；在本例中，我们使用了 System 名称空间中的 Console 类，以写入控制台窗口。C#没有用于输入和输出的内置关键字，而是完全依赖于.NET 类。

几乎所有的 C#程序都使用 System 名称空间中的类，所以假定本章所有的代码文件都包含 using System;语句。

接着，声明一个类 MyFirstClass。但是，因为该类位于 Wrox 名称空间中，所以其完整的名称是 Wrox.MyFirstCSharpClass：

```
class MyFirstCSharpClass
{
```

所有的 C#代码都必须包含在一个类中。类的声明包括 class 关键字，其后是类名和一对花括号。与类相关的所有代码都应放在这对花括号中。

下面声明方法 Main()。每个 C#可执行文件(如控制台应用程序、Windows 应用程序和 Windows 服务)都必须有一个入口点——Main()方法(注意 M 大写)：

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

在程序启动时调用这个方法。该方法要么没有返回值(void)，要么返回一个整数(int)。注意，在 C#中方法的定义如下所示：

```
  [modifiers] return_type MethodName([parameters])
{
   // Method body. NB. This code block is pseudo-code.
}
```

第一个方括号中的内容表示可选关键字。修饰符(modifiers)用于指定用户所定义的方法的某些特性，如可以在什么地方调用该方法。在本例中，有两个修饰符 public 和 static。修饰符 public 表示可以在任何地方访问该方法，所以可以在类的外部调用它。修饰符 static 表示方法不能在类的实例上执行，因此不必先实例化类再调用。这非常重要，因为我们创建的是一个可执行文件，而不是类库。把返回类型设置为 void，在本例中，不包含任何参数。

最后，看看代码语句：

```
Console.WriteLine("Hello from Wrox.");
Console.ReadLine();
return;
```

在本例中，我们只调用了 System.Console 类的 WriteLine()方法，把一行文本写到控制台窗口上。WriteLine()是一个静态方法，在调用之前不需要实例化 Console 对象。

Console.ReadLine()读取用户的输入，添加这行代码会让应用程序等待用户按回车键，之后退出

应用程序。在 Visual Studio 2013 中，控制台窗口会消失。

然后调用 return 退出该方法(因为这是 Main 方法，所以也退出了程序)。在方法头中指定 void，因此没有返回值。

对 C#基本语法有了大致的认识后，下面就详细讨论 C#的各个方面。因为没有变量不可能编写出重要的程序，所以首先介绍 C#中的变量。

2.3 变量

在 C#中声明变量使用下述语法：

```
datatype identifier;
```

例如：

```
int i;
```

该语句声明 int 变量 i。编译器不允许在表达式中使用这个变量，除非用一个值初始化了该变量。声明 i 之后，就可以使用赋值运算符(=)给它赋值：

```
i = 10;
```

还可以在一行代码中声明变量，并初始化它的值：

```
int i = 10;
```

如果在一条语句中声明和初始化了多个变量，那么所有的变量都具有相同的数据类型：

```
int x = 10, y =20; // x and y are both ints
```

要声明不同类型的变量，需要使用单独的语句。在多个变量的声明中，不能指定不同的数据类型：

```
int x = 10;
bool y = true;              // Creates a variable that stores true or false
int x = 10, bool y = true;  // This won't compile!
```

注意上面例子中的“//”和其后的文本，它们是注释。“//”字符串告诉编译器，忽略该行后面的文本，这些文本仅为了让人更好地理解程序，它们并不是程序的一部分。本章后面会详细讨论代码中的注释。

2.3.1 变量的初始化

变量的初始化是 C#强调安全性的另一个例子。简单地说，C#编译器需要用某个初始值对变量进行初始化，之后才能在操作中引用该变量。大多数现代编译器把没有初始化标记为警告，但 C#编译器把它当作错误来看待。这就可以防止我们无意中从其他程序遗留下来的内存中获取垃圾值。

C#有两个方法可确保变量在使用前进行了初始化：

- 变量是类或结构中的字段，如果没有显式初始化，创建这些变量时，其默认值就是 0(类和结构在后面讨论)。
- 方法的局部变量必须在代码中显式初始化，之后才能在语句中使用它们的值。此时，初始化不是在声明该变量时进行的，但编译器会通过方法检查所有可能的路径，如果检测到局部变量在初始化之前就使用了它的值，就会产生错误。

例如，在 C#中不能使用下面的语句：

```
public static int Main()
{
    int d;
    Console.WriteLine(d); // Can't do this! Need to initialize d before use
    return 0;
}
```

注意在这段代码中，演示了如何定义 Main()，使之返回一个 int 类型的数据，而不是 void。

在编译这些代码时，会得到下面的错误消息：

```
Use of unassigned local variable 'd'
```

考虑下面的语句：

```
Something objSomething;
```

在 C#中，这行代码仅会为 Something 对象创建一个引用，但这个引用还没有指向任何对象。对该变量调用方法或属性会导致错误。

在 C#中实例化一个引用对象需要使用 new 关键字。如上所述，创建一个引用，使用 new 关键字把该引用指向存储在堆上的一个对象：

```
objSomething = new Something(); // This creates a Something on the heap
```

2.3.2 类型推断

类型推断(type inference)使用 var 关键字。声明变量的语法有些变化。编译器可以根据变量的初始化值“推断”变量的类型。例如：

```
int someNumber = 0;
```

就变成：

```
var someNumber = 0;
```

即使 someNumber 从来没有声明为 int，编译器也可以确定，只要 someNumber 在其作用域内，就是一个 int。编译后，上面两个语句是等价的。

下面是另一个小例子：

```
using System;

namespace Wrox
{
  class Program
  {
```

```
        static void Main(string[] args)
        {
          var name = "Bugs Bunny";
          var age = 25;
          var isRabbit = true;

          Type nameType = name.GetType();
          Type ageType = age.GetType();
          Type isRabbitType = isRabbit.GetType();

          Console.WriteLine("name is type " + nameType.ToString());
          Console.WriteLine("age is type " + ageType.ToString());
          Console.WriteLine("isRabbit is type " + isRabbitType.ToString());
        }
      }
    }
```

这个程序的输出如下：

```
name is type System.String
age is type System.Int32
isRabbit is type System.Bool
```

需要遵循一些规则：

- 变量必须初始化。否则，编译器就没有推断变量类型的依据。
- 初始化器不能为空。
- 初始化器必须放在表达式中。
- 不能把初始化器设置为一个对象，除非在初始化器中创建了一个新对象。

第 3 章在讨论匿名类型时将详细探讨。

声明了变量，推断出了类型后，就不能改变变量类型了。变量的类型确定后，就遵循其他变量类型遵循的强类型化规则。

2.3.3 变量的作用域

变量的作用域是可以访问该变量的代码区域。一般情况下，确定作用域遵循以下规则：

- 只要类在某个作用域内，其字段(也称为成员变量)也在该作用域内。
- 局部变量存在于表示声明该变量的块语句或方法结束的右花括号之前的作用域内。
- 在 for、while 或类似语句中声明的局部变量存在于该循环体内。

1. 局部变量的作用域冲突

大型程序在不同部分为不同的变量使用相同的变量名很常见。只要变量的作用域是程序的不同部分，就不会有问题，也不会产生多义性。但要注意，同名的局部变量不能在同一作用域内声明两次，所以不能使用下面的代码：

```
int x = 20;
// some more code
int x = 30;
```

考虑下面的代码示例：

```
using System;
namespace Wrox.ProCSharp.Basics
{
  public class ScopeTest
  {
    public static int Main()
    {
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine(i);
      } // i goes out of scope here
      // We can declare a variable named i again, because
      // there's no other variable with that name in scope
      for (int i = 9; i >= 0; i-)
      {
         Console.WriteLine(i);
      } // i goes out of scope here.
      return 0;
    }
  }
}
```

这段代码使用两个for循环打印0~9的数字，再逆序打印0~9的数字。重要的是在同一个方法中，代码中的变量i声明了两次。可以这么做的原因是i在两个不同的循环内部声明，所以变量i对于各自的循环来说是局部变量。

下面是另一个例子：

```
public static int Main()
{
  int j = 20;
  for (int i = 0; i < 10; i++)
  {
    int j = 30; // Can't do this - j is still in scope
    Console.WriteLine(j + i);
  }
  return 0;
}
```

如果试图编译它，就会产生如下错误：

```
ScopeTest.cs(12,15): error CS0136: A local variable named 'j' cannot be declared in
this scope because it would give a different meaning to 'j', which is already used
in a 'parent or current' scope to denote something else.
```

其原因是：变量j是在for循环开始前定义的，在执行for循环时应处于其作用域内，在Main()方法结束执行后，变量j才超出作用域，第2个j(不合法)则在循环的作用域内，该作用域嵌套在Main()方法的作用域内。因为编译器无法区分这两个变量，所以不允许声明第2个变量。

2. 字段和局部变量的作用域冲突

某些情况下，可以区分名称相同(尽管其完全限定的名称不同)、作用域相同的两个标识符。此时编译器允许声明第2个变量。原因是C#在变量之间有一个基本的区分，它把在类型级别声明的变

量看作字段，而把在方法中声明的变量看作局部变量。

考虑下面的代码：

```
using System;
namespace Wrox
{
  class ScopeTest2
  {
    static int j = 20;
    public static void Main()
    {
      int j = 30;
      Console.WriteLine(j);
      return;
    }
  }
}
```

虽然在 Main()方法的作用域内声明了两个变量 j，这段代码也会编译——在类级上定义的 j，在该类删除前是不会超出作用域的(在本例中，当 Main()方法终止，程序结束时，才会删除该类)；以及在 Main()中定义的 j。此时，在 Main()方法中声明的新变量 j 隐藏了同名的类级变量，所以在运行这段代码时，会显示数字 30。

但是，如果要引用类级变量，该怎么办？可以使用语法 object.fieldname，在对象的外部引用类或结构的字段。在上面的例子中，我们访问静态方法中的一个静态字段(静态字段详见下一节)，所以不能使用类的实例，只能使用类本身的名称：

```
..
public static void Main()
{
  int j = 30;
  Console.WriteLine(j);
  Console.WriteLine(ScopeTest2.j);
}
..
```

如果要访问一个实例字段(该字段属于类的一个特定实例)，就需要使用 this 关键字。

2.3.4 常量

顾名思义，常量是其值在使用过程中不会发生变化的变量。在声明和初始化变量时，在变量的前面加上关键字 const，就可以把该变量指定为一个常量：

```
const int a = 100; // This value cannot be changed.
```

常量具有如下特点：

- 常量必须在声明时初始化。指定了其值后，就不能再改写了。
- 常量的值必须能在编译时用于计算。因此，不能用从一个变量中提取的值来初始化常量。如果需要这么做，应使用只读字段(详见第 3 章)。
- 常量总是静态的。但注意，不必(实际上，是不允许)在常量声明中包含修饰符 static。

在程序中使用常量至少有 3 个好处：

- 由于使用易于读取的名称(名称的值易于理解)替代了较难读取的数字或字符串，常量使程序变得更易于阅读。
- 常量使程序更易于修改。例如，在 C#程序中有一个 SalesTax 常量，该常量的值为 6%。如果以后销售税率发生变化，把新值赋给这个常量，就可以修改所有的税款计算结果，而不必查找整个程序去修改税率为 0.06 的每个项。
- 常量更容易避免程序出现错误。如果在声明常量的位置以外的某个地方将另一个值赋给常量，编译器就会报告错误。

2.4 预定义数据类型

前面介绍了如何声明变量和常量，下面要详细讨论 C#中可用的数据类型。与其他语言相比，C#对其可用的类型及其定义有更严格的描述。

2.4.1 值类型和引用类型

在开始介绍 C#中的数据类型之前，理解 C#把数据类型分为两种非常重要：

- 值类型
- 引用类型

下面几节将详细介绍值类型和引用类型的语法。从概念上看，其区别是值类型直接存储其值，而引用类型存储对值的引用。

这两种类型存储在内存的不同地方：值类型存储在堆栈中，而引用类型存储在托管堆上。注意区分某个类型是值类型还是引用类型，因为这种存储位置的不同会有不同的影响。例如，int 是值类型，这表示下面的语句会在内存的两个地方存储值 20：

```
// i and j are both of type int
i = 20;
j = i;
```

但考虑下面的代码。这段代码假定已经定义了一个类 Vector，Vector 是一个引用类型，它有一个 int 类型的成员变量 Value：

```
Vector x, y;
x = new Vector();
x.Value = 30; // Value is a field defined in Vector class
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

要理解的重要一点是在执行这段代码后，只有一个 Vector 对象。x 和 y 都指向包含该对象的内存位置。因为 x 和 y 是引用类型的变量，声明这两个变量只保留了一个引用——而不会实例化给定类型的对象。两种情况下都不会真正创建对象。要创建对象，就必须使用 new 关键字，如上所示。因为 x 和 y 引用同一个对象，所以对 x 的修改会影响 y，反之亦然。因此上面的代码会显示 30 和 50。

C++开发人员应注意，这个语法类似于引用，而不是指针。我们使用.(句点)符号，而不是->来访问对象成员。在语法上，C#引用看起来更类似于 C++引用变量。但是，抛开表面的语法，实际上它类似于 C++指针。

如果变量是一个引用，就可以把其值设置为 null，表示它不引用任何对象：

```
y = null;
```

如果将引用设置为 null，显然就不可能对它调用任何非静态的成员函数或字段，这么做会在运行期间抛出一个异常。

在 C#中，基本数据类型如 bool 和 long 都是值类型。如果声明一个 bool 变量，并给它赋予另一个 bool 变量的值，在内存中就会有两个 bool 值。如果以后修改第一个 bool 变量的值，第二个 bool 变量的值也不会改变。这些类型是通过值来复制的。

相反，大多数更复杂的 C#数据类型，包括我们自己声明的类都是引用类型。它们分配在堆中，其生存期可以跨多个函数调用，可以通过一个或几个别名来访问。CLR 实现一种精细的算法，来跟踪哪些引用变量仍是可以访问的，哪些引用变量已经不能访问了。CLR 会定期删除不能访问的对象，把它们占用的内存返回给操作系统。这是通过垃圾回收器实现的。

把基本类型(如 int 和 bool)规定为值类型，而把包含许多字段的较大类型(通常在有类的情况下)规定为引用类型，C#设计这种方式的原因是可以得到最佳性能。如果要把自己的类型定义为值类型，就应把它声明为一个结构。

2.4.2　CTS 类型

如第 1 章所述，C#认可的基本预定义类型并没有内置于 C#语言中，而是内置于.NET Framework 中。例如，在 C#中声明一个 int 类型的数据时，声明的实际上是.NET 结构 System.Int32 的一个实例。这听起来似乎很深奥，但其意义深远：这表示在语法上，可以把所有的基本数据类型看成支持某些方法的类。例如，要把 int i 转换为 string，可以编写下面的代码：

```
string s = i.ToString();
```

应强调的是，在这种便利语法的背后，类型实际上仍存储为基本类型。基本类型在概念上用.NET 结构表示，所以肯定没有性能损失。

下面看看 C#中定义的内置类型。我们将列出每个类型，以及它们的定义和对应.NET 类型(CTS 类型)的名称。C#有 15 个预定义类型，其中 13 个是值类型，两个是引用类型(string 和 object)。

2.4.3　预定义的值类型

内置的 CTS 值类型表示基本类型，如整型和浮点类型、字符类型和布尔类型。

1. 整型

C#支持 8 个预定义整数类型，如表 2-1 所示。

表2-1

名　称	CTS 类型	说　明	范　围
sbyte	System.SByte	8 位有符号的整数	–128～127 (-2^7～2^7-1)
short	System.Int16	16 位有符号的整数	–32 768～32 767 (-2^{15}～$2^{15}-1$)
int	System.Int32	32 位有符号的整数	–2 147 483 648～2 147 483 647 (-2^{31}～$2^{31}-1$)
long	System.Int64	64 位有符号的整数	–9 223 372 036 854 775 808～ 9 223 372 036 854 775 807 (-2^{63}～$2^{63}-1$)
byte	System.Byte	8 位无符号的整数	0～255 (0～2^8-1)
ushort	System.UInt16	16 位无符号的整数	0～65 535 (0～$2^{16}-1$)
uint	System.UInt32	32 位无符号的整数	0～4 294 967 295 (0～$2^{32}-1$)
ulong	System.UInt64	64 位无符号的整数	0～18 446 744 073 709 551 615 (0～$2^{64}-1$)

一些 C#类型的名称与 C++和 Java 类型一致，但其定义不同。例如，在 C#中，int 总是 32 位带符号的整数。而在 C++中，int 是带符号的整数，但其位数取决于平台(在 Windows 上是 32 位)。在 C#中，所有的数据类型都以与平台无关的方式定义，以备将来 C#和.NET 迁移到其他平台上。

byte 是 0~255(包括 255)的标准 8 位类型。注意，在强调类型的安全性时，C#认为 byte 类型和 char 类型完全不同，它们之间的编程转换必须显式写出。还要注意，与整数中的其他类型不同，byte 类型在默认状态下是无符号的，其有符号的版本有一个特殊的名称 sbyte。

在.NET 中，short 不再很短，现在它有 16 位长。int 类型更长，有 32 位。long 类型最长，有 64 位。所有整数类型的变量都能被赋予十进制或十六进制的值，后者需要 0x 前缀：

```
long x = 0x12ab;
```

如果对一个整数是 int、uint、long 或是 ulong 没有任何显式的声明，则该变量默认为 int 类型。为了把输入的值指定为其他整数类型，可以在数字后面加上如下字符：

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

也可以使用小写字母 u 和 l，但后者会与整数 1 混淆。

2. 浮点类型

C#提供了许多整型数据类型，也支持浮点类型，如表 2-2 所示。

表2-2

名　称	CTS 类型	说　明	位　数	范围(大致)
float	System.Single	32 位单精度浮点数	7	$\pm1.5\times10^{245}$～$\pm3.4\times10^{38}$
double	System.Double	64 位双精度浮点数	15/16	$\pm5.0\times10^{-324}$～$\pm1.7\times10^{308}$

float 数据类型用于较小的浮点数，因为它要求的精度较低。double 数据类型比 float 数据类型大，

提供的精度也大一倍(15 位)。

如果在代码中对某个非整数值(如 12.3)硬编码，则编译器一般假定该变量是 double。如果想指定该值为 float，可以在其后加上字符 F(或 f)：

```
floatf = 12.3F;
```

3. decimal 类型

decimal 类型表示精度更高的浮点数，如表 2-3 所示。

表 2-3

名　称	CTS 类 型	说　明	位　数	范围(大致)
decimal	System.Decimal	128 位高精度十进制数表示法	28	$\pm1.0\times10^{-28}\sim\pm7.9\times10^{28}$

CTS 和 C#一个重要的优点是提供了一种专用类型进行财务计算，这就是 decimal 类型，使用 decimal 类型提供的 28 位的方式取决于用户。换言之，可以用较大的精确度(带有美分)来表示较小的美元值，也可以在小数部分用更多的舍入来表示较大的美元值。但应注意，decimal 类型不是基本类型，所以在计算时使用该类型会有性能损失。

要把数字指定为 decimal 类型，而不是 double、float 或整型，可以在数字的后面加上字符 M(或 m)，如下所示。

```
decimal d = 12.30M;
```

4. bool 类型

C#的 bool 类型用于包含布尔值 true 或 false，如表 2-4 所示。

表 2-4

名　称	CTS 类 型	说　明	位　数	值
bool	System.Boolean	表示 true 或 false	NA	true 或 false

bool 值和整数值不能相互隐式转换。如果变量(或函数的返回类型)声明为 bool 类型，就只能使用值 true 或 false。如果试图使用 0 表示 false，非 0 值表示 true，就会出错。

5. 字符类型

为了保存单个字符的值，C#支持 char 数据类型，如表 2-5 所示。

表 2-5

名　称	CTS 类 型	值
char	System.Char	表示一个 16 位的(Unicode)字符

char 类型的字面量是用单引号括起来的，如'A'。如果把字符放在双引号中，编译器会把它看作字符串，从而产生错误。

除了把 char 表示为字符字面量之外，还可以用 4 位十六进制的 Unicode 值(如'\u0041')、带有数据类型转换的整数值(如(char)65)或十六进制数('\x0041')表示它们。它们还可以用转义序列表示，如表 2-6 所示。

表 2-6

转义序列	字　符
\'	单引号
\"	双引号
\\	反斜杠
\0	空
\a	警告
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

2.4.4 预定义的引用类型

C#支持两种预定义的引用类型，如表 2-7 所示。

表 2-7

名　称	CTS 类 型	说　明
object	System.Object	根类型，CTS 中的其他类型都是从它派生而来的(包括值类型)
string	System.String	Unicode 字符串

1. object 类型

许多编程语言和类结构都提供了根类型，层次结构中的其他对象都从它派生而来。C#和.NET 也不例外。在 C#中，object 类型就是最终的父类型，所有内置类型和用户定义的类型都从它派生而来。这样，object 类型就可以用于两个目的：

- 可以使用 object 引用绑定任何子类型的对象。例如，第 7 章将说明如何使用 object 类型把堆栈中的一个值对象装箱，再移动到堆中。object 引用也可以用于反射，此时必须有代码来处理类型未知的对象。
- object 类型实现了许多一般用途的基本方法，包括 Equals()、GetHashCode()、GetType()和 ToString()。用户定义的类需要使用一种面向对象技术——重写(见第 4 章)，提供其中一些方

法的替代实现代码。例如，重写 ToString()时，要给类提供一个方法，给出类本身的字符串表示。如果类中没有提供这些方法的实现代码，编译器就会使用 object 类型中的实现代码，它们在类中的执行不一定正确。

后面的章节将详细讨论 object 类型。

2. string 类型

C#有 string 关键字，在编译为.NET 类时，它就是 system.String。有了它，像字符串连接和字符串复制这样的操作就很简单了：

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

尽管这是一个值类型的赋值，但 string 是一个引用类型。String 对象被分配在堆上，而不是栈上。因此，当把一个字符串变量赋予另一个字符串时，会得到对内存中同一个字符串的两个引用。但是，string 与引用类型的常见行为有一些区别。例如，字符串是不可改变的。修改其中一个字符串，就会创建一个全新的 string 对象，而另一个字符串不发生任何变化。考虑下面的代码：

```
using System;
class StringExample
{
  public static int Main()
  {
    string s1 = "a string";
    string s2 = s1;
    Console.WriteLine("s1 is " + s1);
    Console.WriteLine("s2 is " + s2);
    s1 = "another string";
    Console.WriteLine("s1 is now " + s1);
    Console.WriteLine("s2 is now " + s2);
    return 0;
  }
}
```

其输出结果为：

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

改变 s1 的值对 s2 没有影响，这与我们期待的引用类型正好相反。当用值"a string"初始化 s1 时，就在堆上分配了一个新的 string 对象。在初始化 s2 时，引用也指向这个对象，所以 s2 的值也是"a string"。但是当现在要改变 s1 的值时，并不会替换原来的值，堆上会为新值分配一个新对象。s2 变量仍指向原来的对象，所以它的值没有改变。这实际上是运算符重载的结果，运算符重载详见第 7 章。基本上，string 类实现为其语义遵循一般的、直观的字符串规则。

字符串字面量放在双引号中("...")；如果试图把字符串放在单引号中，编译器就会把它当作 char

类型，从而引发错误。C#字符串和char一样，可以包含Unicode和十六进制数转义序列。因为这些转义序列以一个反斜杠开头，所以不能在字符串中使用没有经过转义的反斜杠字符，而需要用两个反斜杠字符(\\)来表示它：

```
string filepath = "C:\\ProCSharp\\First.cs";
```

即使用户相信自己可以在任何情况下都记住要这么做，但输入两个反斜杠字符会令人迷惑。幸好，C#提供了另一种替代方式。可以在字符串字面量的前面加上字符@，在这个字符后的所有字符都看成其原来的含义——它们不会解释为转义字符：

```
string filepath = @"C:\ProCSharp\First.cs";
```

甚至允许在字符串字面量中包含换行符：

```
string jabberwocky = @"'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.";
```

那么jabberwocky的值就是：

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

2.5 流控制

本节将介绍C#语言的重要语句：控制程序流的语句，它们不是按代码在程序中的排列位置顺序执行的。

2.5.1 条件语句

条件语句可以根据条件是否满足或根据表达式的值控制代码的执行分支。C#有两个控制代码分支的结构：if语句，测试特定条件是否满足；switch语句，它比较表达式和多个不同的值。

1. if 语句

对于条件分支，C#继承了C和C++的if...else结构。对于用过程语言编程的人，其语法非常直观：

```
if (condition)
   statement(s)
else
   statement(s)
```

如果在条件中要执行多个语句，就需要用花括号({ ... })把这些语句组合为一个块(这也适用于其他可以把语句组合为一个块的C#结构，如for和while循环)。

```
bool isZero;
if (i == 0)
{
   isZero = true;
   Console.WriteLine("i is Zero");
```

```
}
else
{
   isZero = false;
   Console.WriteLine("i is Non-zero");
}
```

还可以单独使用 if 语句，不加最后的 else 语句。也可以合并 else if 子句，测试多个条件。

```
using System;
namespace Wrox
{
    class MainEntryPoint
    {
    static void Main(string[] args)
    {
        Console.WriteLine("Type in a string");
        string input;
        input = Console.ReadLine();
        if (input == "")
        {
           Console.WriteLine("You typed in an empty string.");
        }
        else if (input.Length < 5)
        {
           Console.WriteLine("The string had less than 5 characters.");
        }
        else if (input.Length < 10)
        {
           Console.WriteLine("The string had at least 5 but less than 10
             Characters.");
        }
        Console.WriteLine("The string was " + input);
    }
}
```

添加到 if 子句中的 else if 语句的个数不受限制。

注意在上面的例子中，我们声明了一个字符串变量 input，让用户在命令行上输入文本，把文本填充到 input 中，然后测试该字符串变量的长度。代码还说明了在 C#中如何进行字符串处理。例如，要确定 input 的长度，可以使用 input.Length。

对于 if，要注意的一点是如果条件分支中只有一条语句，就无须使用花括号：

```
if (i == 0) Let's add some brackets here.
   Console.WriteLine("i is Zero");      // This will only execute if i == 0
Console.WriteLine("i can be anything"); // Will execute whatever the
                                        // value of i
```

但是，为了保持一致，许多程序员只要使用 if 语句，就加上花括号。

前面介绍的 if 语句还演示了用于比较数值的一些 C#运算符。特别注意，C#使用“==”对变量进行等于比较。此时不要使用“=”，一个“=”用于赋值。

在 C#中，if 子句中的表达式必须等于布尔值。不能直接测试整数(如从函数中返回的值)，而必须明确地把返回的整数转换为布尔值 true 或 false，例如，将值与 0 或 null 进行比较：

```
if (DoSomething() != 0)
{
   // Non-zero value returned
}
else
{
   // Returned zero
}
```

2. switch 语句

switch...case 语句适合于从一组互斥的分支中选择一个执行分支。其形式是 switch 参数的后面跟一组 case 子句。如果 switch 参数中表达式的值等于某个 case 子句旁边的某个值，就执行该 case 子句中的代码。此时不需要使用花括号把语句组合到块中；只需要使用 break 语句标记每段 case 代码的结尾即可。也可以在 switch 语句中包含一条 default 子句，如果表达式不等于任何 case 子句的值，就执行 default 子句的代码。下面的 switch 语句测试 integerA 变量的值：

```
switch (integerA)
{
   case 1:
      Console.WriteLine("integerA =1");
      break;
   case 2:
      Console.WriteLine("integerA =2");
      break;
   case 3:
      Console.WriteLine("integerA =3");
      break;
   default:
      Console.WriteLine("integerA is not 1,2, or 3");
      break;
}
```

注意 case 的值必须是常量表达式；不允许使用变量。

C 和 C++程序员应很熟悉 switch...case 语句，而 C#的 switch...case 语句更安全。特别是它禁止几乎所有 case 中的失败条件。如果激活了块中靠前的一条 case 子句，后面的 case 子句就不会被激活，除非使用 goto 语句特别标记也要激活后面的 case 子句。编译器会把没有 break 语句的 case 子句标记为错误：

```
Control cannot fall through from one case label ('case 2:') to another
```

在有限的几种情况下，这种失败是允许的，但在大多数情况下，我们不希望出现这种失败，而且这会导致出现很难察觉的逻辑错误。让代码正常工作，而不是出现异常，这样不是更好吗？

但在使用 goto 语句时，会在 switch...cases 中重复出现失败。如果确实想这么做，就应重新考虑设计方案了。下面的代码说明了如何使用 goto 模拟失败，得到的代码会非常混乱：

```
// assume country and language are of type string
switch(country)
{
    case "America":
        CallAmericanOnlyMethod();
```

```
        goto case "Britain";
    case "France":
        language = "French";
        break;
    case "Britain":
        language = "English";
        break;
}
```

但有一种例外情况。如果一条 case 子句为空，就可以从这个 case 跳到下一条 case 上，这样就可以用相同的方式处理两条或多条 case 子句了(不需要 goto 语句)。

```
switch(country)
{
    case "au":
    case "uk":
    case "us":
        language = "English";
        break;
    case "at":
    case "de":
        language = "German";
        break;
}
```

在C#中，switch 语句的一个有趣的地方是case 子句的排放顺序是无关紧要的，甚至可以把 default 子句放在最前面！因此，任何两条 case 都不能相同。这包括值相同的不同常量，所以不能这样编写：

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
    case england:
    case britain: // This will cause a compilation error.
        language = "English";
        break;
}
```

上面的代码还说明了 C#中的 switch 语句与 C++中的 switch 语句的另一个不同之处：在 C#中，可以把字符串用作测试的变量。

2.5.2　循环

C#提供了 4 种不同的循环机制(for、while、do...while 和 foreach)，在满足某个条件之前，可以重复执行代码块。

1. for 循环

C#的 for 循环提供的迭代循环机制是在执行下一次迭代前，测试是否满足某个条件，其语法如下：

```
for (initializer; condition; iterator):
   statement(s)
```

其中：

- initializer 是指在执行第一次循环前要计算的表达式(通常把一个局部变量初始化为循环计数器)。
- condition 是在每次迭代执行新循环前要测试的表达式(它必须等于 true，才能执行下一次迭代)。
- iterator 是每次迭代完要计算的表达式(通常是递增循环计数器)。

当 condition 等于 false 时，迭代停止。

for 循环是所谓的预测试循环，因为循环条件是在执行循环语句前计算的，如果循环条件为假，循环语句就根本不会执行。

for 循环非常适合于一个语句或语句块重复执行预定的次数。下面的例子就是 for 循环的典型用法，这段代码输出从 0~99 的整数：

```
for (int i = 0; i < 100; i=i+1)      // This is equivalent to
                                     // For i = 0 To 99 in VB.
{
   Console.WriteLine(i);
}
```

这里声明了一个 int 类型的变量 i，并把它初始化为 0，用作循环计数器。接着测试它是否小于 100。因为这个条件等于 true，所以执行循环中的代码，显示值 0。然后给该计数器加 1，再次执行该过程。当 i 等于 100 时，循环停止。

实际上，上述编写循环的方式并不常用。C#在给变量加 1 时有一种简化方式，即不使用 i = i+1，而简写为 i++：

```
for (int i = 0; i < 100; i++)
{
   // etc.
      }
```

也可以在上面的例子中给循环变量 i 使用类型推断功能。使用类型推断功能时，循环结构变成：

```
for (var i = 0; i < 100; i++)
..
```

嵌套的 for 循环非常常见，在每次迭代外部的循环时，内部循环都要彻底执行完毕。这种模式通常用于在矩形多维数组中遍历每个元素。最外部的循环遍历每一行，内部的循环遍历某行上的每个列。下面的代码显示数字行，它还使用另一个 Console 方法 Console.Write()，该方法的作用与 Console.WriteLine()相同，但不在输出中添加回车换行符：

```
using System;
namespace Wrox
{
  class MainEntryPoint
  {
    static void Main(string[] args)
    {
      // This loop iterates through rows
      for (int i = 0; i < 100; i+=10)
```

```
        {
          // This loop iterates through columns
          for (int j = i; j < i + 10; j++)
          {
             Console.Write(" " + j);
          }
          Console.WriteLine();
        }
      }
    }
  }
```

尽管 j 是一个整数，但它会自动转换为字符串，以便进行连接。

上述例子的结果是：

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

尽管在技术上，可以在 for 循环的测试条件中计算其他变量，而不计算计数器变量，但这不太常见。也可以在 for 循环中忽略一个表达式(甚至所有表达式)。但此时，要考虑使用 while 循环。

2. while 循环

与 for 循环一样，while 也是一个预测试循环。其语法是类似的，但 while 循环只有一个表达式：

```
while(condition)
   statement(s);
```

与 for 循环不同的是，while 循环最常用于以下情况：在循环开始前，不知道重复执行一个语句或语句块的次数。通常，在某次迭代中，while 循环体中的语句把布尔标志设置为 false，结束循环，如下面的例子所示。

```
bool condition = false;
while (!condition)
{
  // This loop spins until the condition is true.
  DoSomeWork();
  condition = CheckCondition(); // assume CheckCondition() returns a bool
}
```

3. do...while 循环

do...while 循环是 while 循环的后测试版本。该循环的测试条件要在执行完循环体之后执行。因此 do...while 循环适用于至少要将循环体执行一次的情况：

```
bool condition;
do
{
    // This loop will at least execute once, even if Condition is false.
    MustBeCalledAtLeastOnce();
    condition = CheckCondition();
} while (condition);
```

4. foreach 循环

foreach 循环可以迭代集合中的每一项。现在不必考虑集合的概念，第 10 章将详细介绍集合。知道集合是一种包含一系列对象的对象即可。从技术上看，要使用集合对象，就必须支持 IEnumerable 接口。集合的例子有 C#数组、System.Collection 名称空间中的集合类，以及用户定义的集合类。从下面的代码中可以了解 foreach 循环的语法，其中假定 arrayOfInts 是一个整型数组：

```
foreach (int temp in arrayOfInts)
{
   Console.WriteLine(temp);
}
```

其中，foreach 循环每次迭代数组中的一个元素。它把每个元素的值放在 int 型的变量 temp 中，然后执行一次循环迭代。

这里也可以使用类型推断功能。此时，foreach 循环变成：

```
foreach (var temp in arrayOfInts)
..
```

temp 的类型推断为 int，因为这是集合项的类型。

注意，foreach 循环不能改变集合中各项(上面的 temp)的值，所以下面的代码不会编译：

```
foreach (int temp in arrayOfInts)
{
  temp++;
  Console.WriteLine(temp);
}
```

如果需要迭代集合中的各项，并改变它们的值，就应使用 for 循环。

2.5.3 跳转语句

C#提供了许多可以立即跳转到程序中另一行代码的语句，在此，先介绍 goto 语句。

1. goto 语句

goto 语句可以直接跳转到程序中用标签指定的另一行(标签是一个标识符，后跟一个冒号)：

```
goto Label1;
   Console.WriteLine("This won't be executed");
Label1:
   Console.WriteLine("Continuing execution from here");
```

goto 语句有两个限制。不能跳转到像 for 循环这样的代码块中，也不能跳出类的范围，不能退

出 try...catch 块后面的 finally 块(第 16 章将介绍如何用 try...catch...finally 块处理异常)。

goto 语句的名声不太好，在大多数情况下不允许使用它。一般情况下，使用它肯定不是面向对象编程的好方式。

2. break 语句

前面简要提到过 break 语句——在 switch 语句中使用它退出某个 case 语句。实际上，break 也可以用于退出 for、foreach、while 或 do...while 循环，该语句会使控制流执行循环后面的语句。

如果该语句放在嵌套的循环中，就执行最内部循环后面的语句。如果 break 放在 switch 语句或循环外部，就会产生编译错误。

3. continue 语句

continue 语句类似于 break，也必须在 for、foreach、while 或 do...while 循环中使用。但它只退出循环的当前迭代，开始执行循环的下一次迭代，而不是退出循环。

4. return 语句

return 语句用于退出类的方法，把控制权返回方法的调用者。如果方法有返回类型，return 语句必须返回这个类型的值；如果方法返回 void，应使用没有表达式的 return 语句。

2.6 枚举

枚举是用户定义的整数类型。在声明一个枚举时，要指定该枚举的实例可以包含的一组可接受的值。不仅如此，还可以给值指定易于记忆的名称。如果在代码的某个地方，要试图把一个不在可接受范围内的值赋予枚举的一个实例，编译器就会报告一个错误。

从长远来看，创建枚举可以节省大量时间，减少许多麻烦。使用枚举比使用无格式的整数至少有如下 3 个优势：

- 如上所述，枚举可以使代码更易于维护，有助于确保给变量指定合法的、期望的值。
- 枚举使代码更清晰，允许用描述性的名称表示整数值，而不是用含义模糊、变化多端的数来表示。
- 枚举也使代码更易于输入。在给枚举类型的实例赋值时，Visual Studio .NET IDE会通过IntelliSense弹出一个包含可接受值的列表框，减少了按键次数，并能够让我们回忆起可选的值。

可以定义如下的枚举：

```
public enum TimeOfDay
{
  Morning = 0,
  Afternoon = 1,
  Evening = 2
}
```

本例在枚举中使用一个整数值，来表示一天的每个阶段。现在可以把这些值作为枚举的成员来访问。例如，TimeOfDay.Morning 返回数字 0。使用这个枚举一般是把合适的值传送给方法，并在

switch 语句中迭代可能的值。

```
class EnumExample
{
  public static int Main()
  {
    WriteGreeting(TimeOfDay.Morning);
    return 0;
  }
  static void WriteGreeting(TimeOfDay timeOfDay)
  {
    switch(timeOfDay)
    {
      case TimeOfDay.Morning:
        Console.WriteLine("Good morning!");
        break;
      case TimeOfDay.Afternoon:
        Console.WriteLine("Good afternoon!");
        break;
      case TimeOfDay.Evening:
        Console.WriteLine("Good evening!");
        break;
      default:
        Console.WriteLine("Hello!");
        break;
    }
  }
}
```

在 C#中，枚举的真正强大之处是它们在后台会实例化为派生于基类 System.Enum 的结构。这表示可以对它们调用方法，执行有用的任务。注意因为.NET Framework 的执行方式，在语法上把枚举当作结构不会造成性能损失。实际上，一旦代码编译好，枚举就成为基本类型，与 int 和 float 类似。

可以获取枚举的字符串表示，例如使用前面的 TimeOfDay 枚举：

```
TimeOfDay time = TimeOfDay.Afternoon;
Console.WriteLine(time.ToString());
```

会返回字符串 Afternoon。

另外，还可以从字符串中获取枚举值：

```
TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);
```

这段代码说明了如何从字符串获取枚举值，并转换为整数。要从字符串中转换，需要使用静态的 Enum.Parse()方法，这个方法带 3 个参数。第 1 个参数是要使用的枚举类型，其语法是关键字 typeof 后跟放在括号中的枚举类名。typeof 运算符将在第 7 章详细论述。第 2 个参数是要转换的字符串，第 3 个参数是一个 bool，指定在进行转换时是否忽略大小写。最后，注意 Enum.Parse()方法实际上返回一个对象引用——我们需要把这个字符串显式转换为需要的枚举类型(这是一个拆箱操作的例子)。对于上面的代码，将返回 1，作为一个对象，对应于 TimeOfDay.Afternoon 的枚举值。在显式转换为 int 时，会再次生成 1。

System.Enum 上的其他方法可以返回枚举定义中的值的个数或列出值的名称等。详细信息参见

MSDN 文档。

2.7　名称空间

如前所述，名称空间提供了一种组织相关类和其他类型的方式。与文件或组件不同，名称空间是一种逻辑组合，而不是物理组合。在 C#文件中定义类时，可以把它包括在名称空间定义中。以后，在定义另一个类(在另一个文件中执行相关操作)时，就可以在同一个名称空间中包含它，创建一个逻辑组合，该组合告诉使用类的其他开发人员：这两个类是如何相关的以及如何使用它们：

```
namespace CustomerPhoneBookApp
{
  using System;
  public struct Subscriber
  {
     // Code for struct here..
  }
}
```

把一个类型放在名称空间中，可以有效地给这个类型指定一个较长的名称，该名称包括类型的名称空间，名称之间用句点(.)隔开，最后是类名。在上面的例子中，Subscriber 结构的全名是 CustomerPhoneBookApp.Subscriber。这样，有相同短名的不同类就可以在同一个程序中使用了。全名常常称为完全限定的名称。

也可以在名称空间中嵌套其他名称空间，为类型创建层次结构：

```
namespace Wrox
{
  namespace ProCSharp
  {
    namespace Basics
    {
      class NamespaceExample
      {
        // Code for the class here..
      }
    }
  }
}
```

每个名称空间名都由它所在名称空间的名称组成，这些名称用句点分隔开，开头是最外层的名称空间，最后是它自己的短名。所以 ProCSharp 名称空间的全名是 Wrox.ProCSharp，NamespaceExample 类的全名是 Wrox.ProCSharp.Basics.NamespaceExample。

使用这个语法也可以组织自己的名称空间定义中的名称空间，所以上面的代码也可以写为：

```
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
```

```
    {
      // Code for the class here..
    }
}
```

注意不允许声明嵌套在另一个名称空间中的多部分的名称空间。

名称空间与程序集无关。同一个程序集中可以有不同的名称空间，也可以在不同的程序集中定义同一个名称空间中的类型。

应在开始一个项目之前就计划定义名称空间的层次结构。一般可接受的格式是CompanyName.ProjectName.SystemSection。所以在上面的例子中，Wrox 是公司名，ProCSharp 是项目，对于本章，Basics 是部分名。

2.7.1 using 语句

显然，名称空间相当长，输入起来很繁琐，用这种方式指定某个类也不总是必要的。如本章开头所述，C#允许简写类的全名。为此，要在文件的顶部列出类的名称空间，前面加上 using 关键字。在文件的其他地方，就可以使用其类型名称来引用名称空间中的类型了：

```
using System;
using Wrox.ProCSharp;
```

如前所述，几乎所有的 C#源代码都以语句 using System;开头，这仅是因为 Microsoft 提供的许多有用的类都包含在 System 名称空间中。

如果 using 语句引用的两个名称空间包含同名的类型，就必须使用完整的名称(或者至少较长的名称)，确保编译器知道访问哪个类型。例如，假如类 NamespaceExample 同时存在于Wrox.ProCSharp.Basics 和 Wrox.ProCSharp.OOP 名称空间中。如果要在名称空间 Wrox.ProCSharp 中创建一个类 Test，并在该类中实例化一个 NamespaceExample 类，就需要指定使用哪个类：

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;
namespace Wrox.ProCSharp
{
  class Test
  {
    public static int Main()
    {
      Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
      // do something with the nSEx variable.
      return 0;
    }
  }
}
```

因为 using 语句在 C#文件的开头，而 C 和 C++也把#include 语句放在这里，所以从 C++迁移到 C#的程序员常把名称空间与 C++风格的头文件相混淆。不要犯这种错误，using 语句在这些文件之间并没有建立物理链接。C#也没有对应于 C++头文件的部分。

公司应花一些时间开发一种名称空间模式，这样其开发人员才能快速定位他们需要的功能，而且公司内部使用的类名也不会与现有的类库相冲突。本章后面将介绍建立名称空间模式的规则和其他命名约定。

2.7.2 名称空间的别名

using 关键字的另一个用途是给类和名称空间指定别名。如果名称空间的名称非常长，又要在代码中多次引用，但不希望该名称空间的名称包含在 using 指令中(例如，避免类名冲突)，就可以给该名称空间指定一个别名，其语法如下：

```
using alias = NamespaceName;
```

下面的例子(前面例子的修订版本)给 Wrox.ProCSharp.Basics 名称空间指定别名 Introduction，并使用这个别名实例化了一个 NamespaceExample 对象，这个对象是在该名称空间中定义的。注意名称空间别名的修饰符是“::”。因此将先从 Introduction 名称空间别名开始搜索。如果在相同的作用域中引入了一个 Introduction 类，就会发生冲突。即使出现了冲突，“::”运算符也允许引用别名。NamespaceExample 类有一个方法 GetNamespace()，该方法调用每个类都有的 GetType()方法，以访问表示类的类型的 Type 对象。下面使用这个对象来返回类的名称空间名：

```
using System;
using Introduction = Wrox.ProCSharp.Basics;
class Test
{
  public static int Main()
  {
    Introduction::NamespaceExample NSEx =
       new Introduction::NamespaceExample();
    Console.WriteLine(NSEx.GetNamespace());
    return 0;
  }
}
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    public string GetNamespace()
    {
       return this.GetType().Namespace;
    }
  }
}
```

2.8 Main()方法

本章的开头提到过，C#程序是从方法 Main()开始执行的。这个方法必须是类或结构的静态方法，并且其返回类型必须是 int 或 void。

虽然显式指定public修饰符是很常见的，因为按照定义，必须在程序外部调用该方法，但我们给该入口点方法指定什么访问级别并不重要，即使把该方法标记为private，它也可以运行。

2.8.1　多个Main()方法

在编译C#控制台或Windows应用程序时，默认情况下，编译器会在类中查找与上述签名匹配的Main()方法，并使这个类方法成为程序的入口点。如果有多个Main()方法，编译器就会返回一个错误消息。例如，考虑下面的代码DoubleMain.cs：

```
using System;
namespace Wrox
{
  class Client
  {
    public static int Main()
    {
      MathExample.Main();
      return 0;
    }
  }
  class MathExample
  {
    static int Add(int x, int y)
    {
       return x + y;
    }
    public static int Main()
    {
       int i = Add(5,10);
       Console.WriteLine(i);
       return 0;
    }
  }
}
```

上述代码包含两个类，它们都有一个Main()方法。如果按照通常的方式编译这段代码，就会得到下述错误：

```
csc DoubleMain.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.
DoubleMain.cs(7,25): error CS0017: Program
        'DoubleMain.exe' has more than one entry point defined:
        'Wrox.Client.Main()'. Compile with /main to specify the type that
        contains the entry point.
DoubleMain.cs(21,25): error CS0017: Program
        'DoubleMain.exe' has more than one entry point defined:
        'Wrox.MathExample.Main()'. Compile with /main to specify the type that
```

```
        contains the entry point.
```

但是，可以使用/main选项，其后跟Main()方法所属类的全名(包括名称空间)，明确告诉编译器把哪个方法作为程序的入口点：

```
csc DoubleMain.cs /main:Wrox.MathExample
```

2.8.2 给Main()方法传递参数

前面的例子只介绍了不带参数的Main()方法。但在调用程序时，可以让CLR包含一个参数，将命令行参数传递给程序。这个参数是一个字符串数组，传统上称为args(但C#可以接受任何名称)。在启动程序时，程序可以使用这个数组，访问通过命令行传送过来的选项。

下面的例子ArgsExample.cs是在传送给Main()方法的字符串数组中循环，并把每个选项的值写入控制台窗口：

```
using System;
namespace Wrox
{
  class ArgsExample
  {
    public static int Main(string[] args)
    {
      for (int i = 0; i < args.Length; i++)
      {
         Console.WriteLine(args[i]);
      }
      return 0;
    }
  }
}
```

使用命令行就可以编译这段代码。在运行编译好的可执行文件时，可以在程序名的后面加上参数，例如：

```
ArgsExample /a /b /c
/a
/b
/c
```

2.9 有关编译C#文件的更多内容

前面介绍了如何使用csc.exe编译控制台应用程序，但其他类型的应用程序如何编译？如果要引用一个类库，该怎么办？MSDN文档详细介绍了C#编译器的所有编译选项，这里只介绍其中最重要的选项。

要回答第一个问题，应使用/target选项(常简写为/t)来指定要创建的文件类型。文件类型可以是表2-8所示的类型中的一种。

表2-8

选　项	输　出
/t:exe	控制台应用程序 (默认)
/t:library	带有清单的类库
/t:module	没有清单的组件
/t:winexe	Windows 应用程序 (没有控制台窗口)

如果想得到一个可由.NET 运行库加载的非可执行文件(如 DLL)，就必须把它编译为一个库。如果把 C#文件编译为一个模块，就不会创建任何程序集。虽然模块不能由运行库加载，但可以使用/addmodule 选项编译到另一个清单中。

另一个需要注意的选项是/out，该选项可以指定由编译器生成的输出文件名。如果没有指定/out 选项，编译器就会使用输入的 C#文件名，加上目标类型的扩展名来确定输出文件名(如.exe 表示 Windows 或控制台应用程序，.dll 表示类库)。注意/out 和/t(或/target)选项必须放在要编译的文件名前面。

默认状态下，如果在未引用的程序集中引用类型，可以将/reference 或/r 选项与程序集的路径和文件名一起使用。下面的例子说明了如何编译类库，并在另一个程序集中引用这个库。它包含两个文件：

- 类库
- 控制台应用程序，该应用程序调用库中的一个类

第一个文件 MathLibrary.cs 包含 DLL 的代码，为了简单起见，它只包含一个公共类 MathLib 和一个方法，该方法把两个 int 类型的数据加在一起：

```
namespace Wrox
{
  public class MathLib
  {
    public int Add(int x, int y)
    {
        return x + y;
    }
  }
}
```

使用下述命令把这个 C#文件编译为. NET DLL：

```
csc /t:library MathLibrary.cs
```

控制台应用程序 MathClient.cs 将简单地实例化这个对象，调用其 Add()方法，在控制台窗口中显示结果：

```
using System;
namespace Wrox
{
```

```
    class Client
    {
      public static void Main()
      {
        MathLib mathObj = new MathLib();
        Console.WriteLine(mathObj.Add(7,8));
      }
    }
}
```

使用/r 选项编译这个文件，使之指向新编译的 DLL：

```
csc MathClient.cs /r:MathLibrary.dll
```

当然，下面就可以像往常一样运行它了：在命令提示符下输入 MathClient，其结果是显示数字 15——加运算的结果。

2.10 控制台 I/O

现在，读者应基本熟悉了 C#的数据类型以及控制线程如何执行操作这些数据类型的程序。本章还要使用 Console 类的几个静态方法来读写数据，这些方法在编写基本的 C#程序时非常有效，下面就详细介绍它们。

要从控制台窗口中读取一行文本，可以使用 Console.ReadLine()方法，它会从控制台窗口中读取一个输入流(在用户按回车键时停止)，并返回输入的字符串。写入控制台也有两个对应的方法，前面已经使用过它们：

- Console.Write()方法将指定的值写入控制台窗口。
- Console.WriteLine()方法将指定的值写入控制台窗口，但在输出结果的最后添加一个换行符。

所有预定义类型(包括 object)都有这些方法的各种形式(重载)，所以在大多数情况下，在显示值之前不必把它们转换为字符串。

例如，下面的代码允许用户输入一行文本，并显示该文本：

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

Console.WriteLine()还允许用与 C 的 printf()函数类似的方式显示格式化的输出结果。要以这种方式使用 WriteLine()，应传入许多参数。第一个参数是花括号中包含标记的字符串，在这个花括号中，要把后续参数插入到文本中。每个标记都包含一个基于 0 的索引，表示列表中参数的序号。例如，{0}表示列表中的第一个参数，所以下面的代码：

```
int i = 10;
int j = 20;
Console.WriteLine("{0} plus {1} equals {2}", i, j, i + j);
```

会显示：

```
10 plus 20 equals 30
```

也可以为值指定宽度，调整文本在该宽度中的位置，正值表示右对齐，负值表示左对齐。为此可以使用格式{n,w}，其中 n 是参数索引，w 是宽度值。

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n —— \n {2,4}", i, j, i + j);
```

结果如下：

```
  940
+  73
--
 1013
```

最后，还可以添加一个格式字符串以及一个可选的精度值。这里没有列出格式字符串的完整列表，因为如第 9 章所述，我们可以定义自己的格式字符串。但用于预定义类型的主要格式字符串如表 2-9 所示。

表 2-9

字 符 串	说　　明
C	本地货币格式
D	十进制格式，把整数转换为以 10 为基数的数，如果给定一个精度说明符，就加上前导 0
E	科学计数法(指数)格式。精度说明符设置小数位数(默认为 6)。格式字符串的大小写(e 或 E)确定指数符号的大小写
F	固定点格式，精度说明符设置小数位数，可以为 0
G	普通格式，使用 E 或 F 格式取决于哪种格式较简单
N	数字格式，用逗号表示千分符，例如 32 767.44
P	百分数格式
X	十六进制格式，精度说明符用于加上前导 0

注意除 e/E 之外，格式字符串都不需要考虑大小写。

如果要使用格式字符串，应把它放在给出参数个数和字段宽度的标记后面，并用一个冒号把它们分隔开。例如，要把 decimal 值格式化为货币格式，且使用计算机上的地区设置，其精度为两位小数，则使用 C2：

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n —— -\n {2,9:C2}", i, j, i + j);
```

在美国，其结果是：

```
   $940.23
+   $73.70
—— -
 $1,013.93
```

最后一个技巧是，可以使用占位符来代替这些格式字符串，例如：

```
double d = 0.234;
Console.WriteLine("{0:#.00}", d);
```

其结果为.23，因为如果在符号(#)的位置上没有字符，就会忽略该符号(#)，如果在 0 的位置上有一个字符，就用这个字符代替 0，否则就显示 0。

2.11　使用注释

本节的内容是给代码添加注释，该主题表面看来十分简单，但实际可能很复杂。注释有助于阅读代码的其他开发人员理解代码，而且可以用来为开发人员生成代码的文档。

2.11.1　源文件中的内部注释

本章开头提到过，C#使用传统的 C 风格注释方式：单行注释使用// ...，多行注释使用/* ... */：

```
// This is a single-line comment
/* This comment
   spans multiple lines. */
```

单行注释中的任何内容，即从//开始一直到行尾的内容都会被编译器忽略。多行注释中“/*”和“*/”之间的所有内容也会被忽略。显然不能在多行注释中包含“*/”组合，因为这会被当作注释的结尾。

实际上，可以把多行注释放在一行代码中：

```
Console.WriteLine(/* Here's a comment! */ "This will compile.");
```

像这样的内联注释在使用时应小心，因为它们会使代码难以理解。但这样的注释在调试时是非常有用的，例如，在运行代码时要临时使用另一个值：

```
DoSomething(Width, /*Height*/ 100);
```

当然，字符串字面值中的注释字符会按照一般的字符来处理：

```
string s = "/* This is just a normal string .*/";
```

2.11.2　XML 文档

如前所述，除了 C 风格的注释外，C#还有一个非常出色的功能，本章将讨论这一功能：根据特定的注释自动创建 XML 格式的文档说明。这些注释都是单行注释，但都以 3 条斜杠(///)开头，而不是通常的两条斜杠。在这些注释中，可以把包含类型和类型成员的文档说明的 XML 标记放在代码中。

编译器可以识别表 2-10 所示的标记。

表 2-10

标　记	说　明
<c>	把行中的文本标记为代码，例如<c>int i = 10;</c>
<code>	把多行标记为代码
<example>	标记为一个代码示例

(续表)

标　记	说　明
<exception>	说明一个异常类(编译器要验证其语法)
<include>	包含其他文档说明文件的注释(编译器要验证其语法)
<list>	把列表插入文档中
<para>	建立文档的结构
<param>	标记方法的参数(编译器要验证其语法)
<paramref>	表示一个单词是方法的参数(编译器要验证其语法)
<permission>	说明对成员的访问(编译器要验证其语法)
<remarks>	给成员添加描述
<returns>	说明方法的返回值
<see>	提供对另一个参数的交叉引用(编译器要验证其语法)
<seealso>	提供描述中的“参见”部分(编译器要验证其语法)
<summary>	提供类型或成员的简短小结
<typeparam>	用在泛型类型的注释中以说明一个类型参数
<typepararef>	类型参数的名称
<value>	描述属性

要了解它们的工作方式，可以在上一节的 MathLibrary.cs 文件中添加一些 XML 注释。我们给类及其 Add()方法添加一个<summary>元素，也给 Add()方法添加一个<returns>元素和两个<param>元素：

```
// MathLib.cs
namespace Wrox
{
  ///<summary>
  /// Wrox.Math class.
  /// Provides a method to add two integers.
  ///</summary>
  public class MathLib
  {
    ///<summary>
    /// The Add method allows us to add two integers.
    ///</summary>
    ///<returns>Result of the addition (int)</returns>
    ///<param name="x">First number to add</param>
    ///<param name="y">Second number to add</param>
    public int Add(int x, int y)
    {
       return x + y;
    }
  }
}
```

C#编译器可以把 XML 元素从特定的注释中提取出来，并使用它们生成一个 XML 文件。要让编译器为程序集生成 XML 文档，需在编译时指定/doc 选项，后跟要创建的文件名：

```
csc /t:library /doc:MathLibrary.xml MathLibrary.cs
```

如果XML注释没有生成格式正确的XML文档，编译器就生成一个错误。

上面的代码会生成一个XML文件Math.xml，如下所示。

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>MathLibrary</name>
  </assembly>
  <members>
    <member name="T:Wrox.MathLibrary">
        <summary>
        Wrox.MathLibrary class.
        Provides a method to add two integers.
     </summary>
   </member>
   <member name=
       "M:Wrox.MathLibrary.Add(System.Int32,System.Int32)">
     <summary>
       The Add method allows us to add two integers.
     </summary>
     <returns>Result of the addition (int)</returns>
     <param name="x">First number to add</param>
     <param name="y">Second number to add</param>
   </member>
  </members>
</doc>
```

注意，编译器自行完成了一些工作——它创建了一个<assembly>元素，并为该文件中的每个类型或类型成员添加一个<member>元素。每个<member>元素都有一个 name 特性，该特性的值是成员的全名，前面有一个字母，含义如下："T:"表示一个类型，"F:"表示一个字段，"M:"表示一个成员。

2.12 C#预处理器指令

除了前面介绍的常用关键字外，C#还有许多名为“预处理器指令”的命令。这些命令从来不会转化为可执行代码中的命令，但会影响编译过程的各个方面。例如，使用预处理器指令可以禁止编译器编译代码的某一部分。如果计划发布两个版本的代码，即基本版本和拥有更多功能的企业版本，就可以使用这些预处理器指令。在编译软件的基本版本时，使用预处理器指令可以禁止编译器编译与额外功能相关的代码。另外，在编写提供调试信息的代码时，也可以使用预处理器指令。实际上，在销售软件时，一般不希望编译这部分代码。

预处理器指令的开头都有符号#。

C++开发人员应知道，在C和C++中预处理器指令非常重要，但是，在C#中，并没有那么多的预处理器指令，它们的使用也不太频繁。C#提供了其他机制来实现许多C++指令的功能，如定制特性。还要注意，C#并没有一个像C++那样的独立预处理器，所谓的预处理器指令实际上是由编译器处理的。尽管如此，C#仍保留了一些预处理器指令名称，因为这些命令会让人觉得就是预处理器。

下面简要介绍预处理器指令的功能。

2.12.1 #define 和#undef

#define 的用法如下所示：

```
#define DEBUG
```

它告诉编译器存在给定名称的符号，在本例中是 DEBUG。这有点类似于声明一个变量，但这个变量并没有真正的值，只是存在而已。这个符号不是实际代码的一部分，而只在编译器编译代码时存在。在 C#代码中它没有任何意义。

#undef 正好相反——它删除符号的定义：

```
#undef DEBUG
```

如果符号不存在，#undef 就没有任何作用。同样，如果符号已经存在，则#define 也不起作用。

必须把#define 和#undef 命令放在 C#源文件的开头位置，在声明要编译的任何对象的代码之前。

#define 本身并没有什么用，但与其他预处理器指令(特别是#if)结合使用时，它的功能就非常强大了。

这里应注意一般 C#语法的一些变化。预处理器指令不用分号结束，一般一行上只有一条命令。这是因为对于预处理器指令，C#不再要求命令使用分号进行分隔。如果它遇到一条预处理器指令，就会假定下一条命令在下一行上。

2.12.2 #if、#elif、#else 和#endif

这些指令告诉编译器是否要编译某个代码块。考虑下面的方法：

```
int DoSomeWork(double x)
{
  // do something
  #if DEBUG
     Console.WriteLine("x is " + x);
  #endif
}
```

这段代码会像往常那样编译，但 Console.WriteLine 命令包含在#if 子句内。这行代码只有在前面的#define 命令定义了符号 DEBUG 后才执行。当编译器遇到#if 语句后，将先检查相关的符号是否存在，如果符号存在，就编译#if 子句中的代码。否则，编译器会忽略所有的代码，直到遇到匹配的#endif 指令为止。一般是在调试时定义符号 DEBUG，把与调试相关的代码放在#if 子句中。在完成了调试后，就把#define 语句注释掉，所有的调试代码会奇迹般地消失，可执行文件也会变小，最终用户不会被这些调试信息弄糊涂(显然，要做更多的测试，确保代码在没有定义 DEBUG 的情况下也能工作)。这项技术在 C 和 C++编程中十分常见，称为条件编译(conditional compilation)。

#elif (=else if)和#else 指令可以用在#if 块中，其含义非常直观。也可以嵌套#if 块：

```
#define ENTERPRISE
#define W2K
```

```
// further on in the file
#if ENTERPRISE
    // do something
    #if W2K
        // some code that is only relevant to enterprise
        // edition running on W2K
  #endif
#elif PROFESSIONAL
   // do something else
#else
   // code for the leaner version
#endif
```

与 C++中的情况不同，使用#if 不是有条件地编译代码的唯一方式，C#还通过 Conditional 特性提供了另一种机制，详见第 15 章。

#if 和#elif 还支持一组逻辑运算符“!”、“==”、“!=”和“||”。如果符号存在，就被认为是 true，否则为 false，例如：

```
#if W2K && (ENTERPRISE==false) // if W2K is defined but ENTERPRISE isn't
```

2.12.3 #warning 和#error

另两个非常有用的预处理器指令是#warning 和#error，当编译器遇到它们时，会分别产生警告或错误。如果编译器遇到#warning 指令，会给用户显示#warning 指令后面的文本，之后编译继续进行。如果编译器遇到#error 指令，就会给用户显示后面的文本，作为一条编译错误消息，然后会立即退出编译，不会生成 IL 代码。

使用这两条指令可以检查#define 语句是不是做错了什么事，使用#warning 语句可以提醒自己执行某个操作：

```
#if DEBUG && RELEASE
   #error "You've defined DEBUG and RELEASE simultaneously!"
#endif
#warning "Don't forget to remove this line before the boss tests the code!"
   Console.WriteLine("*I hate this job.*");
```

2.12.4 #region 和#endregion

#region 和#endregion 指令用于把一段代码标记为有给定名称的一个块，如下所示。

```
#region Member Field Declarations
  int x;
  double d;
  Currency balance;
#endregion
```

这看起来似乎没有什么用，它不影响编译过程。这些指令的优点是它们可以被某些编辑器识别，包括 Visual Studio .NET 编辑器。这些编辑器可以使用这些指令使代码在屏幕上更好地布局。第 17 章会详细介绍它们。

2.12.5 #line

#line 指令可以用于改变编译器在警告和错误信息中显示的文件名和行号信息。这条指令用得并不多。如果编写代码时，在把代码发送给编译器前，要使用某些软件包改变输入的代码，就可以使用这个指令，因为这意味着编译器报告的行号或文件名与文件中的行号或编辑的文件名不匹配。#line 指令可以用于还原这种匹配。也可以使用语法#line default 把行号还原为默认的行号：

```
#line 164 "Core.cs"  // We happen to know this is line 164 in the file
                     // Core.cs, before the intermediate
                     // package mangles it.
// later on
#line default     // restores default line numbering
```

2.12.6 #pragma

#pragma 指令可以抑制或还原指定的编译警告。与命令行选项不同，#pragma 指令可以在类或方法级别执行，对抑制警告的内容和抑制的时间进行更精细的控制。下面的例子禁止“字段未使用”警告，然后在编译 MyClass 类后还原该警告。

```
#pragma warning disable 169
public class MyClass
{
   int neverUsedField;
}
#pragma warning restore 169
```

2.13 C#编程规则

本节介绍编写 C#程序时应该遵循的准则。大多数 C#开发人员都遵守这些规则，所以在这些规则的指导下编写程序可以方便其他开发人员使用程序的代码。

2.13.1 关于标识符的规则

本节将讨论变量、类、方法等的命名规则。注意本节所介绍的规则不仅是准则，也是 C#编译器强制使用的。

标识符是给变量、用户定义的类型(如类和结构)和这些类型的成员指定的名称。标识符区分大小写，所以 interestRate 和 InterestRate 是不同的变量。确定在 C#中可以使用什么标识符有两条规则：

- 尽管可以包含数字字符，但它们必须以字母或下划线开头。
- 不能把 C#关键字用作标识符。

C#包含如表 2-11 所示的保留关键字。

表 2-11

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	void
delegate	internal	short	volatile
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

如果需要把某一保留字用作标识符(例如，访问一个用另一种语言编写的类)，那么可以在标识符的前面加上前缀符号@，告知编译器其后的内容是一个标识符，而不是C#关键字(所以abstract不是有效的标识符，@abstract才是)。

最后，标识符也可以包含Unicode字符，用语法\uXXXX来指定，其中XXXX是Unicode字符的4位十六进制编码。下面是有效标识符的一些例子：

- Name
- überfluß
- _Identifier
- \u005fIdentifier

最后两个标识符完全相同，可以互换(因为005f是下划线字符的Unicode代码)，所以这些标识符在同一个作用域内不要声明两次。注意虽然从语法上看，在标识符中可以使用下划线字符，但大多数情况下最好不要这么做，因为它不符合Microsoft的变量命名规则，这种命名规则可以确保开发人员使用相同的命名约定，易于阅读他人编写的代码。

2.13.2 用法约定

在任何开发语言中，通常有一些传统的编程风格。这些风格不是语言自身的一部分，而是约定，

例如，变量如何命名，类、方法或函数如何使用等。如果使用某语言的大多数开发人员都遵循相同的约定，不同的开发人员就很容易理解彼此的代码，这一般有助于程序的维护。约定主要取决于语言和环境。例如，在 Windows 平台上编程的 C++开发人员一般使用前缀 psz 或 lpsz 表示字符串：char *pszResult; char *lpszMessage;，但在 UNIX 系统上，则不使用任何前缀：char*Result; char *Message;。

从本书中的示例代码中可以总结出，C#中的约定是命名变量时不使用任何前缀：string Result; string Message;。

变量名用带有前缀字母来表示某种数据类型，这种约定称为 Hungarian 表示法。这样，其他阅读该代码的开发人员就可以立即从变量名中了解它代表什么数据类型。在有了智能编辑器和 IntelliSense 之后，人们普遍认为 Hungarian 表示法是多余的。

在许多语言中，用法约定是从语言的使用过程中逐渐演变而来的，但是 Microsoft 编写的 C#和整个.NET Framework 有非常多的用法约定，详见.NET/C# MSDN 文档。这说明，从一开始，.NET 程序就有非常高的互操作性，开发人员可以以此来理解代码。用法规则还得益于 20 年来面向对象编程的发展，因此相关的新闻组已经仔细考虑了这些用法规则，而且已经为开发团体所接受。所以我们应遵守这些约定。

但要注意，这些规则与语言规范不同。用户应尽可能遵循这些规则。但如果有很好的理由不遵循它们，也不会有什么问题。例如，不遵循这些用法约定，也不会出现编译错误。一般情况下，如果不遵循用法规则，就必须有一个充分的理由。规则应是一个正确的决策，而不是一种束缚。在阅读本书的后续内容时，应注意到在本书的许多示例中，都没有遵循该约定，这通常是因为某些规则适用于大型程序，而不适合于本书中的小示例。如果编写一个完整的软件包，就应遵循这些规则，但它们并不适合于只有 20 行代码的独立程序。在许多情况下，遵循约定会使这些示例难以理解。

编程风格的规则非常多。这里只介绍一些比较重要的规则，以及最适合于用户的规则。如果用户要让代码完全遵循用法规则，就需要参考 MSDN 文档。

1. 命名约定

使程序易于理解的一个重要方面是给对象选择命名的方式，包括变量、方法、类、枚举和名称空间的命名方式。

显然，这些名称应反映对象的功能，且不与其他名称冲突。在.NET Framework 中，一般规则也是变量名要反映变量实例的功能，而不反映数据类型。例如，height 就是一个比较好的变量名，而 integerValue 就不太好。但是，这种规则是一种理想状态，很难达到。在处理控件时，大多数情况下使用 confirmationDialog 和 chooseEmployeeListBox 等变量名比较好，这些变量名说明了变量的数据类型。

名称的约定包括以下几个方面。

(1) 名称的大小写

在许多情况下，名称都应使用 Pascal 大小写形式。 Pascal 大小写形式指名称中单词的首字母大写，如 EmployeeSalary、ConfirmationDialog、PlainTextEncoding。注意，名称空间和类，以及基

类中的成员等的名称都应遵循该规则，最好不要使用带有下划线字符的单词，即名称不应是 employee_salary。其他语言中常量的名称常常全部大写，但在 C#中最好不要这样，因为这种名称很难阅读，而应全部使用 Pascal 大小写形式的命名约定：

```
const int MaximumLength;
```

我们还推荐使用另一种大小写模式：camel 大小写形式。这种形式类似于 Pascal 大小写形式，但名称中第一个单词的首字母不大写，如 employeeSalary、confirmationDialog、plainTextEncoding。有 3 种情况可以使用 camel 大小写形式。

- 类型中所有私有成员字段的名称都应是 camel 大小写形式：

```
private int subscriberId;
```

但要注意成员字段的前缀名常常用一条下划线开头：

```
private int _subscriberId;
```

- 传递给方法的所有参数的名称都应是 camel 大小写形式：

```
public void RecordSale(string salesmanName, int quantity);
```

- camel 大小写形式也可以用于区分同名的两个对象——比较常见的情况是属性封装一个字段：

```
private string employeeName;
public string EmployeeName
{
  get
  {
     return employeeName;
  }
}
```

如果这么做，则私有成员总是使用 camel 大小写形式，而公有的或受保护的成员总是使用 Pascal 大小写形式，这样使用这段代码的其他类就只能使用 Pascal 大小写形式的名称了(除了参数名以外)。

还要注意大小写问题。C#区分大小写，所以在 C#中，仅大小写不同的名称在语法上是正确的，如上面的例子。但是，有时可能从 Visual Basic .NET 应用程序中调用程序集，而 Visual Basic .NET 不区分大小写，如果使用仅大小写不同的名称，就必须使这两个名称不能在程序集的外部访问(上例是可行的，因为仅私有变量使用了 camel 大小写形式的名称)。否则，Visual Basic .NET 中的其他代码就不能正确使用这个程序集。

(2) 名称的风格

名称的风格应保持一致。例如，如果类中的一个方法被命名为 ShowConfirmationDialog()，另一个方法就不能被命名为 ShowDialogWarning()或 WarningDialogShow()，而应是 ShowWarningDialog()。

(3) 名称空间的名称

名称空间的名称非常重要，一定要仔细考虑，以避免一个名称空间的名称与其他名称空间同名。记住，名称空间的名称是.NET 区分共享程序集中对象名的唯一方式。如果软件包的名称空间使用的名称与另一个软件包相同，而这两个软件包都安装在一台计算机上，就会出问题。因此，最好用自己的公司名创建顶级的名称空间，再嵌套技术范围较窄、用户所在小组或部门或者类所在软件包的

名称空间。Microsoft 建议使用如下的名称空间：<CompanyName>.<TechnologyName>，例如：

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

(4) 名称和关键字

名称不应与任何关键字冲突，这非常重要。实际上，如果在代码中，试图给某一项指定与C#关键字同名的名称，就会出现语法错误，因为编译器会假定该名称表示一条语句。但是，由于类可能由其他语言编写的代码访问，所以不能使用其他.NET 语言中的关键字作为对应的名称。一般来说，C++关键字类似于 C#关键字，不太可能与 C++混淆，只有 Visual C++常用的关键字以两个下划线字符开头。与 C#一样，C++关键字都是小写字母，如果要遵循公有类和成员使用 Pascal 风格的名称的约定，则在它们的名称中至少有一个字母大写，因此不会与 C++关键字冲突。另一方面，Visual Basic .NET 的问题会多一些，因为 Visual Basic .NET 的关键字要比 C#的多，而且它不区分大小写，不能依赖于 Pascal 风格的名称来区分类和成员。

表 2-12 列出了 Visual Basic .NET 中的关键字和标准函数调用，无论对 C#公有类使用什么大小写组合，这些名称都不应使用。

表 2-12

Abs	Do	Loc	RGB
Add	Double	Local	Right
AddHandler	Each	Lock	RmDir
AddressOf	Else	LOF	Rnd
Alias	ElseIf	Log	RTrim
And	Empty	Long	SaveSettings
Ansi	End	Loop	Second
AppActivate	Enum	LTrim	Seek
Append	EOF	Me	Select
As	Erase	Mid	SetAttr
Asc	Err	Minute	SetException
Assembly	Error	MIRR	Shared
Atan	Event	MkDir	Shell
Auto	Exit	Module	Short
Beep	Exp	Month	Sign
Binary	Explicit	MustInherit	Sin
BitAnd	ExternalSource	MustOverride	Single
BitNot	False	MyBase	SLN
BitOr	FileAttr	MyClass	Space
BitXor	FileCopy	Namespace	Spc
Boolean	FileDateTime	New	Split

(续表)

ByRef	FileLen	Next	Sqrt
Byte	Filter	Not	Static
ByVal	Finally	Nothing	Step
Call	Fix	NotInheritable	Stop
Case	For	NotOverridable	Str
Catch	Format	Now	StrComp
CBool	FreeFile	NPer	StrConv
CByte	Friend	NPV	Strict
CDate	Function	Null	String
CDbl	FV	Object	Structure
CDec	Get	Oct	Sub
ChDir	GetAllSettings	Off	Switch
ChDrive	GetAttr	On	SYD
Choose	GetException	Open	SyncLock
Chr	GetObject	Option	Tab
CInt	GetSetting	Optional	Tan
Class	GetType	Or	Text
Clear	GoTo	Overloads	Then
CLng	Handles	Overridable	Throw
Close	Hex	Overrides	TimeOfDay
Collection	Hour	ParamArray	Timer
Command	If	Pmt	TimeSerial
Compare	Iif	PPmt	TimeValue
Const	Implements	Preserve	To
Cos	Imports	Print	Today
CreateObject	In	Private	Trim
CShort	Inherits	Property	Try
CSng	Input	Public	TypeName
CStr	InStr	Put	TypeOf
CurDir	Int	PV	UBound
Date	Integer	QBColor	UCase
DateAdd	Interface	Raise	Unicode
DateDiff	Ipmt	RaiseEvent	Unlock
DatePart	IRR	Randomize	Until
DateSerial	Is	Rate	Val

(续表)

DateValue	IsArray	Read	Weekday
Day	IsDate	ReadOnly	While
DDB	IsDbNull	ReDim	Width
Decimal	IsNumeric	Remove	With
Declare	Item	RemoveHandler	WithEvents
Default	Kill	Rename	Write
Delegate	Lcase	Replace	WriteOnly
DeleteSetting	Left	Reset	Xor
Dim	Lib	Resume	Year

2. 属性和方法的使用

类中出现混乱的一个方面是某个特定数量是用属性还是方法来表示。这没有硬性规定，但一般情况下，如果该对象的外观像一个变量，就应使用属性来表示它(属性详见第 3 章)，即：

- 客户端代码应能读取它的值，最好不要使用只写属性，例如，应使用 SetPassword()方法，而不是 Password 只写属性。
- 读取该值不应花太长的时间。实际上，如果它是一个属性，通常表明读取过程花的时间相对较短。
- 读取该值不应有任何明显的和不希望的负面效应。设置属性的值，不应有与该属性不直接相关的负面效应。设置对话框的宽度会改变该对话框在屏幕上的外观，这是可以的，因为它与该属性相关。
- 可以按照任何顺序设置属性。尤其在设置属性时，最好不要因为还没有设置另一个相关的属性而抛出一个异常。例如，如果为了使用访问数据库的类，需要设置 ConnectionString、UserName 和 Password，应确保已经实现了该类，这样用户才能按照任何顺序设置它们。
- 顺序读取属性也应有相同的效果。如果属性的值可能会出现预料不到的改变，就应把它编写为一个方法。在监控汽车的运动的类中，把 speed 设置为属性就不合适，而应使用 GetSpeed()方法；另一方面，应把 Weight 和 EngineSize 设置为属性，因为对于给定的对象，它们是不变的。

如果要编码的相关项满足上述所有条件，就把它设置为属性，否则就应使用方法。

3. 字段的用法

字段的用法非常简单。字段应总是私有的，但在某些情况下也可以把常量或只读字段设置为公有。原因是如果把字段设置为公有，就不利于在以后扩展或修改类。

遵循上面的规则就可以培养良好的编程习惯，而且这些规则应与面向对象编程的风格一起使用。

最后要记住以下有用的备注：Microsoft 在保持一致性方面相当谨慎，在编写.NET 基类时遵循了它自己的规则。在编写.NET 代码时应很好地遵循这些规则，对于基类来说，就是要弄清楚类、成员、名称空间的命名方式和类层次结构的工作方式等。类与基类之间的一致性有助于提高可读性和

可维护性。

2.14 小结

本章介绍了一些 C#基本语法，包括编写简单的 C#程序需要掌握的内容。我们讲述了许多基础知识，但其中有许多是熟悉 C 风格语言(甚至 JavaScript)的开发人员能立即领悟的。

C#语法与 C++/Java 语法非常类似，但仍存在一些细微区别。在许多领域，将这些语法与功能结合起来会提高编码速度，如高质量的字符串处理功能。C#还有一个已定义的强类型系统，该系统基于值类型和引用类型的区别。第 3 章和第 4 章将介绍 C#的面向对象编程特性。

第 3 章

对象和类型

本章要点

- 类和结构的区别
- 类成员
- 按值和按引用传送参数
- 方法重载
- 构造函数和静态构造函数
- 只读字段
- 部分类
- 静态类
- 弱引用
- Object 类，其他类型都从该类派生而来

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- MathTest
- MathTestWeakReference
- ParameterTest

3.1 创建及使用类

到目前为止，我们介绍了组成 C#语言的主要模块，包括变量、数据类型和程序流语句，并简要介绍了一个只包含 Main()方法的完整小例子。但还没有介绍如何把这些内容组合在一起，构成一个完整的程序，其关键就在于对类的处理。这就是本章的主题。第 4 章将介绍继承以及与继承相关的特性。

本章将讨论与类相关的基本语法，但假定你已经熟悉了使用类的基本原则，例如，知道构造函数或属性的含义，因此本章主要阐述如何把这些原则应用于C#代码。

3.2 类和结构

类和结构实际上都是创建对象的模板，每个对象都包含数据，并提供了处理和访问数据的方法。类定义了类的每个对象(称为实例)可以包含什么数据和功能。例如，如果一个类表示一个顾客，就可以定义字段 CustomerID、FirstName、LastName 和 Address，以包含该顾客的信息。还可以定义处理在这些字段中存储的数据的功能。接着，就可以实例化表示某个顾客的类的对象，为这个实例设置相关字段的值，并使用其功能。

```
class PhoneCustomer
{
   public const string DayOfSendingBill = "Monday";
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

结构与类的区别是它们在内存中的存储方式、访问方式(类是存储在堆(heap)上的引用类型，而结构是存储在栈(stack)上的值类型)和它们的一些特征(如结构不支持继承)。较小的数据类型使用结构可提高性能。但在语法上，结构与类非常相似，主要的区别是使用关键字 struct 代替 class 来声明结构。例如，如果希望所有的 PhoneCustomer 实例都分布在栈上，而不是分布在托管堆上，就可以编写下面的语句：

```
struct PhoneCustomerStruct
{
   public const string DayOfSendingBill = "Monday";
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

对于类和结构，都使用关键字 new 来声明实例：这个关键字创建对象并对其进行初始化。在下面的例子中，类和结构的字段值都默认为 0：

```
PhoneCustomer myCustomer = new PhoneCustomer(); // works for a class
PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct();// works for a struct
```

在大多数情况下，类要比结构常用得多。因此，我们先讨论类，然后指出类和结构的区别，以及选择使用结构而不使用类的特殊原因。但除非特别说明，否则就可以假定用于类的代码也适用于结构。

3.3　类

类中的数据和函数称为类的成员。Microsoft 的正式术语对数据成员和函数成员进行了区分。除了这些成员外，类还可以包含嵌套的类型(如其他类)。成员的可访问性可以是 public、protected、internal protected、private 或 internal。第 5 章将详细解释各种可访问性。

3.3.1　数据成员

数据成员是包含类的数据——字段、常量和事件的成员。数据成员可以是静态数据。类成员总是实例成员，除非用 static 进行显式的声明。

字段是与类相关的变量。前面的例子已经使用了 PhoneCustomer 类中的字段。

一旦实例化 PhoneCustomer 对象，就可以使用语法 Object.FieldName 来访问这些字段，如下例所示：

```
PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = "Simon";
```

常量与类的关联方式和变量与类的关联方式相同。使用 const 关键字来声明常量。如果把它声明为 public，就可以在类的外部访问它。

```
class PhoneCustomer
{
   public const string DayOfSendingBill = "Monday";
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

事件是类的成员，在发生某些行为(如改变类的字段或属性，或者进行了某种形式的用户交互操作)时，它可以让对象通知调用方。客户可以包含所谓“事件处理程序”的代码来响应该事件。第 8 章将详细介绍事件。

3.3.2　函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(finalizer)、运算符以及索引器。

- 方法是与某个类相关的函数，与数据成员一样，函数成员默认为实例成员，使用 static 修饰符可以把方法定义为静态方法。
- 属性是可以从客户端访问的函数组，其访问方式与访问类的公共字段类似。C#为读写类中的属性提供了专用语法，所以不必使用那些名称中嵌有 Get 或 Set 的方法。因为属性的这种语法不同于一般函数的语法，在客户端代码中，虚拟的对象被当作实际的东西。
- 构造函数是在实例化对象时自动调用的特殊函数。它们必须与所属的类同名，且不能有返回类型。构造函数用于初始化字段的值。
- 终结器类似于构造函数，但是在 CLR 检测到不再需要某个对象时调用它。它们的名称与类相同，但前面有一个“~”符号。不可能预测什么时候调用终结器。第 14 章将介绍终结器。

- 运算符执行的最简单的操作就是加法和减法。在两个整数相加时，严格地说，就是对整数使用"+"运算符。C#还允许指定把已有的运算符应用于自己的类(运算符重载)。第7章将详细论述运算符。
- 索引器允许对象以数组或集合的方式进行索引。

1. 方法

注意，正式的C#术语区分函数和方法。在C#术语中，"函数成员"不仅包含方法，也包含类或结构的一些非数据成员，如索引器、运算符、构造函数和析构函数等，甚至还有属性。这些都不是数据成员，字段、常量和事件才是数据成员。

(1)方法的声明

在C#中，方法的定义包括任意方法修饰符(如方法的可访问性)、返回值的类型，然后依次是方法名、输入参数的列表(用圆括号括起来)和方法体(用花括号括起来)。

```
[modifiers] return_type MethodName([parameters])
{
   // Method body
}
```

每个参数都包括参数的类型名和在方法体中的引用名称。但如果方法有返回值，return 语句就必须与返回值一起使用，以指定出口点，例如：

```
public bool IsSquare(Rectangle rect)
{
   return (rect.Height == rect.Width);
}
```

这段代码使用了一个表示矩形的.NET 基类 System.Drawing.Rectangle。

如果方法没有返回值，就把返回类型指定为 void，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面包含一对空的圆括号()。此时 return 语句就是可选的——当到达右花括号时，方法会自动返回。注意方法可以包含任意多条 return 语句：

```
public bool IsPositive(int value)
{
   if (value < 0)
      return false;
   return true;
}
```

(2)调用方法

在下面的例子中，MathTest 说明了类的定义和实例化、方法的定义和调用的语法。除了包含 Main()方法的类之外，它还定义了类 MathTest，该类包含几个方法和一个字段。

```
using System;

namespace Wrox
{
   class MainEntryPoint
   {
```

```
        static void Main()
        {
            // Try calling some static functions.
            Console.WriteLine("Pi is " + MathTest.GetPi());
            int x = MathTest.GetSquareOf(5);
            Console.WriteLine("Square of 5 is " + x);

            // Instantiate a MathTest object
            MathTest math = new MathTest(); // this is C#'s way of
                                            // instantiating a reference type
            // Call nonstatic methods
            math.value = 30;
            Console.WriteLine(
                "Value field of math variable contains " + math.value);
            Console.WriteLine("Square of 30 is " + math.GetSquare());
        }
    }

    // Define a class named MathTest on which we will call a method
    classMathTest
    {
        publicint value;

        publicintGetSquare()
        {
            return value*value;
        }

        public static intGetSquareOf(int x)
        {
            return x*x;
        }

        public static double GetPi()
        {
            return 3.14159;
        }
    }
}
```

运行 MathTest 示例，会得到如下结果：

```
Pi is 3.14159
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900
```

从代码中可以看出，MathTest 类包含一个字段和一个方法，该字段包含一个数字，该方法计算该数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算作为参数传入的数字的平方。

这个类有一些功能并不是设计 C#程序的好例子。例如，GetPi()通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

(3) 给方法传递参数

参数可以通过引用或通过值传递给方法。在变量通过引用传递给方法时，被调用的方法得到的

就是这个变量，更准确地说，是指向内存中变量的指针。所以在方法内部对变量进行的任何改变在方法退出后仍旧有效。而如果变量通过值传送给方法，被调用的方法得到的是变量的一个相同副本，也就是说，在方法退出后，对变量进行的修改会丢失。对于复杂的数据类型，按引用传递的效率更高，因为在按值传递时，必须复制大量的数据。

在 C#中，除非特别指定，所有的引用类型都通过引用传递，所有的值类型都通过值来传递。但是，在理解引用类型的含义时需要注意。因为引用类型的变量只包含对象的引用，作为参数传递的正是这个引用，而不是对象本身，所以对底层对象的修改会保留下来。相反，值类型的变量包含的是实际数据，所以传递给方法的是数据本身的副本。例如，int 通过值传递给方法，对应方法对该 int 的值所做的任何改变都没有改变原 int 对象的值。但如果把数组或其他引用类型(如类)传递给方法，对应的方法就会使用该引用改变这个数组中的值，而新值会反射在原始数组对象上。

下面的例子 ParameterTest.cs 说明了用作参数的值类型和引用类型的区别：

```
using System;

namespace Wrox
{
   class ParameterTest
   {
      static void SomeFunction(int[] ints, int i)
      {
         ints[0] = 100;
         i = 100;
      }

      public static int Main()
      {
         int i = 0;
         int[] ints = { 0, 1, 2, 4, 8 };
         // Display the original values.
         Console.WriteLine("i = " + i);
         Console.WriteLine("ints[0] = " + ints[0]);
         Console.WriteLine("Calling SomeFunction.");

         // After this method returns, ints will be changed,
         // but i will not.
         SomeFunction(ints, i);
         Console.WriteLine("i = " + i);
         Console.WriteLine("ints[0] = " + ints[0]);
         return 0;
      }
   }
}
```

结果如下：

```
ParameterTest.exe
i = 0
ints[0] = 0
Calling SomeFunction ...
i = 0
ints[0] = 100
```

注意，i 的值保持不变，而在 ints 中改变的值在原始数组中也改变了。

注意字符串的行为方式有所不同，因为字符串是不可变的(如果改变字符串的值，就会创建一个全新的字符串)，所以字符串无法采用一般引用类型的行为方式。在方法调用中，对字符串所做的任何改变都不会影响原始字符串。这一点将在第 9 章详细讨论。

(4) ref 参数

如前所述，通过值传送变量是默认的，也可以迫使值参数通过引用传送给方法。为此，要使用 ref 关键字。如果把一个参数传递给方法，且这个方法的输入参数前带有 ref 关键字，则该方法对变量所做的任何改变都会影响原始对象的值：

```
static void SomeFunction(int[] ints, ref int i)
{
   ints[0] = 100;
   i = 100; // The change to i will persist after SomeFunction() exits.
}
```

在调用该方法时，还需要添加 ref 关键字：

```
SomeFunction(ints, ref i);
```

最后，C#仍要求对传递给方法的参数进行初始化，理解这一点也非常重要。在传递给方法之前，无论是按值传递，还是按引用传递，任何变量都必须初始化。

(5) out 参数

在 C 风格的语言中，函数常常能从一个例程中输出多个值，这使用输出参数实现，只要把输出的值赋予通过引用传递给方法的变量即可。通常，变量通过引用传递的初值并不重要，这些值会被函数重写，函数甚至从来没有使用过它们。

如果可以在 C#中使用这种约定，就会非常方便。但 C#要求变量在被引用前必须用一个初值进行初始化。尽管在把输入变量传递给函数前，可以用没有意义的值初始化它们，因为函数将使用真实、有意义的值初始化它们，但是这样做是没有必要的，有时甚至会引起混乱。但有一种方法能够简化 C#编译器所坚持的输入参数的初始化。

编译器使用 out 关键字来初始化。在方法的输入参数前面加上 out 前缀时，传递给该方法的变量可以不初始化。该变量通过引用传递，所以在从被调用的方法中返回时，对应方法对该变量进行的任何改变都会保留下来。在调用该方法时，还需要使用 out 关键字，与在定义该方法时一样：

```
static void SomeFunction(out inti)
{
   i = 100;
}

public static int Main()
{
   int i; // note how i is declared but not initialized.
   SomeFunction(out i);
   Console.WriteLine(i);
   return 0;
}
```

(6) 命名参数

参数一般需要按定义的顺序传送给方法。命名参数允许按任意顺序传递。所以下面的方法：

```
string FullName(string firstName, string lastName)
{
   return firstName + " " + lastName;
}
```

下面的方法调用会返回相同的全名：

```
FullName("John", "Doe");
FullName(lastName: "Doe", firstName: "John");
```

如果方法有几个参数，就可以在同一个调用中混合使用位置参数和命名参数。

(7) 可选参数

参数也可以是可选的。必须为可选参数提供默认值。可选参数还必须是方法定义的最后的参数。所以下面的方法声明是不正确的：

```
void TestMethod(int optionalNumber = 10, int notOptionalNumber)
{
   System.Console.Write(optionalNumber + notOptionalNumber);
}
```

要使这个方法正常工作，就必须在最后定义 optionalNumber 参数。

(8) 方法的重载

C#支持方法的重载——方法的几个版本有不同的签名(即，方法名相同，但参数的个数和/或类型不同)。为了重载方法，只需要声明同名但参数个数或类型不同的方法即可：

```
class ResultDisplayer
{
   void DisplayResult(string result)
   {
      // implementation
   }

   void DisplayResult(int result)
   {
      // implementation
   }
}
```

如果不能使用可选参数，就可以使用方法重载来达到此目的：

```
class MyClass
{
   int DoSomething(int x) // want 2nd parameter with default value 10
   {
      DoSomething(x, 10);
   }

   intDoSomething(int x, int y)
   {
      // implementation
   }
}
```

在任何语言中，对于方法重载，如果调用了错误的重载方法，就有可能出现运行错误。第 4 章

将讨论如何使代码避免这些错误。现在，知道 C#在重载方法的参数方面有一些小限制即可：

- 两个方法不能仅在返回类型上有区别。
- 两个方法不能仅根据参数是声明为 ref 还是 out 来区分。

2. 属性

属性(property)的概念是：它是一个方法或一对方法，在客户端代码看来，它(们)是一个字段。例如 Windows 窗体的 Height 属性。假定有下面的代码：

```
// mainForm is of type System.Windows.Forms
mainForm.Height = 400;
```

执行这段代码时，窗口的高度设置为 400，因此窗口会在屏幕上重新设置大小。在语法上，上面的代码类似于设置一个字段，但实际上是调用了属性访问器，它包含的代码重新设置了窗体的大小。

在 C#中定义属性，可以使用下面的语法：

```
public string SomeProperty
{
   get
   {
      return "This is the property value.";
   }
   set
   {
      // do whatever needs to be done to set the property.
   }
}
```

get 访问器不带任何参数，且必须返回属性声明的类型。也不应为 set 访问器指定任何显式参数，但编译器假定它带一个参数，其类型也与属性相同，并表示为 value。例如，下面的代码包含一个属性 Age，它设置了一个字段 age。在这个例子中，age 表示属性 Age 的后备变量。

```
private int age;

public int Age
{
   get
   {
      return age;
   }
   set
   {
      age = value;
   }
}
```

注意这里所用的命名约定。我们采用 C#的区分大小写模式，使用相同的名称，但公有属性采用 Pascal 大小写形式命名，如果存在一个等价的私有字段，则它采用 camel 大小写形式命名。一些开发人员喜欢使用把下划线作为前缀的字段名，如_age，这会为识别字段提供极大的便利。

(1) 只读和只写属性

在属性定义中省略 set 访问器，就可以创建只读属性。因此，如下代码把 Name 变成只读属性：

```
private string name;

public string Name
{
   get
   {
      return Name;
   }
}
```

同样，在属性定义中省略 get 访问器，就可以创建只写属性。但是，这是不好的编程方式，因为这可能会使客户端代码的作者感到迷惑。一般情况下，如果要这么做，最好使用一个方法替代。

(2) 属性的访问修饰符

C#允许给属性的 get 和 set 访问器设置不同的访问修饰符，所以属性可以有公有的 get 访问器和私有或受保护的 set 访问器。这有助于控制属性的设置方式或时间。在下面的代码示例中，注意 set 访问器有一个私有访问修饰符，而 get 访问器没有任何访问修饰符。这表示 get 访问器具有属性的访问级别。在 get 和 set 访问器中，必须有一个具备属性的访问级别。如果 get 访问器的访问级别是 protected，就会产生一个编译错误，因为这会使两个访问器的访问级别都不是属性。

```
public string Name
{
   get
   {
      return _name;
   }
   private set
   {
      _name = value;
   }
}
```

(3) 自动实现的属性

如果属性的 set 和 get 访问器中没有任何逻辑，就可以使用自动实现的属性。这种属性会自动实现后备成员变量。前面 Age 示例的代码如下：

```
publicint Age {get; set;}
```

不需要声明 private int Age;。编译器会自动创建它。

使用自动实现的属性，就不能在属性设置中验证属性的有效性。所以在上面的例子中，不能检查是否设置了无效的年龄。但必须有两个访问器。尝试把该属性设置为只读属性，就会出错：

```
public int Age {get;}
```

但是，每个访问器的访问级别可以不同。因此，下面的代码是合法的：

```
public int Age {get; private set;}
```

内联

一些开发人员可能会担心，前面我们列举了许多情况，其中标准C#编码方式导致了大材小用，例如，通过属性访问字段，而不是直接访问字段。这些额外的函数调用是否会增加系统开销，导致性能下降？其实，不需要担心这种编程方式会在C#中带来性能损失。C#代码会编译为IL，然后在运行时JIT编译为本地可执行代码。JIT编译器可生成高度优化的代码，并在适当的时候随意地内联代码(即，用内联代码来替代函数调用)。如果实现某个方法或属性仅是调用另一个方法，或返回一个字段，则该方法或属性肯定是内联的。但要注意，在何处内联代码完全由CLR决定。我们无法使用像C++中inline这样的关键字来控制哪些方法是内联的。

3. 构造函数

声明基本构造函数的语法就是声明一个与包含的类同名的方法，但该方法没有返回类型：

```
public class MyClass
{
   public MyClass()
   {
   }
   // rest of class definition
```

没有必要给类提供构造函数，到目前为止本书的例子中没有提供这样的构造函数。一般情况下，如果没有提供任何构造函数，编译器会在后台创建一个默认的构造函数。这是一个非常基本的构造函数，它只能把所有的成员字段初始化为标准的默认值(例如，引用类型为空引用，数值数据类型为0，bool为false)。这通常就足够了，否则就需要编写自己的构造函数。

构造函数的重载遵循与其他方法相同的规则。换言之，可以为构造函数提供任意多的重载，只要它们的签名有明显的区别即可：

```
public MyClass() // zeroparameter constructor
{
   // construction code
}
public MyClass(int number) // another overload
{
   // construction code
}
```

但是，如果提供了带参数的构造函数，编译器就不会自动提供默认的构造函数。只有在没有定义任何构造函数时，编译器才会自动提供默认的构造函数。在下面的例子中，因为定义了一个带单个参数的构造函数，编译器会假定这是可用的唯一构造函数，所以它不会隐式地提供其他构造函数：

```
public class MyNumber
{
   private int number;
   public MyNumber(int number)
   {
      this.number = number;
   }
}
```

上面的代码还说明，一般使用 this 关键字区分成员字段和同名的参数。如果试图使用无参数的构造函数实例化 MyNumber 对象，就会得到一个编译错误：

```
MyNumber numb = new MyNumber(); // causes compilation error
```

注意，可以把构造函数定义为 private 或 protected，这样不相关的类就不能访问它们：

```
public class MyNumber
{
   private int number;
   private MyNumber(int number) // another overload
   {
      this.number = number;
   }
}
```

这个例子没有为 MyNumber 定义任何公有的或受保护的构造函数。这就使 MyNumber 不能使用 new 运算符在外部代码中实例化(但可以在 MyNumber 中编写一个公有静态属性或方法，以实例化该类)。这在下面两种情况下是有用的：

- 类仅用作某些静态成员或属性的容器，因此永远不会实例化它
- 希望类仅通过调用某个静态成员函数来实例化(这就是所谓对象实例化的类工厂方法)

(1) 静态构造函数

C#的一个新特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次，而前面的构造函数是实例构造函数，只要创建类的对象，就会执行它。

```
class MyClass
{
   static MyClass()
   {
      // initialization code
   }
   // rest of class definition
}
```

编写静态构造函数的一个原因是，类有一些静态字段或属性，需要在第一次使用类之前，从外部源中初始化这些静态字段和属性。

.NET 运行库没有确保什么时候执行静态构造函数，所以不应把要求在某个特定时刻(例如，加载程序集时)执行的代码放在静态构造函数中。也不能预计不同类的静态构造函数按照什么顺序执行。但是，可以确保静态构造函数至多运行一次，即在代码引用类之前调用它。在 C#中，通常在第一次调用类的任何成员之前执行静态构造函数。

注意，静态构造函数没有访问修饰符，其他 C#代码从来不调用它，但在加载类时，总是由.NET 运行库调用它，所以像 public 或 private 这样的访问修饰符就没有任何意义。出于同样原因，静态构造函数不能带任何参数，一个类也只能有一个静态构造函数。很显然，静态构造函数只能访问类的静态成员，不能访问类的实例成员。

无参数的实例构造函数与静态构造函数可以在同一个类中同时定义。尽管参数列表相同，但这并不矛盾，因为在加载类时执行静态构造函数，而在创建实例时执行实例构造函数，所以何时执行哪个构造函数不会有冲突。

如果多个类都有静态构造函数，先执行哪个静态构造函数就不确定。此时静态构造函数中的代码不应依赖于其他静态构造函数的执行情况。另一方面，如果任何静态字段有默认值，就在调用静态构造函数之前指定它们。

下面用一个例子来说明静态构造函数的用法，该例子的思想基于包含用户首选项的程序(假定用户首选项存储在某个配置文件中)。为了简单起见，假定只有一个用户首选项——BackColor，它表示要在应用程序中使用的背景色。因为这里不想编写从外部数据源中读取数据的代码，所以假定该首选项在工作日的背景色是红色，在周末的背景色是绿色。程序仅在控制台窗口中显示首选项——但这足以说明静态构造函数是如何工作的。

```
namespace Wrox.ProCSharp.StaticConstructorSample
{
   public class UserPreferences
   {
      public static readonly Color BackColor;

      static UserPreferences()
      {
         DateTime now = DateTime.Now;
         if (now.DayOfWeek == DayOfWeek.Saturday
            || now.DayOfWeek == DayOfWeek.Sunday)
            BackColor = Color.Green;
         else
            BackColor = Color.Red;
      }

      private UserPreferences()
      {
      }
   }
}
```

这段代码说明了颜色首选项如何存储在静态变量中，该静态变量在静态构造函数中进行初始化。把这个字段声明为只读类型，这表示其值只能在构造函数中设置。本章后面将详细介绍只读字段。这段代码使用了 Microsoft 在 Framework 类库中支持的两个有用的结构 System.DateTime 和 System.Drawing.Color。DateTime 结构实现了静态属性 Now 和实例属性 DayOfWeek，Now 属性返回当前时间，DayOfWeek 属性计算出某个日期是星期几。Color 用于存储颜色，它实现了各种静态属性，如本例使用的 Red 和 Green，本例返回常用的颜色。为了使用 Color 结构，需要在编译时引用 System.Drawing.dll 程序集，且必须为 System.Drawing 名称空间添加一条 using 语句：

```
using System;
usingSystem.Drawing;
```

用下面的代码测试静态构造函数：

```
class MainEntryPoint
{
   static void Main(string[] args)
   {
      Console.WriteLine("User-preferences: BackColor is: " +
                        UserPreferences.BackColor.ToString());
   }
}
```

编译并运行这段代码，会得到如下结果：

```
User-preferences: BackColor is: Color [Red]
```

当然，如果在周末执行上述代码，颜色设置就是 Green。

(2) 从构造函数中调用其他构造函数

有时，在一个类中有几个构造函数，以容纳某些可选参数，这些构造函数包含一些共同的代码。例如，下面的情况：

```
class Car
{
   private string description;
   private uint nWheels;
   public Car(string description, uint nWheels)
   {
      this.description = description;
      this.nWheels = nWheels;
   }

   public Car(string description)
   {
      this.description = description;
      this.nWheels = 4;
   }
// etc.
```

这两个构造函数初始化了相同的字段，显然，最好把所有的代码放在一个地方。C#有一个特殊的语法，称为构造函数初始化器，可以实现此目的：

```
class Car
{
   private string description;
   private uint nWheels;

   public Car(string description, uint nWheels)
   {
      this.description = description;
      this.nWheels = nWheels;
   }

   public Car(string description): this(description, 4)
   {
   }
   // etc
```

这里，this 关键字仅调用参数最匹配的那个构造函数。注意，构造函数初始化器在构造函数的函数体之前执行。现在假定运行下面的代码：

```
Car myCar = newCar("Proton Persona");
```

在本例中，在带一个参数的构造函数的函数体执行之前，先执行带两个参数的构造函数(但在本例中，因为在带一个参数的构造函数的函数体中没有代码，所以没有区别)。

C#构造函数初始化器可以包含对同一个类的另一个构造函数的调用(使用前面介绍的语法)，也

可以包含对直接基类的构造函数的调用(使用相同的语法，但应使用 base 关键字代替 this)。初始化器中不能有多个调用。

3.3.3　只读字段

常量的概念就是一个包含不能修改的值的变量，常量是 C#与大多数编程语言共有的。但是，常量不必满足所有的要求。有时可能需要一些变量，其值不应改变，但在运行之前其值是未知的。C#为这种情形提供了另一种类型的变量：只读字段。

readonly 关键字比 const 灵活得多，允许把一个字段设置为常量，但还需要执行一些计算，以确定它的初始值。其规则是可以在构造函数中给只读字段赋值，但不能在其他地方赋值。只读字段还可以是一个实例字段，而不是静态字段，类的每个实例可以有不同的值。与 const 字段不同，如果要把只读字段设置为静态，就必须显式声明它。

如果有一个用于编辑文档的 MDI 程序，因为要注册，所以需要限制可以同时打开的文档数。现在假定要销售该软件的不同版本，而且顾客可以升级他们的版本，以便同时打开更多的文档。显然，不能在源代码中对最大文档数进行硬编码，而是需要一个字段表示这个最大文档数。这个字段必须是只读的——每次启动程序时，从注册表键或其他文件存储中读取。代码如下所示：

```
public class DocumentEditor
{
   public static readonly uint MaxDocuments;

   static DocumentEditor()
   {
      MaxDocuments = DoSomethingToFindOutMaxNumber();
   }
}
```

在本例中，字段是静态的，因为每次运行程序的实例时，只需要存储最大文档数一次。这就是在静态构造函数中初始化它的原因。如果只读字段是一个实例字段，就要在实例构造函数中初始化它。例如，假定编辑的每个文档都有一个创建日期，但不允许用户修改它(因为这会覆盖过去的日期)。注意，该字段也是公有的，我们不需要把只读字段设置为私有，因为按照定义，它们不能在外部修改(这条规则也适用于常量)。

如前所述，日期用基类 System.DateTime 表示。下面的代码使用带有 3 个参数(年份、月份和月份中的日)的 System.DateTime 构造函数，可以从 MSDN 文档中找到这个构造函数和其他 DateTime 构造函数的更多信息。

```
public class Document
{
   public readonly DateTime CreationDate;

   public Document()
   {
      // Read in creation date from file. Assume result is 1 Jan 2002
      // but in general this can be different for different instances
      // of the class
      CreationDate = new DateTime(2002, 1, 1);
      }
   }
```

在上面的代码段中，CreationDate 和 MaxDocuments 的处理方式与任何其他字段相同，但因为它们是只读的，所以不能在构造函数外部赋值：

```
void SomeMethod()
{
   MaxDocuments = 10; // compilation error here. MaxDocuments is readonly
}
```

还要注意，在构造函数中不必给只读字段赋值。如果没有赋值，它的值就是其特定数据类型的默认值，或者在声明时给它初始化的值。这适用于只读的静态字段和实例字段。

3.4 匿名类型

第 2 章讨论了 var 关键字，它用于表示隐式类型化的变量。var 与 new 关键字一起使用时，可以创建匿名类型。匿名类型只是一个继承自 Object 且没有名称的类。该类的定义从初始化器中推断，类似于隐式类型化的变量。

如果需要一个对象包含某个人的姓氏、中间名和名字，则声明如下：

```
var captain = new {FirstName = "James", MiddleName = "T", LastName = "Kirk"};
```

这会生成一个包含 FirstName、MiddleName 和 LastName 属性的对象。如果创建另一个对象，如下所示：

```
var doctor = new {FirstName = "Leonard", MiddleName = "", LastName = "McCoy"};
```

captain 和 doctor 的类型就相同。例如，可以设置 captain = doctor。

如果所设置的值来自于另一个对象，就可以简化初始化器。如果已经有一个包含 FirstName、MiddleName 和 LastName 属性的类，且有该类的一个实例(person)，captain 对象就可以初始化为：

```
var captain = new {person.FirstName, person.MiddleName, person.LastName};
```

person 对象的属性名应投射到新对象名 captain。所以 captain 对象应有 FirstName、MiddleName 和 LastName 属性。

这些新对象的类型名未知。编译器为类型“伪造”了一个名称，但只有编译器才能使用它。我们不能也不应使用新对象上的任何类型反射，因为这不会得到一致的结果。

3.5 结构

前面介绍了类如何封装程序中的对象，也介绍了如何将它们存储在堆中，通过这种方式可以在数据的生存期上获得很大的灵活性，但性能会有一定的损失。因为托管堆的优化，这种性能损失比较小。但是，有时仅需要一个小的数据结构。此时，类提供的功能多于我们需要的功能，由于性能原因，最好使用结构。看看下面的例子：

```
class Dimensions
{
```

```
    public double Length;
    public double Width;
}
```

上面的代码定义了类 Dimensions，它只存储了某一项的长度和宽度。假定编写一个布置家具的程序，让人们试着在计算机上重新布置家具，并存储每件家具的尺寸。表面看来使字段变为公共字段会违背编程规则，但这里的关键是我们实际上并不需要类的全部功能。现在只有两个数字，把它们当成一对来处理，要比单个处理方便一些。既不需要很多方法，也不需要从类中继承，也不希望.NET 运行库在堆中遇到麻烦和性能问题，只需要存储两个 double 类型的数据即可。

为此，只需要修改代码，用关键字 struct 代替 class，定义一个结构而不是类，如本章前面所述：

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

为结构定义函数与为类定义函数完全相同。下面的代码说明了结构的构造函数和属性：

```
struct Dimensions
{
    public double Length;
    public double Width;

    public Dimensions(double length, double width)
    {
        Length=length;
        Width=width;
    }

    public double Diagonal
    {
        get
        {
          return Math.Sqrt(Length*Length + Width*Width);
        }
    }
}
```

结构是值类型，不是引用类型。它们存储在栈中或存储为内联(inline)(如果它们是存储在堆中的另一个对象的一部分)，其生存期的限制与简单的数据类型一样。

- 结构不支持继承。
- 对于结构，构造函数的工作方式有一些区别。尤其是编译器总是提供一个无参数的默认构造函数，它是不允许替换的。
- 使用结构，可以指定字段如何在内存中布局(第 15 章在介绍特性时将详细论述这个问题)。

因为结构实际上是把数据项组合在一起，有时大多数或者全部字段都声明为public。严格来说，这与编写.NET代码的规则相反——根据Microsoft，字段(除了const字段之外)应总是私有的，并由公有属性封装。但是，对于简单的结构，许多开发人员都认为公有字段是可接受的编程方式。

下面几节将详细说明类和结构之间的区别。

3.5.1　结构是值类型

虽然结构是值类型，但在语法上常常可以把它们当作类来处理。例如，在上面的 Dimensions 类的定义中，可以编写下面的代码：

```
Dimensions point = new Dimensions();
point.Length = 3;
point.Width = 6;
```

注意，因为结构是值类型，所以 new 运算符与类和其他引用类型的工作方式不同。new 运算符并不分配堆中的内存，而是只调用相应的构造函数，根据传送给它的参数，初始化所有的字段。对于结构，可以编写下述完全合法的代码：

```
Dimensions point;
point.Length = 3;
point.Width = 6;
```

如果 Dimensions 是一个类，就会产生一个编译错误，因为 point 包含一个未初始化的引用——不指向任何地方的一个地址，所以不能给其字段设置值。但对于结构，变量声明实际上是为整个结构在栈中分配空间，所以就可以为它赋值了。但要注意下面的代码会产生一个编译错误，编译器会抱怨用户使用了未初始化的变量：

```
Dimensions point;
Double D = point.Length;
```

结构遵循其他数据类型都遵循的规则：在使用前所有的元素都必须进行初始化。在结构上调用 new 运算符，或者给所有的字段分别赋值，结构就完全初始化了。当然，如果结构定义为类的成员字段，在初始化包含的对象时，该结构会自动初始化为 0。

结构是会影响性能的值类型，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在栈中。在结构超出了作用域被删除时，速度也很快，不需要等待垃圾回收。负面影响是，只要把结构作为参数来传递或者把一个结构赋予另一个结构(如 A=B，其中 A 和 B 是结构)，结构的所有内容就被复制，而对于类，则只复制引用。这样就会有性能损失，根据结构的大小，性能损失也不同。注意，结构主要用于小的数据结构。但当把结构作为参数传递给方法时，应把它作为 ref 参数传递，以避免性能损失——此时只传递了结构在内存中的地址，这样传递速度就与在类中的传递速度一样快了。但如果这样做，就必须注意被调用的方法可以改变结构的值。

3.5.2　结构和继承

结构不是为继承设计的。这意味着：它不能从一个结构中继承。唯一的例外是对应的结构(和 C#中的其他类型一样)最终派生于类 System.Object。因此，结构也可以访问 System.Object 的方法。在结构中，甚至可以重写 System.Object 中的方法——如重写 ToString()方法。结构的继承链是：每个结构派生自 System.ValueType 类，System.ValueType 类又派生自 System.Object。ValueType 并没有给 Object 添加任何新成员，但提供了一些更适合结构的实现方式。注意，不能为结构提供其他基类：每个结构都派生自 ValueType。

3.5.3 结构的构造函数

为结构定义构造函数的方式与为类定义构造函数的方式相同,但不允许定义无参数的构造函数。这看起来似乎没有意义，但其原因隐藏在.NET运行库的实现方式中。在一些极罕见的情况中，.NET运行库不能调用用户提供的自定义无参数构造函数，因此Microsoft干脆采用一种非常简单的方式：禁止在C#的结构内使用无参数的构造函数。

前面说过，默认构造函数把数值字段都初始化为0，把引用类型字段初始化为null，且总是隐式地给出，即使提供了其他带参数的构造函数，也是如此。提供字段的初始值也不能绕过默认构造函数。下面的代码会产生编译错误：

```
struct Dimensions
{
   public double Length = 1; // error. Initial values not allowed
   publicdouble Width = 2;  // error. Initial values not allowed
}
```

当然，如果Dimensions声明为一个类，这段代码就不会有编译错误。

另外，可以像类那样为结构提供Close()或Dispose()方法。第14章将讨论Dispose()方法。

3.6 弱引用

在应用程序代码内实例化一个类或结构时，只要有代码引用它，就会形成强引用。例如，如果有一个类MyClass()，并创建了一个变量myClassVariable来引用该类的对象，那么只要myClassVariable在作用域内，就存在对MyClass对象的强引用，如下所示：

```
MyClass myClassVariable = new MyClass();
```

这意味着垃圾回收器不会清理MyClass对象使用的内存。一般而言这是好事，因为可能需要访问MyClass对象，但是如果MyClass对象很大，并且不经常访问呢？此时可以创建对象的弱引用。

弱引用允许创建和使用对象，但是垃圾回收器运行时(第14章将介绍垃圾回收)，就会回收对象并释放内存。由于存在潜在的bug和性能问题，一般不会这么做，但是在特定的情况下使用弱引用是很合理的。

弱引用是使用WeakReference类创建的。因为对象可能在任意时刻被回收，所以在引用该对象前必须确认它存在。以前面的MathTest类为例，这次使用WeakReference类创建对它的弱引用：

```
static void Main()
{
   // Instantiate a weak reference to MathTest object
   WeakReference mathReference = new WeakReference(new MathTest());
   MathTest math;
   if(mathReference.IsAlive)
   {
      math = mathReference.Target as MathTest;
      math.Value = 30;
      Console.WriteLine("Value field of math variable contains " + math.Value);
      Console.WriteLine("Square of 30 is " + math.GetSquare());
   }
```

```
    else
    {
       Console.WriteLine("Reference is not available.");
    }

    GC.Collect();

    if(mathReference.IsAlive)
    {
       math = mathReference.Target as MathTest;
    }
    else
    {
       Console.WriteLine("Reference is not available.");
    }
}
```

创建 mathReference 时，会向其构造函数传递一个新的 MathTest 对象。MathTest 对象成为了 WeakReference 对象的目标。想要使用 MathTest 对象时，就需要先检查 mathReference 对象以确保其未被回收。IsAlive 属性就用于这个目的。如果 IsAlive 为 true，就从目标属性得到 MathTest 对象的引用。注意，因为 Target 属性返回的是 Object 类型，所以必须将其强制转换为 MathTest 类型。

然后，调用垃圾回收器(GC.Collect())，并尝试再次获得 MathTest 对象。这一次，IsAlive 属性返回 false，如果确实想要使用 MathTest 对象，就必须实例化一个新的 MathTest 对象。

3.7 部分类

partial 关键字允许把类、结构、方法或接口放在多个文件中。一般情况下，一个类全部驻留在单个文件中。但有时，多个开发人员需要访问同一个类，或者某种类型的代码生成器生成了一个类的某部分，所以把类放在多个文件中是有益的。

partial 关键字的用法是：把 partial 放在 class、struct 或 interface 关键字的前面。在下面的例子中，TheBigClass 类驻留在两个不同的源文件 BigClassPart1.cs 和 BigClassPart2.cs 中：

```
//BigClassPart1.cs
partial class TheBigClass
{
   public void MethodOne()
   {
   }
}

//BigClassPart2.cs
partial class TheBigClass
{
   public void MethodTwo()
   {
   }
}
```

编译包含这两个源文件的项目时，会创建一个 TheBigClass 类，它有两个方法 MethodOne()和 MethodTwo()。

如果声明类时使用了下面的关键字，这些关键字就必须应用于同一个类的所有部分：

- public
- private
- protected
- internal
- abstract
- sealed
- new
- 一般约束

在嵌套的类型中，只要 partial 关键字位于 class 关键字的前面，就可以嵌套部分类。在把部分类编译到类型中时，属性、XML 注释、接口、泛型类型的参数属性和成员会合并。有如下两个源文件：

```
//BigClassPart1.cs
[CustomAttribute]
partial class TheBigClass: TheBigBaseClass, IBigClass
{
  public void MethodOne()
  {
  }
}

//BigClassPart2.cs
[AnotherAttribute]
partial class TheBigClass: IOtherBigClass
{
  public void MethodTwo()
  {
  }
}
```

编译后，等价的源文件变成：

```
[CustomAttribute]
[AnotherAttribute]
partial class TheBigClass: TheBigBaseClass, IBigClass, IOtherBigClass
{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}
```

3.8 静态类

本章前面讨论了静态构造函数和它们如何初始化静态的成员变量。如果类只包含静态的方法和

属性，该类就是静态的。静态类在功能上与使用私有静态构造函数创建的类相同。不能创建静态类的实例。使用 static 关键字，编译器可以检查用户是否不经意间给该类添加了实例成员。如果是，就生成一个编译错误。这可以确保不创建静态类的实例。静态类的语法如下所示：

```
static class StaticUtilities
{
   public static void HelperMethod()
   {
   }
}
```

调用 HelperMethod()不需要 StaticUtilities 类型的对象。使用类型名即可进行该调用：

```
StaticUtilities.HelperMethod();
```

3.9 Object 类

前面提到，所有的.NET 类都派生自 System.Object。实际上，如果在定义类时没有指定基类，编译器就会自动假定这个类派生自 Object。本章没有使用继承，所以前面介绍的每个类都派生自 System.Object(如前所述，对于结构，这个派生是间接的：结构总是派生自 System.ValueType，System.ValueType 又派生自 System.Object)。

其实际意义在于，除了自己定义的方法和属性等外，还可以访问为 Object 定义的许多公有的和受保护的成员方法。这些方法可用于自己定义的所有其他类中。

3.9.1 System.Object()方法

下面将简要总结每个方法的作用，3.9.2 小节详细论述 ToString()方法。

- ToString()方法：是获取对象的字符串表示的一种便捷方式。当只需要快速获取对象的内容，以进行调试时，就可以使用这个方法。在数据的格式化方面，它几乎没有提供选择：例如，在原则上日期可以表示为许多不同的格式，但 DateTime.ToString()没有在这方面提供任何选择。如果需要更复杂的字符串表示，例如，考虑用户的格式化首选项或区域性(区域)，就应实现 IFormattable 接口(详见第 9 章)。
- GetHashCode()方法：如果对象放在名为映射(也称为散列表或字典)的数据结构中，就可以使用这个方法。处理这些结构的类使用该方法确定把对象放在结构的什么地方。如果希望把类用作字典的一个键，就需要重写 GetHashCode()方法。实现该方法重载的方式有一些相当严格的限制，这些将在第 10 章介绍字典时讨论。
- Equals()(两个版本)和 ReferenceEquals()方法：注意有 3 个用于比较对象相等性的不同方法，这说明.NET Framework 在比较相等性方面有相当复杂的模式。这 3 个方法和比较运算符“==”在使用方式上有微妙的区别。而且，在重写带一个参数的虚 Equals()方法时也有一些限制，因为 System.Collections 名称空间中的一些基类要调用该方法，并希望它以特定的方式执行。第 7 章在介绍运算符时将探讨这些方法的使用。
- Finalize()方法：第 13 章将介绍这个方法，它最接近 C++风格的析构函数，在引用对象作为垃圾被回收以清理资源时调用它。Object 中实现的 Finalize()方法实际上什么也没有做，因

而被垃圾回收器忽略。如果对象拥有对未托管资源的引用，则在该对象被删除时，就需要删除这些引用，此时一般要重写 Finalize()。垃圾收集器不能直接删除这些对未托管资源的引用，因为它只负责托管的资源，于是它只能依赖用户提供的 Finalize()。

- GetType()方法：这个方法返回从 System.Type 派生的类的一个实例。这个对象可以提供对象成员所属类的更多信息，包括基本类型、方法、属性等。System.Type 还提供了.NET 的反射技术的入口点。这个主题详见第 15 章。
- MemberwiseClone()方法：这是 System.Object 中唯一没有在本书的其他地方详细论述的方法。不需要讨论这个方法，因为它在概念上相当简单，它只复制对象，并返回对副本的一个引用(对于值类型，就是一个装箱的引用)。注意，得到的副本是一个浅表复制，即它复制了类中的所有值类型。如果类包含内嵌的引用，就只复制引用，而不复制引用的对象。这个方法是受保护的，所以不能用于复制外部的对象。该方法不是虚方法，所以不能重写它的实现代码。

3.9.2 ToString()方法

第 2 章已经提到了 ToString()方法，它是快速获取对象的字符串表示的最便捷方式。

例如：

```
inti = 50;
string str = i.ToString(); // returns "50"
```

下面是另一个例子：

```
enum Colors {Red, Orange, Yellow};
// later on in code...
Colors favoriteColor = Colors.Orange;
string str = favoriteColor.ToString(); // returns "Orange"
```

Object.ToString()声明为虚方法，在这些例子中，实现该方法的代码都是为 C#预定义数据类型重写过的代码，以返回这些类型的正确字符串表示。Colors 枚举是一个预定义的数据类型，它实际上实现为一个派生自 System.Enum 的结构，而 System.Enum 有一个相当智能的 ToString()重写方法，它处理用户定义的所有枚举。

如果不在自己定义的类中重写 ToString()，该类将只继承 System.Object 的实现方式——它显示类的名称。如果希望 ToString()返回一个字符串，其中包含类中对象的值信息，就需要重写它。下面用一个例子 Money 来说明这一点。在该例子中，定义一个非常简单的类 Money，它表示美元数。Money 只是 decimal 类的包装器，但它提供了一个 ToString()方法。注意，这个方法必须声明为 override，因为它将替代(重写)Object 提供的 ToString()方法。第 4 章将详细讨论重写。该例子的完整代码如下所示(注意它还说明了如何使用属性封装字段)：

```
using System;

namespace Wrox
{
  class MainEntryPoint
  {
    static void Main(string[] args)
```

```
      {
        Money cash1 = new Money();
        cash1.Amount = 40M;
        Console.WriteLine("cash1.ToString() returns: " + cash1.ToString());
        Console.ReadLine();
      }
    }
    public class Money
    {
      private decimal amount;

      public decimal Amount
      {
        get
        {
           return amount;
        }
        set
        {
           amount = value;
        }
      }
      public override string ToString()
      {
         return "$" + Amount.ToString();
      }
    }

}
```

这个例子仅说明了 C#的语法特性。C#已经有表示货币量的预定义类型 decimal。所以在现实生活中，不必编写这样的类来重复该功能，除非要给它添加其他各种方法。在许多情况下，由于格式化要求，也可以使用 String.Format()方法(详见第 8 章)来表示货币字符串，而不是 ToString()。

在 Main()方法中，先实例化一个 Money 对象，再调用 ToString()，执行该方法的重写版本。运行这段代码，会得到如下结果：

```
cash1.ToString() returns: $40
```

3.10 扩展方法

有许多扩展类的方式。如果有类的源代码，继承(如第 4 章所述)就是给对象添加功能的好方法。但如果没有源代码，则可以使用扩展方法，它允许改变一个类，但不需要该类的源代码。

扩展方法是静态方法，它是类的一部分，但实际上没有放在类的源代码中。假定上例中的 Money 类需要一个方法 AddToAmount(decimal amountToAdd)。但是，由于某种原因，程序集最初的源代码不能直接修改。此时必须创建一个静态类，把方法 AddToAmount()添加为一个静态方法。对应的代码如下：

```
namespace Wrox
{
```

```
    public static class MoneyExtension
    {
      public static void AddToAmount(this Money money, decimal amountToAdd)
      {
         money.Amount +=.amountToAdd;
      }
    }
}
```

注意 AddToAmount()方法的参数。对于扩展方法，第一个参数是要扩展的类型，它放在 this 关键字的后面。这告诉编译器，这个方法是 Money 类型的一部分。在这个例子中，Money 是要扩展的类型。在扩展方法中，可以访问所扩展类型的所有公有方法和属性。

在主程序中，AddToAmount()方法看起来像是另一个方法。它没有显示第一个参数，也不能对它进行任何处理。要使用新方法，需要执行如下调用，这与其他方法相同：

```
cash1.AddToAmount(10M);
```

即使扩展方法是静态的，也要使用标准的实例方法语法。注意这里使用 cash1 实例变量来调用 AddToAmount()，而没有使用类型名。

如果扩展方法与类中的某个方法同名，就从来不会调用扩展方法。类中已有的任何实例方法优先。

3.11 小结

本章介绍了 C#中声明和处理对象的语法，论述了如何声明静态和实例字段、属性、方法和构造函数。还讨论了 C#中新增的且其他语言的 OOP 模型中没有的新特性。例如，静态构造函数提供了初始化静态字段的方式，利用结构可以定义高性能的类型，不需要使用托管的堆。我们还阐述了 C#中的所有类型最终都派生自类 System.Object，这说明所有的类型都开始于一组基本的实用方法，包括 ToString()。本章多次提到了继承，第 4 章将介绍 C#中的实现(implementation)继承和接口继承。

第 4 章

继　　承

本章要点

- 继承的类型
- 实现继承
- 访问修饰符
- 接口

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- BankAccounts.cs
- CurrentAccounts.cs
- MortimerPhones.cs

4.1 继承

第 3 章介绍了如何使用 C#中的各个类，其重点是如何定义单个类(或单个结构)中的方法、属性、构造函数和其他成员。尽管已说明所有的类最终都派生于 System.Object 类，但并没有说明如何创建继承类的层次结构。继承是本章的主题。本章将讨论 C#和.NET Framework 如何处理继承。

4.2 继承的类型

首先介绍 C#在继承方面支持和不支持的功能。

4.2.1 实现继承和接口继承

在面向对象的编程中，有两种截然不同的继承类型：实现继承和接口继承。

- **实现继承**：表示一个类型派生于一个基类型，它拥有该基类型的所有成员字段和函数。在实现继承中，派生类型采用基类型的每个函数的实现代码，除非在派生类型的定义中指定重写某个函数的实现代码。在需要给现有的类型添加功能，或许多相关的类型共享一组重要的公共功能时，这种类型的继承非常有用。
- **接口继承**：表示一个类型只继承了函数的签名，没有继承任何实现代码。在需要指定该类型具有某些可用的特性时，最好使用这种类型的继承。

C#支持实现继承和接口继承。它们都内置于语言和架构中，因此可以根据应用程序的体系结构选择合适的继承。

4.2.2 多重继承

一些语言(如 C++)支持所谓的“多重继承”，即一个类派生自多个类。使用多重继承的优点是有争议的：一方面，毫无疑问，可以使用多重继承编写非常复杂、但很紧凑的代码，如 C++ ATL 库。另一方面，使用多重实现继承的代码常常很难理解和调试(这也可以从 C++ ATL 库中看出)。如前所述，简化健壮代码的编写工作是开发 C#的重要设计目标。因此，C#不支持多重实现继承。而 C#又允许类型派生自多个接口——多重接口继承。这说明，C#类可以派生自另一个类和任意多个接口。更准确地说，因为 System.Object 是一个公共的基类，所以每个 C#类(除了 Object 类之外)都有一个基类，还可以有任意多个基接口。

4.2.3 结构和类

第 3 章区分了结构(值类型)和类(引用类型)。使用结构的一个限制是结构不支持继承，但每个结构都自动派生自 System.ValueType。不能编码实现类型层次的结构，但结构可以实现接口。换言之，结构并不支持实现继承，但支持接口继承。定义结构和类可以总结为：

- 结构总是派生自 System.ValueType，它们还可以派生自任意多个接口。
- 类总是派生自 System.Object 或用户选择的另一个类，它们还可以派生自任意多个接口。

4.3 实现继承

如果要声明派生自另一个类的一个类，就可以使用下面的语法：

```
classMyDerivedClass: MyBaseClass
{
   // functions and data members here
}
```

这个语法非常类似于 C++和 Java 中的语法，但是，C++程序员习惯于使用公共和私有继承的概念，要注意 C#不支持私有继承，因此在基类名上没有 public 或 private 限定符。支持私有继承只会大大增加语言的复杂性，实际上私有继承在 C++中也很少使用。

如果类(或结构)也派生自接口，则用逗号分隔列表中的基类和接口：

```
public class MyDerivedClass: MyBaseClass, IInterface1, IInterface2
{
   // etc.
}
```

对于结构，语法如下：

```
public struct MyDerivedStruct: IInterface1, IInterface2
{
   // etc.
}
```

如果在类定义中没有指定基类，C#编译器就假定 System.Object 是基类。因此下面的两段代码生成相同的结果：

```
class MyClass: Object // derives from System.Object
{
   // etc.
}
```

和

```
class MyClass // derives from System.Object
{
   // etc.
}
```

第二种形式比较常用，因为它较简单。

因为 C#支持 object 关键字，它用作 System.Object 类的假名，所以也可以编写下面的代码：

```
class MyClass: object // derives from System.Object
{
   // etc.
}
```

如果要引用 Object 类，就可以使用 object 关键字，智能编辑器(如 Visual Studio .NET)会识别它，因此便于编辑代码。

4.3.1 虚方法

把一个基类函数声明为 virtual，就可以在任何派生类中重写该函数：

```
class MyBaseClass
{
  public virtual string VirtualMethod()
  {
     return "This method is virtual and defined in MyBaseClass";
  }
}
```

也可以把属性声明为 virtual。对于虚属性或重写属性，语法与非虚属性相同，但要在定义中添加关键字 virtual，其语法如下所示：

```
public virtual string ForeName
{
    get { return foreName;}
    set { foreName = value;}
}
private string foreName;
```

为了简单起见，下面的讨论将主要集中于方法，但其规则也适用于属性。

C#中虚函数的概念与标准 OOP 的概念相同：可以在派生类中重写虚函数。在调用方法时，会调用该类对象的合适方法。在 C#中，函数在默认情况下不是虚拟的，但(除了构造函数以外)可以显式地声明为 virtual。这遵循 C++的方式，即从性能的角度来看，除非显式指定，否则函数就不是虚拟的。而在 Java 中，所有的函数都是虚拟的。但 C#的语法与 C++的语法不同，因为 C#要求在派生类的函数重写另一个函数时，要使用 override 关键字显式声明：

```
class MyDerivedClass: MyBaseClass
{
  public override string VirtualMethod()
  {
     return "This method is an override defined in MyDerivedClass.";
  }
}
```

重写方法的语法避免了 C++中很容易发生的潜在运行错误：当派生类的方法签名无意中与基类版本略有差别时，该方法就不能重写基类的方法。在 C#中，这会出现一个编译错误，因为编译器会认为函数已标记为 override，但没有重写其基类的方法。

成员字段和静态函数都不能声明为 virtual，因为这个概念只对类中的实例函数成员有意义。

4.3.2 隐藏方法

如果签名相同的方法在基类和派生类中都进行了声明，但该方法没有分别声明为 virtual 和 override，派生类方法就会隐藏基类方法。

在大多数情况下，是要重写方法，而不是隐藏方法，因为隐藏方法会造成对于给定类的实例调用错误方法的危险。但是，如下面的例子所示，C#语法可以确保开发人员在编译时收到这个潜在错误的警告，从而使隐藏方法(如果这确实是用户的本意)更加安全。这也是类库开发人员得到的版本方面的好处。

假定有一个类 HisBaseClass：

```
class HisBaseClass
{
    // various members
}
```

在将来的某一刻，要编写一个派生类，用它给 HisBaseClass 添加某个功能，特别是要添加该基类中目前没有的方法——MyGroovyMethod()：

```
class MyDerivedClass: HisBaseClass
{
  public int MyGroovyMethod()
  {
```

```
        // some groovy implementation
        return 0;
    }
}
```

一年后，基类的编写者决定扩展基类的功能。为了保持一致，他也添加了一个名为 MyGroovyMethod()的方法，该方法的名称和签名与前面添加的方法相同，但并不完成相同的工作。在使用基类的新方法编译代码时，程序在应该调用哪个方法上就会有潜在的冲突。这在 C#中完全合法，但因为 MyGroovyMethod()与基类的 MyGroovyMethod()不相关，运行这段代码就可能会产生意外的结果。C#可以很好地处理这种冲突。

此时，编译时系统会发出警告。在 C#中，要隐藏一个方法应使用 new 关键字声明，如下所示：

```
class MyDerivedClass: HisBaseClass
{
  public new int MyGroovyMethod()
  {
    // some groovy implementation
    return 0;
  }
}
```

但是，新添加的 MyGroovyMethod()没有声明为 new，所以编译器会认为它隐藏了基类的方法，但没有显式声明，因此系统会发出一个警告(这也适用于是否把 MyGroovyMethod()声明为 virtual)。如果愿意，就可以给新方法重命名。最好这么做，因为这会避免许多冲突。但是，如果觉得重命名方法不可能(例如，已经针对其他公司把软件发布为一个库，所以无法修改方法的名称)，则所有的已有客户端代码仍能正确运行，选择新添加的 MyGroovyMethod()。这是因为访问这个方法的任何已有代码必须通过对 MyDerivedClass(或进一步派生的类)的引用进行选择。

已有的代码不能通过对 HisBaseClass 类的引用访问这个方法，因为在对 HisBaseClass 类的早期版本进行编译时，会产生一个编译错误。这个问题只会发生在将来编写的客户端代码上。C#会发出一个警告，告诉用户在将来的代码中可能会出问题——用户应注意这个警告，不要试图在将来添加的代码中通过对 HisBaseClass 的引用调用新的 MyGroovyMethod()方法，但所有已有的代码仍会正常工作。这是比较微妙的，但它很好地说明了 C#如何处理类的不同版本。

4.3.3 调用函数的基类版本

C#有一种特殊的语法用于从派生类中调用方法的基类版本：base.<MethodName>()。例如，假定派生类中的一个方法要返回基类的方法 90%的返回值，就可以使用下面的语法：

```
class CustomerAccount
{
  public virtual decimal CalculatePrice()
  {
    // implementation
    return 0.0M;
  }
}
class GoldAccount: CustomerAccount
{
  public override decimal CalculatePrice()
```

```
    {
       return base.CalculatePrice() * 0.9M;
    }
}
```

注意，可以使用 base.<MethodName>()语法调用基类中的任何方法，不必从同一个方法的重载中调用它。

4.3.4 抽象类和抽象函数

C#允许把类和函数声明为abstract。抽象类不能实例化，而抽象函数不能直接实现，必须在非抽象的派生类中重写。显然，抽象函数本身也是虚拟的(尽管也不需要提供virtual关键字，实际上，如果提供了该关键字，就会产生一个语法错误)。如果类包含抽象函数，则该类也是抽象的，也必须声明为抽象的：

```
abstract class Building
{
   public abstract decimal CalculateHeatingCost(); // abstract method
}
```

C++开发人员还要注意术语上的细微差别：在C++中，抽象函数常常描述为纯虚函数，而在C#中，仅使用抽象这个术语。

4.3.5 密封类和密封方法

C#允许把类和方法声明为sealed。对于类，这表示不能继承该类；对于方法，这表示不能重写该方法。

```
sealed class FinalClass
{
   // etc
}
class DerivedClass: FinalClass       // wrong. Will give compilation error
{
   // etc
}
```

在把类或方法标记为sealed时，最可能的情形是：如果要对库、类或自己编写的其他类作用域之外的类或方法进行操作，则重写某些功能会导致代码混乱。也可以因商业原因把类或方法标记为sealed，以防第三方以违反授权协议的方式扩展该类。但一般情况下，在把类或成员标记为 sealed 时要小心，因为这么做会严重限制它的使用方式。即使认为它不能对继承自一个类或重写类的某个成员发挥作用，仍有可能在将来的某个时刻，有人会遇到我们没有预料到的情形，此时这么做就很有用。.NET基类库大量使用了密封类，使希望从这些类中派生出自己的类的第三方开发人员无法访问这些类。例如，string就是一个密封类。

把方法声明为sealed也可以实现类似的目的：

```
class MyClass: MyClassBase
{
  public sealed override void FinalMethod()
  {
     // etc.
  }
}
class DerivedClass: MyClass
{
  public override void FinalMethod() // wrong. Will give compilation error
  {
  }
}
```

要在方法或属性上使用 sealed 关键字，必须先从基类上把它声明为要重写的方法或属性。如果基类上不希望有重写的方法或属性，就不要把它声明为 virtual。

4.3.6 派生类的构造函数

第 3 章介绍了单个类的构造函数是如何工作的。这样，就产生了一个有趣的问题，在开始为层次结构中的类(这个类继承了其他也可能有自定义构造函数的类)定义自己的构造函数时，会发生什么情况？

假定没有为任何类定义任何显式的构造函数，这样编译器就会为所有的类提供默认的初始化构造函数，在后台会进行许多操作，但编译器可以很好地解决类的层次结构中的所有问题，每个类中的每个字段都会初始化为对应的默认值。但在添加了一个我们自己的构造函数后，就要通过派生类的层次结构高效地控制构造过程，因此必须确保构造过程顺利进行，不要出现不能按照层次结构进行构造的问题。

为什么派生类会有某些特殊的问题？原因是在创建派生类的实例时，实际上会有多个构造函数起作用。要实例化的类的构造函数本身不能初始化类，还必须调用基类中的构造函数。这就是为什么要通过层次结构进行构造的原因。

为了说明为什么必须调用基类的构造函数，下面是手机公司 MortimerPhones 开发的一个例子。这个例子包含一个抽象类 GenericCustomer，它表示顾客。还有一个(非抽象)类 Nevermore60Customer，它表示采用特定付费方式(称为 Nevermore60 付费方式)的顾客。所有的顾客都有一个名字，它由一个私有字段表示。在 Nevermore60 付费方式中，顾客前几分钟的电话费比较高，需要一个字段 highCostMinutesUsed，它详细说明了每个顾客该如何支付这些较高的电话费。抽象类 GenericCustomer 的定义如下所示：

```
abstract class GenericCustomer
{
  private string name;
  // lots of other methods etc.
}
class Nevermore60Customer: GenericCustomer
{
  privateu int highCostMinutesUsed;
  // other methods etc.
}
```

不要担心在这些类中实现的其他方法，因为这里仅考虑构造过程。如果下载了本章的示例代码，就会发现类的定义仅包含构造函数。

下面看看使用 new 运算符实例化 Nevermore60Customer 时，会发生什么情况：

```
GenericCustomer customer = new Nevermore60Customer();
```

显然，成员字段 name 和 highCostMinutesUsed 都必须在实例化 customer 时进行初始化。如果没有提供自己的构造函数，而是仅依赖默认的构造函数，那么 name 会初始化为 null 引用，highCostMinutesUsed 初始化为 0。下面详细讨论其过程。

highCostMinutesUsed 字段没有问题：编译器提供的默认 Nevermore60Customer 构造函数会把它初始化为 0。

那么 name 呢？看看类定义，显然，Nevermore60Customer 构造函数不能初始化这个值。字段 name 声明为 private，这意味着派生的类不能访问它。默认的 Nevermore60Customer 构造函数甚至不知道存在这个字段。唯一知道这个字段的代码项是 GenericCustomer 的其他成员，这意味着如果对 name 进行初始化，就必须在 GenericCustomer 的某个构造函数中进行。无论类层次结构有多大，这种情况都会一直延续到最终的基类 System.Object 上。

理解了上面的问题后，就可以明白实例化派生类时会发生什么样的情况了。假定默认的构造函数一直在使用：编译器首先找到它试图实例化的类的构造函数，在本例中是 Nevermore60Customer，这个默认 Nevermore60Customer 构造函数首先要做的是为其直接基类 GenericCustomer 运行默认构造函数，然后 GenericCustomer 构造函数为其直接基类 System.Object 运行默认构造函数，System.Object 没有任何基类，所以它的构造函数就执行，并把控制权返回给 GenericCustomer 构造函数。现在执行 GenericCustomer 构造函数，把 name 初始化为 null，再把控制权返回给 Nevermore60Customer 构造函数，接着执行这个构造函数，把 highCostMinutesUsed 初始化为 0，并退出。此时，Nevermore60Customer 实例就已经成功地构造和初始化了。

所有操作的最终结果是，构造函数的调用顺序是先调用 System.Object，再按照层次结构由上向下进行，直到到达编译器要实例化的类为止。还要注意在这个过程中，每个构造函数都初始化它自己的类中的字段。这是它的一般工作方式，在开始添加自己的构造函数时，也应尽可能遵循这条规则。

注意构造函数的执行顺序。最先调用的总是基类的构造函数。也就是说，派生类的构造函数可以在执行过程中调用它可以访问的任何基类方法、属性和任何其他成员，因为基类已经构造出来了，其字段也初始化了。这也意味着，如果派生类不喜欢初始化基类的方式，那么只要它能访问基类的数据，就可以改变数据的初始值。但是，好的编程方式几乎总是应尽可能避免这种情况，让基类构造函数来处理其字段。

理解了构造过程后，就可以开始添加自己的构造函数了。

1. 在层次结构中添加无参数的构造函数

首先讨论最简单的情况，在层次结构中用一个无参数的构造函数来替换默认的构造函数后，看看会发生什么情况。假定要把每个人的名字初始化为字符串"<no name>"，而不是 null 引用。就可以修改 GenericCustomer 中的代码，如下所示：

```
public abstract class GenericCustomer
{
  private string name;
  public GenericCustomer()
    : base() // We could omit this line without affecting the compiled code.
  {
     name = "<no name>";
  }
```

添加这段代码后，代码运行正常。Nevermore60Customer 仍有自己的默认构造函数，所以上面描述的事件的顺序保持不变，但编译器会使用自定义 GenericCustomer 构造函数，而不是生成默认的构造函数，所以 name 字段按照需要总是初始化为"<no name>"。

注意，在定制的构造函数中，在执行 GenericCustomer 构造函数前，添加了一个对基类构造函数的调用，使用的语法与前面解释如何让构造函数的不同重载版本互相调用时使用的语法相同。唯一的区别是，这次使用的关键字是 base，而不是 this，表示这是基类的构造函数，而不是要调用的当前类的构造函数。在 base 关键字后面的圆括号中没有参数，这非常重要，因为没有给基类构造函数传送任何参数，所以编译器必须调用无参数的构造函数。其结果是编译器会插入要调用 System.Object 构造函数的代码，这正好与默认情况相同。

实际上，可以省略这行代码，只加上为本章中大多数构造函数编写的代码：

```
public GenericCustomer()
{
   name = "<no name>";
}
```

如果编译器没有在左花括号的前面找到对另一个构造函数的任何引用，它就会假定我们要调用基类的构造函数——这符合默认构造函数的工作方式。

base 和 this 关键字是调用另一个构造函数时允许使用的唯一关键字，其他关键字都会产生编译错误。还要注意只能指定唯一一个其他的构造函数。

到目前为止，这段代码运行正常。但是，要通过构造函数的层次结构把进度弄乱的最好方法是把构造函数声明为私有：

```
private GenericCustomer()
{
   name = "<no name>";
}
```

如果试图这样做，就会产生一个有趣的编译错误，如果不理解构造是如何按照层次结构由上而下的顺序工作的，这个错误就会让人摸不着头脑。

```
'Wrox.ProCSharp.GenericCustomer.GenericCustomer()' is inaccessible due to its protection
level
```

有趣的是，该错误没有发生在 GenericCustomer 类中，而是发生在 Nevermore60Customer 派生类中。编译器试图为 Nevermore60Customer 生成默认的构造函数，但又做不到，因为默认的构造函数应调用无参数的 GenericCustomer 构造函数。把该构造函数声明为 private，它就不可能访问派生类了。如果为 GenericCustomer 提供一个带参数的构造函数，但同时没有提供一个无参数的构造函数，

也会发生类似的错误。在本例中，编译器不能为 GenericCustomer 生成默认构造函数，所以当编译器试图为任何派生类生成默认构造函数时，它会再次发现它不能做到这一点，因为没有无参数的基类构造函数可调用。这个问题的解决方法是为派生类添加自己的构造函数——实际上不需要在这些构造函数中做任何工作，这样，编译器就不会为这些派生类生成任何默认构造函数了。

前面介绍了所有的理论知识，下面用一个例子来说明如何给类的层次结构添加构造函数。下面为 MortimerPhones 示例添加带参数的构造函数。

2. 在层次结构中添加带参数的构造函数

首先是带一个参数的 GenericCustomer 构造函数，它仅在顾客提供其姓名时才实例化顾客：

```
abstract class GenericCustomer
{
  private string name;
  public GenericCustomer(string name)
  {
     this.name = name;
  }
```

到目前为止，代码正常运行，但刚才说过，在编译器试图为派生类创建默认构造函数时，会产生一个编译错误，因为编译器为 Nevermore60Customer 生成的默认构造函数会试图调用一个无参数的 GenericCustomer 构造函数，但 GenericCustomer 没有这样的构造函数。因此，需要为派生类提供一个构造函数，来避免这个编译错误：

```
class Nevermore60Customer: GenericCustomer
{
  private uint highCostMinutesUsed;
  public Nevermore60Customer(string name)
    :   base(name)
  {
  }
```

现在，Nevermore60Customer 对象的实例化只有在提供了包含顾客姓名的字符串时才能进行，这正是我们需要的。有趣的是 Nevermore60Customer 构造函数对这个字符串所做的处理。它本身不能初始化 name 字段，因为它不能访问基类中的私有字段，但可以把顾客姓名传送给基类，以便 GenericCustomer 构造函数处理。具体方法是，把先执行的基类构造函数指定为把顾客姓名作为参数的构造函数。除此之外，它不需要执行任何操作。

下面讨论如果要处理不同的重载构造函数和一个类的层次结构，会发生什么情况。最终，假定 Nevermore60 的顾客通过朋友联系到 MortimerPhones，即 MortimerPhones 公司中有一个人是朋友，因此通过与朋友签约可以获得折扣。这表示在构造一个 Nevermore60Customer 时，还需要传递联系人的姓名。在现实生活中，构造函数必须利用该姓名去完成更复杂的工作，如处理折扣等，但这里只是把联系人的姓名存储到另一个字段中。

此时，Nevermore60Customer 的定义如下所示：

```
class Nevermore60Customer: GenericCustomer
{
  public Nevermore60Customer(string name, string referrerName)
```

```
    : base(name)
    {
      this.referrerName = referrerName;
    }

    private string referrerName;
    private uint highCostMinutesUsed;
```

该构造函数将姓名作为参数，把它传递给 GenericCustomer 构造函数进行处理。referrerName 是一个需要声明的变量，这样构造函数才能在其主体中处理这个参数。

但是，并不是所有的 Nevermore60Customers 都有联系人，所以还需要有一个不需要此参数的构造函数(或为它提供默认值的构造函数)。实际上，我们指定如果没有联系人，referrerName 字段就设置为"<None>"，使用如下带一个参数的构造函数：

```
public Nevermore60Customer(string name)
  : this(name, "<None>")
{
}
```

这样就正确建立了所有的构造函数。执行下面的代码行时，检查事件链很有益：

```
GenericCustomer customer = new Nevermore60Customer("Arabel Jones");
```

编译器认为它需要带一个字符串参数的构造函数，所以它确认的构造函数就是刚才定义的最后一个构造函数，如下所示。

```
public Nevermore60Customer(string Name)
  : this(Name, "<None>")
```

在实例化 customer 时，就会调用这个构造函数。之后立即把控制权传递给对应的 Nevermore60Customer 构造函数，该构造函数带两个参数，分别是"Arabel Jones"和"<None>"。在这个构造函数中，把控制权依次传递给 GenericCustomer 构造函数，该构造函数带有 1 个参数，即字符串"Arabel Jones"。然后这个构造函数把控制权传送给 System.Object 默认构造函数。现在才能执行这些构造函数，首先执行 System.Object 构造函数。接着执行 GenericCustomer 构造函数，它初始化 name 字段。然后带有两个参数的 Nevermore60Customer 构造函数得到控制权，把 referrerName 初始化为"<None>"。最后，执行 Nevermore60Customer 构造函数，该构造函数带有 1 个参数——这个构造函数什么也不做。

这个过程非常简洁，设计也很合理。每个构造函数都负责处理相应变量的初始化。在这个过程中，正确地实例化了类，以备使用。如果在为类编写自己的构造函数时遵循同样的规则，就会发现，即便是最复杂的类也可以顺利地初始化，并且不会出现任何问题。

4.4 修饰符

前面已经遇到许多所谓的修饰符，即应用于类型或成员的关键字。修饰符可以指定方法的可见性，如 public 或 private；还可以指定一项的本质，如方法是 virtual 或 abstract。C#有许多访问修饰符，下面讨论完整的修饰符列表。

4.4.1 可见性修饰符

表 4-1 中的修饰符确定了是否允许其他代码访问某一项。

表4-1

修 饰 符	应 用 于	说 明
public	所有类型或成员	任何代码均可以访问该项
protected	类型和内嵌类型的所有成员	只有派生的类型能访问该项
internal	所有类型或成员	只能在包含它的程序集中访问该项
private	类型和内嵌类型的所有成员	只能在它所属的类型中访问该项
protected internal	类型和内嵌类型的所有成员	只能在包含它的程序集和派生类型的任何代码中访问该项

注意，类型定义可以是内部或公有的，这取决于是否希望在包含类型的程序集外部访问它：

```
public class MyClass
{
   // etc.
```

不能把类型定义为 protected、private 和 protected internal，因为这些修饰符对于包含在名称空间中的类型没有意义。因此这些修饰符只能应用于成员。但是，可以用这些修饰符定义嵌套的类型(即，包含在其他类型中的类型)，因为在这种情况下，类型也具有成员的状态。于是，下面的代码是合法的：

```
public class OuterClass
{
  protected class InnerClass
  {
   // etc.
  }
  // etc.
}
```

如果有嵌套的类型，则内部的类型总是可以访问外部类型的所有成员。所以，在上面的代码中，InnerClass 中的代码可以访问 OuterClass 的所有成员，甚至可以访问 OuterClass 的私有成员。

4.4.2 其他修饰符

表 4-2 中的修饰符可以应用于类型的成员，而且有不同的用途。在应用于类型时，其中的几个修饰符也是有意义的。

表4-2

修 饰 符	应 用 于	说 明
new	函数成员	成员用相同的签名隐藏继承的成员
static	所有成员	成员不作用于类的具体实例
virtual	仅函数成员	成员可以由派生类重写
abstract	仅函数成员	虚拟成员定义了成员的签名，但没有提供实现代码
override	仅函数成员	成员重写了继承的虚拟或抽象成员

(续表)

修 饰 符	应 用 于	说 明
sealed	类、方法和属性	对于类，不能继承自密封类。对于属性和方法，成员重写已继承的虚拟成员，但任何派生类中的任何成员都不能重写该成员。该修饰符必须与 override 一起使用
extern	仅静态[DllImport]方法	成员在外部用另一种语言实现

4.5 接口

如前所述，如果一个类派生自一个接口，声明这个类就会实现某些函数。并不是所有的面向对象语言都支持接口，所以本节将详细介绍 C#接口的实现。下面列出 Microsoft 预定义的一个接口 System.IDisposable 的完整定义。IDisposable 包含一个方法 Dispose()，该方法由类实现，用于清理代码：

```
public interface IDisposable
{
   void Dispose();
}
```

上面的代码说明，声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的实现方式。一般情况下，接口只能包含方法、属性、索引器和事件的声明。

不能实例化接口，它只能包含其成员的签名。接口既不能有构造函数(如何构建不能实例化的对象？)也不能有字段(因为这隐含了某些内部的实现方式)。接口定义也不允许包含运算符重载，尽管这不是因为声明它们在原则上有什么问题，而是因为接口通常是公共协定，包含运算符重载会引起一些与其他.NET 语言不兼容的问题，如 Visual Basic .NET，因为它不支持运算符重载。

在接口定义中还不允许声明关于成员的修饰符。接口成员总是公有的，不能声明为虚拟或静态。如果需要，就应由实现的类来声明，因此最好实现执行的类来声明访问修饰符，就像本节的代码那样。

例如 IDisposable。如果类希望声明为公有类型，以便它实现方法 Dispose()，该类就必须实现 IDisposable。在 C#中，这表示该类派生自 IDisposable 类。

```
class SomeClass: IDisposable
{
  // This class MUST contain an implementation of the
  // IDisposable.Dispose() method, otherwise
  // you get a compilation error.
  public void Dispose()
  {
     // implementation of Dispose() method
  }
  // rest of class
}
```

在这个例子中，如果 SomeClass 派生自 IDisposable 类，但不包含与 IDisposable 类中签名相同的

Dispose()实现代码，就会得到一个编译错误，因为该类破坏了实现 IDisposable 的一致协定。当然，编译器允许类有一个不派生自 IDisposable 类的 Dispose()方法。问题是其他代码无法识别出 SomeClass 类支持 IDisposable 特性。

IDisposable 是一个相当简单的接口，它只定义了一个方法。大多数接口都包含许多成员。

4.5.1 定义和实现接口

下面开发一个遵循接口继承规范的小例子来说明如何定义和使用接口。这个例子建立在银行账户的基础上。假定编写代码，最终允许在银行账户之间进行计算机转账业务。许多公司可以实现银行账户，但它们一致认为，表示银行账户的所有类都实现接口 IBankAccount。该接口包含一个用于存取款的方法和一个返回余额的属性。这个接口还允许外部代码识别由不同银行账户实现的各种银行账户类。我们的目的是允许银行账户彼此通信，以便在账户之间进行转账业务，但还没有介绍这个功能。

为了使例子简单一些，我们把本例子的所有代码都放在同一个源文件中，但实际上不同的银行账户类不仅会编译到不同的程序集中，而且这些程序集位于不同银行的不同机器上。但这些内容对于我们的目的过于复杂了。为了保留一定的真实性，我们为不同的公司定义不同的名称空间。

首先，需要定义 IBankAccount 接口：

```
namespace Wrox.ProCSharp
{
  public interface IBankAccount
  {
    void PayIn(decimal amount);
    bool Withdraw(decimal amount);
    decimal Balance { get; }
  }
}
```

注意，接口的名称为 IBankAccount。接口名称通常以字母 I 开头，以便知道这是一个接口。

如第 2 章所述，在大多数情况下，.NET 的用法规则不鼓励采用所谓的 Hungarian 表示法，在名称的前面加一个字母，表示所定义对象的类型。接口是少数几个推荐使用 Hungarian 表示法的例外之一。

现在可以编写表示银行账户的类了。这些类不必彼此相关，它们可以是完全不同的类。但它们都表示银行账户，因为它们都实现了 IBankAccount 接口。

下面是第一个类，一个由 Royal Bank of Venus 运行的存款账户：

```
namespace Wrox.ProCSharp.VenusBank
{
```

```
public class SaverAccount: IBankAccount
{
  private decimal balance;
  public void PayIn(decimal amount)
  {
     balance += amount;
  }
  public bool Withdraw(decimal amount)
  {
    if (balance >= amount)
    {
      balance -= amount;
      return true;
    }
    Console.WriteLine("Withdrawal attempt failed.");
    return false;
  }
  public decimal Balance
  {
    get
    {
       return balance;
    }
  }
  public override string ToString()
  {
     returnString.Format("Venus Bank Saver: Balance = {0,6:C}", balance);
  }
}
}
```

实现这个类的代码的作用一目了然。其中包含一个私有字段 balance，当存款或取款时就调整这个字段。如果因为账户中的金额不足而取款失败，就会显示一条错误消息。还要注意，因为我们要使代码尽可能简单，所以不实现额外的属性，如账户持有人的姓名。在现实生活中，这是最基本的信息，但对于本例不必要这么复杂。

在这段代码中，唯一有趣的一行是类的声明：

```
public class SaverAccount: IBankAccount
```

SaverAccount 派生自一个接口 IBankAccount，我们没有明确指出任何其他基类(当然这表示 SaverAccount 直接派生自 System.Object)。另外，从接口中派生完全独立于从类中派生。

SaverAccount 派生自 IBankAccount，表示它获得了 IBankAccount 的所有成员，但接口实际上并不实现其方法，所以 SaverAccount 必须提供这些方法的所有实现代码。如果缺少实现代码，编译器就会产生错误。接口仅表示其成员的存在性，类负责确定这些成员是虚拟还是抽象的(但只有在类本身是抽象的，这些函数才能是抽象的)。在本例中，接口的任何函数不必是虚拟的。

为了说明不同的类如何实现相同的接口，下面假定 Planetary Bank of Jupiter 还实现一个类 GoldAccount 来表示其银行账户中的一个：

```
namespaceWrox.ProCSharp.JupiterBank
{
  public class GoldAccount: IBankAccount
  {
```

```
    // etc
  }
}
```

这里没有列出 GoldAccount 类的细节，因为在本例中它基本上与 SaverAccount 的实现代码相同。GoldAccount 与 SaverAccount 没有关系，它们只是碰巧实现相同的接口而已。

有了自己的类后，就可以测试它们了。首先需要一些 using 语句：

```
using System;
usingWrox.ProCSharp;
usingWrox.ProCSharp.VenusBank;
usingWrox.ProCSharp.JupiterBank;
```

然后需要一个 Main()方法：

```
namespaceWrox.ProCSharp
{
  classMainEntryPoint
  {
    static void Main()
    {
      IBankAccountvenusAccount = new SaverAccount();
      IBankAccountjupiterAccount = new GoldAccount();
      venusAccount.PayIn(200);
      venusAccount.Withdraw(100);
      Console.WriteLine(venusAccount.ToString());
      jupiterAccount.PayIn(500);
      jupiterAccount.Withdraw(600);
      jupiterAccount.Withdraw(100);
      Console.WriteLine(jupiterAccount.ToString());
    }
  }
}
```

这段代码(如果下载本例子，就会发现它在 BankAccounts.cs 文件中)的执行结果如下：

```
C:>BankAccounts
Venus Bank Saver: Balance = £100.00
Withdrawal attempt failed.
Jupiter Bank Saver: Balance = £400.00
```

在这段代码中，要点是把两个引用变量声明为 IBankAccount 引用的方式。这表示它们可以指向实现这个接口的任何类的任何实例。但我们只能通过这些引用调用接口的一部分方法——如果要调用由类实现的但不在接口中的方法，就需要把引用强制转换为合适的类型。在这段代码中，我们调用了 ToString()(不是 IBankAccount 实现的)，但没有进行任何显式的强制转换，这只是因为 ToString()是一个 System.Object 方法，因此 C#编译器知道任何类都支持这个方法(换言之，从任何接口到 System.Object 的数据类型强制转换是隐式的)。第 7 章将介绍强制转换的语法。

接口引用完全可以看成类引用——但接口引用的强大之处在于，它可以引用任何实现该接口的类。例如，我们可以构造接口数组，其中数组的每个元素都是不同的类：

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
```

```
accounts[1] = new GoldAccount();
```

但注意，如果编写了如下代码，就会生成一个编译错误：

```
accounts[1] = new SomeOtherClass();    // SomeOtherClass does NOT implement
                                       // IBankAccount: WRONG!!
```

这会导致一个如下所示的编译错误：

```
Cannot implicitly convert type 'Wrox.ProCSharp. SomeOtherClass' to
  'Wrox.ProCSharp.IBankAccount'
```

4.5.2 派生的接口

接口可以彼此继承，其方式与类的继承方式相同。下面通过定义一个新的接口ITransferBankAccount来说明这个概念，该接口的功能与IBankAccount相同，只是又定义了一个方法，把资金直接转到另一个账户上。

```
namespaceWrox.ProCSharp
{
  public interface ITransferBankAccount: IBankAccount
  {
     boolTransferTo(IBankAccount destination, decimal amount);
  }
}
```

因为ITransferBankAccount派生自IBankAccount，所以它拥有IBankAccount的所有成员和它自己的成员。这表示实现(派生自)ITransferBankAccount的任何类都必须实现IBankAccount的所有方法和在ITransferBankAccount中定义的新方法TransferTo()。没有实现所有这些方法就会产生一个编译错误。

注意，TransferTo()方法对于目标账户使用了IBankAccount接口引用。这说明了接口的用途：在实现并调用这个方法时，不必知道转账的对象类型，只需知道该对象实现IBankAccount即可。

下面说明ITransferBankAccount：假定Planetary Bank of Jupiter还提供了一个当前账户。CurrentAccount类的大多数实现代码与SaverAccount和GoldAccount的实现代码相同(这仅是为了使例子更简单，一般是不会这样的)，所以在下面的代码中，我们仅突出显示了不同的地方：

```
public class CurrentAccount: ITransferBankAccount
{
  private decimal balance;
  public void PayIn(decimal amount)
  {
     balance += amount;
  }
  publicbool Withdraw(decimal amount)
  {
    if (balance >= amount)
    {
      balance -= amount;
      return true;
    }
    Console.WriteLine("Withdrawal attempt failed.");
    return false;
```

```
    }
    public decimal Balance
    {
      get
      {
          return balance;
      }
    }
    publicboolTransferTo(IBankAccount destination, decimal amount)
    {
      bool result;
      result = Withdraw(amount);
      if (result)
      {
          destination.PayIn(amount);
      }
      return result;
    }
    public override string ToString()
    {
       returnString.Format("Jupiter Bank Current Account: Balance = {0,6:C}",balance);
    }
}
```

可以用下面的代码验证该类：

```
static void Main()
{
  IBankAccountvenusAccount = new SaverAccount();
  ITransferBankAccountjupiterAccount = new CurrentAccount();
  venusAccount.PayIn(200);
  jupiterAccount.PayIn(500);
  jupiterAccount.TransferTo(venusAccount, 100);
  Console.WriteLine(venusAccount.ToString());
  Console.WriteLine(jupiterAccount.ToString());
}
```

这段代码(CurrentAccounts.cs)的结果如下所示，可以验证，其中说明了正确的转账金额：

```
C:>CurrentAccount
Venus Bank Saver: Balance = £300.00
Jupiter Bank Current Account: Balance = £400.00
```

4.6 小结

本章介绍了如何在 C#中进行继承。C#支持多接口继承和单一实现继承，还提供了许多有用的语法结构，以使代码更健壮，如 override 关键字，它表示函数应在何时重写基类函数，new 关键字表示函数在何时隐藏基类函数，构造函数初始化器的硬性规则可以确保构造函数以健壮的方式进行交互操作。

第 5 章

泛　　型

本章要点

- 泛型概述
- 创建泛型类
- 泛型类的特性
- 泛型接口
- 泛型结构
- 泛型方法

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 链表对象
- 链表示例
- 文档管理器
- Variance
- 泛型方法
- 专用

5.1 泛型概述

.NET 自从 2.0 版本开始就支持泛型。泛型不仅是 C#编程语言的一部分，而且与程序集中的IL(Intermediate Language，中间语言)代码紧密地集成。有了泛型，就可以创建独立于被包含类型的类和方法了。我们不必给不同的类型编写功能相同的许多方法或类，只创建一个方法或类即可。

另一个减少代码的选项是使用 Object 类，但 Object 类不是类型安全的。泛型类使用泛型类型，并可以根据需要用特定的类型替换泛型类型。这就保证了类型安全性：如果某个类型不支持泛型类，

编译器就会出现错误。

泛型不仅限于类，本章还将介绍用于接口和方法的泛型。用于委托的泛型参见第8章。

泛型并不是一个全新的结构，其他语言中有类似的概念。例如，C++模板就与泛型相似。但是，C++模板和.NET 泛型之间有一个很大的区别。对于 C++模板，在用特定的类型实例化模板时，需要模板的源代码。相反，泛型不仅是C#语言的一种结构，而且是CLR 定义的。所以，即使泛型类是在C#中定义的，也可以在 Visual Basic 中用一个特定的类型实例化该泛型。

下面几节介绍泛型的优点和缺点，尤其是：

- 性能
- 类型安全性
- 二进制代码重用
- 代码的扩展
- 命名约定

5.1.1 性能

泛型的一个主要优点是性能。第10章介绍了 System.Collections 和 System.Collections.Generic 名称空间的泛型和非泛型集合类。对值类型使用非泛型集合类，在把值类型转换为引用类型，和把引用类型转换为值类型时，需要进行装箱和拆箱操作。

装箱和拆箱详见第7章，这里仅简要复习一下这些术语。

值类型存储在栈上，引用类型存储在堆上。C#类是引用类型，结构是值类型。.NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，int可以赋予一个对象。从值类型转换为引用类型称为装箱。如果方法需要把一个对象作为参数，同时传递一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以使用拆箱操作转换为值类型。在拆箱时，需要使用类型强制转换运算符。

下面的例子显示了 System.Collections 名称空间中的 ArrayList 类。ArrayList 存储对象，Add()方法定义为需要把一个对象作为参数，所以要装箱一个整数类型。在读取 ArrayList 中的值时，要进行拆箱，把对象转换为整数类型。可以使用类型强制转换运算符把 ArrayList 集合的第一个元素赋予变量 i1，在访问 int 类型的变量 i2 的 foreach 语句中，也要使用类型强制转换运算符：

```
var list = new ArrayList();
list.Add(44); // boxing - convert a value type to a reference type

int i1 = (int)list[0];  // unboxing - convert a reference type to
                        // a value type
foreach (int i2 in list)
{
   Console.WriteLine(i2); // unboxing
}
```

装箱和拆箱操作很容易使用，但性能损失比较大，遍历许多项时尤其如此。

System.Collections.Generic 名称空间中的 List<T>类不使用对象，而是在使用时定义类型。在下

面的例子中，List<T>类的泛型类型定义为 int，所以 int 类型在 JIT 编译器动态生成的类中使用，不再进行装箱和拆箱操作：

```
var list = new List<int>();
list.Add(44); // no boxing — value types are stored in the List<int>

int i1 = list[0]; // no unboxing, no cast needed

foreach (int i2 in list)
{
   Console.WriteLine(i2);
}
```

5.1.2 类型安全

泛型的另一个特性是类型安全。与 ArrayList 类一样，如果使用对象，就可以在这个集合中添加任意类型。下面的例子在 ArrayList 类型的集合中添加一个整数、一个字符串和一个 MyClass 类型的对象：

```
var list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

如果这个集合使用下面的 foreach 语句迭代，而该 foreach 语句使用整数元素来迭代，编译器就会编译这段代码。但并不是集合中的所有元素都可以强制转换为 int，所以会出现一个运行异常：

```
foreach (int i in list)
{
   Console.WriteLine(i);
}
```

错误应尽早发现。在泛型类 List<T>中，泛型类型 T 定义了允许使用的类型。有了 List<int>的定义，就只能把整数类型添加到集合中。编译器不会编译这段代码，因为 Add()方法的参数无效：

```
var list = new List<int>();
list.Add(44);
list.Add("mystring"); // compile time error
list.Add(new MyClass()); // compile time error
```

5.1.3 二进制代码的重用

泛型允许更好地重用二进制代码。泛型类可以定义一次，并且可以用许多不同的类型实例化。不需要像 C++模板那样访问源代码。

例如，System.Collections.Generic 名称空间中的 List<T>类用一个 int、一个字符串和一个 MyClass 类型实例化：

```
var list = new List<int>();
list.Add(44);

var stringList = new List<string>();
stringList.Add("mystring");

var myClassList = new List<MyClass>();
```

```
myClassList.Add(new MyClass());
```

泛型类型可以在一种语言中定义，在任何其他.NET 语言中使用。

5.1.4 代码的扩展

在用不同的特定类型实例化泛型时，会创建多少代码？因为泛型类的定义会放在程序集中，所以用特定类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JIT 编译器把泛型类编译为本地代码时，会给每个值类型创建一个新类。引用类型共享同一个本地类的所有相同的实现代码。这是因为引用类型在实例化的泛型类中只需要 4 个字节的内存地址(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中，同时因为每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

5.1.5 命名约定

如果在程序中使用泛型，在区分泛型类型和非泛型类型时就会有一定的帮助。下面是泛型类型的命名规则：

- 泛型类型的名称用字母 T 作为前缀。
- 如果没有特殊的要求，泛型类型允许用任意类替代，且只使用了一个泛型类型，就可以用字符 T 作为泛型类型的名称。

```
public class List<T> { }

public class LinkedList<T> { }
```

- 如果泛型类型有特定的要求(例如，它必须实现一个接口或派生自基类)，或者使用了两个或多个泛型类型，就应给泛型类型使用描述性的名称：

```
public delegate void EventHandler<TEventArgs>(object sender,
  TEventArgs e);

public delegate TOutput Converter<TInput, TOutput>(TInput from);

public class SortedList<TKey, TValue> { }
```

5.2 创建泛型类

首先介绍一个一般的、非泛型的简化链表类，它可以包含任意类型的对象，以后再把这个类转化为泛型类。

在链表中，一个元素引用下一个元素。所以必须创建一个类，它将对象封装在链表中，并引用下一个对象。类 LinkedListNode 包含一个属性 Value，该属性用构造函数初始化。另外，LinkedListNode 类包含对链表中下一个元素和上一个元素的引用，这些元素都可以从属性中访问(代码文件 LinkedListObjects/LinkedListNode.cs)。

```
public class LinkedListNode
{
  public LinkedListNode(object value)
```

```
    {
      this.Value = value;
    }

    public object Value { get; private set; }

    public LinkedListNode Next { get; internal set; }
    public LinkedListNode Prev { get; internal set; }
  }
```

LinkedList 类包含 LinkedListNode 类型的 First 和 Last 属性，它们分别标记了链表的头尾。AddLast()方法在链表尾添加一个新元素。首先创建一个 LinkedListNode 类型的对象。如果链表是空的，First 和 Last 属性就设置为该新元素；否则，就把新元素添加为链表中的最后一个元素。通过实现 GetEnumerator()方法，可以用 foreach 语句遍历链表。GetEnumerator()方法使用 yield 语句创建一个枚举器类型。

```
  public class LinkedList: IEnumerable
  {
    public LinkedListNode First { get; private set; }
    public LinkedListNode Last { get; private set; }

    public LinkedListNode AddLast(object node)
    {
      var newNode = new LinkedListNode(node);
      if (First == null)
      {
        First = newNode;
        Last = First;
      }
      else
      {
        LinkedListNode previous = Last;
        Last.Next = newNode;
        Last = newNode;
        Last.Prev = previous;
      }
      return newNode;
    }

    public IEnumerator GetEnumerator()
    {
      LinkedListNode current = First;
      while (current != null)
      {
        yield return current.Value;
        current = current.Next;
      }
    }
  }
```

yield 语句创建一个枚举器的状态机，详细介绍请参见第 6 章。

现在可以对于任意类型使用 LinkedList 类了。在下面的代码段中，实例化了一个新 LinkedList 对象，添加了两个整数类型和一个字符串类型。整数类型要转换为一个对象，所以执行装箱操作，如前面所述。通过 foreach 语句执行拆箱操作。在 foreach 语句中，链表中的元素被强制转换为整数，所以对于链表中的第 3 个元素，会发生一个运行异常，因为把它强制转换为 int 时会失败(代码文件 LinkedLisObjects/Program.cs)。

```
var list1 = new LinkedList();
list1.AddLast(2);
list1.AddLast(4);
list1.AddLast("6");

foreach (int i in list1)
{
  Console.WriteLine(i);
}
```

下面创建链表的泛型版本。泛型类的定义与一般类类似，只是要使用泛型类型声明。之后，泛型类型就可以在类中用作一个字段成员，或者方法的参数类型。LinkedListNode 类用一个泛型类型 T 声明。属性 Value 的类型是 T，而不是 object。构造函数也变为可以接受 T 类型的对象。也可以返回和设置泛型类型，所以属性 Next 和 Prev 的类型是 LinkedListNode<T>(代码文件 LinkedListSample/LinkedListNode.cs)。

```
public class LinkedListNode<T>
{
  public LinkedListNode(T value)
  {
    this.Value = value;
  }

  public T Value { get; private set; }
  public LinkedListNode<T> Next { get; internal set; }
  public LinkedListNode<T> Prev { get; internal set; }
}
```

下面的代码把 LinkedList 类也改为泛型类。LinkedList<T>包含 LinkedListNode<T>元素。LinkedList 中的类型 T 定义了类型 T 的属性 First 和 Last。AddLast()方法现在接受类型 T 的参数，并实例化 LinkedListNode<T>类型的对象。

除了 IEnumerable 接口，还有一个泛型版本 IEnumerable<T>。IEnumerable<T>派生自 IEnumerable，添加了返回 IEnumerator<T>的 GetEnumerator()方法，LinkedList<T>实现泛型接口 IEnumerable<T>(代码文件 LinkedListSample/LinkedList.cs)。

枚举与接口 IEnumerable 和 IEnumerator 详见第 6 章。

```
public class LinkedList<T>: IEnumerable<T>
{
```

```
    public LinkedListNode<T> First { get; private set; }
    public LinkedListNode<T> Last { get; private set; }

    public LinkedListNode<T> AddLast(T node)
    {
      var newNode = new LinkedListNode<T>(node);
      if (First == null)
      {
        First = newNode;
        Last = First;
      }
      else
      {
        LinkedListNode<T> previous = Last;
        Last.Next = newNode;
        Last = newNode;
        Last.Prev = previous;
      }
      return newNode;
    }

    public IEnumerator<T> GetEnumerator()
    {
      LinkedListNode<T> current = First;

      while (current != null)
      {
        yield return current.Value;
        current = current.Next;
      }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
      return GetEnumerator();
    }
  }
```

使用泛型 LinkedList<T>，可以用 int 类型实例化它，且无须装箱操作。如果不使用 AddLast()方法传递 int，就会出现一个编译错误。使用泛型 IEnumerable<T>，foreach 语句也是类型安全的，如果 foreach 语句中的变量不是 int，就会出现一个编译错误(代码文件 LinkedListSample/Program.cs)。

```
  var list2 = new LinkedList<int>();
  list2.AddLast(1);
  list2.AddLast(3);
  list2.AddLast(5);

  foreach (int i in list2)
  {
    Console.WriteLine(i);
  }
```

同样，可以对于字符串类型使用泛型 LinkedList<T>，将字符串传递给 AddLast()方法。

```
var list3 = new LinkedList<string>();
list3.AddLast("2");
list3.AddLast("four");
list3.AddLast("foo");

foreach (string s in list3)
{
  Console.WriteLine(s);
}
```

每个处理对象类型的类都可以有泛型实现方式。另外，如果类使用了层次结构，泛型就非常有助于消除类型强制转换操作。

5.3 泛型类的功能

在创建泛型类时，还需要一些其他C#关键字。例如，不能把null赋予泛型类型。此时，如下一节所述，可以使用default关键字。如果泛型类型不需要Object类的功能，但需要调用泛型类上的某些特定方法，就可以定义约束。

本节讨论如下主题：

- 默认值
- 约束
- 继承
- 静态成员

首先介绍一个使用泛型文档管理器的示例。文档管理器用于从队列中读写文档。先创建一个新的控制台项目DocumentManager，并添加DocumentManager<T>类。AddDocument()方法将一个文档添加到队列中。如果队列不为空，IsDocumentAvailable只读属性就返回true(代码文件DocumentManager/DocumentManager.cs)。

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

namespace Wrox.ProCSharp.Generics
{
  public class DocumentManager<T>
  {
    private readonly Queue<T> documentQueue = new Queue<T>();

    public void AddDocument(T doc)
    {
      lock (this)
      {
        documentQueue.Enqueue(doc);
      }
    }
```

```
    public bool IsDocumentAvailable
    {
      get { return documentQueue.Count > 0; }
    }
  }
}
```

第21章将讨论线程和lock语句。

5.3.1　默认值

现在给DocumentManager<T>类添加一个GetDocument()方法。在这个方法中，应把类型T指定为null。但是，不能把null赋予泛型类型。原因是泛型类型也可以实例化为值类型，而null只能用于引用类型。为了解决这个问题，可以使用default关键字。通过default关键字，将null赋予引用类型，将0赋予值类型。

```
public T GetDocument()
{
  T doc = default(T);
  lock (this)
  {
    doc = documentQueue.Dequeue();
  }
  return doc;
}
```

default关键字根据上下文可以有多种含义。switch语句使用default定义默认情况。在泛型中，根据泛型类型是引用类型还是值类型，泛型default用于将泛型类型初始化为null或0。

5.3.2　约束

如果泛型类需要调用泛型类型中的方法，就必须添加约束。

对于DocumentManager<T>，文档的所有标题应在DisplayAllDocuments()方法中显示。Document类实现带有Title和Content属性的IDocument接口(代码文件DocumentManager/Document.cs)：

```
public interface IDocument
{
  string Title { get; set; }
  string Content { get; set; }
}

public class Document: IDocument
{
  public Document()
  {
  }

  public Document(string title, string content)
  {
```

```
    this.Title = title;
    this.Content = content;
  }

  public string Title { get; set; }
  public string Content { get; set; }
}
```

要使用 DocumentManager<T>类显示文档，可以将类型 T 强制转换为 IDocument 接口，以显示标题(代码文件 DocumentManager/DocumentManager.cs)：

```
public void DisplayAllDocuments()
{
  foreach (T doc in documentQueue)
  {
    Console.WriteLine(((IDocument)doc).Title);
  }
}
```

问题是，如果类型 T 没有实现 IDocument 接口，这个类型强制转换就会导致一个运行异常。最好给 DocumentManager<TDocument>类定义一个约束：TDocument 类型必须实现 IDocument 接口。为了在泛型类型的名称中指定该要求，将 T 改为 TDocument。where 子句指定了实现 IDocument 接口的要求。

```
public class DocumentManager<TDocument>
    where TDocument: IDocument
{
```

这样就可以编写 foreach 语句，从而使类型 TDocument 包含属性 Title。Visual Studio IntelliSense 和编译器都会提供这个支持。

```
public void DisplayAllDocuments()
{
  foreach (TDocument doc in documentQueue)
  {
    Console.WriteLine(doc.Title);
  }
}
```

在 Main()方法中，用 Document 类型实例化 DocumentManager<T>类，而 Document 类型实现了需要的 IDocument 接口。接着添加和显示新文档，检索其中一个文档(代码文件 DocumentManager/Program.cs)：

```
static void Main()
{
  var dm = new DocumentManager<Document>();
  dm.AddDocument(new Document("Title A", "Sample A"));
  dm.AddDocument(new Document("Title B", "Sample B"));

  dm.DisplayAllDocuments();

  if (dm.IsDocumentAvailable)
  {
```

```
        Document d = dm.GetDocument();
        Console.WriteLine(d.Content);
      }
    }
```

DocumentManager 现在可以处理任何实现了 IDocument 接口的类。

在示例应用程序中，介绍了接口约束。泛型支持几种约束类型，如表 5-1 所示。

表 5-1

约　束	说　明
where T : struct	对于结构约束，类型 T 必须是值类型
where T : class	类约束指定类型 T 必须是引用类型
where T : IFoo	指定类型 T 必须实现接口 IFoo
where T : Foo	指定类型 T 必须派生自基类 Foo
where T : new()	这是一个构造函数约束，指定类型 T 必须有一个默认构造函数
where T1 : T2	这个约束也可以指定，类型 T1 派生自泛型类型 T2。该约束也称为裸类型约束

只能为默认构造函数定义构造函数约束，不能为其他构造函数定义构造函数约束。

使用泛型类型还可以合并多个约束。where T : IFoo, new()约束和 MyClass<T>声明指定，类型 T 必须实现 IFoo 接口，且必须有一个默认构造函数。

```
public class MyClass<T>
    where T: IFoo, new()
{
//...
```

在 C#中，where 子句的一个重要限制是，不能定义必须由泛型类型实现的运算符。运算符不能在接口中定义。在 where 子句中，只能定义基类、接口和默认构造函数。

5.3.3 继承

前面创建的 LinkedList<T>类实现了 IEnumerable<T>接口：

```
public class LinkedList<T>: IEnumerable<T>
{
    //...
```

泛型类型可以实现泛型接口，也可以派生自一个类。泛型类可以派生自泛型基类：

```
public class Base<T>
{
}
```

```
public class Derived<T>: Base<T>
{
}
```

其要求是必须重复接口的泛型类型，或者必须指定基类的类型，如下例所示：

```
public class Base<T>
{
}

public class Derived<T>: Base<string>
{
}
```

于是，派生类可以是泛型类或非泛型类。例如，可以定义一个抽象的泛型基类，它在派生类中用一个具体的类型实现。这允许对特定类型执行特殊的操作：

```
public abstract class Calc<T>
{
  public abstract T Add(T x, T y);
  public abstract T Sub(T x, T y);
}

public class IntCalc: Calc<int>
{
  public override int Add(int x, int y)
  {
    return x + y;
  }

  public override int Sub(int x, int y)
  {
    return x - y;
  }
}
```

5.3.4 静态成员

泛型类的静态成员需要特别关注。泛型类的静态成员只能在类的一个实例中共享。下面看一个例子，其中 StaticDemo<T>类包含静态字段 x：

```
public class StaticDemo<T>
{
  public static int x;
}
```

由于同时对一个 string 类型和一个 int 类型使用了 StaticDemo<T>类，所以存在两组静态字段：

```
StaticDemo<string>.x = 4;
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // writes 4
```

5.4　泛型接口

使用泛型可以定义接口，在接口中定义的方法可以带泛型参数。在链表的示例中，就实现了IEnumerable<out T>接口，它定义了 GetEnumerator()方法，以返回 IEnumerator<out T>。.NET 为不同的情况提供了许多泛型接口，例如 IComparable<T>、ICollection<T>和 IExtensibleObject<T>。同一个接口常常存在比较老的非泛型版本，例如，.NET 1.0 有基于对象的 IComparable 接口。IComparable<in T>基于一个泛型类型：

```
public interface IComparable<in T>

{
  int CompareTo(T other);
}
```

比较老的非泛型接口 IComparable 需要一个带 CompareTo()方法的对象。这需要强制转换为特定的类型，例如，Person 类要使用 LastName 属性，就需要使用 CompareTo()方法：

```
public class Person: IComparable
{
  public int CompareTo(object obj)
  {
    Person other = obj as Person;
    return this.lastname.CompareTo(other.LastName);
  }
  //
```

实现泛型版本时，不再需要将 object 的类型强制转换为 Person：

```
public class Person: IComparable<Person>
{
  public int CompareTo(Person other)
  {
    return this.LastName.CompareTo(other.LastName);
  }
  //...
```

5.4.1　协变和抗变

在.NET 4 之前，泛型接口是不变的。.NET 4 通过协变和抗变为泛型接口和泛型委托添加了一个重要的扩展。协变和抗变指对参数和返回值的类型进行转换。例如，可以给一个需要 Shape 参数的方法传送 Rectangle 参数吗？下面用示例说明这些扩展的优点。

在.NET 中，参数类型是协变的。假定有 Shape 和 Rectangle 类，Rectangle 派生自 Shape 基类。声明 Display()方法是为了接受 Shape 类型的对象作为其参数：

```
public void Display(Shape o) { }
```

现在可以传递派生自 Shape 基类的任意对象。因为 Rectangle 派生自 Shape，所以 Rectangle 满足 Shape 的所有要求，编译器接受这个方法调用：

```
var r = new Rectangle { Width= 5, Height=2.5 };
```

```
Display(r);
```

方法的返回类型是抗变的。当方法返回一个 Shape 时，不能把它赋予 Rectangle，因为 Shape 不一定总是 Rectangle。反过来是可行的：如果一个方法像 GetRectangle()方法那样返回一个 Rectangle，

```
public Rectangle GetRectangle();
```

就可以把结果赋予某个 Shape：

```
Shape s = GetRectangle();
```

在.NET Framework 4 版本之前，这种行为方式不适用于泛型。在 C# 4 中，扩展后的语言支持泛型接口和泛型委托的协变和抗变。下面开始定义 Shape 基类和 Rectangle 类(代码文件 Variance/Shape.cs 和 Rectangle.cs)：

```
public class Shape
{
  public double Width { get; set; }
  public double Height { get; set; }

  public override string ToString()
  {
    return String.Format("Width: {0}, Height: {1}", Width, Height);
  }
}

public class Rectangle: Shape
{
}
```

5.4.2 泛型接口的协变

如果泛型类型用 out 关键字标注，泛型接口就是协变的。这也意味着返回类型只能是 T。接口 IIndex 与类型 T 是协变的，并从一个只读索引器中返回这个类型(代码文件 Variance/IIndex.cs)：

```
public interface IIndex<out T>
{
  T this[int index] { get; }
  int Count { get; }
}
```

IIndex<T>接口用 RectangleCollection 类来实现。RectangleCollection 类为泛型类型 T 定义了 Rectangle：

如果对接口 IIndex 使用了读写索引器，就把泛型类型 T 传递给方法，并从方法中检索这个类型。这不能通过协变来实现——泛型类型必须定义为不变的。不使用 out 和 in 标注，就可以把类型定义为不变的(代码文件 Variance/RectangleCollection)。

```
public class RectangleCollection: IIndex<Rectangle>
{
```

```
    private Rectangle[] data = new Rectangle[3]
    {
      new Rectangle { Height=2, Width=5 },
      new Rectangle { Height=3, Width=7 },
      new Rectangle { Height=4.5, Width=2.9 }
    };

    private static RectangleCollection coll;
    public static RectangleCollection GetRectangles()
    {
      return coll ?? (coll = new RectangleCollection());
    }

    public Rectangle this[int index]
    {
      get
      {
        if (index < 0 || index > data.Length)
          throw new ArgumentOutOfRangeException("index");
        return data[index];
      }
    }
    public int Count
    {
      get
      {
        return data.Length;
      }
    }
  }
```

RectangleCollection.GetRectangles()方法使用了本章后面将会介绍的合并运算符(coalescing operator)。如果变量 coll 为 null，那么将会调用运算符的右侧，以创建 RectangleCollection 的一个新实例，并将其赋给变量 coll。之后，会从 GetRectangles()方法中返回变量 coll。

RectangleCollection.GetRectangle()方法返回一个实现 IIndex<Rectangle>接口的 RectangleCollection 类，所以可以把返回值赋予 IIndex<Rectangle>类型的变量 rectangle。因为接口是协变的，所以也可以把返回值赋予 IIndex<Shape>类型的变量。Shape 不需要 Rectangle 没有提供的内容。使用 shapes 变量，就可以在 for 循环中使用接口中的索引器和 Count 属性(代码文件 Variance/Program.cs)：

```
static void Main()
{
  IIndex<Rectangle> rectangles = RectangleCollection.GetRectangles();
  IIndex<Shape> shapes = rectangles;

  for (int i = 0; i < shapes.Count; i++)
  {
    Console.WriteLine(shapes[i]);
```

```
    }
}
```

5.4.3 泛型接口的抗变

如果泛型类型用 in 关键字标注，泛型接口就是抗变的。这样，接口只能把泛型类型 T 用作其方法的输入(代码文件 Variance/IDisplay.cs)：

```
public interface IDisplay<in T>
{
  void Show(T item);
}
```

ShapeDisplay 类实现 IDisplay<Shape>，并使用 Shape 对象作为输入参数(代码文件 Variance/ShapeDisplay.cs)：

```
public class ShapeDisplay: IDisplay<Shape>
{
  public void Show(Shape s)
  {
    Console.WriteLine("{0} Width: {1}, Height: {2}", s.GetType().Name,
                      s.Width, s.Height);
  }
}
```

创建 ShapeDisplay 的一个新实例，会返回 IDisplay<Shape>，并把它赋予 shapeDisplay 变量。因为 IDisplay<T>是抗变的，所以可以把结果赋予 IDisplay<Rectangle>，其中 Rectangle 派生自 Shape。这次接口的方法只能把泛型类型定义为输入，而 Rectangle 满足 Shape 的所有要求(代码文件 Variance/Program.cs)：

```
static void Main()
{
  //...

  IDisplay<Shape> shapeDisplay = new ShapeDisplay();
  IDisplay<Rectangle> rectangleDisplay = shapeDisplay;
  rectangleDisplay.Show(rectangles[0]);

}
```

5.5 泛型结构

与类相似，结构也可以是泛型的。它们非常类似于泛型类，只是没有继承特性。本节介绍泛型结构 Nullable<T>，它由.NET Framework 定义。

.NET Framework 中的一个泛型结构是 Nullable<T>。数据库中的数字和编程语言中的数字有显著不同的特征，因为数据库中的数字可以为空，而 C#中的数字不能为空。Int32 是一个结构，而结构的实现同值类型，所以结构不能为空。这种区别常常令人很头痛，映射数据也要多做许多辅助工作。这个问题不仅存在于数据库中，也存在于把 XML 数据映射到.NET 类型。

一种解决方案是把数据库和 XML 文件中的数字映射为引用类型，因为引用类型可以为空值。但这也会在运行期间带来额外的系统开销。

使用 Nullable<T>结构很容易解决这个问题。下面的代码段说明了如何定义 Nullable<T>的一个简化版本。结构 Nullable<T>定义了一个约束：其中的泛型类型 T 必须是一个结构。把类定义为泛型类型后，就没有低系统开销这个优点了，而且因为类的对象可以为空，所以对类使用 Nullable<T>类型是没有意义的。除了 Nullable<T>定义的 T 类型之外，唯一的系统开销是 hasValue 布尔字段，它确定是设置对应的值，还是使之为空。除此之外，泛型结构还定义了只读属性 HasValue 和 Value，以及一些操作符重载。把 Nullable<T>类型强制转换为 T 类型的操作符重载是显式定义的，因为当 hasValue 为 false 时，它会抛出一个异常。强制转换为 Nullable<T>类型的操作符重载定义为隐式的，因为它总是能成功地转换：

```
public struct Nullable<T>
   where T: struct
{
  public Nullable(T value)
  {
    this.hasValue = true;
    this.value = value;
  }
  private bool hasValue;
  public bool HasValue
  {
    get
    {
      return hasValue;
    }
  }

  private T value;
  public T Value
  {
    get
    {
      if (!hasValue)
      {
        throw new InvalidOperationException("no value");
      }
      return value;
    }
  }

  public static explicit operator T(Nullable<T> value)
  {
    return value.Value;
  }
  publicstatic implicit operator Nullable<T>(T value)
  {
    return new Nullable<T>(value);
  }

  public override string ToString()
```

```
    {
      if (!HasValue)
        return String.Empty;
      return this.value.ToString();
    }
  }
```

在这个例子中，Nullable<T>用 Nullable<int>实例化。变量 x 现在可以用作一个 int，进行赋值或使用运算符执行一些计算。这是因为强制转换了 Nullable<T>类型的运算符。但是，x 还可以为空。Nullable<T>的 HasValue 和 Value 属性可以检查是否有一个值，该值是否可以访问：

```
Nullable<int> x;
x = 4;
x += 3;
if (x.HasValue)
{
  int y = x.Value;
}
x = null;
```

因为可空类型使用得非常频繁，所以 C#有一种特殊的语法，它用于定义可空类型的变量。定义这类变量时，不使用泛型结构的语法，而使用“?”运算符。在下面的例子中，变量 x1 和 x2 都是可空的 int 类型的实例：

```
Nullable<int> x1;
int? x2;
```

可空类型可以与 null 和数字比较，如上所示。这里，x 的值与 null 比较，如果 x 不是 null，它就与小于 0 的值比较：

```
int? x = GetNullableType();
if (x == null)
{
  Console.WriteLine("x is null");
}
else if (x < 0)
{
  Console.WriteLine("x is smaller than 0");
}
```

知道了 Nullable<T>是如何定义的之后，下面就使用可空类型。可空类型还可以与算术运算符一起使用。变量 x3 是变量 x1 和 x2 的和。如果这两个可空变量中任何一个的值是 null，它们的和就是 null。

```
int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;
```

这里调用的 GetNullableType()只是一个占位符，它对于任何方法都返回一个可空的 int。为了进行测试，简单起见，可以使实现的 GetNullableType()返回 null 或返回任意整数。

非可空类型可以转换为可空类型。从非可空类型转换为可空类型时，在不需要强制类型转换的地方可以进行隐式转换。这种转换总是成功的：

```
int y1 = 4;
int? x1 = y1;
```

但从可空类型转换为非可空类型可能会失败。如果可空类型的值是 null，并且把 null 值赋予非可空类型，就会抛出 InvalidOperationException 类型的异常。这就是需要类型强制转换运算符进行显式转换的原因：

```
int? x1 = GetNullableType();
int y1 = (int)x1;
```

如果不进行显式类型转换，还可以使用合并运算符从可空类型转换为非可空类型。合并运算符的语法是“??”，为转换定义了一个默认值，以防可空类型的值是 null。这里，如果 x1 是 null，y1 的值就是 0。

```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

5.6 泛型方法

除了定义泛型类之外，还可以定义泛型方法。在泛型方法中，泛型类型用方法声明来定义。泛型方法可以在非泛型类中定义。

Swap<T>()方法把 T 定义为泛型类型，该泛型类型用于两个参数和一个变量 temp：

```
void Swap<T>(ref T x, ref T y)
{
  T temp;
  temp = x;
  x = y;
  y = temp;
}
```

把泛型类型赋予方法调用，就可以调用泛型方法：

```
int i = 4;
int j = 5;
Swap<int>(ref i, ref j);
```

但是，因为 C#编译器会通过调用 Swap()方法来获取参数的类型，所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用：

```
int i = 4;
int j = 5;
Swap(ref i, ref j);
```

5.6.1 泛型方法示例

下面的例子使用泛型方法累加集合中的所有元素。为了说明泛型方法的功能，下面使用包含

Name 和 Balance 属性的 Account 类(代码文件 GenericMethods/Account.cs)：

```
public class Account
{
  public string Name { get; private set; }
  public decimal Balance { get; private set; }

  public Account(string name, Decimal balance)
  {
    this.Name = name;
    this.Balance = balance;
  }
}
```

其中应累加余额的所有账户操作都添加到 List<Account>类型的账户列表中(代码文件 GenericMethods/Program.cs)：

```
var accounts = new List<Account>()
{
  new Account("Christian", 1500),
  new Account("Stephanie", 2200),
  new Account("Angela", 1800),
  new Account("Matthias", 2400)
};
```

累加所有 Account 对象的传统方式是用 foreach 语句遍历所有的 Account 对象，如下所示。foreach 语句使用 IEnumerable 接口迭代集合的元素，所以 AccumulateSimple()方法的参数是 IEnumerable 类型。foreach 语句处理实现 IEnumerable 接口的每个对象。这样，AccumulateSimple()方法就可以用于所有实现 IEnumerable<Account>接口的集合类。在这个方法的实现代码中，直接访问 Account 对象的 Balance 属性(代码文件 GenericMethods/Algorithm.cs)：

```
public static class Algorithm
{
  public static decimal AccumulateSimple(IEnumerable<Account> source)
  {
    decimal sum = 0;
    foreach (Account a in source)
    {
      sum += a.Balance;
    }
    return sum;
  }
}
```

AccumulateSimple()方法的调用方式如下：

```
decimal amount = Algorithm.AccumulateSimple(accounts);
```

5.6.2 带约束的泛型方法

第一个实现代码的问题是，它只能用于 Account 对象。使用泛型方法就可以避免这个问题。

Accumulate()方法的第二个版本接受实现了 IAccount 接口的任意类型。如前面的泛型类所述，

泛型类型可以用 where 子句来限制。用于泛型类的这个子句也可以用于泛型方法。Accumulate()方法的参数改为 IEnumerable<T>。IEnumerable<T>是泛型集合类实现的泛型接口(代码文件 GenericMethods/ Algorithms.cs)。

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)
    where TAccount: IAccount
{
  decimal sum = 0;

  foreach (TAccount a in source)
  {
    sum += a.Balance;
  }
  return sum;
}
```

重构的 Account 类现在实现接口 IAccount(代码文件 GenericMethods/Account.cs):

```
public class Account: IAccount
{
  //...
```

IAccount 接口定义了只读属性 Balance 和 Name(代码文件 GenericMethods/IAccount.cs):

```
public interface IAccount
{
  decimal Balance { get; }
  string Name { get; }
}
```

将 Account 类型定义为泛型类型参数，就可以调用新的 Accumulate()方法(代码文件 GenericMethods/Program.cs):

```
decimal amount = Algorithm.Accumulate<Account>(accounts);
```

因为编译器会从方法的参数类型中自动推断出泛型类型参数，所以以如下方式调用 Accumulate()方法是有效的：

```
decimal amount = Algorithm.Accumulate(accounts);
```

5.6.3 带委托的泛型方法

泛型类型实现 IAccount 接口的要求过于严格。下面的示例提示了，如何通过传递一个泛型委托来修改 Accumulate()方法。第 8 章详细介绍了如何使用泛型委托，以及如何使用 Lambda 表达式。

这个 Accumulate()方法使用两个泛型参数 T1 和 T2。第一个参数 T1 用于实现了 IEnumerable<T1>参数的集合，第二个参数使用泛型委托 Fun<T1, T2, TResult>。其中，第 2 个和第 3 个泛型参数都是 T2 类型。需要传递的方法有两个输入参数(T1 和 T2)和一个 T2 类型的返回值(代码文件 GenericMethods/Algorithm.cs):

```
public static T2 Accumulate<T1, T2>(IEnumerable<T1> source,
                                    Func<T1, T2, T2> action)
```

```
    T2 sum = default(T2);
    foreach (T1 item in source)
    {
      sum = action(item, sum);
    }
    return sum;
  }
```

在调用这个方法时，需要指定泛型参数类型，因为编译器不能自动推断出该类型。对于方法的第 1 个参数，所赋予的 accounts 集合是 IEnumerable<Account>类型。对于第 2 个参数，使用一个 Lambda 表达式来定义 Account 和 decimal 类型的两个参数，返回一个小数。对于每一项，通过 Accumulate()方法调用这个 Lambda 表达式(代码文件 GenericMethods/Program.cs)：

```
decimal amount = Algorithm.Accumulate<Account, decimal>(
                        accounts, (item, sum) => sum += item.Balance);
```

不要为这种语法伤脑筋。该示例仅说明了扩展 Accumulate()方法的可能方式。第 8 章详细介绍了 Lambda 表达式。

5.6.4 泛型方法规范

泛型方法可以重载，为特定的类型定义规范。这也适用于带泛型参数的方法。Foo()方法定义了两个版本，第 1 个版本接受一个泛型参数，第 2 个版本是用于 int 参数的专用版本。在编译期间，会使用最佳匹配。如果传递了一个 int，就选择带 int 参数的方法。对于任何其他参数类型，编译器会选择方法的泛型版本(代码文件 Specialization/Program.cs)：

```
public class MethodOverloads
{
  public void Foo<T>(T obj)
  {
    Console.WriteLine("Foo<T>(T obj), obj type: {0}", obj.GetType().Name);
  }

  public void Foo(int x)
  {
    Console.WriteLine("Foo(int x)");
  }

  public void Bar<T>(T obj)
  {
    Foo(obj);
  }
}
```

Foo()方法现在可以通过任意参数类型来调用。下面的示例代码给该方法传递了一个 int 和一个 string：

```
static void Main()
{
  var test = new MethodOverloads();
  test.Foo(33);
  test.Foo("abc");
}
```

运行该程序，可以从输出中看出选择了最佳匹配的方法：

```
Foo(int x)
Foo<T>(T obj), obj type: String
```

需要注意的是，所调用的方法是在编译期间定义的，而不是运行期间。这很容易举例说明：添加一个调用 Foo()方法的 Bar()泛型方法，并传递泛型参数值：

```
public class MethodOverloads
{
  // ...

  public void Bar<T>(T obj)
  {
    Foo(obj);
  }
```

Main()方法现在改为调用传递一个 int 值的 Bar()方法：

```
static void Main()
{
  var test = new MethodOverloads();
  test.Bar(44);
```

从控制台的输出可以看出，Bar()方法选择了泛型 Foo()方法，而不是用 int 参数重载的 Foo()方法。原因是编译器是在编译期间选择 Bar()方法调用的 Foo()方法。由于 Bar()方法定义了一个泛型参数，而且泛型 Foo()方法匹配这个类型，所以调用了 Foo()方法。在运行期间给 Bar()方法传递一个 int 值不会改变这一点。

```
Foo<T>(T obj), obj type: Int32
```

5.7 小结

本章介绍了 CLR 中一个非常重要的特性：泛型。通过泛型类可以创建独立于类型的类，泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式。我们介绍了如何实现相应的算法(尤其是操作和谓词)以用于不同的类，而且它们都是类型安全的。泛型委托可以去除集合中的算法。

本书还将探讨泛型的更多特性和用法。第 8 章介绍了常常实现为泛型的委托，第 10 章论述了泛型集合类，第 11 章讨论了泛型扩展方法。第 6 章说明如何对于数组使用泛型方法。

第 6 章

数　组

本章要点

- 简单数组
- 多维数组
- 锯齿数组
- Array 类
- 作为参数的数组
- 枚举
- 元组
- 结构比较

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- SimpleArrays
- SortingSample
- ArraySegment
- YieldDemo
- StructuralComparison

6.1 同一类型和不同类型的多个对象

如果需要使用同一类型的多个对象，就可以使用集合(参见第 10 章)和数组。C#用特殊的记号声明、初始化和使用数组。Array 类在后台发挥作用，它为数组中元素的排序和过滤提供了几个方法。使用枚举器，可以迭代数组中的所有元素。

如果需要使用不同类型的多个对象，可以使用 Tuple(元组)类型。详细内容参见 6.8 节。

6.2　简单数组

如果需要使用同一类型的多个对象，就可以使用数组。数组是一种数据结构，它可以包含同一类型的多个元素。

6.2.1　数组的声明

在声明数组时，应先定义数组中元素的类型，其后是一对空方括号和一个变量名。例如，下面声明了一个包含整型元素的数组：

```
int[]myArray
```

6.2.2　数组的初始化

声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。为此，应使用 new 运算符，指定数组中元素的类型和数量来初始化数组的变量。下面指定了数组的大小。

```
myArray = new int[4];
```

值类型和引用类型请参见第 3 章。

在声明和初始化数组后，变量 myArray 就引用了 4 个整型值，它们位于托管堆上，如图 6-1 所示。

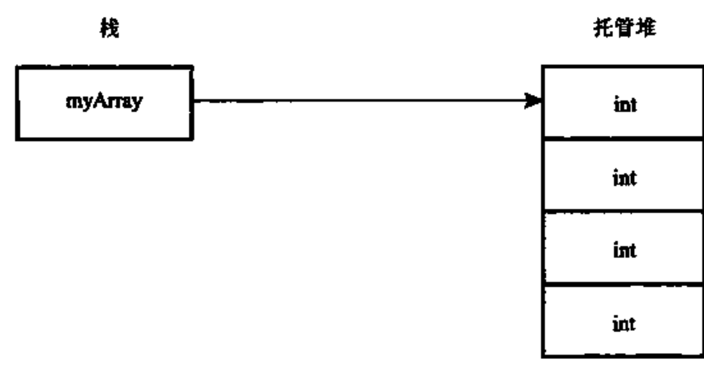


图 6-1

在指定了数组的大小后，如果不复制数组中的所有元素，就不能重新设置数组的大小。如果事先不知道数组中应包含多少个元素，就可以使用集合。集合请参见第 10 章。

除了在两个语句中声明和初始化数组之外，还可以在一个语句中声明和初始化数组：

```
int[] myArray = new int[4];
```

还可以使用数组初始化器为数组的每个元素赋值。数组初始化器只能在声明数组变量时使用，不能在声明数组之后使用。

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

如果用花括号初始化数组，则还可以不指定数组的大小，因为编译器会自动统计元素的个数：

```
int[] myArray = new int[] {4, 7, 11, 2};
```

使用 C#编译器还有一种更简化的形式。使用花括号可以同时声明和初始化数组，编译器生成的代码与前面的例子相同：

```
int[] myArray = {4, 7, 11, 2};
```

6.2.3 访问数组元素

在声明和初始化数组后，就可以使用索引器访问其中的元素了。数组只支持有整型参数的索引器。

通过索引器传递元素编号，就可以访问数组。索引器总是以 0 开头，表示第一个元素。可以传递给索引器的最大值是元素个数减 1，因为索引从 0 开始。在下面的例子中，数组 myArray 用 4 个整型值声明和初始化。用索引器对应的值 0、1、2 和 3 就可以访问该数组中的元素。

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0];  // read first element
int v2 = myArray[1];  // read second element
myArray[3] = 44;      // change fourth element
```

如果使用错误的索引器值(大于数组的长度)，就会抛出 IndexOutOfRangeException 类型的异常。

如果不知道数组中的元素个数，则可以在 for 语句中使用 Length 属性：

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

除了使用 for 语句迭代数组中的所有元素之外，还可以使用 foreach 语句：

```
foreach (var val in myArray)
{
  Console.WriteLine(val);
}
```

foreach 语句利用了本章后面讨论的 IEnumerable 和 IEnumerator 接口。

6.2.4 使用引用类型

除了能声明预定义类型的数组，还可以声明自定义类型的数组。下面用 Person 类来说明，这个类有自动实现的属性 Firstname 和 Lastname，以及从 Object 类重写的 ToString()方法(代码文件 SimpleArrays/Person.cs)：

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public override string ToString()
  {
    return String.Format("{0} {1}", FirstName, LastName);
  }
}
```

声明一个包含两个 Person 元素的数组与声明一个 int 数组类似：

```
Person[] myPersons = new Person[2];
```

但是必须注意，如果数组中的元素是引用类型，就必须为每个数组元素分配内存。若使用了数组中未分配内存的元素，就会抛出 NullReferenceException 类型的异常。

第 16 章介绍了错误和异常的详细内容。

使用从 0 开始的索引器，可以为数组的每个元素分配内存：

```
myPersons[0] = new Person { FirstName="Ayrton", LastName="Senna" };
myPersons[1] = new Person { FirstName="Michael", LastName="Schumacher" };
```

图 6-2 显示了 Person 数组中的对象在托管堆中的情况。myPersons 是存储在栈上的一个变量，该变量引用了存储在托管堆上的 Person 元素对应的数组。这个数组有足够容纳两个引用的空间。数组中的每一项都引用了一个 Person 对象，而这些 Person 对象也存储在托管堆上。

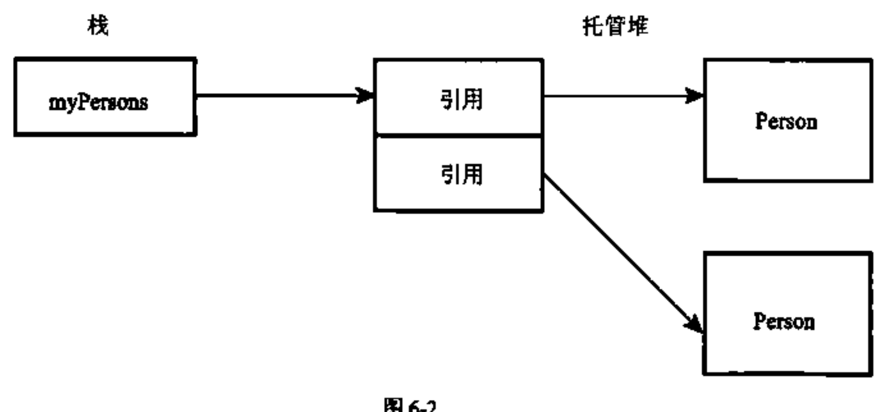


图 6-2

与 int 类型一样，也可以对自定义类型使用数组初始化器：

```
Person[] myPersons2 =
{
  new Person { FirstName="Ayrton", LastName="Senna"},
  new Person { FirstName="Michael", LastName="Schumacher"}
};
```

6.3 多维数组

一般数组(也称为一维数组)用一个整数来索引。多维数组用两个或多个整数来索引。

图6-3是二维数组的数学表示法，该数组有3行3列。第1行的值是1、2和3，第3行的值是7、8和9。

$$a = \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}$$

图6-3

在C#中声明这个二维数组，需要在方括号中加上一个逗号。数组在初始化时应指定每一维的大小(也称为阶)。接着，就可以使用两个整数作为索引器来访问数组中的元素：

```
int[,] twodim = new int[3, 3];
twodim[0, 0] = 1;
twodim[0, 1] = 2;
twodim[0, 2] = 3;
twodim[1, 0] = 4;
twodim[1, 1] = 5;
twodim[1, 2] = 6;
twodim[2, 0] = 7;
twodim[2, 1] = 8;
twodim[2, 2] = 9;
```

声明数组之后，就不能修改其阶数了。

如果事先知道元素的值，则也可以使用数组索引器来初始化二维数组。在初始化数组时，使用一个外层的花括号，每一行用包含在外层花括号中的内层花括号来初始化。

```
int[,] twodim = {
                    {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9}
                };
```

使用数组初始化器时，必须初始化数组的每个元素，不能遗漏任何元素。

在花括号中使用两个逗号，就可以声明一个三维数组：

```
int[,,] threedim = {
            { { 1, 2 }, { 3, 4 } },
            { { 5, 6 }, { 7, 8 } },
```

```
        { { 9, 10 }, { 11, 12 } }
    };

Console.WriteLine(threedim[0, 1, 1]);
```

6.4　锯齿数组

二维数组的大小对应于一个矩形，如对应的元素个数为 3×3。而锯齿数组的大小设置比较灵活，在锯齿数组中，每一行都可以有不同的大小。

图 6-4 比较了有 3×3 个元素的二维数组和锯齿数组。图 6-4 中的锯齿数组有 3 行，第 1 行有两个元素，第 2 行有 6 个元素，第 3 行有 3 个元素。

二维数组

1	2	3
4	5	6
7	8	9

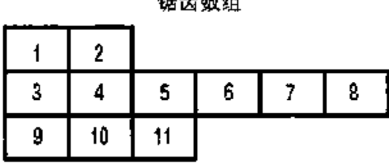


图 6-4

在声明锯齿数组时，要依次放置左右括号。在初始化锯齿数组时，只在第 1 对方括号中设置该数组包含的行数。定义各行中元素个数的第 2 个方括号设置为空，因为这类数组的每一行包含不同的元素个数。之后，为每一行指定行中的元素个数：

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] { 1, 2 };
jagged[1] = new int[6] { 3, 4, 5, 6, 7, 8 };
jagged[2] = new int[3] { 9, 10, 11 };
```

迭代锯齿数组中所有元素的代码可以放在嵌套的 for 循环中。在外层的 for 循环中迭代每一行，在内层的 for 循环中迭代一行中的每个元素：

```
for (int row = 0; row < jagged.Length; row++)
{
  for (int element = 0; element < jagged[row].Length; element++)
  {
    Console.WriteLine("row: {0}, element: {1}, value: {2}", row, element,
                      jagged[row][element]);
  }
}
```

该迭代结果显示了所有的行和每一行中的各个元素：

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
```

```
row: 2, element: 0, value: 9
row: 2, element: 1, value: 10
row: 2, element: 2, value: 11
```

6.5 Array 类

用方括号声明数组是 C#中使用 Array 类的表示法。在后台使用 C#语法，会创建一个派生自抽象基类 Array 的新类。这样，就可以使用 Array 类为每个 C#数组定义的方法和属性了。例如，前面就使用了 Length 属性，或者使用 foreach 语句迭代数组。其实这是使用了 Array 类中的 GetEnumerator()方法。

Array 类实现的其他属性有 LongLength 和 Rank。如果数组包含的元素个数超出了整数的取值范围，就可以使用 LongLength 属性来获得元素个数。使用 Rank 属性可以获得数组的维数。

下面通过了解不同的功能来看看 Array 类的其他成员。

6.5.1 创建数组

Array 类是一个抽象类，所以不能使用构造函数来创建数组。但除了可以使用 C#语法创建数组实例之外，还可以使用静态方法 CreateInstance()创建数组。如果事先不知道元素的类型，该静态方法就非常有用，因为类型可以作为 Type 对象传递给 CreateInstance()方法。

下面的例子说明了如何创建类型为 int、大小为 5 的数组。CreateInstance()方法的第 1 个参数应是元素的类型，第 2 个参数定义数组的大小。可以用 SetValue()方法设置对应元素的值，用 GetValue()方法读取对应元素的值(代码文件 SimpleArrays/Program.cs)：

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
  intArray1.SetValue(33, i);
}

for (int i = 0; i < 5; i++)
{
  Console.WriteLine(intArray1.GetValue(i));
}
```

还可以将已创建的数组强制转换成声明为 int[]的数组：

```
    int[] intArray2 = (int[])intArray1;
```

CreateInstance()方法有许多重载版本，可以创建多维数组和不基于 0 的数组。下面的例子就创建了一个包含 2×3 个元素的二维数组。第一维基于 1，第二维基于 10：

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

SetValue()方法设置数组的元素，其参数是每一维的索引：

```
racers.SetValue(new Person
```

```
{
  FirstName = "Alain",
  LastName = "Prost"
}, index1: 1, index2: 10);
racers.SetValue(new Person
{
  FirstName = "Emerson",
  LastName = "Fittipaldi"
}, 1, 11);
racers.SetValue(new Person
{
  FirstName = "Ayrton",
  LastName = "Senna"
}, 1, 12);
racers.SetValue(new Person
{
  FirstName = "Michael",
  LastName = "Schumacher"
}, 2, 10);
racers.SetValue(new Person
{
  FirstName = "Fernando",
  LastName = "Alonso"
}, 2, 11);
racers.SetValue(new Person
{
  FirstName = "Jenson",
  LastName = "Button"
}, 2, 12);
```

尽管数组不是基于 0，但可以用一般的 C#表示法将它赋予一个变量。只需要注意不要超出边界即可：

```
Person[,] racers2 = (Person[,])racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

6.5.2 复制数组

因为数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个引用同一数组的变量。而复制数组，会使数组实现 ICloneable 接口。这个接口定义的 Clone()方法会创建数组的浅表副本。

如果数组的元素是值类型，以下代码段就会复制所有值，如图 6-5 所示：

```
int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();
```

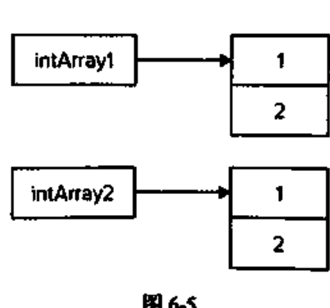


图 6-5

如果数组包含引用类型，则不复制元素，而只复制引用。图 6-6 显示了变量 beatles 和 beatlesClone，其中 beatlesClone 通过从 beatles 中调用 Clone()方法来创建。beatles 和 beatlesClone 引用的 Person 对象是相同的。如果修改 beatlesClone 中一个元

素的属性，就会改变 beatles 中的对应对象(代码文件 SimpleArray/Program.cs)。

```
Person[] beatles = {
                        new Person { FirstName="John", LastName="Lennon" },
                        new Person { FirstName="Paul", LastName="McCartney" }
                   };
Person[] beatlesClone = (Person[])beatles.Clone();
```

图 6-6

除了使用 Clone()方法之外，还可以使用 Array.Copy()方法创建浅表副本。但 Clone()方法和 Copy()方法有一个重要区别：Clone()方法会创建一个新数组，而 Copy()方法必须传递阶数相同且有足够元素的已有数组。

如果需要包含引用类型的数组的深层副本，就必须迭代数组并创建新对象。

6.5.3 排序

Array 类使用 QuickSort 算法对数组中的元素进行排序。Sort()方法需要数组中的元素实现 IComparable 接口。因为简单类型(如 System.String 和 System.Int32)实现 IComparable 接口，所以可以对包含这些类型的元素排序。

在示例程序中，数组 name 包含 string 类型的元素，这个数组可以排序(代码文件 SortingSample/Program.cs)。

```
string[] names = {
        "Christina Aguilera",
        "Shakira",
        "Beyonce",
        "Lady Gaga"
    };

Array.Sort(names);
foreach (var name in names)
{
  Console.WriteLine(name);
}
```

该应用程序的输出是排好序的数组：

```
Beyonce
Christina Aguilera
```

```
Lady Gaga
Shakira
```

如果对数组使用自定义类，就必须实现 IComparable 接口。这个接口只定义了一个方法 CompareTo()，如果要比较的对象相等，该方法就返回 0。如果该实例应排在参数对象的前面，该方法就返回小于 0 的值。如果该实例应排在参数对象的后面，该方法就返回大于 0 的值。

修改 Person 类，使之实现 IComparable<Person>接口。先使用 String 类中 CompareTo()方法对 LastName 的值进行比较。如果 LastName 的值相同，就比较 FirstName(代码文件 SortingSample/Program.cs)：

```
public class Person: IComparable<Person>
{
  public int CompareTo(Person other)
  {
    if (other == null) return 1;
    int result = string.Compare(this.LastName, other.LastName);
    if (result == 0)
    {
    result = string.Compare(this.FirstName, other.FirstName);
    }
    return result;
  }
  //...
```

现在可以按照姓氏对 Person 对象对应的数组排序(代码文件 SortingSample/Program.cs)：

```
Person[] persons = {
    new Person { FirstName="Damon", LastName="Hill" },
    new Person { FirstName="Niki", LastName="Lauda" },
    new Person { FirstName="Ayrton", LastName="Senna" },
    new Person { FirstName="Graham", LastName="Hill" }
};

Array.Sort(persons);
foreach (var p in persons)
{
   Console.WriteLine(p);
}
```

使用 Person 类的排序功能，会得到按姓氏排序的姓名：

```
Damon Hill
Graham Hill
Niki Lauda
Ayrton Senna
```

如果 Person 对象的排序方式与上述不同，或者不能修改在数组中用作元素的类，就可以实现 IComparer 接口或 IComparer<T>接口。这两个接口定义了方法 Compare()。要比较的类必须实现这两个接口之一。IComparer 接口独立于要比较的类。这就是 Compare()方法定义了两个要比较的参数的原因。其返回值与 IComparable 接口的 CompareTo()方法类似。

类 PersonComparer 实现了 IComparer<Person>接口，可以按照 firstName 或 lastName 对 Person

对象排序。枚举 PersonCompareType 定义了可用于 PersonComparer 的排序选项：FirstName 和 LastName。排序方式由 PersonComparer 类的构造函数定义，在该构造函数中设置了一个 PersonCompareType 值。实现 Compare()方法时用一个 switch 语句指定是按 FirstName 还是 LastName 排序(代码文件 SortingSample/PersonComparer.cs)。

```
public enum PersonCompareType
{
  FirstName,
  LastName
}

public class PersonComparer: IComparer<Person>
{
  private PersonCompareType compareType;

  public PersonComparer(PersonCompareType compareType)
  {
    this.compareType = compareType;
  }

  public int Compare(Person x, Person y)
  {
    if (x == null && y == null) return 0;
    if (x == null) return 1;
    if (y == null) return -1;

    switch (compareType)
    {
      case PersonCompareType.FirstName:
        return string.Compare(x.FirstName, y.FirstName);
      case PersonCompareType.LastName:
        return string.Compare(x.LastName, y.LastName);
      default:
        throw new ArgumentException("unexpected compare type");
    }
  }
}
```

现在，可以将一个 PersonComparer 对象传递给 Array.Sort()方法的第 2 个参数。下面按名字对 persons 数组排序(代码文件 SortingSample/Program.cs)：

```
Array.Sort(persons, new PersonComparer(PersonCompareType.FirstName));
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

persons 数组现在按名字排序：

```
Ayrton Senna
Damon Hill
Graham Hill
Niki Lauda
```

Array 类还提供了 Sort 方法，它需要将一个委托作为参数。这个参数可以传递给方法，从而比较两个对象，而不需要依赖 IComparable 或 IComparer 接口。第 8 章将介绍如何使用委托。

6.6 数组作为参数

数组可以作为参数传递给方法，也可以从方法中返回。要返回一个数组，只需要把数组声明为返回类型，如下面的方法 GetPersons()所示：

```
static Person[] GetPersons()
{
  return new Person[] {
      new Person { FirstName="Damon", LastName="Hill" },
      new Person { FirstName="Niki", LastName="Lauda" },
      new Person { FirstName="Ayrton", LastName="Senna" },
      new Person { FirstName="Graham", LastName-"Hill" }
  };
}
```

要把数组传递给方法，应把数组声明为参数，如下面的 DisplayPersons()方法所示：

```
static void DisplayPersons(Person[] persons)
{
   //...
```

6.6.1 数组协变

数组支持协变。这表示数组可以声明为基类，其派生类型的元素可以赋予数组元素。

例如，可以声明一个 object[]类型的参数，给它传递一个 Person[]：

```
static void DisplayArray(object[] data)
{
  //...
}
```

数组协变只能用于引用类型，不能用于值类型。另外，数组协变有一个问题，它只能通过运行时异常来解决。如果把 Person 数组赋予 object 数组，object 数组就可以使用派生自 object 的任何元素。例如，编译器允许把字符串传递给数组元素。但因为 object数组引用 Person数组，所以会出现一个运行时异常 ArrayTypeMismatchException。

6.6.2 ArraySegment<T>

结构 ArraySegment<T>表示数组的一段。如果需要使用不同的方法处理某个大型数组的不同部分，那么可以把相应的数组部分复制到各个方法中。此时，与创建多个数组相比，更有效的方法是

使用一个数组，将整个数组传递给不同的方法。这些方法只使用数组的某个部分。方法的参数除了数组以外，还应包括数组内的偏移量以及该方法应该使用的元素数。这样一来，方法就需要至少 3 个参数。当使用数组段时，只需要一个参数就可以了。ArraySegment<T>结构包含了关于数组段的信息(偏移量和元素个数)。

SumOfSegments()方法提取一组 ArraySegment<int>元素，计算该数组段定义的所有整数之和，并返回整数和(代码文件 ArraySegmentSample/Program.cs)：

```
static int SumOfSegments(ArraySegment<int>[] segments)
{
  int sum = 0;
  foreach (var segment in segments)
  {
    for (int i = segment.Offset; i < segment.Offset +
        segment.Count; i++)
    {
      sum += segment.Array[i];
    }
  }
  return sum;
}
```

使用这个方法时，传递了一个数组段。第一个数组元素从 ar1 的第一个元素开始，引用了 3 个元素；第二个数组元素从 ar2 的第 4 个元素开始，引用了 3 个元素：

```
int[] ar1 = { 1, 4, 5, 11, 13, 18 };
int[] ar2 = { 3, 4, 5, 18, 21, 27, 33 };

var segments = new ArraySegment<int>[2]
{
  new ArraySegment<int>(ar1, 0, 3),
  new ArraySegment<int>(ar2, 3, 3)
};
var sum = SumOfSegments(segments);
```

数组段不复制原数组的元素，但原数组可以通过 ArraySegment<T>访问。如果数组段中的元素改变了，这些变化就会反映到原数组中。

6.7　枚举

在 foreach 语句中使用枚举，可以迭代集合(参见第 10 章)中的元素，且无须知道集合中的元素个数。foreach 语句使用了一个枚举器。图 6-7 显示了调用 foreach 方法的客户端和集合之间的关系。数组或集合实现带 GetEumerator()方法的 IEumerable 接口。GetEumerator()方法返回一个实现 IEumerable 接口的枚举。接着，foreach 语句就可以使用 IEumerable 接口迭代集合了。

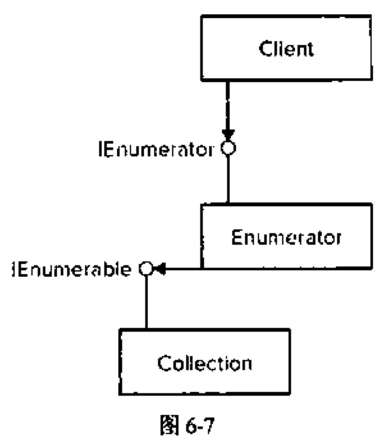


图 6-7

GetEnumerator()方法用 IEnumerable 接口定义。foreach 语句并不真的需要在集合类中实现这个接口。有一个名为 GetEnumerator()的方法，它返回实现了 IEnumerator 接口的对象就足够了。

6.7.1 IEnumerator 接口

foreach 语句使用 IEnumerator 接口的方法和属性，迭代集合中的所有元素。为此，IEnumerator 定义了 Current 属性，来返回光标所在的元素，该接口的 MoveNext()方法移动到集合的下一个元素上，如果有这个元素，该方法就返回 true。如果集合不再有更多的元素，该方法就返回 false。

这个接口的泛型版本 IEnumerator<T>派生自接口 IDisposable，因此定义了 Dispose()方法，来清理枚举器占用的资源。

IEnumerator 接口还定义了 Reset()方法，以与 COM 交互操作。许多.NET 枚举器通过抛出 NotSupportedException 类型的异常，来实现这个方法。

6.7.2 foreach 语句

C#的 foreach 语句不会解析为 IL 代码中的 foreach 语句。C#编译器会把 foreach 语句转换为 IEnumerable 接口的方法和属性。下面是一条简单的 foreach 语句，它迭代 persons 数组中的所有元素，并逐个显示它们：

```
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

foreach 语句会解析为下面的代码段。首先，调用 GetEnumerator()方法，获得数组的一个枚举器。在 while 循环中——只要 MoveNext()返回 true——就用 Current 属性访问数组中的元素：

```
IEnumerator<Person> enumerator = persons.GetEnumerator();
```

```
while (enumerator.MoveNext())
{
  Person p = enumerator.Current;
  Console.WriteLine(p);
}
```

6.7.3 yield 语句

自 C#的第 1 个版本以来，使用 foreach 语句可以轻松地迭代集合。在 C# 1.0 中，创建枚举器仍需要做大量的工作。C# 2.0 添加了 yield 语句，以便于创建枚举器。yield return 语句返回集合的一个元素，并移动到下一个元素上。yield break 可停止迭代。

下一个例子是用 yield return 语句实现一个简单集合的代码。HelloCollection 类包含 GetEnumerator()方法。该方法的实现代码包含两条 yield return 语句，它们分别返回字符串 Hello 和 World(代码文件 YieldDemo/Program.cs)。

```
using System;
using System.Collections;

namespace Wrox.ProCSharp.Arrays
{
  public class HelloCollection
  {
    public IEnumerator<string> GetEnumerator()
    {
      yield return "Hello";
      yield return "World";
    }
  }
```

包含 yield 语句的方法或属性也称为迭代块。迭代块必须声明为返回 IEnumerator 或 IEnumerable 接口，或者这些接口的泛型版本。这个块可以包含多条 yield return 语句或 yield break 语句，但不能包含 return 语句。

现在可以用 foreach 语句迭代集合了：

```
  public void HelloWorld()
  {
    var helloCollection = new HelloCollection();
    foreach (var s in helloCollection)
    {
      Console.WriteLine(s);
    }
  }
}
```

使用迭代块，编译器会生成一个 yield 类型，其中包含一个状态机，如下面的代码段所示。yield 类型实现 IEnumerator 和 IDisposable 接口的属性和方法。在下面的例子中，可以把 yield 类型看作内部类 Enumerator。外部类的 GetEnumerator()方法实例化并返回一个新的 yield 类型。在 yield 类型中，变量 state 定义了迭代的当前位置，每次调用 MoveNext()时，当前位置都会改变。MoveNext()

封装了迭代块的代码，并设置了 current 变量的值，从而使 Current 属性根据位置返回一个对象。

```
public class HelloCollection
{
  public IEnumerator GetEnumerator()
  {
    return new Enumerator(0);
  }

  public class Enumerator: IEnumerator<string>, IEnumerator, IDisposable
  {
    private int state;
    private string current;

    public Enumerator(int state)
    {
      this.state = state;
    }
    bool System.Collections.IEnumerator.MoveNext()
    {
      switch (state)
      {
        case 0:
          current = "Hello";
          state = 1;
          return true;
        case 1:
          current = "World";
          state = 2;
          return true;
        case 2:
          break;
      }

      return false;
    }

    void System.Collections.IEnumerator.Reset()
    {
      throw new NotSupportedException();
    }

    string System.Collections.Generic.IEnumerator<string>.Current
    {
      get
      {
        return current;
      }
    }
    object System.Collections.IEnumerator.Current
    {
      get
      {
        return current;
      }
```

```
    }

    void IDisposable.Dispose()
    {
    }
  }
}
```

yield 语句会生成一个枚举器，而不仅仅生成一个包含的项的列表。这个枚举器通过 foreach 语句调用。从 foreach 中依次访问每一项时，就会访问枚举器。这样就可以迭代大量的数据，而无须一次把所有的数据都读入内存。

1. 迭代集合的不同方式

在下面这个比 Hello World 示例略大但比较真实的示例中，可以使用 yield return 语句，以不同方式迭代集合的类。类 MusicTitles 可以用默认方式通过 GetEnumerator()方法迭代标题，用 Reverse()方法逆序迭代标题，用 Subset()方法迭代子集(代码文件 YieldDemo/MusicTitles.cs)：

```
public class MusicTitles
{
  string[] names = { "Tubular Bells", "Hergest Ridge", "Ommadawn",
                     "Platinum" };

  public IEnumerator<string> GetEnumerator()
  {
    for (int i = 0; i < 4; i++)
    {
      yield return names[i];
    }
  }

  public IEnumerable<string> Reverse()
  {
    for (int i = 3; i >= 0; i--)
    {
      yield return names[i];
    }
  }

  public IEnumerable<string> Subset(int index, int length)
  {
    for (int i = index; i < index + length; i++)
    {
      yield return names[i];
    }
  }
}
```

类支持的默认迭代是定义为返回 IEnumerator 的 GetEnumerator()方法。命名的迭代返回 IEnumerable。

迭代字符串数组的客户端代码先使用 GetEnumerator()方法，该方法不必在代码中编写，因为这是 foreach 语句默认使用的方法。然后逆序迭代标题，最后将索引和要迭代的项数传递给 Subset()方法，来迭代子集(代码文件 YieldDemo/Program.cs)：

```
var titles = new MusicTitles();
foreach (var title in titles)
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("reverse");
foreach (var title in titles.Reverse())
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("subset");
foreach (var title in titles.Subset(2, 2))
{
  Console.WriteLine(title);
}
```

2. 用 yield return 返回枚举器

使用 yield 语句还可以完成更复杂的任务，例如，从 yield return 中返回枚举器。在 TicTacToe 游戏中有 9 个域，玩家轮流在这些域中放置一个“十”字或一个圆。这些移动操作由 GameMoves 类模拟。方法 Cross()和 Circle()是创建迭代类型的迭代块。变量 cross 和 circle 在 GameMoves 类的构造函数中设置为 Cross()和 Circle()方法。这些字段不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 Cross()迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 8，就用 yield break 停止迭代；否则，就在每次迭代中返回 yield 类型 circle 的枚举对象。Circle()迭代块非常类似于 Cross()迭代块，只是它在每次迭代中返回 cross 迭代器类型(代码文件 YieldDemo/GameMoves.cs)。

```
public class GameMoves
{
  private IEnumerator cross;
  private IEnumerator circle;

  public GameMoves()
  {
    cross = Cross();
    circle = Circle();
  }
```

```
    private int move = 0;
    const int MaxMoves = 9;

    public IEnumerator Cross()
    {
      while (true)
      {
        Console.WriteLine("Cross, move {0}", move);
        if (++move >= MaxMoves)
          yield break;
        yield return circle;
      }
    }

    public IEnumerator Circle()
    {
      while (true)
      {
        Console.WriteLine("Circle, move {0}", move);
        if (++move >= MaxMoves)
          yield break;
        yield return cross;
      }
    }
  }
```

在客户端程序中，可以以如下方式使用 GameMoves 类。将枚举器设置为由 game.Cross()返回的枚举器类型，以设置第一次移动。在 while 循环中，调用 enumerator.MoveNext()。第一次调用 enumerator.MoveNext()时，会调用 Cross()方法，Cross()方法使用 yield 语句返回另一个枚举器。返回的值可以用 Current 属性访问，并设置为 enumerator 变量，用于下一次循环：

```
var game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
  enumerator = enumerator.Current as IEnumerator;
}
```

这个程序的输出会显示交替移动的情况，直到最后一次移动：

```
Cross, move 0
Circle, move 1
Cross, move 2
Circle, move 3
Cross, move 4
Circle, move 5
Cross, move 6
Circle, move 7
Cross, move 8
```

6.8 元组

数组合并了相同类型的对象，而元组合并了不同类型的对象。元组起源于函数编程语言(如 F#)，在这些语言中频繁使用元组。在.NET Framework 中，元组可用于所有的.NET 语言。

.NET Framework 定义了 8 个泛型 Tuple 类(自.NET 4.0 以来)和一个静态 Tuple 类，它们用作元组的工厂。不同的泛型 Tuple 类支持不同数量的元素。例如，Tuple<T1>包含一个元素，Tuple<T1, T2>包含两个元素，依此类推。

方法 Divide()返回包含两个成员的元组 Tuple<int, int>。泛型类的参数定义了成员的类型，它们都是整数。元组用静态 Tuple 类的静态 Create()方法创建。Create()方法的泛型参数定义了要实例化的元组类型。新建的元组用 result 和 reminder 变量初始化，返回这两个变量相除的结果(代码文件 TupleSample/Program.cs)：

```
public static Tuple<int, int> Divide(int dividend, int divisor)
{
  int result = dividend / divisor;
  int reminder = dividend % divisor;

  return Tuple.Create<int, int>(result, reminder);
}
```

下面的代码说明了 Divide()方法的调用。可以用属性 Item1 和 Item2 访问元组的项：

```
var result = Divide(5, 2);
Console.WriteLine("result of division: {0}, reminder: {1}",
    result.Item1, result.Item2);
```

如果元组包含的项超过 8 个，就可以使用带 8 个参数的 Tuple 类定义。最后一个模板参数是 TRest，表示必须给它传递一个元组。这样，就可以创建带任意个参数的元组了。

下面说明这个功能：

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

其中，最后一个模板参数是一个元组类型，所以可以创建带任意多项的元组：

```
var tuple = Tuple.Create<string, string, string, int, int, int, double,
    Tuple<int, int>>("Stephanie", "Alina", "Nagel", 2009, 6, 2, 1.37,
       Tuple.Create<int, int>(52, 3490));
```

6.9 结构比较

数组和元组都实现接口 IStructuralEquatable 和 IStructuralComparable。这两个接口都是.NET 4 新增的，不仅可以比较引用，还可以比较内容。这些接口都是显式实现的，所以在使用时需要把数组和元组强制转换为这个接口。IstructuralEquatable 接口用于比较两个元组或数组是否有相同的内容，IStructuralComparable 接口用于给元组或数组排序。

对于说明 IStructuralEquatable 接口的示例，使用实现 IEquatable 接口的 Person 类。IEquatable 接

口定义了一个强类型化的 Equals()方法，以比较 FirstName 和 LastName 属性的值(代码文件 StructuralComparison/Person.cs)：

```
public class Person: IEquatable<Person>
{
  public int Id { get; private set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public override string ToString()
  {
    return String.Format("{0}, {1} {2}", Id, FirstName, LastName);
  }

  public override bool Equals(object obj)
  {
    if (obj == null)
      return base.Equals(obj);
    return Equals(obj as Person);
  }

  public override int GetHashCode()
  {
    return Id.GetHashCode();
  }

  public bool Equals(Person other)
  {
    if (other == null)
      return base.Equals(other);

    return this.Id == other.Id && this.FirstName == other.FirstName &&
        this.LastName == other.LastName;
  }
}
```

现在创建了两个包含 Person 项的数组。这两个数组通过变量名 janet 包含相同的 Person 对象，和两个内容相同的不同 Person 对象。比较运算符“!=”返回 true，因为这其实是两个变量 persons1 和 persons2 引用的两个不同数组。因为 Array 类没有重写带一个参数的 Equals()方法，所以用“==”运算符比较引用也会得到相同的结果，即这两个变量不相同(代码文件 StructuralComparison/Program.cs)：

```
var janet = new Person { FirstName = "Janet", LastName = "Jackson" };
Person[] persons1 = {
  new Person
  {
    FirstName = "Michael",
    LastName = "Jackson"
  },
  janet
};
Person[] persons2 = {
  new Person
```

```
    {
      FirstName = "Michael",
      LastName = "Jackson"
    },
    janet
};
if (persons1 != persons2)
  Console.WriteLine("not the same reference");
```

对于 IStructuralEquatable 接口定义的 Equals()方法，它的第一个参数是 object 类型，第二个参数是 IEqualityComparer 类型。调用这个方法时，通过传递一个实现了 IEqualityComparer<T>的对象，就可以定义如何进行比较。通过 EqualityComparer<T>类完成 IEqualityComparer 的一个默认实现。这个实现检查该类型是否实现了 IEquatable 接口，并调用 IEquatable.Equals()方法。如果该类型没有实现 IEquatable，就调用 Object 基类中的 Equals()方法进行比较。

Person 实现 IEquatable<Person>，在此过程中比较对象的内容，而数组的确包含相同的内容：

```
if ((persons1 as IStructuralEquatable).Equals(persons2,
     EqualityComparer<Person>.Default))
{
  Console.WriteLine("the same content");
}
```

下面看看如何对元组执行相同的操作。这里创建了两个内容相同的元组实例。当然，因为引用 t1 和 t2 引用了两个不同的对象，所以比较运算符“!=”返回 true：

```
var t1 = Tuple.Create<int, string>(1, "Stephanie");
var t2 = Tuple.Create<int, string>(1, "Stephanie");
if (t1 != t2)
  Console.WriteLine("not the same reference to the tuple");
```

Tuple<>类提供了两个 Equals()方法：一个重写了 Object 基类中的 Equals()方法，并把 object 作为参数，第二个由 IStructuralEqualityComparer 接口定义，并把 object 和 IEqualityComparer 作为参数。可以给第一个方法传送另一个元组，如下所示。这个方法使用 EqualityComparer<object>.Default 获取一个 ObjectEqualityComparer<object>，以进行比较。这样，就会调用 Object.Equals()方法比较元组的每一项。如果每一项都返回 true，Equals()方法的最终结果就是 true，这里因为 int 和 string 值都相同，所以返回 true：

```
if (t1.Equals(t2))
Console.WriteLine("the same content");
```

还可以使用类 TupleComparer 创建一个自定义的 IEqualityComparer，如下所示。这个类实现了 IEqualityComparer 接口的两个方法 Equals()和 GetHashCode()：

```
class TupleComparer: IEqualityComparer
{
  public new bool Equals(object x, object y)
  {
    return x.Equals(y);
  }
```

```
  public int GetHashCode(object obj)
  {
    return obj.GetHashCode();
  }
}
```

实现IEqualityComparer接口的Equals()方法需要new修饰符或者隐式实现的接口，因为基类Object也定义了带两个参数的静态Equals()方法。

使用TupleComparer，给Tuple<T1, T2>类的Equals()方法传递一个新实例。Tuple类的Equals()方法为要比较的每一项调用TupleComparer的Equals()方法。所以，对于Tuple<T1, T2>类，要调用两次TupleComparer，以检查所有项是否相等：

```
if (t1.Equals(t2, new TupleComparer()))
Console.WriteLine("equals using TupleComparer");
```

6.10 小结

本章介绍了创建和使用简单数组、多维数组和锯齿数组的C#表示法。C#数组在后台使用Array类，这样就可以用数组变量调用这个类的属性和方法。

我们还探讨了如何使用IComparable和IComparer接口给数组中的元素排序，描述了如何使用和创建枚举器、IEnumerable和IEnumerator接口，以及yield语句。

最后介绍了如何在数组中组织相同类型的对象，在元组中组织不同类型的对象。

第7章介绍运算符和强制类型转换。

第 7 章

运算符和类型强制转换

本章要点

- C#中的运算符
- 处理引用类型和值类型时相等的含义
- 基本数据类型之间的数据转换
- 使用装箱技术把值类型转换为引用类型
- 通过类型强制转换在引用类型之间转换
- 重载标准的运算符以支持自定义类型
- 给自定义类型添加类型强制转换运算符

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- SimpleCurrency
- SimpleCurrency2
- VectorStruct
- VectorStructMoreOverloads

7.1 运算符和类型转换

前几章介绍了使用 C#编写程序所需要的大部分知识。本章将首先讨论基本语言元素，接着论述 C#语言的扩展功能。

7.2 运算符

C 和 C++开发人员应很熟悉大多数 C#运算符，这里为新程序员和 Visual Basic 开发人员介绍最

重要的运算符，并揭示 C#中的一些新变化。

C#支持表 7-1 中的运算符。

表 7-1

类　别	运　算　符
算术运算符	+ –* / %
逻辑运算符	& | ^ ~ && || !
字符串连接运算符	+
增量和减量运算符	++ ––
移位运算符	<< >>
比较运算符	== != <> <=> =
赋值运算符	= += –= *= /= %= &= |= ^= <<= >>=
成员访问运算符(用于对象和结构)	.
索引运算符(用于数组和索引器)	[]
类型转换运算符	()
条件运算符(三元运算符)	?:
委托连接和删除运算符(见第 8 章)	+ –
对象创建运算符	new
类型信息运算符	sizeof is typeof as
溢出异常控制运算符	checked unchecked
间接寻址运算符	[]
名称空间别名限定符(见第 2 章)	::
空合并运算符	??

如表 7-2 所示，有 4 个运算符(sizeof、*、->和&)只能用于不安全的代码(这些代码忽略了 C#的类型安全性检查)，这些不安全的代码见第 14 章的讨论。还要注意，sizeof 运算符在早期版本的.NET Framework 1.0 和 1.1 中使用，它需要不安全模式。自从.NET Framework 2.0 版本以来，就不需要这个运算符了。

表 7-2

类　别	运　算　符
运算符关键字	sizeof(仅用于.NET Framework 1.0 和 1.1)
运算符	*、->、&

使用 C#运算符的一个最大缺点是，与 C 风格的语言一样，对于赋值(=)和比较(==)运算 C#使用不同的运算符。例如，下述语句表示“x 等于 3”：

```
x = 3;
```

如果要比较 x 和另一个值，就需要使用两个等号(==)：

```
if (x == 3)
{

}
```

C#非常严格的类型安全规则防止出现常见的 C 错误，也就是在逻辑语句中使用赋值运算符代替比较运算符。在 C#中，下述语句会产生一个编译错误：

```
if (x = 3)
{

}
```

习惯使用与字符(&)来连接字符串的 Visual Basic 程序员必须改变这个习惯。在 C#中，使用加号(+)连接字符串，而“&”符号表示两个不同整数值的按位 AND 运算。“|”符号则在两个整数之间执行按位 OR 运算。Visual Basic 程序员可能还没有使用过取模(%)运算符，它返回除运算的余数，例如，如果 x 等于 7，则 x % 5 会返回 2。

在 C#中很少会用到指针，因此也很少用到间接寻址运算符(->)。使用它们的唯一场合是在不安全的代码块中，因为只有在此 C#才允许使用指针。指针和不安全的代码见第 14 章。

7.2.1 运算符的简化操作

表 7-3 列出了 C#中的全部简化赋值运算符。

表 7-3

运算符的简化操作	等 价 于
x++,++x	x = x + 1
x--,--x	x = x-1
x+= y	x = x + y
x-= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x |= y	x = x | y

为什么用两个例子来说明“++”增量和“--”减量运算符？把运算符放在表达式的前面称为前置，把运算符放在表达式的后面称为后置。要点是注意它们的行为方式有所不同。

增量或减量运算符可以作用于整个表达式，也可以作用于表达式的内部。当 x++和++x 单独占一行时，它们的作用是相同的，对应于语句 x = x + 1。但当它们用于较长的表达式的内部时，把运算符放在前面(++x)会在计算表达式之前递增 x，换言之，递增了 x 后，在表达式中使用新值进行计

算。而把运算符放在后面(x++)会在计算表达式之后递增 x——使用 x 的原始值计算表达式。下面的例子使用“++”增量运算符说明了它们的区别：

```
int x = 5;

if (++x == 6) // true - x is incremented to 6 before the evaluation
{
   Console.WriteLine("This will execute");
}

if (x++ == 7) // false - x is incremented to 7 after the evaluation
{
   Console.WriteLine("This won't");
}
```

判断第一个 if 条件得到 true，因为在计算表达式之前，x 从 5 递增为 6。第二条 if 语句中的条件为 false，因为在计算完整个表达式(x == 6)后，x 才递增为 7。

前置运算符--x 和后置运算符 x--与此类似，但它们是递减，而不是递增。

其他简化运算符，如+=和-=需要两个操作数，通过对第 1 个操作数执行算术、逻辑或按位运算，从而改变第 1 个操作数的值。例如，下面两行代码是等价的：

```
x += 5;
x = x + 5;
```

下面介绍在 C#代码中频繁使用的基本运算符和类型强制转换运算符。

1. 条件运算符

条件运算符(?:)也称为三元运算符，是 if...else 结构的简化形式。其名称的出处是它带有 3 个操作数。它首先判断一个条件，如果条件为真，就返回一个值；如果条件为假，则返回另一个值。其语法如下：

```
condition ? true_value: false_value
```

其中 condition 是要判断的布尔表达式，true_value 是 condition 为 true 时返回的值，false_value 是 condition 为 false 时返回的值。

恰当地使用三元运算符，可以使程序非常简洁。它特别适合于给被调用的函数提供两个参数中的一个。使用它可以把布尔值转换为字符串值 true 或 false。它也很适合于显示正确的单数形式或复数形式，例如：

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man": "men");
Console.WriteLine(s);
```

如果 x 等于 1，这段代码就显示 1 man；如果 x 等于其他数，就显示其正确的复数形式。但要注意，如果结果需要本地化为不同的语言，就必须编写更复杂的例程，以考虑到不同语言的不同语法。

2. checked 和 unchecked 运算符

考虑下面的代码：

```
byte b = 255;
b++;
Console.WriteLine(b.ToString());
```

byte 数据类型只能包含 0~255 的数，所以递增 b 的值会导致溢出。CLR 如何处理这个溢出取决于许多因素，包括编译器选项，所以只要有未预料到的溢出风险，就需要用某种方式确保得到我们希望的结果。

为此，C#提供了 checked 和 unchecked 运算符。如果把一个代码块标记为 checked，CLR 就会执行溢出检查，如果发生溢出，就抛出 OverflowException 异常。下面修改代码，使之包含 checked 运算符：

```
byte b = 255;
checked
{
   b++;
}
Console.WriteLine(b.ToString());
```

运行这段代码，就会得到一条错误信息：

```
Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an
overflow at Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)
```

用/checked 编译器选项进行编译，就可以检查程序中所有未标记代码中的溢出。

如果要禁止溢出检查，则可以把代码标记为 unchecked：

```
byte b = 255;
unchecked
{
   b++;
}
Console.WriteLine(b.ToString());
```

在本例中，不会抛出异常，但会丢失数据——因为 byte 数据类型不能包含 256，溢出的位会丢弃，所以 b 变量得到的值是 0。

注意，unchecked 是默认行为。只有在需要把几行未检查的代码放在一个显式地标记为 checked 的大代码块中，才需要显式地使用 unchecked 关键字。

3. is 运算符

is 运算符可以检查对象是否与特定的类型兼容。“兼容”表示对象或者是该类型，或者派生自该类型。例如，要检查变量是否与 object 类型兼容，可以使用下面的代码：

```
int i = 10;
if (i is object)
{
   Console.WriteLine("i is an object");
}
```

int 和所有 C#数据类型一样，也从 object 继承而来；表达式 i is object 将为 true，并显示相应的消息。

4. as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，as 运算符就会返回 null 值。如下面的代码所示，如果 object 引用实际上不引用 string 实例，把 object 引用转换为 string 就会返回 null：

```
object o1 = "Some String";
object o2 = 5;

string s1 = o1 as string; // s1 = "Some String"
string s2 = o2 as string; // s2 = null
```

as 运算符允许在一步中进行安全的类型转换，不需要先使用 is 运算符测试类型，再执行转换。

5. sizeof 运算符

使用 sizeof 运算符可以确定栈中值类型需要的长度(单位是字节)：

```
Console.WriteLine(sizeof(int));
```

其结果是显示数字 4，因为 int 有 4 个字节。

如果对复杂类型(和非基本类型)使用 sizeof 运算符，就需要把代码放在 unsafe 块中，如下所示：

```
unsafe
{
  Console.WriteLine(sizeof(Customer));
}
```

第 14 章将详细论述不安全的代码。

6. typeof 运算符

typeof 运算符返回一个表示特定类型的 System.Type 对象。例如，typeof(string)返回表示 System.String 类型的 Type 对象。在使用反射技术动态地查找对象的相关信息时，这个运算符很有用。第 15 章将介绍反射。

7. 可空类型和运算符

对于布尔类型，可以给它指定 true 或 false 值。但是，要把该类型的值定义为 undefined，该怎么办？此时使用可空类型可以给应用程序提供一个独特的值。如果在程序中使用可空类型，就必须考虑 null 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 null，其结果就是 null。例如：

```
int? a = null;

int? b = a + 4; // b = null
int? c = a * 5; // c = null
```

但是在比较可空类型时，只要有一个操作数是 null，比较的结果就是 false。即不能因为一个条件是 false，就认为该条件的对立面是 true，这在使用非可空类型的程序中很常见。例如：

```
int? a = null;
int? b = -5;

if (a > = b)
   Console.WriteLine("a > = b");
else
   Console.WriteLine("a < b");
```

null 值的可能性表示，不能随意合并表达式中的可空类型和非可空类型，详见 7.3.1 小节的内容。

8. 空合并运算符

空合并运算符(??)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 null 可能的值。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型；第二个操作数必须与第一个操作数的类型相同，或者可以隐含地转换为第一个操作数的类型。空合并运算符的计算如下：

- 如果第一个操作数不是 null，整个表达式就等于第一个操作数的值。
- 如果第一个操作数是 null，整个表达式就等于第二个操作数的值。

例如：

```
int? a = null;
int b;

b = a ?? 10; // b has the value 10
a = 3;
b = a ?? 10; // b has the value 3
```

如果第二个操作数不能隐含地转换为第一个操作数的类型，就生成一个编译错误。

7.2.2　运算符的优先级

表 7-4 显示了 C#运算符的优先级。表 7-4 顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)。

表 7-4

组	运　算　符
初级运算符	() . [] x++ x-- new typeof sizeof checked unchecked
一元运算符	+ – ! ~ ++x --x 和数据类型强制转换
乘/除运算符	* / %
加/减运算符	+ –
移位运算符	<< >>
关系运算符	< ><= >= is as
比较运算符	== !=
按位 AND 运算符	&
按位 XOR 运算符	^
按位 OR 运算符	|
布尔 AND 运算符	&&
布尔 OR 运算符	||
条件运算符	?:
赋值运算符	= += -= *= /= %= &= |= ^= <<= >>= >>>=

在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用圆括号指定运算符的执行顺序，可以使代码更整洁，避免出现潜在的冲突。

7.3 类型的安全性

第 1 章提到中间语言(IL)可以对其代码强制实现强类型安全性。强类型化支持.NET 提供的许多服务，包括安全性和语言的交互性。因为 C#这种语言会编译为 IL，所以 C#也是强类型的。此外，这说明数据类型并不总是无缝地可互换的。本节将介绍基元类型之间的转换。

C#也支持不同引用类型之间的轮换，在与其他类型相互转换时还允许定义所创建的数据类型的行为方式。本章稍后将详细讨论这两个主题。

另一方面，泛型可以避免对一些常见的情形进行类型转换，详见第 5 章和第 10 章。

7.3.1 类型转换

我们常常需要把数据从一种类型转换为另一种类型。考虑下面的代码：

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

在试图编译这些代码时，会得到一条错误消息：

```
Cannot implicitly convert type 'int' to 'byte'
```

问题是，我们把两个 byte 型数据加在一起时，应返回 int 型结果，而不是另一个 byte。这是因为 byte 包含的数据只能为 8 位，所以把两个 byte 型数据加在一起，很容易得到不能存储在单个字节中的值。如果要把结果存储在一个 byte 变量中，就必须把它转换回一个 byte。C#支持两种转换方式：隐式转换和显式转换。

1. 隐式转换

只要能保证值不会发生任何变化，类型转换就可以自动(隐式)进行。这就是前面代码失败的原因：试图从 int 转换为 byte，而可能丢失了 3 个字节的数据。编译器不允许这么做，除非我们明确告诉它这就是我们希望的！如果在 long 类型变量中而不是 byte 类型变量中存储结果，就不会有问题了：

```
byte value1 = 10;
byte value2 = 23;
long total;          // this will compile fine
total = value1 + value2;
Console.WriteLine(total);
```

这是因为 long 类型变量包含的数据字节比 byte 类型多，所以没有丢失数据的危险。在这些情况下，编译器会很顺利地转换，我们也不需要显式提出要求。

表 7-5 列出了 C#支持的隐式类型转换。

表 7-5

源 类 型	目 的 类 型
sbyte	short、int、long、float、double、decimal、BigInteger
byte	short、ushort、int、uint、long、ulong、float、double、decimal、BigInteger
short	int、long、float、double、decimal、BigInteger
ushort	int、uint、long、ulong、float、double、decimal、BigInteger
int	long、float、double、decimal、BigInteger
uint	long、ulong、float、double、decimal、BigInteger
long、ulong	float、double、decimal、BigInteger
float	double、BigInteger
char	ushort、int、uint、long、ulong、float、double、decimal、BigInteger

注意，只能从较小的整数类型隐式地转换为较大的整数类型，不能从较大的整数类型隐式地转换为较小的整数类型。也可以在整数和浮点数之间转换，然而，其规则略有不同。尽管可以在相同大小的类型之间转换，如 int/uint 转换为 float，long/ulong 转换为 double，但是也可以从 long/ulong 转换回 float。这样做可能会丢失 4 个字节的数据，但这仅表示得到的 float 值比使用 double 得到的值精度低，编译器认为这是一种可以接受的错误，因为值的数量级不会受到影响。无符号的变量可以转换为有符号的变量，只要无符号的变量值的大小在有符号的变量的范围之内即可。

在隐式地转换值类型时，可空类型需要考虑其他因素：

- 可空类型隐式地转换为其他可空类型，应遵循表 7-5 中非可空类型的转换规则。即 int?隐式地转换为 long?、float?、double?和 decimal?。
- 非可空类型隐式地转换为可空类型也遵循表 7-5 中的转换规则，即 int 隐式地转换为 long?、float?、double?和 decimal?。
- 可空类型不能隐式地转换为非可空类型，此时必须进行显式转换，如下一节所述。这是因为可空类型的值可以是 null，但非可空类型不能表示这个值。

2. 显式转换

有许多场合不能隐式地转换类型，否则编译器会报告错误。下面是不能进行隐式转换的一些场合：

- int 转换为 short——会丢失数据。
- int 转换为 uint——会丢失数据。
- uint 转换为 int——会丢失数据。
- float 转换为 int——会丢失小数点后面的所有数据。
- 任何数字类型转换为 char——会丢失数据。
- decimal 转换为任何数字类型——因为 decimal 类型的内部结构不同于整数和浮点数。
- int?转换为 int——可空类型的值可以是 null。

但是，可以使用类型强制转换(cast)显式地执行这些转换。在把一种类型强制转换为另一种类型时，要有意地迫使编译器进行转换。类型强制转换的一般语法如下：

```
long val = 30000;
int i = (int)val;   // A valid cast. The maximum int is 2147483647
```

这表示，把强制转换的目标类型名放在要转换的值之前的圆括号中。对于熟悉 C 的程序员，这是类型强制转换的典型语法。对于熟悉 C++类型强制转换关键字(如 static_cast)的程序员，这些关键字在 C#中不存在，必须使用 C 风格的旧语法。

这种类型强制转换是一种比较危险的操作，即使在从 long 转换为 int 这样简单的类型强制转换过程中，如果原来 long 的值比 int 的最大值还大，就会出问题：

```
long val = 3000000000;
int i = (int)val;         // An invalid cast. The maximum int is 2147483647
```

在本例中，不会报告错误，也得不到期望的结果。如果运行上面的代码，输出结果存储在 i 中，则其值为：

```
-1294967296
```

最好假定显式转换强制转换不会给出希望的结果。如前所述，C#提供了一个 checked 运算符，使用它可以测试操作是否会导致算术溢出。使用 checked 运算符可以检查类型强制转换是否安全，如果不安全，就会迫使运行库抛出一个溢出异常：

```
long val = 3000000000;
int i = checked((int)val);
```

记住，所有的显式类型强制转换都可能不安全，在应用程序中应包含代码来处理可能失败的类

型强制转换。第 16 章将使用 try 和 catch 语句引入结构化异常处理。

使用类型强制转换可以把大多数基本数据类型从一种类型转换为另一种类型。例如，给 price 加上 0.5，再把结果强制转换为 int：

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

这会把价格四舍五入为最接近的美元数。但在这个转换过程中，小数点后面的所有数据都会丢失。因此，如果要使用这个修改过的价格进行更多的计算，最好不要使用这种转换；如果要输出全部计算完或部分计算完的近似值，且不希望用小数点后面的多位数据去麻烦用户，这种转换就很好。

下面的例子说明了把无符号整数转换为 char 时会发生的情况：

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

结果是 ASCII 码为 43 的字符，即“+”号。可以尝试数字类型(包括 char)之间的任何转换，这种转换是可行的，例如，把 decimal 转换为 char，或把 char 转换为 decimal。

值类型之间的转换并不仅限于孤立的变量。还可以把类型为 double 的数组元素转换为类型为 int 的结构成员变量：

```
struct ItemDetails
{
  public string Description;
  public int ApproxPrice;
}

//..

double[] Prices = { 25.30, 26.20, 27.40, 30.00 };

ItemDetails id;
id.Description = "Hello there.";
id.ApproxPrice = (int)(Prices[0] + 0.5);
```

要把一个可空类型转换为非可空类型，或转换为另一个可空类型，其中可能会丢失数据，就必须使用显式的类型强制转换。甚至在底层基本类型相同的元素之间进行转换时，也要使用显式的类型强制转换，例如，int?转换为 int，或 float?转换为 float。这是因为可空类型的值可以是 null，非可空类型不能表示这个值。只要可以在两种等价的非可空类型之间进行显式的类型强制转换，对应可空类型之间显式的类型强制转换就可以进行。但如果从可空类型强制转换为非可空类型，且变量的值是 null，就会抛出 InvalidOperationException 异常。例如：

```
int? a = null;
int  b = (int)a;     // Will throw exception
```

小心地使用显式的类型强制转换，就可以把简单值类型的任何实例转换为几乎任何其他类型。但在进行显式的类型转换时有一些限制，就值类型来说，只能在数字、char 类型和 enum 类型之间转换。不能直接把布尔型强制转换为其他类型，也不能把其他类型转换为布尔型。

如果需要在数字和字符串之间转换，就可以使用.NET 类库中提供的一些方法。Object 类实现了一个 ToString()方法，该方法在所有的.NET 预定义类型中都进行了重写，并返回对象的字符串表示：

```
int i = 10;
string s = i.ToString();
```

同样，如果需要分析一个字符串，以检索一个数字或布尔值，就可以使用所有预定义值类型都支持的 Parse()方法：

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); // Add 50 to prove it is really an int
```

注意，如果不能转换字符串(例如，要把字符串 Hello 转换为一个整数)，Parse()方法就会通过抛出一个异常注册一个错误。第 15 章将介绍异常。

7.3.2 装箱和拆箱

第 2 章介绍了所有类型，包括简单的预定义类型(如 int 和 char)和复杂类型(如从 object 类型中派生的类和结构)。下面可以像处理对象那样处理字面值：

```
string s = 10.ToString();
```

但是，C#数据类型可以分为在栈上分配内存的值类型和在托管堆上分配内存的引用类型。如果 int 不过是栈上一个 4 字节的值，该如何在它上面调用方法？

C#的实现方式是通过一个有点魔术性的方式，即装箱(boxing)。装箱和拆箱(unboxing)可以把值类型转换为引用类型，并把引用类型转换回值类型。这包含在 7.6 节中，因为这是基本的操作，即把值强制转换为 object 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“箱子”。

该转换可以隐式地进行，如上面的例子所述。还可以显式地进行转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

拆箱用于描述相反的过程，其中以前装箱的值类型强制转换回值类型。这里使用术语“强制转换”，是因为这种转换是显式进行的。其语法类似于前面的显式类型转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;         // Box the int
int mySecondNumber = (int)myObject;    // Unbox it back into an int
```

只能对以前装箱的变量进行拆箱。当 myObject 不是装箱后的 int 型时，如果执行最后一行代码，就会在运行期间抛出一个异常。

这里有一个警告。在拆箱时，必须非常小心，确保得到的值变量有足够的空间存储拆箱的值中的所有字节。例如，C#的 int 只有 32 位，所以把 long 值(64 位)拆箱为 int 时，会导致一个 InvalidCastException 异常：

```
long myLongNumber = 333333423;
object myObject = (object)myLongNumber;
int myIntNumber = (int)myObject;
```

7.4　比较对象的相等性

在讨论了运算符并简要介绍了相等运算符后，就应考虑在处理类和结构的实例时，“相等”意味着什么。理解对象相等的机制对逻辑表达式的编程非常重要，另外，对实现运算符重载和类型强制转换也非常重要，本章后面将讨论运算符重载。

对象相等的机制有所不同，这取决于比较的是引用类型(类的实例)还是值类型(基本数据类型，结构或枚举的实例)。下面分别介绍引用类型和值类型的相等性。

7.4.1　比较引用类型的相等性

System.Object 定义了 3 个不同的方法，来比较对象的相等性：ReferenceEquals()和两个版本的 Equals()。再加上比较运算符(==)，实际上有 4 种比较相等性的方式。这些方法有一些细微的区别，下面就介绍这些方法。

1. ReferenceEquals()方法

ReferenceEquals()是一个静态方法，测试两个引用是否引用类的同一个实例，特别是两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以 System.Object 的实现代码保持不变。如果提供的两个引用引用同一个对象实例，则 ReferenceEquals()总是返回 true；否则就返回 false。但是它认为 null 等于 null：

```
SomeClass x, y;
x = new SomeClass();
y = new SomeClass();
bool B1 = ReferenceEquals(null, null); // returns true
bool B2 = ReferenceEquals(null,x);     // returns false
bool B3 = ReferenceEquals(x, y);       // returns false because x and y
                                       // point to different objects
```

2. 虚拟的 Equals()方法

Equals()虚拟版本的 System.Object 实现代码也可以比较引用。但因为这个方法是虚拟的，所以可以在自己的类中重写它，从而按值来比较对象。特别是如果希望类的实例用作字典中的键，就需要重写这个方法，以比较相关值。否则，根据重写 Object.GetHashCode()的方式，包含对象的字典类要么不工作，要么工作的效率非常低。在重写 Equals()方法时要注意，重写的代码不会抛出异常。同理，这是因为如果抛出异常，字典类就会出问题，一些在内部调用这个方法的.NET 基类也可能出问题。

3. 静态的 Equals()方法

Equals()的静态版本与其虚拟实例版本的作用相同，其区别是静态版本带有两个参数，并对它们进行相等性比较。这个方法可以处理两个对象中有一个是 null 的情况，因此，如果一个对象可能是 null，这个方法就可以抛出异常，提供额外的保护。静态重载版本首先要检查传递给它的引用是否为 null。如果它们都是 null，就返回 true(因为 null 与 null 相等)。如果只有一个引用是 null，它就返回 false。如果两个引用实际上引用了某个对象，它就调用 Equals()的虚拟实例版本。这表示在重写

Equals()的实例版本时，其效果相当于也重写了静态版本。

4. 比较运算符(==)

最好将比较运算符看作严格的值比较和严格的引用比较之间的中间选项。在大多数情况下，下面的代码表示正在比较引用：

```
bool b = (x == y);  // x, y object references
```

但是，如果把一些类看作值，其含义就会比较直观，这是可以接受的。在这些情况下，最好重写比较运算符，以执行值的比较。后面将讨论运算符的重载，但它的一个明显例子是 System.String 类，Microsoft 重写了这个运算符，以比较字符串的内容，而不比较它们的引用。

7.4.2 比较值类型的相等性

在比较值类型的相等性时，采用与引用类型相同的规则：ReferenceEquals()用于比较引用，Equals()用于比较值，比较运算符可以看作一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，进而才能对它们执行方法。另外，Microsoft 已经在 System.ValueType 类中重载了实例方法 Equals()，以便对值类型进行合适的相等性测试。如果调用 sA.Equals(sB)，其中 sA 和 sB 是某个结构的实例，则根据 sA 和 sB 是否在其所有的字段中包含相同的值，而返回 true 或 false。另一方面，在默认情况下，不能对自己的结构重载“==”运算符。在表达式中使用(sA == sB)会导致一个编译错误，除非在代码中为该结构提供了“==”的重载版本。

另外，ReferenceEquals()在应用于值类型时，它总是返回 false，因为为了调用这个方法，值类型需要装箱到对象中。即使使用下面的代码：

```
bool b = ReferenceEquals(v,v);  // v is a variable of some value type
```

也会返回 false，因为在转换每个参数时，v 都会被单独装箱，这意味着会得到不同的引用。出于上述原因，调用 ReferenceEquals()来比较值类型实际上没有什么意义，所以不能调用它。

尽管 System.ValueType 提供的 Equals()的默认重写版本肯定足以应付绝大多数自定义的结构，但仍可以针对自己的结构再次重写它，以提高性能。另外，如果值类型包含作为字段的引用类型，就需要重写 Equals()，以便为这些字段提供合适的语义，因为 Equals()的默认重写版本仅比较它们的地址。

7.5 运算符重载

本节将介绍为类或结构定义的另一种类型的成员：运算符重载。C++开发人员应很熟悉运算符重载。但是，因为这个概念对 Java 和 Visual Basic 开发人员是全新的，所以这里要解释一下。C++开发人员可以直接跳到主要的运算符重载示例上。

运算符重载的关键是在对象上不能总是只调用方法或属性，有时还需要做一些其他工作，例如，对数值进行相加、相乘或逻辑操作(如比较对象)等。假定已经定义了一个表示数学矩阵的类。在数学领域中，矩阵可以相加和相乘，就像数字一样。所以可以编写下面的代码：

```
Matrix a, b, c;
```

```
// assume a, b and c have been initialized
Matrix d = c * (a + b);
```

通过重载运算符，就可以告诉编译器，“+”和“*”对 Matrix 对象进行什么操作，以编写上面的代码。如果用不支持运算符重载的语言编写代码，就必须定义一个方法，以执行这些操作。结果肯定不太直观，可能如下所示。

```
Matrix d = c.Multiply(a.Add(b));
```

学习到现在，像“+”和“*”这样的运算符只能用于预定义的数据类型，原因很简单：编译器知道所有常见的运算符对于这些数据类型的含义。例如，它知道如何把两个 long 加起来，或者如何对两个 double 执行相除操作，并生成合适的中间语言代码。但在定义自己的类或结构时，必须告诉编译器：什么方法可以调用，每个实例存储了什么字段等所有信息。同样，如果要对自定义的类使用运算符，就必须告诉编译器相关的运算符在这个类的上下文中的含义。此时就要定义运算符的重载。

要强调的另一个问题是重载不仅仅限于算术运算符。还需要考虑比较运算符 ==、<、>、!=、>=和<=。例如，语句 if(a==b)。对于类，这个语句在默认状态下会比较引用 a 和 b。检测这两个引用是否指向内存中的同一个地址，而不是检测两个实例实际上是否包含相同的数据。对于 string 类，这种操作就会重写，于是比较字符串实际上就是比较每个字符串的内容。可以对自己的类进行这样的操作。对于结构，“==”运算符在默认状态下不做任何工作。试图比较两个结构，看看它们是否相等，就会产生一个编译错误，除非显式地重载了“==”，告诉编译器如何进行比较。

在许多情况下，重载运算符允许生成可读性更高、更直观的代码，包括：

- 在数学领域中，几乎包括所有的数学对象：坐标、矢量、矩阵、张量和函数等。如果编写一个程序执行某些数学或物理建模，就肯定会用类表示这些对象。
- 图形程序在计算屏幕上的位置时，也使用与数学或坐标相关的对象。
- 表示大量金钱的类(例如，在财务程序中)。
- 字处理或文本分析程序也有表示语句、子句等的类，可以使用运算符合并语句(这是字符串连接的一种比较复杂的版本)。

但是，也有许多类型与运算符重载并不相关。不恰当地使用运算符重载，会使使用类型的代码很难理解。例如，把两个 DateTime 对象相乘，在概念上没有任何意义。

7.5.1 运算符的工作方式

为了理解运算符是如何重载的，考虑一下在编译器遇到运算符时会发生什么情况很有用。用加法运算符(+)作为例子，假定编译器处理下面的代码：

```
int myInteger = 3;
uint myUnsignedInt = 2;
double myDouble = 4.0;
long myLong = myInteger + myUnsignedInt;
double myOtherDouble = myDouble + myInteger;
```

考虑当编译器遇到这行代码时会发生什么情况：

```
long myLong = myInteger + myUnsignedInt;
```

编译器知道它需要把两个整数加起来，并把结果赋予一个 long 型变量。调用一个方法把数字加在一起时，表达式 myInteger + myUnsignedInt 是一种非常直观和方便的语法。该方法接受两个参数 myInteger 和 myUnsignedInt，并返回它们的和。所以编译器完成的任务与任何方法调用一样——它会根据参数类型查找最匹配的“+”运算符重载，这里是带两个整数参数的“+”运算符重载。与一般的重载方法一样，预定义的返回类型不会因为编译器所调用方法的哪个版本而影响编译器的选择。在本例中调用的重载方法接受两个 int 参数，返回一个 int，这个返回值随后会转换为一个 long。

下一行代码让编译器使用“+”运算符的另一个重载版本：

```
double myOtherDouble = myDouble + myInteger;
```

在这个实例中，参数是一个 double 类型的数据和一个 int 类型的数据，但“+”运算符没有带这种复合参数的重载形式，所以编译器认为，最匹配的“+”运算符重载是把两个 double 作为其参数的版本，并隐式地把 int 强制转换为 double。把两个 double 加在一起与把两个整数加在一起完全不同，浮点数存储为一个尾数和一个指数。把它们加在一起要按位移动一个 double 的尾数，从而使两个指数有相同的值，然后把尾数加起来，移动所得尾数的位，调整其指数，保证答案有尽可能高的精度。

现在，看看如果编译器遇到下面的代码，会发生什么：

```
Vector vect1, vect2, vect3;
// initialize vect1 and vect2
vect3 = vect1 + vect2;
vect1 = vect1*2;
```

其中，Vector 是结构，稍后再定义它。编译器知道它需要把两个 Vector 实例加起来，即 vect1 和 vect2。它会查找“+”运算符的重载，重载的“+”运算符把两个 Vector 实例作为参数。

如果编译器找到这样的重载版本，它就调用该运算符的实现代码。如果找不到，它就要看看有没有可以用作最佳匹配的其他“+”运算符的重载，例如某个运算符重载对应的两个参数是其他数据类型，但可以隐式地转换为 Vector 实例。如果编译器找不到合适的运算符重载，就会产生一个编译错误，就像找不到其他方法调用的合适重载版本一样。

7.5.2　运算符重载的示例：Vector 结构

本小节将开发一个结构 Vector，来说明运算符重载，这个 Vector 结构表示一个三维数学矢量。如果数学不是你的强项，不必担心，我们会使这个例子尽可能简单。三维矢量只是 3 个(double)数字的一个集合，说明物体和原点之间的距离，表示数字的变量是 x、y 和 z，x 表示物体与原点在 x 方向上的距离，y 表示它与原点在 y 方向上的距离，z 表示高度。把这 3 个数字组合起来，就得到总距离。例如，如果 x=3.0, y=3.0, z=1.0，一般可以写作(3.0, 3.0, 1.0)，表示物体与原点在 x 方向上的距离是 3 个单位，与原点在 y 方向上的距离是 3 个单位，高度为 1 个单位。

矢量可以与矢量或数字相加或相乘。在这里我们还使用术语“标量”，它是数字的数学用语——在 C#中，就是一个 double。相加的作用很明显。如果先移动(3.0, 3.0, 1.0)矢量对应的距离，再移动(2.0,–4.0,–4.0)矢量对应的距离，总移动量就是把这两个矢量加起来。矢量的相加指把每个坐标轴对应的元素分别相加，因此得到(5.0,–1.0,–3.0)。此时，数学表达式总是写成 c=a+b，其中 a 和 b 是矢量，c 是结果矢量。这与使用 Vector 结构的方式一样。

这个例子将作为一个结构来开发，而不是类，但这并不重要。运算符重载用于结构和类时，其工作方式是一样的。

下面是 Vector 的定义——包含成员字段、构造函数和重写的一个 ToString()方法，以便轻松地查看 Vector 的内容，最后是运算符重载：

```
namespace Wrox.ProCSharp.OOCSharp
{
  struct Vector
  {
    public double x, y, z;

    public Vector(double x, double y, double z)
    {
      this.x = x;
      this.y = y;
      this.z = z;
    }

    public Vector(Vector rhs)
    {
      x = rhs.x;
      y = rhs.y;
      z = rhs.z;
    }

    public override string ToString()
    {
     return "( " + x + ", " + y + ", " + z + " )";
    }
```

这里提供了两个构造函数，通过传递每个元素的值，或者提供另一个复制其值的 Vector，来指定矢量的初始值。第二个构造函数带一个 Vector 参数，它通常称为复制构造函数，因为它们允许通过复制另一个实例来初始化一个类或结构实例。注意，为了简单起见，把字段设置为 public。也可以把它们设置为 private，编写相应的属性来访问它们，这样做不会改变这个程序的功能，只是代码会复杂一些。

下面是 Vector 结构的有趣部分——为“+”运算符提供支持的运算符重载：

```
    public static Vector operator + (Vector lhs, Vector rhs)
    {
      Vector result = new Vector(lhs);
      result.x += rhs.x;
      result.y += rhs.y;
      result.z += rhs.z;

      return result;
    }
  }
}
```

运算符重载的声明方式与方法相同，但 operator 关键字告诉编译器，它实际上是一个自定义的运算符重载，后面是相关运算符的实际符号，在本例中就是“+”。返回类型是在使用这个运算符时获得的类型。在本例中，把两个矢量加起来会得到另一个矢量，所以返回类型也是 Vector。对于这个“+”运算符重载，返回类型与包含的类一样，但这种情况并不是必需的，在本示例中稍后将看到。两个参数就是要操作的对象。对于二元运算符(它带两个参数)，如“+”和“-”运算符，第一个参数是运算符左边的值，第二个参数是运算符右边的值。

一般把运算符左边的参数命名为 lhs，运算符右边的的参数命名为 rhs。

C#要求所有的运算符重载都声明为 public 和 static，这表示它们与它们的类或结构相关联，而不是与某个特定实例相关联，所以运算符重载的代码体不能访问非静态类成员，也不能访问 this 标识符；这是可以的，因为参数提供了运算符执行其任务所需要知道的所有输入数据。

前面介绍了声明运算符“+”的语法，下面看看运算符内部的情况：

```
{
  Vector result = new Vector(lhs);
  result.x += rhs.x;
  result.y += rhs.y;
  result.z += rhs.z;

  return result;
}
```

这部分代码与声明方法的代码完全相同，显然，它返回一个矢量，其中包含前面定义的 lhs 和 rhs 的和，即把 x、y 和 z 成员分别相加。

下面需要编写一些简单的代码来测试 Vector 结构：

```
static void Main()
{
  Vector vect1, vect2, vect3;

  vect1 = new Vector(3.0, 3.0, 1.0);
  vect2 = new Vector(2.0, -4.0, -4.0);
  vect3 = vect1 + vect2;

  Console.WriteLine("vect1 = " + vect1.ToString());
  Console.WriteLine("vect2 = " + vect2.ToString());
  Console.WriteLine("vect3 = " + vect3.ToString());
}
```

把这些代码另存为 Vectors.cs，编译并运行它，结果如下：

```
vect1 = ( 3, 3, 1 )
vect2 = ( 2, -4, -4 )
vect3 = ( 5, -1, -3 )
```

1. 添加更多的重载

矢量除了可以相加之外，还可以相乘、相减和比较它们的值。本节通过添加几个运算符重载，扩展了这个 Vector 例子。这并不是一个功能齐全的真实的 Vector 类型，但足以说明运算符重载的其他方面了。首先要重载乘法运算符，以支持标量和矢量的相乘以及矢量和矢量的相乘。

矢量乘以标量只意味着矢量的元素分别与标量相乘，例如，2×(1.0, 2.5, 2.0)就等于(2.0, 5.0, 4.0)。相关的运算符重载如下所示：

```
public static Vector operator * (double lhs, Vector rhs)
{
   return new Vector(lhs * rhs.x, lhs * rhs.y, lhs * rhs.z);
}
```

但这还不够，如果 a 和 b 声明为 Vector 类型，就可以编写下面的代码：

```
b = 2 * a;
```

编译器会隐式地把整数 2 转换为 double 类型，以匹配运算符重载的签名。但不能编译下面的代码：

```
b = a * 2;
```

编译器处理运算符重载的方式和方法重载是一样的。它会查看给定运算符的所有可用重载，找到与之最匹配的那个运算符重载。上面的语句要求第一个参数是 Vector，第二个参数是整数，或者可以隐式转换为整数的其他数据类型。我们没有提供这样一个重载。有一个运算符重载，其参数依次是一个 double 和一个 Vector，但编译器不能交换参数的顺序，所以这是不行的。还需要显式地定义一个运算符重载，其参数依次是一个 Vector 和一个 double，有两种方式可以实现这样的运算符重载。第一种方式对矢量乘法进行分解，和处理所有运算符的方式一样，显式执行矢量相乘操作：

```
public static Vector operator * (Vector lhs, double rhs)
{
   return new Vector(rhs * lhs.x, rhs * lhs.y, rhs *lhs.z);
}
```

前面已经编写了实现相乘操作的代码，最好重用该代码：

```
public static Vector operator * (Vector lhs, double rhs)
{
   return rhs * lhs;
}
```

这段代码会告诉编译器，如果有 Vector 和 double 的相乘操作，编译器就颠倒参数的顺序，调用另一个运算符重载。本章的示例代码使用第二个版本，因为它看起来比较简洁，同时阐述了该行为的思想。利用这个版本可以编写出可维护性更好的代码，因为不需要复制代码，就可在两个独立的重载中执行相乘操作。

下一个要重载的乘法运算符支持矢量相乘。在数学上，矢量相乘有两种方式，但这里我们感兴趣的是点积或内积，其结果实际上是一个标量。这就是我们介绍这个例子的原因：算术运算符不必返回与定义它们的类相同的类型。

在数学术语中，如果有两个矢量(x, y, z)和(X, Y, Z)，其内积就定义为 x・X + y・Y + z・Z 的值。

两个矢量这样相乘很奇怪，但这实际上很有用，因为它可以用于计算各种其他的数。当然，如果要使用 Direct3D 或 DirectDraw 编写代码来显示复杂的 3D 图形，那么在计算对象放在屏幕上的什么位置时，中间常常需要编写代码来计算矢量的内积。这里我们关心的是使用 Vector 编写出 double X = a • b，其中 a 和 b 是两个矢量对象，并计算出它们的点积。相关的运算符重载如下所示：

```
public static double operator * (Vector lhs, Vector rhs)
{
   return lhs.x * rhs.x + lhs.y * rhs.y + lhs.z * rhs.z;
}
```

理解了算术运算符后，就可以用一个简单的测试方法来检验它们是否能正常运行：

```
static void Main()
{
    // stuff to demonstrate arithmetic operations
    Vector vect1, vect2, vect3;
    vect1 = new Vector(1.0, 1.5, 2.0);
    vect2 = new Vector(0.0, 0.0, -10.0);

    vect3 = vect1 + vect2;

    Console.WriteLine("vect1 = " + vect1);
    Console.WriteLine("vect2 = " + vect2);
    Console.WriteLine("vect3 = vect1 + vect2 = " + vect3);
    Console.WriteLine("2*vect3 = " + 2*vect3);
    vect3 += vect2;

    Console.WriteLine("vect3+=vect2 gives " + vect3);

    vect3 = vect1*2;

    Console.WriteLine("Setting vect3=vect1*2 gives " + vect3);

    double dot = vect1*vect3;

    Console.WriteLine("vect1*vect3 = " + dot);
}
```

运行代码(Vectors2.cs)，得到如下所示的结果：

VECTORS2

```
vect1 = ( 1, 1.5, 2 )
vect2 = ( 0, 0, -10 )
vect3 = vect1 + vect2 = ( 1, 1.5, -8 )
2*vect3 = ( 2, 3, -16 )
vect3+=vect2 gives ( 1, 1.5, -18 )
Setting vect3=vect1*2 gives ( 2, 3, 4 )
vect1*vect3 = 14.5
```

这说明，运算符重载会给出正确的结果，但如果仔细看看测试代码，就会惊奇地注意到，实际上它使用的是没有重载的运算符——相加赋值运算符(+=)：

```
vect3 += vect2;

Console.WriteLine("vect3 += vect2 gives " + vect3);
```

虽然“+=”一般用作单个运算符，但实际上它对应的操作分为两步：相加和赋值。与C++语言不同，C#不允许重载“=”运算符，但如果重载“+”运算符，编译器就会自动使用“+”运算符的重载来执行“+=”运算符的操作。-=、*=、/=和&=等所有赋值运算符也遵循此规则。

2. 比较运算符的重载

本章前面介绍过，C#中有6个比较运算符，它们分为3对：

- ==和!=
- >和<
- >=和<=

C#语言要求成对重载比较运算符。即，如果重载了“==”，也就必须重载“!=”；否则会产生编译错误。另外，比较运算符必须返回布尔类型的值。这是它们与算术运算符的根本区别。例如，两个数相加或相减的结果，理论上取决于数的类型。前面提到，两个Vector对象的相乘会得到一个标量。另一个例子是.NET基类System.DateTime，两个DateTime实例相减，得到的结果不是一个DateTime，而是一个System.TimeSpan实例。相比之下，如果比较运算得到的不是布尔类型的值，就没有任何意义。

在重载“==”和“!=”时，还必须重载从System.Object中继承的Equals()和GetHashCode()方法，否则会产生一个编译警告。原因是Equals()方法应实现与“==”运算符相同类型的相等逻辑。

除了这些区别外，重载比较运算符所遵循的规则与重载算术运算符相同。但比较两个数并不像想象的那么简单。例如，如果只比较两个对象引用，就是比较存储对象的内存地址。比较运算符很少进行这样的比较，所以必须编写代码重载运算符，比较对象的值，并返回相应的布尔结果。下面对Vector结构重载“==”和“!=”运算符。首先是实现“==”重载的代码：

```
public static bool operator == (Vector lhs, Vector rhs)
{
  if (lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z)
    return true;
  else
    return false;
}
```

这种方式仅根据矢量元素的值，来对它们进行相等性比较。对于大多数结构，这就是我们希望的，但在某些情况下，可能需要仔细考虑相等的含义。例如，如果有嵌入的类，那么是应比较引用是否指向同一个对象(浅度比较)，还是应比较对象的值是否相等(深度比较)？

浅度比较是比较对象是否指向内存中的同一个位置，而深度比较是比较对象的值和属性是否相等。应根据具体情况进行相等检查，从而有助于确定要验证什么。

不要通过调用从 System.Object 中继承的 Equals()方法的实例版本，来重载比较运算符。如果这么做，在 objA 是 null 时判断(objA==objB)，就会产生一个异常，因为.NET 运行库会试图判断 null.Equals(objB)。采用其他方法(重写 Equals()方法以调用比较运算符)比较安全。

还需要重载运算符“!=”，采用的方式如下：

```
public static bool operator != (Vector lhs, Vector rhs)
{
  return ! (lhs == rhs);
}
```

像往常一样，用一些测试代码检查重写方法的工作情况。这次定义 3 个 Vector 对象，并进行比较：

```
static void Main()
{
  Vector vect1, vect2, vect3;

  vect1 = new Vector(3.0, 3.0, -10.0);
  vect2 = new Vector(3.0, 3.0, -10.0);
  vect3 = new Vector(2.0, 3.0, 6.0);

  Console.WriteLine("vect1==vect2 returns " + (vect1==vect2));
  Console.WriteLine("vect1==vect3 returns " + (vect1==vect3));
  Console.WriteLine("vect2==vect3 returns " + (vect2==vect3));

  Console.WriteLine();

  Console.WriteLine("vect1!=vect2 returns " + (vect1!=vect2));
  Console.WriteLine("vect1!=vect3 returns " + (vect1!=vect3));
  Console.WriteLine("vect2!=vect3 returns " + (vect2!=vect3));
}
```

编译这些代码(代码下载中的 Vectors3.cs 示例)，会得到以下编译器警告，因为我们没有为 Vector 重写 Equals()。对于本例，这并不重要，所以忽略它。

```
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
for Microsoft (R) .NET Framework version 4.0
Copyright (C) Microsoft Corporation. All rights reserved.

Vectors3.cs(5,11): warning CS0660: 'Wrox.ProCSharp.OOCSharp.Vector' defines
        operator == or operator != but does not override Object.Equals(object o)
Vectors3.cs(5,11): warning CS0661: 'Wrox.ProCSharp.OOCSharp.Vector' defines
        operator == or operator != but does not override Object.GetHashCode()
```

在命令行上运行该示例，生成如下结果：

VECTORS3

```
vect1==vect2 returns True
vect1==vect3 returns False
vect2==vect3 returns False
```

```
vect1!=vect2 returns False
vect1!=vect3 returns True
vect2!=vect3 returns True
```

3. 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 7-6 所示。

表 7-6

<table>
<tr><th>类　别</th><th>运 算 符</th><th>限　制</th></tr>
<tr><td>算术二元运算符</td><td>+、*、/、-、%</td><td>无</td></tr>
<tr><td>算术一元运算符</td><td>+、-、++、--</td><td>无</td></tr>
<tr><td>按位二元运算符</td><td>&、|、^、<<、>></td><td>无</td></tr>
<tr><td>按位一元运算符</td><td>!、~、true、false</td><td>true 和 false 运算符必须成对重载</td></tr>
<tr><td>比较运算符</td><td>==、!=、>=、<、<=、></td><td>比较运算符必须成对重载</td></tr>
<tr><td>赋值运算符</td><td>+=、-=、*=、/=、>>=、<<=、%=、&=、|=、^=</td><td>不能显式地重载这些运算符，在重写单个运算符(如+、-、%等)时，它们会被隐式地重写</td></tr>
<tr><td>索引运算符</td><td>[]</td><td>不能直接重载索引运算符。第 2 章介绍的索引器成员类型允许在类和结构上支持索引运算符</td></tr>
<tr><td>数据类型强制转换运算符</td><td>()</td><td>不能直接重载类型强制转换运算符。用户定义的类型强制转换(本章的后面介绍)允许定义定制的类型强制转换行为</td></tr>
</table>

7.6 用户定义的类型强制转换

本章前面(见 7.3.1 小节中“2.显式转换”部分)介绍了如何在预定义的数据类型之间转换数值，这通过类型强制转换过程来完成。C#允许进行两种不同数据类型的强制转换：隐式强制转换和显式强制转换。本节将讨论这两种类型的强制转换。

显式强制转换要在代码中显式地标记强制转换，应该在圆括号中写出目标数据类型：

```
int I = 3;
long l = I;          // implicit
short s = (short)I;  // explicit
```

对于预定义的数据类型，当类型强制转换可能失败或丢失某些数据时，需要显式强制转换。例如：

- 把 int 转换为 short 时，因为 short 可能不够大，不能包含对应 int 的数值。
- 把有符号的数据类型转换为无符号的数据类型时，如果有符号的变量包含一个负值，就会得到不正确的结果。
- 把浮点数转换为整数数据类型时，数字的小数部分会丢失。
- 把可空类型转换为非可空类型时，null 值会导致异常。

此时应在代码中进行显式强制转换，告诉编译器你知道这会有丢失数据的危险，因此编写代码时要把这种可能性考虑在内。

C#允许定义自己的数据类型(结构和类)，这意味着需要某些工具支持在自定义的数据类型之间进行类型强制转换。方法是把类型强制转换运算符定义为相关类的一个成员运算符，类型强制转换运算符必须标记为隐式或显式，以说明希望如何使用它。我们应遵循与预定义的类型强制转换相同的规则，如果知道无论在源变量中存储什么值，类型强制转换总是安全的，就可以把它定义为隐式强制转换。然而，如果某些数值可能会出错，如丢失数据或抛出异常，就应把数据类型转换定义为显式强制转换。

如果源数据值会使类型强制转换失败，或者可能会抛出异常，就应把任何自定义类型强制转换定义为显式强制转换。

定义类型强制转换的语法类似于本章前面介绍的重载运算符。这并不是偶然的，类型强制转换在某种情况下可以看作是一种运算符，其作用是从源类型转换为目标类型。为了说明这种语法，下面的代码是从本节后面介绍的结构 Currency 示例中节选的：

```
public static implicit operator float (Currency value)
{
   // processing
}
```

运算符的返回类型定义了类型强制转换操作的目标类型，它有一个参数，即要转换的源对象。这里定义的类型强制转换可以隐式地把 Currency 型的值转换为 float 型。注意，如果数据类型转换声明为隐式，编译器就可以隐式或显式地使用这个转换。如果数据类型转换声明为显式，编译器就只能显式地使用它。与其他运算符重载一样，类型强制转换必须同时声明为 public 和 static。

C++开发人员应注意，这种情况与 C++中的用法不同，在 C++中，类型强制转换针对于类的实例成员。

7.6.1　实现用户定义的类型强制转换

本节将在示例 SimpleCurrency(和往常一样，其代码可以下载)中介绍隐式和显式的用户定义的类型强制转换的用法。在这个示例中，定义一个结构 Currency，它包含一个正的 USD($)金额。C#为此提供了 decimal 类型，但如果要进行比较复杂的财务处理，仍可以编写自己的结构和类来表示相应的金额，在这样的类上实现特定的方法。

类型强制转换的语法对于结构和类是一样的。本示例定义了一个结构，但如果把 Currency 声明为类，也是可以的。

首先，Currency 结构的定义如下所示。

```
struct Currency
{
  public uint Dollars;
```

```
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
      this.Dollars = dollars;
      this.Cents = cents;
    }

    public override string ToString()
    {
     return string.Format("${0}.{1,-2:00}", Dollars,Cents);
    }
  }
```

Dollars 和 Cents 字段使用无符号的数据类型，可以确保 Currency 实例只能包含正值。这样限制，是为了在后面说明显式强制转换的一些要点。可以像这样使用一个类来存储公司员工的薪水信息。人们的薪水不会是负值！为了使类比较简单，我们把字段声明为 public，但通常应把它们声明为 private，并为 Dollars 和 Cents 字段定义相应的属性。

下面先假定要把 Currency 实例转换为 float 值，其中 float 值的整数部分表示美元。换言之，应编写下面的代码：

```
Currency balance = new Currency(10,50);
float f = balance; // We want f to be set to 10.5
```

为此，需要定义一种类型强制转换。给 Currency 的定义添加下述代码：

```
public static implicit operator float (Currency value)
{
   return value.Dollars + (value.Cents/100.0f);
}
```

这种类型强制转换是隐式的。在本例中这是一个合理的选择，因为在 Currency 的定义中，可以存储在 Currency 中的值也都可以存储在 float 中。在这种强制转换中，不应出现任何错误。

这里有一点欺骗性：实际上，当把 uint 转换为 float 时，精确度会降低，但 Microsoft 认为这种错误并不重要，因此把从 uint 到 float 的强制转换都当作隐式转换。

但是，如果把 float 型转换为 Currency 型，就不能保证转换肯定成功了：float 型可以存储负值，而 Currency 实例不能，float 型存储的数值的数量级要比 Currency 型的(uint) Dollars 字段大得多。所以，如果 float 值包含一个不合适的值，把它转换为 Currency 型就会得到意想不到的结果。因此，从 float 型转换到 Currency 型就应定义为显式转换。下面是我们的第一次尝试，这次不会得到正确的结果，但有助于解释原因：

```
public static explicit operator Currency (float value)
{
  uint dollars = (uint)value;
  ushort cents = (ushort)((value-dollars)*100);
  return new Currency(dollars, cents);
}
```

下面的代码可以成功编译：

```
float amount = 45.63f;
  Currency amount2 = (Currency)amount;
```

但是，下面的代码会抛出一个编译错误，因为它试图隐式地使用一个显式的类型强制转换：

```
float amount = 45.63f;
  Currency amount2 = amount;   // wrong
```

把数据类型强制转换声明为显式，就是警告开发人员要小心，因为可能会丢失数据。但这不是我们希望的 Currency 结构的行为方式。下面编写一个测试程序，并运行该示例。其中有一个 Main() 方法，它实例化了一个 Currency 结构，并试图进行几个转换。在这段代码的开头，以两种不同的方式计算 balance 的值(因为要使用它们来说明后面的内容)：

```
static void Main()
{
  try
  {
    Currency balance = new Currency(50,35);

    Console.WriteLine(balance);
    Console.WriteLine("balance is " + balance);
    Console.WriteLine("balance is (using ToString()) " + balance.ToString());

    float balance2= balance;

    Console.WriteLine("After converting to float, = " + balance2);

    balance = (Currency) balance2;

    Console.WriteLine("After converting back to Currency, = " + balance);
    Console.WriteLine("Now attempt to convert out of range value of " +
                      "-$50.50 to a Currency:");

    checked
    {
      balance = (Currency) (-50.50);
      Console.WriteLine("Result is " + balance.ToString());
    }
  }
  catch(Exception e)
  {
   Console.WriteLine("Exception occurred: " + e.Message);
  }
}
```

注意，所有的代码都放在一个 try 块中，来捕获在类型强制转换过程中发生的任何异常。在 checked 块中还添加了把超出范围的值转换为 Currency 的测试代码，试图捕获负值。运行这段代码(SimpleCurrency)，得到如下所示的结果：

```
SIMPLECURRENCY

50.35
```

```
Balance is $50.35
Balance is (using ToString()) $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$100.00 to a Currency:
Result is $4294967246.00
```

这个结果表示代码并没有像我们希望的那样工作。首先，从 float 转换回 Currency 得到一个错误的结果$50.34，而不是$50.35。其次，在试图转换明显超出范围的值时，没有生成异常。

第一个问题是由舍入错误引起的。如果类型强制转换用于把 float 转换为 uint，计算机就会截去多余的数字，而不是四舍五入它。计算机以二进制方式存储数字，而不是十进制，小数部分 0.35 不能用二进制小数来精确表示(像 1/3 这样的分数不能精确地表示为十进制小数，它应等于循环小数 0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式精确地表示。把该数字乘以 100，就会得到一个小于 35 的数字，它截去了 34 美分。显然在本例中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能的四舍五入操作。

幸运的是，Microsoft 编写了一个类 System.Convert 来完成该任务。System.Convert 对象包含大量的静态方法来完成各种数字转换，我们需要使用的是 Convert.ToUInt16()。注意，在使用 System.Convert 类的方法时会造成额外的性能损失，所以只应在需要时才使用它们。

下面看看为什么没有抛出期望的溢出异常。此处的问题是溢出异常实际发生的位置根本不在 Main()例程中——它是在强制转换运算符的代码中发生的，该代码在 Main()方法中调用，而且没有标记为 checked。

其解决方法是确保类型强制转换本身也在 checked 环境下进行。进行了这两个修改后，修订后的转换代码如下所示。

```
public static explicit operator Currency (float value)
{
  checked
  {
    uint dollars = (uint)value;
    ushort cents = Convert.ToUInt16((value-dollars)*100);
    return new Currency(dollars, cents);
  }
}
```

注意，使用 Convert.ToUInt16()计算数字的美分部分，如上所示，但没有使用它计算数字的美元部分。在计算美元值时不需要使用 System.Convert，因为在此我们希望截去 float 值。

值得注意的是，System.Convert 类的方法还执行它们自己的溢出检查。因此对于本例的情况，不需要把对 Convert.ToUInt16()的调用放在 checked 环境下。但把 value 显式地强制转换为美元值仍需要 checked 环境。

这里没有给出这个新的 checked 强制转换的结果，因为在本节后面还要对 SimpleCurrency 示例进行一些修改。

如果定义了一种使用非常频繁的类型强制转换，其性能也非常好，就可以不进行任何错误检查。如果对用户定义的强制转换和没有检查的错误进行了清晰的说明，这也是一种合法的解决方案。

1. 类之间的类型强制转换

Currency 示例仅涉及与 float(一种预定义的数据类型)来回转换的类。但类型转换不一定会涉及任何简单的数据类型。定义不同结构或类的实例之间的类型强制转换是完全合法的，但有两个限制：

- 如果某个类派生自另一个类，就不能定义这两个类之间的类型强制转换(这些类型的类型转换已经存在)。
- 类型强制转换必须在源数据类型或目标数据类型的内部定义。

要说明这些要求，假定有如图 7-1 所示的类层次结构。

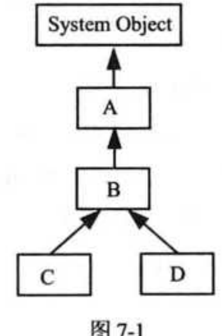


图 7-1

换言之，类 C 和 D 间接派生于 A。在这种情况下，在 A、B、C 或 D 之间唯一合法的自定义类型强制转换就是类 C 和 D 之间的转换，因为这些类并没有互相派生。这段代码如下所示(假定希望类型强制转换是显式的，这是在用户定义的类之间定义类型强制转换的通常情况)：

```
public static explicit operator D(C value)
{
   // and so on
}
public static explicit operator C(D value)
{
   // and so on
}
```

对于这些类型强制转换，可以选择放置定义的地方——在 C 的类定义内部，或者在 D 的类定义内部，但不能在其他地方定义。C#要求把类型强制转换的定义放在源类(或结构)或目标类(或结构)的内部。它的副作用是不能定义两个类之间的类型强制转换，除非至少可以编辑其中一个类的源代码。这是因为，这样可以防止第三方把类型强制转换引入类中。

一旦在一个类的内部定义了类型强制转换，就不能在另一个类中定义相同的类型强制转换。显然，对于每一种转换只能有一种类型强制转换，否则编译器就不知道该选择哪个类型强制转换了。

2. 基类和派生类之间的类型强制转换

要了解这些类型强制转换是如何工作的，首先看看源和目标的数据类型都是引用类型的情况。考虑两个类 MyBase 和 MyDerived，其中 MyDerived 直接或间接派生自 MyBase。

首先是从 MyDerived 到 MyBase 的转换，代码如下(假定可以使用构造函数)：

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

在本例中，是从 MyDerived 隐式地强制转换为 MyBase。这是可行的，因为对类型 MyBase 的

任何引用都可以引用 MyBase 类的对象或派生自 MyBase 的对象。在 OO 编程中，派生类的实例实际上是基类的实例，但加上了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上定义了。

下面看看另一种方式，编写下面的代码：

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject;  // OK
MyDerived derivedCopy2 = (MyDerived) baseObject;     // Throws exception
```

上面的代码都是合法的 C#代码(从语法的角度来看，它是合法的)，它说明了把基类强制转换为派生类。但是，在执行时最后一条语句会抛出一个异常。在进行类型强制转换时，会检查被引用的对象。因为基类引用原则上可以引用一个派生类的实例，所以这个对象可能是要强制转换的派生类的一个实例。如果是这样，强制转换就会成功，派生的引用被设置为引用这个对象。但如果该对象不是派生类(或者派生于这个类的其他类)的一个实例，强制转换就会失败，并抛出一个异常。

注意，编译器已经提供了基类和派生类之间的强制转换，这种转换实际上并没有对对象进行任何数据转换。如果要进行的转换是合法的，它们也仅是把新引用设置为对对象的引用。这些强制转换在本质上与用户定义的强制转换不同。例如，在前面的 SimpleCurrency 示例中，我们定义了 Currency 结构和 float 数之间的强制转换。在 float 到 Currency 的强制转换中，实际上实例化了一个新的 Currency 结构，并用要求的值初始化它。在基类和派生类之间的预定义强制转换则不是这样。如果实际上要把 MyBase 实例转换为真实的 MyDerived 对象，该对象的值根据 MyBase 实例的内容来确定，就不能使用类型强制转换语法。最合适的选项通常是定义一个派生类的构造函数，它以基类的实例作为参数，让这个构造函数完成相关的初始化：

```
class DerivedClass: BaseClass
{
  public DerivedClass(BaseClass rhs)
  {
     // initialize object from the Base instance
  }
  // etc.
```

3. 装箱和拆箱数据类型强制转换

前面主要讨论了基类和派生类之间的数据类型强制转换，其中，基类和派生类都是引用类型。类似的规则也适用于强制转换值类型，尽管在转换值类型时，不可能仅仅复制引用，还必须复制一些数据。

当然，不能从结构或基元值类型中派生。所以基本结构和派生结构之间的强制转换总是基元类型或结构与 System.Object 之间的转换(理论上可以在结构和 System.ValueType 之间进行强制转换，但一般很少这么做)。

从结构(或基本类型)到 object 的强制转换总是一种隐式的强制转换，因为这种强制转换是从派生类型到基类型的转换，即第 2 章简要介绍的装箱过程。例如，Currency 结构：

```
Currency balance = new Currency(40,0);
object baseCopy = balance;
```

在执行上述隐式的强制转换时，balance 的内容被复制到堆上，放在一个装箱的对象上，baseCopy 对象引用被设置为该对象。实际上在后台发生的情况是：在最初定义 Currency 结构时，.NET Framework 隐式地提供另一个(隐藏的)类，即装箱的 Currency 类，它包含与 Currency 结构相同的所有字段，但它是一个引用类型，存储在堆上。无论定义的这个值类型是一个结构，还是一个枚举，定义它时都存在类似的装箱引用类型，对应于所有的基元值类型，如 int、double 和 uint 等。不能也不必在源代码中直接通过编程访问某些装箱类，但在把一个值类型强制转换为 object 时，它们是在后台工作的对象。在隐式地把 Currency 转换为 object 时，会实例化一个装箱的 Currency 实例，并用 Currency 结构中的所有数据进行初始化。在上面的代码中，baseCopy 对象引用的就是这个已装箱的 Currency 实例。通过这种方式，就可以实现从派生类型到基类型的强制转换，并且，值类型的语法与引用类型的语法一样。

强制转换的另一种方式称为拆箱。与在基引用类型和派生引用类型之间的强制转换一样，这是一种显式的强制转换，因为如果要强制转换的对象不是正确的类型，就会抛出一个异常：

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject; // OK
Currency derivedCopy2 = (Currency)baseObject;    // Exception thrown
```

上述代码的工作方式与前面的引用类型中的代码一样。把 derivedObject 强制转换为 Currency 会成功进行，因为 derivedObject 实际上引用的是装箱 Currency 实例——强制转换的过程是把已装箱的 Currency 对象的字段复制到一个新的 Currency 结构中。第二种强制转换会失败，因为 baseObject 没有引用已装箱的 Currency 对象。

在使用装箱和拆箱时，这两个过程都把数据复制到新装箱或拆箱的对象上，理解这一点非常重要。这样，例如，对装箱对象的操作就不会影响原始值类型的内容。

7.6.2 多重类型强制转换

在定义类型强制转换时必须考虑的一个问题是，如果在进行要求的数据类型转换时，C#编译器没有可用的直接强制转换方式，C#编译器就会寻找一种转换方式，把几种强制转换合并起来。例如，在 Currency 结构中，假定编译器遇到下面几行代码：

```
Currency balance = new Currency(10,50);
long amount = (long)balance;
double amountD = balance;
```

首先初始化一个 Currency 实例，再把它转换为一个 long 数。问题是不能定义这样的强制转换。但是，这段代码仍可以编译成功。因为编译器知道我们已经定义一个从 Currency 到 float 的隐式强制转换，而且它知道如何显式地从 float 强制转换为 long。所以它会把这行代码编译为中间语言(IL)代码，IL 代码首先把 balance 转换为 float，再把结果转换为 long。把 balance 转换为 double 型时，在上述代码的最后一行中也是这样。因为从 Currency 到 float 的强制转换和从 float 到 double 的预定义强制转换都是隐式的，所以可以在编写代码时把这种转换当作一种隐式转换。如果要显式地指定强制转换过程，则可以编写如下代码：

```
Currency balance = new Currency(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

但是，在大多数情况下，这会使代码变得比较复杂，因此是不必要的。相比之下，下面的代码会产生一个编译错误：

```
Currency balance = new Currency(10,50);
long amount = balance;
```

原因是编译器可以找到的最佳匹配的转换仍是首先转换为 float，再转换为 long。但从 float 到 long 的转换需要显式地指定。

并非所有这些转换都会带来太多的麻烦。毕竟转换的规则非常直观，主要是为了防止在开发人员不知情的情况下丢失数据。但是，在定义类型强制转换时如果不小心，编译器就有可能指定一条导致不期望的结果的路径。例如，假定编写 Currency 结构的其他小组成员要把一个 uint 转换为 Currency，其中该 uint 中包含了美分的总数(美分不是美元，因为我们不希望丢掉美元的小数部分)。为此应编写如下代码来实现强制转换：

```
public static implicit operator Currency (uint value)
{
   return new Currency(value/100u, (ushort)(value%100));
} // Do not do this!
```

注意，在这段代码中，第一个 100 后面的 u 可以确保把 value/100u 解释为一个 uint 数。如果写成 value/100，编译器就会把它解释为一个 int 型的值，而不是 uint 型的值。

在这段代码中清楚地标注了“不要这么做”。下面说明其原因。看看下面的代码段，它把包含 350 的一个 uint 转换为一个 Currency，再转换回 uint。那么在执行完这段代码后，bal2 中又将包含什么？

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

答案不是 350，而是 3！而且这是符合逻辑的。我们把 350 隐式地转换为 Currency，得到的结果是 balance.Dollars=3，balance.Cents=50。然后编译器进行通常的操作，为转换回 uint 指定最佳路径。balance 最终会被隐式地转换为 float 型(其值为 3.5)，然后显式地转换为 uint 型，其值为 3。

当然，在其他示例中，转换为另一种数据类型后，再转换回来有时会丢失数据。例如，把包含 5.8 的 float 数转换为 int 数，再转换回 float 数，会丢失数字中的小数部分，得到 5，但原则上丢失数字的小数部分和一个整数被大于 100 的数整除的情况略有区别。Currency 现在成了一种相当危险的类，它会对整数进行一些奇怪的操作。

问题是，在转换过程中如何解释整数存在冲突。从 Currency 到 float 的强制转换会把整数 1 解释为 1 美元，但从 uint 到 Currency 的强制转换会把这个整数解释为 1 美分，这是很糟糕的一个示例。如果希望类易于使用，就应确保所有的强制转换都按一种互相兼容的方式执行，即这些转换直观上应得到相同的结果。在本例中，显然要重新编写从 uint 到 Currency 的强制转换，把整数值 1 解释为 1 美元：

```
public static implicit operator Currency (uint value)
{
   return new Currency(value, 0);
}
```

偶尔也会觉得这种新的转换方式可能根本不需要。但实际上这种转换方式可能非常有用。没有

这种强制转换，编译器在执行从 uint 到 Currency 的转换时，就只能通过 float 来进行。此时直接转换的效率要高得多，所以进行这种额外的强制转换会提高性能，但需要确保它的结果与通过 float 进行转换得到的结果相同。在其他情况下，也可以为不同的预定义数据类型分别定义强制转换，让更多的转换隐式地执行，而不是显式地执行，但本例不是这样。

测试这种强制转换是否兼容，应确定无论使用什么转换路径，它是否都能得到相同的结果(而不是像在从 float 到 int 的转换过程中那样丢失数据)。Currency 类就是一个很好的示例。看看下面的代码：

```
Currency balance = new Currency(50, 35);
ulong bal = (ulong) balance;
```

目前，编译器只能采用一种方式来完成这个转换：把 Currency 隐式地转换为 float，再显式地转换为 ulong。从 float 到 ulong 的转换需要显式转换，本例就显式指定了这个转换，所以编译是成功的。

但假定要添加另一种强制转换，从 Currency 隐式地转换为 uint，就需要修改 Currency 结构，添加从 uint 到 Currency 的强制转换和从 Currency 到 uint 的强制转换。这段代码可以作为 SimpleCurrency2 示例：

```
public static implicit operator Currency (uint value)
{
   return new Currency(value, 0);
}

public static implicit operator uint (Currency value)
{
   return value.Dollars;
}
```

现在，编译器从 Currency 转换到 ulong 可以使用另一条路径：先从 Currency 隐式地转换为 uint，再隐式地转换为 ulong。该采用哪条路径？ C#有一些严格的规则(本书不详细讨论这些规则，读者可参阅 MSDN 文档)，告诉编译器如何确定哪条是最佳路径。但最好自己设计类型强制转换，让所有的转换路径都得到相同的结果(但没有精确度的损失)，此时编译器选择哪条路径就不重要了(在本例中，编译器会选择 Currency→uint→ulong 路径，而不是 Currency→float→ulong 路径)。

为了测试 SimpleCurrency2 示例，给 SimpleCurrency 的测试程序添加如下代码：

```
try
{
  Currency balance = new Currency(50,35);

  Console.WriteLine(balance);
  Console.WriteLine("balance is " + balance);
  Console.WriteLine("balance is (using ToString()) " + balance.ToString());

  uint balance3 = (uint) balance;

  Console.WriteLine("Converting to uint gives " + balance3);
```

运行这个示例，得到如下所示的结果：

SIMPLECURRENCY2

```
50
balance is $50.35
```

```
balance is (using ToString()) $50.35
Converting to uint gives 50
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$50.50 to a Currency:
Result is $4294967246.00
```

这个结果显示了到 uint 的转换是成功的，但在转换过程中丢失了 Currency 的美分部分(小数部分)。把负的 float 类型强制转换为 Currency 也产生了预料中的溢出异常，因为 float 到 Currency 的强制转换本身定义了一个 checked 环境。

但是，这个输出结果也说明了进行强制转换时最后一个要注意的潜在问题：结果的第一行没有正确显示余额，显示了 50，而不是$50.35。在下面的代码中：

```
Console.WriteLine(balance);
Console.WriteLine("balance is " + balance);
Console.WriteLine("balance is (using ToString()) " + balance.ToString());
```

只有最后两行把 Currency 正确地显示为一个字符串。这是为什么？问题是在把类型强制转换和方法重载合并起来时，会出现另一个不希望的错误源。下面用倒序的方式解释这段代码。

第 3 行的 Console.WriteLine()语句显式地调用 Currency.ToString()方法，以确保 Currency 显示为一个字符串。第 2 行代码没有这么做。字符串“balance is”传递给 Console.WriteLine()，告诉编译器这个参数应解释为字符串。因此要隐式地调用 Currency.ToString()方法。

但第 1 行的 Console.WriteLine()方法只把原始 Currency 结构传递给 Console.WriteLine()。目前 Console.WriteLine()有许多重载版本，但它们的参数都不是 Currency 结构。所以编译器会到处搜索，看看它能把 Currency 强制转换为什么，以便与 Console.WriteLine()的一个重载方法匹配。如上所示，Console.WriteLine()的一个重载方法可以快速而高效地显示 uint，且其参数是一个 uint。因此应把 Currency 隐式地强制转换为 uint。

实际上，Console.WriteLine()有另一个重载方法，它的参数是一个 double 数，结果显示该 double 数的值。如果仔细看看第一个 SimpleCurrency 示例的结果，就会发现该结果的第 1 行就是使用这个重载方法把 Currency 显示为一个 double 数。在这个示例中，没有直接把 Currency 强制转换为 uint，所以编译器选择 Currency→float→double 作为可用于 Console.WriteLine()重载方法首选的强制转换方式。但在 SimpleCurrency2 中可以直接强制转换为 uint，所以编译器会选择该路径。

结论是：如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些强制转换方式进行数据转换，从而决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果存在任何疑问，最好指定显式地使用哪种强制转换。

7.7　小结

本章介绍了 C#提供的标准运算符，描述了对象的相等机制，讨论了编译器如何把一种标准数据类型转换为另一种标准数据类型。还阐述了如何使用运算符重载在自己的数据类型上实现自定义运算符。最后，学习了运算符重载的一种特殊类型，即类型强制转换运算符，它允许用户指定如何将自定义类型的实例转换为其他数据类型。

第 8 章

委托、lambda 表达式和事件

本章要点

- 委托
- lambda 表达式
- 闭包
- 事件
- 弱事件

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 简单委托(Simple Delegates)
- 冒泡排序(Bubble Sorter)
- lambda 表达式(lambda Expressions)
- 事件示例(Events Sample)
- 弱事件(Weak Events)

8.1 引用方法

委托是寻址方法的.NET 版本。在 C++中，函数指针只不过是一个指向内存位置的指针，它不是类型安全的。我们无法判断这个指针实际指向什么，像参数和返回类型等项就更无从知晓了。而.NET 委托完全不同，委托是类型安全的类，它定义了返回类型和参数的类型。委托类不仅包含对方法的引用，也可以包含对多个方法的引用。

lambda 表达式与委托直接相关。当参数是委托类型时，就可以使用 lambda 表达式实现委托引用的方法。

本章学习委托和 lambda 表达式的基础知识，说明如何通过 lambda 表达式实现委托调用，并阐

述.NET 如何将委托用作实现事件的方式。

8.2 委托

当要把方法传送给其他方法时，需要使用委托。要了解它们的含义，可以看看下面一行代码：

```
int i = int.Parse("99");
```

我们习惯于把数据作为参数传递给方法，如上面的例子所示。所以，给方法传递另一个方法听起来有点奇怪。而有时某个方法执行的操作并不是针对数据进行的，而是要对另一个方法进行操作。更麻烦的是，在编译时我们不知道第二个方法是什么，这个信息只能在运行时得到，所以需要把第二个方法作为参数传递给第一个方法。这听起来很令人迷惑，下面用几个示例来说明：

- **启动线程和任务**——在 C#中，可以告诉计算机并行运行某些新的执行序列，同时运行当前的任务。这种序列就称为线程，在其中一个基类 System.Threading.Thread 的一个实例上使用方法 Start()，就可以启动一个线程。如果要告诉计算机启动一个新的执行序列，就必须说明要在哪里启动该序列。必须为计算机提供开始启动的方法的细节，即 Thread 类的构造函数必须带有一个参数，该参数定义了线程调用的方法。
- **通用库类**——许多库包含执行各种标准任务的代码。这些库通常可以自我包含。这样在编写库时，就会知道任务该如何执行。但是有时在任务中还包含子任务，只有使用该库的客户端代码才知道如何执行这些子任务。例如，编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能对任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户端代码也必须告诉类如何比较要排序的特定对象。换言之，客户端代码必须给类传递某个可以调用并且进行这种比较的合适方法的细节。
- **事件**——一般是通知代码发生了什么事件。GUI 编程主要处理事件。在引发事件时，运行库需要知道应执行哪个方法。这就需要把处理事件的方法作为一个参数传递给委托。这些将在本章后面讨论。

在 C 和 C++中，只能提取函数的地址，并作为一个参数传递它。C 没有类型安全性。可以把任何函数传递给需要函数指针的方法。但是，这种直接方法不仅会导致一些关于类型安全性的问题，而且没有意识到：在进行面向对象编程时，几乎没有方法是孤立存在的，而是在调用方法前通常需要与类实例相关联。所以.NET Framework 在语法上不允许使用这种直接方法。如果要传递方法，就必须把方法的细节封装在一种新类型的对象中，即委托。委托只是一种特殊类型的对象，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是一个或多个方法的地址。

8.2.1 声明委托

在 C#中使用一个类时，分两个阶段。首先，需要定义这个类，即告诉编译器这个类由什么字段和方法组成。然后(除非只使用静态方法)，实例化类的一个对象。使用委托时，也需要经过这两个步骤。首先必须定义要使用的委托，对于委托，定义它就是告诉编译器这种类型的委托表示哪种类型的方法。然后，必须创建该委托的一个或多个实例。编译器在后台将创建表示该委托的一个类。

定义委托的语法如下：

```
delegate void IntMethodInvoker(int x);
```

在这个示例中，定义了一个委托 IntMethodInvoker，并指定该委托的每个实例都可以包含一个方法的引用，该方法带有一个 int 参数，并返回 void。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所表示的方法的签名和返回类型等全部细节。

理解委托的一种好方式是把委托当作这样一件事情，它给方法的签名和返回类型指定名称。

假定要定义一个委托 TwoLongsOp，该委托表示的方法有两个 long 型参数，返回类型为 double。可以编写如下代码：

```
delegate double TwoLongsOp(long first, long second);
```

或者要定义一个委托，它表示的方法不带参数，返回一个 string 型的值，可以编写如下代码：

```
delegate string GetAString();
```

其语法类似于方法的定义，但没有方法体，定义的前面要加上关键字 delegate。因为定义委托基本上是定义一个新类，所以可以在定义类的任何相同地方定义委托。也就是说，可以在另一个类的内部定义，也可以在任何类的外部定义，还可以在名称空间中把委托定义为顶层对象。根据定义的可见性和委托的作用域，可以在委托的定义上应用任意常见的访问修饰符：public、private、protected 等：

```
public delegate string GetAString();
```

实际上，"定义一个委托"是指"定义一个新类"。委托实现为派生自基类 System.MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 System.Delegate。C#编译器能识别这个类，会使用其委托语法，因此我们不需要了解这个类的具体执行情况。这是 C#与基类共同合作，使编程更易完成的另一个范例。

定义好委托后，就可以创建它的一个实例，从而用它存储特定方法的细节。

但是，在术语方面有一个问题。类有两个不同的术语："类"表示较广义的定义，"对象"表示类的实例。但委托只有一个术语。在创建委托的实例时，所创建的委托的实例仍称为委托。必须从上下文中确定委托的确切含义。

8.2.2　使用委托

下面的代码段说明了如何使用委托。这是在 int 上调用 ToString()方法的一种相当冗长的方式(代码

文件 GetAStringDemo/Program.cs)：

```
private delegate string GetAString();

static void Main()
    {
      int x = 40;
      GetAString firstStringMethod = new GetAString(x.ToString);
      Console.WriteLine("String is {0}", firstStringMethod());
      // With firstStringMethod initialized to x.ToString(),
      // the above statement is equivalent to saying
      // Console.WriteLine("String is {0}", x.ToString());
    }
```

在这段代码中，实例化了类型为 GetAString 的一个委托，并对它进行初始化，使它引用整型变量 x 的 ToString()方法。在 C#中，委托在语法上总是接受一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数并返回一个字符串的方法来初始化 firstStringMethod 变量，就会产生一个编译错误。注意，因为 int.ToString()是一个实例方法(不是静态方法)，所以需要指定实例(x)和方法名来正确地初始化委托。

下一行代码使用这个委托来显示字符串。在任何代码中，都应提供委托实例的名称，后面的圆括号中应包含调用该委托中的方法时使用的任何等效参数。所以在上面的代码中，Console.WriteLine()语句完全等价于注释语句中的代码行。

实际上，给委托实例提供圆括号与调用委托类的 Invoke()方法完全相同。因为 firstStringMethod 是委托类型的一个变量，所以 C#编译器会用 firstStringMethod.Invoke()代替 firstStringMethod()。

```
firstStringMethod();
firstStringMethod.Invoke();
```

为了减少输入量，只要需要委托实例，就可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个 C#特性就是有效的。下面的示例用 GetAString 委托的一个新实例初始化 GetAString 类型的 firstStringMethod 变量：

```
GetAString firstStringMethod = new GetAString(x.ToString);
```

只要用变量 x 把方法名传送给变量 firstStringMethod，就可以编写出作用相同的代码：

```
GetAString firstStringMethod = x.ToString;
```

C#编译器创建的代码是一样的。由于编译器会用 firstStringMethod 检测需要的委托类型，因此它创建 GetAString 委托类型的一个实例，用对象 x 把方法的地址传送给构造函数。

调用上述方法名时输入形式不能为 x.ToString()(不要输入圆括号)，也不能把它传送给委托变量。输入圆括号调用一个方法。调用 x.ToString()方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。

委托推断可以在需要委托实例的任何地方使用。委托推断也可以用于事件，因为事件基于委托(参见本章后面的内容)。

委托的一个特征是它们的类型是安全的，可以确保被调用的方法的签名是正确的。但有趣的是，它们不关心在什么类型的对象上调用该方法，甚至不考虑该方法是静态方法，还是实例方法。

给定委托的实例可以引用任何类型的任何对象上的实例方法或静态方法——只要方法的签名匹配于委托的签名即可。

为了说明这一点，扩展上面的代码，让它使用 firstStringMethod 委托在另一个对象上调用其他两个方法，其中一个是实例方法，另一个是静态方法。为此，使用本章前面定义的 Currency 结构。Currency 结构有自己的 ToString()重载方法和一个与 GetCurrencyUnit()的签名相同的静态方法。这样，就可以用同一个委托变量调用这些方法了(代码文件 GetAStringDemo/Currency.cs)：

```
struct Currency
{
  public uint Dollars;
  public ushort Cents;

  public Currency(uint dollars, ushort cents)
  {
    this.Dollars = dollars;
    this.Cents = cents;
  }

  public override string ToString()
  {
   return string.Format("${0}.{1,2:00}", Dollars,Cents);
  }

  public static string GetCurrencyUnit()
  {
   return "Dollar";
  }

  public static explicit operator Currency (float value)
  {
    checked
    {
      uint dollars = (uint)value;
      ushort cents = (ushort)((value - dollars) * 100);
      return new Currency(dollars, cents);
    }
  }

  public static implicit operator float (Currency value)
  {
   return value.Dollars + (value.Cents / 100.0f);
  }

  public static implicit operator Currency (uint value)
  {
   return new Currency(value, 0);
```

```
    }

    public static implicit operator uint (Currency value)
    {
      return value.Dollars;
    }
  }
```

下面就可以使用 GetAString 实例，代码如下所示：

```
private delegate string GetAString();

static void Main()
{
  int x = 40;
  GetAString firstStringMethod = x.ToString;
  Console.WriteLine("String is {0}", firstStringMethod());

  Currency balance = new Currency(34, 50);

  // firstStringMethod references an instance method
  firstStringMethod = balance.ToString;
  Console.WriteLine("String is {0}", firstStringMethod());

  // firstStringMethod references a static method
  firstStringMethod = new GetAString(Currency.GetCurrencyUnit);
  Console.WriteLine("String is {0}", firstStringMethod());
}
```

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上引用的不同方法，甚至可以指定静态方法，或者在类的不同类型的实例上引用的方法，只要每个方法的签名匹配委托定义即可。

运行应用程序，会得到委托引用的不同方法的输出结果：

```
String is 40
String is $34.50
String is Dollar
```

但是，我们实际上还没有说明把一个委托传递给另一个方法的具体过程，也没有得到任何特别有用的结果。调用 int 和 Currency 对象的 ToString()的方法要比使用委托直观得多！但是，需要用一个相当复杂的示例来说明委托的本质，才能真正领会到委托的用处。下面就是两个委托的示例。第一个示例仅使用委托来调用两个不同的操作。它说明了如何把委托传递给方法，如何使用委托数组，但这仍没有很好地说明：没有委托，就不能完成很多工作。第二个示例就复杂得多了，它有一个类 BubbleSorter，该类实现一个方法，按照升序排列一个对象数组。没有委托很难编写出这个类。

8.2.3 简单的委托示例

在这个示例中，定义一个类 MathOperations，它有两个静态方法，对 double 类型的值执行两个操作，然后使用该委托调用这些方法。这个数学类如下所示：

```
class MathOperations
```

```
{
  public static double MultiplyByTwo(double value)
  {
    return value * 2;
  }

  public static double Square(double value)
  {
    return value * value;
  }
}
```

下面调用这些方法(代码文件 SimpleDelegate/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  delegate double DoubleOp(double x);

  class Program
  {
    static void Main()
    {
      DoubleOp[] operations =
      {
        MathOperations.MultiplyByTwo,
        MathOperations.Square
      };

      for (int i=0; i < operations.Length; i++)
      {
        Console.WriteLine("Using operations[{0}]:", i);
        ProcessAndDisplayNumber(operations[i], 2.0);
        ProcessAndDisplayNumber(operations[i], 7.94);
        ProcessAndDisplayNumber(operations[i], 1.414);
        Console.WriteLine();
      }
    }

    static void ProcessAndDisplayNumber(DoubleOp action, double value)
    {
      double result = action(value);
      Console.WriteLine("Value is {0}, result of operation is {1}",
                        value, result);
    }
  }
}
```

在这段代码中，实例化了一个 DoubleOp 委托的数组(记住，一旦定义了委托类，基本上就可以实例化它的实例，就像处理一般的类那样——所以把一些委托的实例放在数组中是可以的)。该数组的每个元素都初始化为由 MathOperations 类实现的不同操作。然后遍历这个数组，把每个操作应用到 3 个不同的值上。这说明了使用委托的一种方式——把方法组合到一个数组中来使用，这样就可

以在循环中调用不同的方法了。

这段代码的关键一行是把每个委托传递给 ProcessAndDisplayNumber 方法，例如：

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

其中传递了委托名，但不带任何参数。假定 operations[i]是一个委托，其语法是：

- operations[i]表示“这个委托”。换言之，就是委托表示的方法。
- operations[i](2.0)表示“实际上调用这个方法，参数放在圆括号中”。

ProcessAndDisplayNumber 方法定义为把一个委托作为其第一个参数：

```
static void ProcessAndDisplayNumber(DoubleOp action, double value)
```

然后，在这个方法中，调用：

```
double result = action(value);
```

这实际上是调用 action 委托实例封装的方法，其返回结果存储在 result 中。运行这个示例，得到如下所示的结果：

```
SimpleDelegate
Using operations[0]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 15.88
Value is 1.414, result of operation is 2.828

Using operations[1]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 63.0436
Value is 1.414, result of operation is 1.999396
```

8.2.4　Action<T>和 Func<T>委托

除了为每个参数和返回类型定义一个新委托类型之外，还可以使用 Action<T>和 Func<T>委托。泛型 Action<T>委托表示引用一个 void 返回类型的方法。这个委托类存在不同的变体，可以传递至多 16 种不同的参数类型。没有泛型参数的 Action 类可调用没有参数的方法。Action<in T>调用带一个参数的方法，Action<in T1, in T2>调用带两个参数的方法，Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8>调用带 8 个参数的方法。

Func<T>委托可以以类似的方式使用。Func<T>允许调用带返回类型的方法。与 Action<T>类似，Func<T>也定义了不同的变体，至多也可以传递 16 个参数类型和一个返回类型。Func<out TResult>委托类型可以调用带返回类型且无参数的方法，Func<in T, out TResult>调用带一个参数的方法，Func<in T1, in T2, in T3, in T4,out TResult>调用带 4 个参数的方法。

8.2.3 小节的示例声明了一个委托，其参数是 double 类型，返回类型是 double：

```
delegate double DoubleOp(double x);
```

除了声明自定义委托 DoubleOp 之外，还可以使用 Func<in T, out TResult>委托。可以声明一个该委托类型的变量，或者该委托类型的数组，如下所示：

```
Func<double, double>[] operations =
```

```
{
  MathOperations.MultiplyByTwo,
  MathOperations.Square
};
```

使用它，并将 ProcessAndDisplayNumber()方法作为参数：

```
static void ProcessAndDisplayNumber(Func<double, double> action,
                                     double value)
{
  double result = action(value);
  Console.WriteLine("Value is {0}, result of operation is {1}",
                    value, result);
}
```

8.2.5 BubbleSorter 示例

下面的示例将说明委托的真正用途。我们要编写一个类 BubbleSorter，它实现一个静态方法 Sort()，这个方法的第一个参数是一个对象数组，把该数组按照升序重新排列。例如，假定传递给它的是 int 数组：{0, 5, 6, 2, 1}，则返回的结果应是{0, 1, 2, 5, 6}。

冒泡排序算法非常著名，是一种简单的排序方法。它适合于一小组数字，因为对于大量的数字(超过 10 个)，还有更高效的算法。冒泡排序算法重复遍历数组，比较每一对数字，按照需要交换它们的位置，从而把最大的数字逐步移动到数组的最后。对于给 int 排序，进行冒泡排序的方法如下所示：

```
bool swapped = true;
do
{
  swapped = false;
  for (int i = 0; i < sortArray.Length - 1; i++)
  {
    if (sortArray[i] > sortArray[i+1])) // problem with this test
    {
      int temp = sortArray[i];
      sortArray[i] = sortArray[i + 1];
      sortArray[i + 1] = temp;
      swapped = true;
    }
  }
} while (swapped);
```

它非常适合于 int，但我们希望 Sort()方法能给任何对象排序。换言之，如果某段客户端代码包含 Currency 结构或自定义的其他类和结构的数组，就需要对该数组排序。这样，上面代码中的 if(sortArray[i] < sortArray[i+1])就有问题了，因为它需要比较数组中的两个对象，看看哪一个更大。可以对 int 进行这样的比较，但如何对没有实现“<”运算符的新类进行比较？答案是能识别该类的客户端代码必须在委托中传递一个封装的方法，这个方法可以进行比较。另外，不给 temp 变量使用 int 类型，而使用泛型类型就可以实现泛型方法 Sort()。

对于接受类型 T 的泛型方法 Sort<T>()，需要一个比较方法，其两个参数的类型是 T，if 比较的返回类型是布尔类型。这个方法可以从 Func<T1, T2, TResult>委托中引用，其中 T1 和 T2 的类型相同：Func<T, T, bool>。

给 Sort<T>方法指定下述签名：

```
static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
```

这个方法的文档说明，comparison 必须引用一个方法，该方法带有两个参数，如果第一个参数的值“小于”第二个参数，就返回 true。

设置完毕后，下面定义 BubbleSorter 类(代码文件 BubbleSorter/BubbleSorter.cs)：

```
class BubbleSorter
{
  static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
  {
    bool swapped = true;
    do
    {
      swapped = false;
      for (int i = 0; i < sortArray.Count - 1; i++)
      {
        if (comparison(sortArray[i+1], sortArray[i]))
        {
          T temp = sortArray[i];
          sortArray[i] = sortArray[i + 1];
          sortArray[i + 1] = temp;
          swapped = true;
        }
      }
    } while (swapped);
  }
}
```

为了使用这个类，需要定义另一个类，从而建立要排序的数组。在本例中，假定 Mortimer Phones 移动电话公司有一个员工列表，要根据他们的薪水进行排序。每个员工分别由类 Employee 的一个实例表示，如下所示(代码文件 BubbleSorter/Employee.cs)：

```
class Employee
{
  public Employee(string name, decimal salary)
  {
    this.Name = name;
    this.Salary = salary;
  }

  public string Name { get; private set; }
  public decimal Salary { get; private set; }

  public override string ToString()
  {
    return string.Format("{0}, {1:C}", Name, Salary);
  }

  public static bool CompareSalary(Employee e1, Employee e2)
  {
    return e1.Salary < e2.Salary;
  }
}
```

注意，为了匹配 Func<T, T, bool>委托的签名，在这个类中必须定义 CompareSalary，它的参数是两个 Employee 引用，并返回一个布尔值。在实现比较的代码中，根据薪水进行比较。

下面编写一些客户端代码，完成排序(代码文件 BubbleSorter/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      Employee[] employees =
      {
        new Employee("Bugs Bunny", 20000),
        new Employee("Elmer Fudd", 10000),
        new Employee("Daffy Duck", 25000),
        new Employee("Wile Coyote", 1000000.38m),
        new Employee("Foghorn Leghorn", 23000),
        new Employee("RoadRunner", 50000)
      };

      BubbleSorter.Sort(employees, Employee.CompareSalary);

      foreach (var employee in employees)
      {
       Console.WriteLine(employee);
      }
    }
  }
}
```

运行这段代码，正确显示按照薪水排列的 Employee，如下所示：

```
BubbleSorter
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
RoadRunner, $50,000.00
Wile Coyote, $1,000,000.38
```

8.2.6 多播委托

前面使用的每个委托都只包含一个方法调用。调用委托的次数与调用方法的次数相同。如果要调用多个方法，就需要多次显式调用这个委托。但是，委托也可以包含多个方法。这种委托称为多播委托。如果调用多播委托，就可以按顺序连续调用多个方法。为此，委托的签名就必须返回 void；否则，就只能得到委托调用的最后一个方法的结果。

可以使用返回类型为 void 的 Action<double>委托(代码文件 MulticastDelegates/Program.cs)：

```
class Program
{
  static void Main()
```

```
{
  Action<double> operations = MathOperations.MultiplyByTwo;
  operations += MathOperations.Square;
```

在前面的示例中，因为要存储对两个方法的引用，所以实例化了一个委托数组。而这里只是在同一个多播委托中添加两个操作。多播委托可以识别运算符“+”和“+=”。另外，还可以扩展上述代码中的最后两行，如下所示：

```
Action<double> operation1 = MathOperations.MultiplyByTwo;
Action<double> operation2 = MathOperations.Square;
Action<double> operations = operation1 + operation2;
```

多播委托还识别运算符“–”和“–=”，以从委托中删除方法调用。

> 根据后面的内容，多播委托实际上是一个派生自 System.MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 System.Delegate。System.MulticastDelegate 的其他成员允许把多个方法调用链接为一个列表。

为了说明多播委托的用法，下面把 SimpleDelegate 示例转换为一个新示例 MulticastDelegate。现在需要委托引用返回 void 的方法，就应重写 MathOperations 类中的方法，让它们显示其结果，而不是返回它们：

```
class MathOperations
{
  public static void MultiplyByTwo(double value)
  {
    double result = value * 2;
    Console.WriteLine("Multiplying by 2: {0} gives {1}", value, result);
  }

  public static void Square(double value)
  {
    double result = value * value;
    Console.WriteLine("Squaring: {0} gives {1}", value, result);
  }
}
```

为了适应这个改变，也必须重写 ProcessAndDisplayNumber()方法：

```
static void ProcessAndDisplayNumber(Action<double> action, double value)
{
  Console.WriteLine();
  Console.WriteLine("ProcessAndDisplayNumber called with value = {0}",
                    value);
  action(value);
}
```

下面测试多播委托，其代码如下：

```
static void Main()
{
```

```
    Action<double> operations = MathOperations.MultiplyByTwo;
    operations += MathOperations.Square;

    ProcessAndDisplayNumber(operations, 2.0);
    ProcessAndDisplayNumber(operations, 7.94);
    ProcessAndDisplayNumber(operations, 1.414);
    Console.WriteLine();
  }
```

现在，每次调用 ProcessAndDisplayNumber()方法时，都会显示一条消息，说明它已经被调用。然后，下面的语句会按顺序调用 action 委托实例中的每个方法：

```
action(value);
```

运行这段代码，得到如下所示的结果：

```
MulticastDelegate

ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

ProcessAndDisplayNumber called with value = 7.94
Multiplying by 2: 7.94 gives 15.88
Squaring: 7.94 gives 63.0436

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.999396
```

如果正在使用多播委托，就应知道对同一个委托调用方法链的顺序并未正式定义。因此应避免编写依赖于以特定顺序调用方法的代码。

通过一个委托调用多个方法还可能导致一个大问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的其中一个方法抛出一个异常，整个迭代就会停止。下面是 MulticastIteration 示例。其中定义了一个简单的委托 Action，它没有参数并返回 void。这个委托打算调用 One()和 Two()方法，这两个方法满足委托的参数和返回类型要求。注意 One()方法抛出了一个异常(代码文件 MulticastDelegateWithIteration/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void One()
    {
      Console.WriteLine("One");
      throw new Exception("Error in one");
    }

    static void Two()
    {
```

```
        Console.WriteLine("Two");
      }
```

在 Main()方法中，创建了委托 d1，它引用方法 One()，接着把 Two()方法的地址添加到同一个委托中。调用 d1 委托，就可以调用这两个方法。在 try/catch 块中捕获异常：

```
      static void Main()
      {
        Action d1 = One;
        d1 += Two;

        try
        {
          d1();
        }
        catch (Exception)
        {
          Console.WriteLine("Exception caught");
        }
      }
    }
  }
```

委托只调用了第一个方法。因为第一个方法抛出了一个异常，所以委托的迭代会停止，不再调用 Two()方法。没有指定调用方法的顺序时，结果会有所不同。

```
One
Exception Caught
```

错误和异常详见第 16 章。

在这种情况下，为了避免这个问题，应自己迭代方法列表。Delegate 类定义 GetInvocationList()方法，它返回一个 Delegate 对象数组。现在可以使用这个委托调用与委托直接相关的方法，捕获异常，并继续下一次迭代。

```
static void Main()
{
  Action d1 = One;
  d1 += Two;

  Delegate[] delegates = d1.GetInvocationList();
  foreach (Action d in delegates)
  {
    try
    {
      d();
    }
    catch (Exception)
    {
      Console.WriteLine("Exception caught");
    }
```

```
    }
  }
```

修改了代码后，运行应用程序，会看到在捕获了异常后，将继续迭代下一个方法。

```
One
Exception caught
Two
```

8.2.7　匿名方法

到目前为止，要想使委托工作，方法必须已经存在(即委托是用它将调用的方法的相同签名定义的)。但还有另外一种使用委托的方式：即通过匿名方法。匿名方法是用作委托的参数的一段代码。

用匿名方法定义委托的语法与前面的定义并没有区别。但在实例化委托时，就有区别了。下面是一个非常简单的控制台应用程序，它说明了如何使用匿名方法(代码文件 AnonymousMethods/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      string mid = ", middle part,";

      Func<string, string> anonDel = delegate(string param)
      {
        param += mid;
        param += " and this was added to the string.";
        return param;
      };
      Console.WriteLine(anonDel("Start of string"));

    }
  }
}
```

Func<string, string>委托接受一个字符串参数，返回一个字符串。anonDel 是这种委托类型的变量。不是把方法名赋予这个变量，而是使用一段简单的代码：它前面是关键字 delegate，后面是一个字符串参数。

可以看出，该代码块使用方法级的字符串变量 mid，该变量是在匿名方法的外部定义的，并把它添加到要传递的参数中。接着代码返回该字符串值。在调用委托时，把一个字符串作为参数传递，将返回的字符串输出到控制台上。

匿名方法的优点是减少了要编写的代码。不必定义仅由委托使用的方法。在为事件定义委托时，这是非常明显的(本章后面探讨事件)。这有助于降低代码的复杂性，尤其是定义了好几个事件时，代码会显得比较简单。使用匿名方法时，代码执行速度并没有加快。编译器仍定义了一个方法，该方法只有一个自动指定的名称，我们不需要知道这个名称。

在使用匿名方法时，必须遵循两条规则。在匿名方法中不能使用跳转语句(break、goto 或 continue)

跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

在匿名方法内部不能访问不安全的代码。另外，也不能访问在匿名方法外部使用的 ref 和 out 参数。但可以使用在匿名方法外部定义的其他变量。

如果需要用匿名方法多次编写同一个功能，就不要使用匿名方法。此时与复制代码相比，编写一个命名方法比较好，因为该方法只需要编写一次，以后可通过名称引用它。

从 C# 3.0 开始，可以使用 lambda 表达式替代匿名方法。

8.3 lambda 表达式

自 C# 3.0 开始，就可以使用一种新语法把实现代码赋予委托：lambda 表达式。只要有委托参数类型的地方，就可以使用 lambda 表达式。前面使用匿名方法的例子可以改为使用 lambda 表达式。

lambda 表达式的语法比匿名方法简单。如果所调用的方法有参数，且不需要参数，匿名方法的语法就比较简单，因为这样不需要提供参数。

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      string mid = ", middle part,";

      Func<string, string> lambda = param =>
        {
          param += mid;
          param += " and this was added to the string.";
          return param;
        };

      Console.WriteLine(lambda("Start of string"));
    }
  }
}
```

lambda 运算符"=>"的左边列出了需要的参数。lambda 运算符的右边定义了赋予 lambda 变量的方法的实现代码。

8.3.1 参数

lambda 表达式有几种定义参数的方式。如果只有一个参数，只写出参数名就足够了。下面的 lambda 表达式使用了参数 s。因为委托类型定义了一个 string 参数，所以 s 的类型就是 string。实现代码调用 String.Format()方法来返回一个字符串，在调用该委托时，就把字符串写到控制台上：

```
Func<string, string> oneParam = s =>
```

```
    String.Format("change uppercase {0}", s.ToUpper());
Console.WriteLine(oneParam("test"));
```

如果委托使用多个参数，就把参数名放在花括号中。这里参数 x 和 y 的类型是 double，由 Func<double, double, double>委托定义：

```
Func<double, double, double> twoParams = (x, y) => x * y;
Console.WriteLine(twoParams(3, 2));
```

为了方便，可以在花括号中给变量名添加参数类型。如果编译器不能匹配重载后的版本，那么使用参数类型可以帮助找到匹配的委托：

```
Func<double, double, double> twoParamsWithTypes = (double x, double y) => x * y;
Console.WriteLine(twoParamsWithTypes(4, 2));
```

8.3.2　多行代码

如果 lambda 表达式只有一条语句，在方法块内就不需要花括号和 return 语句，因为编译器会添加一条隐式的 return 语句。

```
Func<double, double> square = x => x * x;
```

添加花括号、return 语句和分号是完全合法的，通常这比不添加这些符号更容易阅读：

```
Func<double, double> square = x =>
{
  return x * x;
}
```

但是，如果在 lambda 表达式的实现代码中需要多条语句，就必须添加花括号和 return 语句：

```
Func<string, string> lambda = param =>
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
```

8.3.3　闭包

通过 lambda 表达式可以访问 lambda 表达式块外部的变量。这称为闭包。闭包是一个非常好的功能，但如果使用不当，也会非常危险。

在下面的示例中，Func<int, int>类型的 lambda 表达式需要一个 int 参数，返回一个 int。该 lambda 表达式的参数用变量 x 定义。实现代码还访问了 lambda 表达式外部的变量 someVal。只要不假设在调用 f 时，lambda 表达式创建了一个以后使用的新方法，这似乎没有什么问题。看看下面这个代码块，调用 f 的返回值应是 x 加 5 的结果，但似乎不是这样：

```
int someVal = 5;
Func<int, int> f = x => x + someVal;
```

假定以后要修改变量 someVal，于是调用 lambda 表达式时，会使用 someVal 的新值。调用 f(3) 的结果是 10：

```
someVal = 7;
Console.WriteLine(f(3));
```

特别是，通过另一个线程调用 lambda 表达式时，我们可能不知道进行了这个调用，也不知道外部变量的当前值是什么。

现在我们也许会奇怪，如何在 lambda 表达式的内部访问 lambda 表达式外部的变量。为了理解这一点，看看编译器在定义 lambda 表达式时做了什么。对于 lambda 表达式 x => x + someVal，编译器会创建一个匿名类，它有一个构造函数来传递外部变量。该构造函数取决于从外部传递进来的变量个数。对于这个简单的例子，构造函数接受一个 int。匿名类包含一个匿名方法，其实现代码、参数和返回类型由 lambda 表达式定义：

```
public class AnonymousClass
{
  private int someVal;
  public AnonymousClass(int someVal)
  {
    this.someVal = someVal;
  }
  public int AnonymousMethod(int x)
  {
    return x + someVal;
  }
}
```

使用 lambda 表达式并调用该方法，会创建匿名类的一个实例，并传递调用该方法时变量的值。

8.3.4　使用 foreach 语句的闭包

针对闭包，C# 5.0 中的 foreach 语句有了一个很大的改变。在下面的例子中，首先用值 10、20、30 填充了一个名为 values 的列表。变量 funcs 引用一个泛型列表，其中每个对象都引用 Func<int> 类型的委托。第一条 foreach 语句添加了 funcs 列表中的每个元素。添加到项中的函数使用 lambda 表达式定义。该 lambda 表达式使用了一个变量 val，该变量在 lambda 表达式的外部定义为 foreach 语句的循环变量。第二条 foreach 语句迭代 funcs 列表，以调用列表中引用的每个函数：

```
var values = new List<int>() { 10, 20, 30 };
var funcs = new List<Func<int>>();

foreach (var val in values)
{
  funcs.Add(() => val);
}
foreach (var f in funcs)
{
  Console.WriteLine((f()));
}
```

在 C# 5.0 中，这段代码的结果发生了变化。使用 C# 4 或更早版本的编译器时，会在控制台中输出 30 三次。在第一个 foreach 循环中使用闭包时，所创建的函数是在调用时、而不是在迭代时获得 val 变量的值。第 6 章已经介绍过，编译器会从 foreach 语句创建一个 while 循环。在 C# 4 中，编译器在 while 循环外部定义循环变量，在每次迭代中重用这个变量。因此，在循环结束时，该变量

的值就是最后一次迭代时的值。要想在使用 C# 4 时，让代码的结果为 10、20、30，必须将代码改为使用一个局部变量，并将这个局部变量传入 lambda 表达式。这样，每次迭代时就将保留一个不同的值。

```
var values = new List<int>() { 10, 20, 30 };
var funcs = new List<Func<int>>();

foreach (var val in values)
{
  var v = val;
  funcs.Add(() => v);
}
foreach (var f in funcs)
{
 Console.WriteLine((f()));
}
```

在 C# 5.0 中，不再需要做这种代码修改(即将代码修改为局部变量)。C# 5.0 会在 while 循环的代码块中创建一个不同的局部循环变量，所以值会自动得到保留。这是 C# 4.0 和 C# 5.0 的区别，必须知道这一点。

lambda 表达式可以用于类型为委托的任意地方。类型是 Expression 或 Expression<T>时，也可以使用 lambda 表达式。此时编译器会创建一个表达式树，详见第 11 章。

8.4 事件

事件基于委托，为委托提供了一种发布/订阅机制。在架构内到处都能看到事件。在 Windows 应用程序中，Button 类提供了 Click 事件。这类事件就是委托。触发 Click 事件时调用的处理程序方法需要定义，其参数由委托类型定义。

在本节的示例代码中，事件用于连接 CarDealer 类和 Consumer 类。CarDealer 类提供了一个新车到达时触发的事件。Consumer 类订阅该事件，以获得新车到达的通知。

8.4.1 事件发布程序

从 CarDealer 类开始，它基于事件提供一个订阅。CarDealer 类用 event 关键字定义了类型为 EventHandler<CarInfoEventArgs>的 NewCarInfo 事件。在 NewCar()方法中，通过调用 RaiseNewCarInfo 方法触发 NewCarInfo 事件。这个方法的实现检查委托是否为空，如果不为空，就引发事件(代码文件 EventSample/CarDealer.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  public class CarInfoEventArgs: EventArgs
  {
```

```
        public CarInfoEventArgs(string car)
        {
            this.Car = car;
        }

        public string Car { get; private set; }
    }

    public class CarDealer
    {
        public event EventHandler<CarInfoEventArgs> NewCarInfo;

        public void NewCar(string car)
        {
          Console.WriteLine("CarDealer, new car {0}", car);

          RaiseNewCarInfo(car);
        }

        protected virtual void RaiseNewCarInfo(string car)
        {
            EventHandler<CarInfoEventArgs> newCarInfo = NewCarInfo;
            if (newCarInfo != null)
            {
              newCarInfo(this, new CarInfoEventArgs(car));
            }
        }
    }
}
```

CarDealer 类提供了 EventHandler<CarInfoEventArgs>类型的 NewCarInfo 事件。作为一个约定，事件一般使用带两个参数的方法，其中第一个参数是一个对象，包含事件的发送者，第二个参数提供了事件的相关信息。第二个参数随不同的事件类型而不同。.NET 1.0 为所有不同数据类型的事件定义了几百个委托。有了泛型委托 EventHandler<T>后，这就不再需要委托了。EventHandler<TEventArgs>定义了一个处理程序，它返回 void，接受两个参数。对于 EventHandler<TEventArgs>，第一个参数必须是 object 类型，第二个参数是 T 类型。EventHandler<TEventArgs>还定义了一个关于 T 的约束：它必须派生自基类 EventArgs，CarInfoEventArgs 就派生自基类 EventArgs：

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

委托 EventHandler<TEventArgs>的定义如下：

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
where TEventArgs: EventArgs
```

在一行上定义事件是 C#的简化记法。编译器会创建一个 EventHandler<CarInfoEventArgs>委托类型的变量，并添加方法，以便从委托中订阅和取消订阅。该简化记法的较长形式如下所示。这非常类似于自动属性和完整属性之间的关系。对于事件，使用 add 和 remove 关键字添加和删除委托的处理程序：

```
private delegate EventHandler<CarInfoEventArgs> newCarInfo;
public event EventHandler<CarInfoEventArgs> NewCarInfo
```

```
{
  add
  {
    newCarInfo += value;
  }
  remove
  {
    newCarInfo-= value;
  }
}
```

如果不仅需要添加和删除事件处理程序，定义事件的长记法就很有用，例如，需要为多个线程访问添加同步操作。WPF 控件使用长记法给事件添加冒泡和隧道功能。事件的冒泡和隧道详见第 29 章。

CarDealer 类在 RaiseNewCarInfo 方法中触发事件。使用委托 NewCarInfo 和花括号可以调用给事件订阅的所有处理程序。注意与多播委托一样，方法的调用顺序无法保证。为了更多地控制处理程序的调用，可以使用 Delegate 类的 GetInvocationList()方法，访问委托列表中的每一项，并独立地调用每个方法，如上所示。

在触发事件之前，需要检查委托 NewCarInfo 是否不为空。如果没有订阅处理程序，委托就是空。

```
protected virtual void RaiseNewCarInfo(string car)
{
  var newCarInfo = NewCarInfo;
  if (newCarInfo != null)
  {
    newCarInfo(this, new CarInfoEventArgs(car));
  }
}
```

8.4.2 事件侦听器

Consumer 类用做事件侦听器。这个类订阅了 CarDealer 类的事件，并定义了 NewCarIsHere 方法，该方法满足 EventHandler<CarInfoEventArgs>委托的要求，其参数类型是 object 和 CarInfoEventArgs (代码文件 EventsSample/Consumer.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  public class Consumer
  {
    private string name;

    public Consumer(string name)
    {
      this.name = name;
    }

    public void NewCarIsHere(object sender, CarInfoEventArgs e)
```

```
        {
          Console.WriteLine("{0}: car {1} is new", name, e.Car);
        }
      }
    }
```

现在需要连接事件发布程序和订阅器。为此使用 CarDealer 类的 NewCarInfo 事件，通过“+=”创建一个订阅。消费者 michael(变量)订阅了事件，接着消费者 sebastian(变量)也订阅了事件，然后 michael(变量)通过“-=”取消了订阅(代码文件 EventsSample/Program.cs)。

```
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      var dealer = new CarDealer();

      var michael = new Consumer("Michael");
      dealer.NewCarInfo += michael.NewCarIsHere;

      dealer.NewCar("Ferrari");

      var sebastian = new Consumer("Sebastian");
      dealer.NewCarInfo += sebastian.NewCarIsHere;

      dealer.NewCar("Mercedes");

      dealer.NewCarInfo -= michael.NewCarIsHere;

      dealer.NewCar("Red Bull Racing");
    }
  }
}
```

运行应用程序，一辆 Ferrari 到达，Michael 得到了通知。因为之后 Sebastian 也注册了该订阅，所以 Michael 和 Sebastian 都获得了新 Mercedes 的通知。接着 Michael 取消了订阅，所以只有 Sebastian 获得了 Red Bull 的通知。

```
CarDealer, new car Ferrari
Michael: car Ferrari is new
CarDealer, new car Mercedes
Michael: car Mercedes is new
Sebastian: car Mercedes is new
CarDealer, new car Red Bull
Sebastian: car Red Bull is new
```

8.4.3 弱事件

通过事件，直接连接到发布程序和侦听器。但垃圾回收有一个问题。例如，如果侦听器不再直接引用，发布程序就仍有一个引用。垃圾回收器不能清空侦听器占用的内存，因为发布程序仍保有一个引用，会针对侦听器触发事件。

这种强连接可以通过弱事件模式来解决，即使用 WeakEventManager 作为发布程序和侦听器之间的中介。

前面的示例把 CarDealer 作为发布程序，把 Consumer 作为侦听器，下一节将修改这个示例，以使用弱事件模式。

动态创建订阅器时，为了避免出现资源泄露，必须特别留意事件。也就是说，需要在订阅器离开作用域(不再需要它)之前，确保取消对事件的订阅。另一种方法就是使用弱事件。

1. 弱事件管理器

要使用弱事件，需要创建一个派生自 WeakEventManager 类的类。WeakEventManager 类在程序集 WindowsBase 的名称空间 System.Windows 中定义。

WeakCarInfoEventManager 类是弱事件管理器类，它管理 NewCarInfo 事件的发布程序和侦听器之间的连接。因为这个类实现了单态模式，所以只创建一个实例。静态属性 CurrentManager 创建了一个 WeakCarInfoEventManager 类型的对象(如果它不存在)，并返回对该对象的引用。WeakCarInfoEventManager.CurrentManager 用于访问 WeakCarInfoEventManager 类中的单态对象。

对于弱事件模式，弱事件管理器类需要静态方法 AddListener()和 RemoveListener()。侦听器使用这些方法连接发布程序，断开与发布程序的连接，而不是直接使用发布程序中的事件。侦听器还需要实现稍后介绍的接口 IWeakEventListener。通过 AddListener()和 RemoveListener()方法，调用 WeakEventManager 基类中的方法，来添加和删除侦听器。

对于 WeakCarInfoEventManager 类，还需要重写基类的 StartListening()和 StopListening()方法。添加第一个侦听器时调用 StartListening()方法，删除最后一个侦听器时调用 StopListening()方法。StartListening()方法和 StopListening()方法从弱事件管理器中订阅和取消订阅一个方法，以侦听发布程序中的事件。如果弱事件管理器类需要连接到不同的发布程序类型上，就可以在源对象中检查类型信息，之后进行类型强制转换。接着使用基类的 DeliverEvent()方法，把事件传递给侦听器。DeliverEvent()方法在侦听器中调用 IWeakEventListener 接口中的 ReceiveWeakEvent()方法(代码文件 WeakEventsSample/WeakCarInfoEventManager.cs)：

```
using System.Windows;

namespace Wrox.ProCSharp.Delegates
{
  public class WeakCarInfoEventManager: WeakEventManager
  {
    public static void AddListener(object source, IWeakEventListener listener)
    {
      CurrentManager.ProtectedAddListener(source, listener);
    }

    public static void RemoveListener(object source, IWeakEventListener listener)
    {
      CurrentManager.ProtectedRemoveListener(source, listener);
```

```
        }

        public static WeakCarInfoEventManager CurrentManager
        {
          get
          {
            var manager = GetCurrentManager(typeof(WeakCarInfoEventManager))
                as WeakCarInfoEventManager;
            if (manager == null)
            {
              manager = new WeakCarInfoEventManager();
              SetCurrentManager(typeof(WeakCarInfoEventManager), manager);
            }
            return manager;
          }
        }

        protected override void StartListening(object source)
        {
          (source as CarDealer).NewCarInfo += CarDealer_NewCarInfo;
        }

        void CarDealer_NewCarInfo(object sender, CarInfoEventArgs e)
        {
          DeliverEvent(sender, e);
        }

        protected override void StopListening(object source)
        {
          (source as CarDealer).NewCarInfo = CarDealer_NewCarInfo;
        }
      }
    }
```

WPF 使用弱事件模式和事件管理器类 CollectionChangedEventManager、CurrentChangedEventManager、CurrentChangingEventManager、PropertyChangedEventManager、DataChangedEventManager 和 LostFocusEventManager。

对于发布程序类 CarDealer，不需要做任何修改，其实现代码与前面相同。

2. 事件侦听器

侦听器需要改为实现 IWeakEventListener 接口。这个接口定义了 ReceiveWeakEvent()方法，触发事件时，从弱事件管理器中调用这个方法。在该方法的实现代码中，应从触发的事件中调用 NewCarIsHere()方法(代码文件 WeakEventsSample/Consumer.cs)。

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.Delegates
{
```

```
  public class Consumer: IWeakEventListener
  {
    private string name;

    public Consumer(string name)
    {
      this.name = name;
    }

    public void NewCarIsHere(object sender, CarInfoEventArgs e)
    {
      Console.WriteLine("{0}: car {1} is new", name, e.Car);
    }

    bool IWeakEventListener.ReceiveWeakEvent(Type managerType, object sender,
              EventArgs e)
    {
      NewCarIsHere(sender, e as CarInfoEventArgs);
      return true;
    }
  }
}
```

在 Main 方法中，连接发布程序和侦听器，该连接现在使用 WeakCarInfoEventManager 类的 AddListener()和 RemoveListener()静态方法(代码文件 WeakEventsSample/Program.cs)。

```
static void Main()
{
  var dealer = new CarDealer();

  var michael = new Consumer("Michael");
  WeakCarInfoEventManager.AddListener(dealer, michael);

  dealer.NewCar("Mercedes");

  var sebastian = new Consumer("Sebastian");
  WeakCarInfoEventManager.AddListener(dealer, sebastian);

  dealer.NewCar("Ferrari");

  WeakCarInfoEventManager.RemoveListener(dealer, michael);

  dealer.NewCar("Red Bull Racing");
}
```

实现了弱事件模式后，发布程序和侦听器就不再强连接了。当不再引用侦听器时，它就会被垃圾回收。

3. 泛型弱事件管理器

.NET 4.5 为弱事件管理器提供了新的实现。泛型类 WeakEventManager<TEventSource, TEventArgs> 派生自基类 WeakEventManager，它显著简化了弱事件的处理。使用这个类时，不再需要为每个事件实现一个自定义的弱事件管理器，也不需要让事件的消费者实现接口 IWeakEventsListener。所要做

的就是使用泛型弱事件管理器订阅事件。

订阅事件的主程序现在改为使用泛型 WeakEventManager，其事件源为 CarDealer 类型，随事件一起传递的事件参数为 CarInfoEventArgs 类型。WeakEventManager 类定义了 AddHandler 方法来订阅事件，使用 RemoveHandler 方法来取消订阅事件。然后，程序的工作方式与以前一样，但是代码少了许多：

```
var dealer = new CarDealer();

var michael = new Consumer("Michael");
WeakEventManager<CarDealer, CarInfoEventArgs>.AddHandler(dealer,
    "NewCarInfo", michael.NewCarIsHere);

dealer.NewCar("Mercedes");

var sebastian = new Consumer("Sebastian");
WeakEventManager<CarDealer, CarInfoEventArgs>.AddHandler(dealer,
    "NewCarInfo", sebastian.NewCarIsHere);

dealer.NewCar("Ferrari");

WeakEventManager<CarDealer, CarInfoEventArgs>.RemoveHandler(dealer,
    "NewCarInfo", michael.NewCarIsHere);

dealer.NewCar("Red Bull Racing");
```

8.5 小结

本章介绍了委托、lambda 表达式和事件的基础知识，解释了如何声明委托，如何给委托列表添加方法，如何实现通过委托和 lambda 表达式调用的方法，并讨论了声明事件处理程序来响应事件的过程，以及如何创建自定义事件，使用引发事件的模式。

.NET 开发人员将大量使用委托和事件，特别是在开发 Windows 应用程序时。事件是.NET 开发人员监控应用程序执行时出现的各种 Windows 消息的方式，否则就必须监控 WndProc，捕获 WM_MOUSEDOWN 消息，而不是获取按钮的鼠标 Click 事件。

在设计大型应用程序时，使用委托和事件可以减少依赖性和层的耦合，并能开发出具有更高重用性的组件。

lambda 表达式是委托的 C#语言特性。通过它们可以减少需要编写的代码量。lambda 表达式不仅仅用于委托，详见第 11 章。

第 9 章介绍字符串和正则表达式。

第 9 章

字符串和正则表达式

本章要点

- 创建字符串
- 格式化表达式
- 使用正则表达式

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Encoder.cs
- Encoder2.cs
- FormattableVector.cs
- RegularExpressionPlayground.cs
- StringEncoder.cs

从本书前面开始，我们一直在使用字符串，但可能没有意识到，在 C#中 string 关键字的映射实际上指向.NET 基类 System.String。System.String 是一个功能非常强大且用途非常广泛的基类，但它不是.NET 库中唯一与字符串相关的类。本章首先复习一下 System.String 的特性，再介绍如何使用其他的.NET 库类来处理字符串，特别是 System.Text 和 System.Text.RegularExpressions 名称空间中的类。本章主要介绍下述内容：

- **创建字符串**——如果多次修改一个字符串，例如，在显示字符串或将其传递给其他方法或应用程序前，创建一个较长的字符串，String 类就会变得效率低下。对于这种情况，应使用另一个类 System.Text.StringBuilder，因为它是专门为这种情况设计的。
- **格式化表达式**——这些格式化表达式将用于后面几章中的 Console.WriteLine()方法。格式化表达式使用两个有效的接口 IFormatProvider 和 IFormattable 来处理。在自己的类上实现这两个接口，实际上就可以定义自己的格式化序列，这样，Console.WriteLine()和类似的类就可以以指定的方式显示类的值。

- **正则表达式**——.NET 还提供了一些非常复杂的类来识别字符串，或从长字符串中提取满足某些复杂条件的子字符串。例如，找出字符串中所有重复出现的某个字符或一组字符，或者找出以 s 开头且至少包含一个 n 的所有单词，或者找出遵循雇员 ID 或社会安全号码结构的字符串。虽然可以使用 String 类，编写方法来完成这类处理，但这类方法编写起来比较繁琐。而使用 System.Text.RegularExpressions 名称空间中的类就比较简单，System.Text.RegularExpressions 专门用于完成这类处理。

9.1 System.String 类

在介绍其他字符串类之前，先快速复习一下 String 类中一些可用的方法。

System.String 是一个类，专门用于存储字符串，允许对字符串进行许多操作。由于这种数据类型非常重要，C#提供了它自己的关键字和相关的语法，以便于使用这个类来轻松地处理字符串。

使用运算符重载可以连接字符串：

```
string message1 = "Hello"; // returns "Hello"
message1 += ", There"; // returns "Hello, There"
string message2 = message1 + "!"; // returns "Hello, There!"
```

C#还允许使用类似于索引器的语法来提取指定的字符：

```
string message = "Hello";
char char4 = message[4];   // returns 'o'. Note the string is zero-indexed
```

这个类可以完成许多常见的任务，如替换字符、删除空白和把字母变成大写形式等。可用的方法如表 9-1 所示。

表 9-1

方　法	作　用
Compare	比较字符串的内容，考虑文化背景(区域)，判断某些字符是否相等
CompareOrdinal	与 Compare 一样，但不考虑文化背景
Concat	把多个字符串实例合并为一个实例
CopyTo	把从选定的下标开始的特定数量的字符复制到数组的一个全新实例中
Format	格式化包含各种值的字符串和如何格式化每个值的说明符
IndexOf	定位字符串中第一次出现某个给定子字符串或字符的位置
IndexOfAny	定位字符串中第一次出现某个字符或一组字符的位置
Insert	把一个字符串实例插入到另一个字符串实例的指定索引处
Join	合并字符串数组，创建一个新字符串
LastIndexOf	与 IndexOf 一样，但定位最后一次出现的位置
LastIndexOfAny	与 IndexOfAny 一样，但定位最后一次出现的位置
PadLeft	在字符串的左侧，通过添加指定的重复字符填充字符串
PadRight	在字符串的右侧，通过添加指定的重复字符填充字符串
Replace	用另一个字符或子字符串替换字符串中给定的字符或子字符串

(续表)

方　法	作　用
Split	在出现给定字符的地方，把字符串拆分为一个子字符串数组
Substring	在字符串中检索给定位置的子字符串
ToLower	把字符串转换为小写形式
ToUpper	把字符串转换为大写形式
Trim	删除首尾的空白

表 9-1 并不完整，但可以让你明白字符串所提供的功能。

9.1.1　创建字符串

如上所述，String 类是一个功能非常强大的类，它实现许多很有用的方法。但是，String 类存在一个问题：重复修改给定的字符串，效率会很低，它实际上是一个不可变的数据类型，一旦对字符串对象进行了初始化，该字符串对象就不能改变了。表面上修改字符串内容的方法和运算符实际上创建一个新字符串，根据需要，可以把旧字符串的内容复制到新字符串中。例如，下面的代码：

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

在执行这段代码时，首先，创建一个 System.String 类型的对象，并把它初始化为文本"Hello from all the guys at Wrox Press. "。注意句号后面有一个空格。此时.NET 运行库会为该字符串分配足够的内存来保存这个文本(39 个字符)，再设置变量 greetingText，来表示这个字符串实例。

从语法上看，下一行代码是把更多的文本添加到字符串中。实际上并非如此，而是创建一个新字符串实例，给它分配足够的内存，以存储合并的文本(共 103 个字符)。把最初的文本"Hello from all the people at Wrox Press. "复制到这个新字符串中，再加上额外的文本"We do hope you enjoy this book as much as we enjoyed writing it."。然后更新存储在变量 greetingText 中的地址，使变量正确地指向新的字符串对象。现在没有引用旧的字符串对象——不再有变量引用它，下一次垃圾收集器清理应用程序中所有未使用的对象时，就会删除它。

这本身还不坏，但假定要对这个字符串编码，在字母表中，用 ASCII 码靠后的字符替代其中的每个字母(标点符号除外)，作为非常简单的加密模式的一部分。这就会把该字符串变成"Ifmmp gspn bmm uif hvst bu Xspy Qsftt. Xf ep ipqf zpv fokpz uijt cppl bt nvdi bt xf fokpzfe xsjujoh ju"。完成这个任务有好几种方式，但最简单、最高效的一种(假定只使用 String 类)是使用 String.Replace()方法，该方法把字符串中指定的子字符串用另一个子字符串代替。使用 Replace()，对文本进行编码的代码如下所示：

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";

for(int i = 'z'; i>= 'a'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}
```

```
for(int i = 'Z'; i>='A'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}

Console.WriteLine("Encoded:\n" + greetingText);
```

为了简单起见，这段代码没有把 Z 换成 A，也没有把 z 换成 a。这些字符分别编码为[和{。

在本示例中，Replace()方法以一种智能的方式工作，在某种程度上，它并没有创建一个新字符串，除非它实际上要对旧字符串进行某些改变。原来的字符串包含 23 个不同的小写字母，和 3 个不同的大写字母。所以 Replace()就分配一个新字符串，共 26 次，每个新字符串都包含 103 个字符。因此加密过程需要在堆上有一个总共能存储 2678 个字符的字符串对象，最终将等待被垃圾收集！显然，如果使用字符串频繁进行文字处理，应用程序就会遇到严重的性能问题。

为了解决这类问题，Microsoft 提供了 System.Text.StringBuilder 类。StringBuilder 类不像 String 类那样能够支持非常多的方法。在 StringBuilder 类上可以进行的处理仅限于替换和追加或删除字符串中的文本。但是，它的工作方式非常高效。

在使用 String 类构造一个字符串时，要给它分配足够的内存来保存字符串。然而，StringBuilder 类通常分配的内存会比它需要的更多。开发人员可以选择指定 StringBuilder 要分配多少内存，但如果没有指定，在默认情况下就根据初始化 StringBuilder 实例时的字符串长度来确定内存的大小。StringBuilder 类有两个主要的属性：

- Length 指定字符串的实际长度。
- Capacity 指定字符串在分配的内存中的最大长度。

对字符串的修改就在赋予 StringBuilder 实例的内存块中进行，这就大大提高了追加子字符串和替换单个字符的效率。删除或插入子字符串仍然效率低下，因为这需要移动随后的字符串。只有执行扩展字符串容量的操作，才需要给字符串分配新内存，才可能移动包含的整个字符串。在添加额外的容量时，从经验来看，如果 StringBuilder 类检测到容量超出，且容量没有设置新值，就会使自己的容量翻倍。

例如，如果使用 StringBuilder 对象构造最初的欢迎字符串，就可以编写下面的代码：

```
StringBuilder greetingBuilder =
   new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we enjoyed
   writing it");
```

为了使用 StringBuilder 类，需要在代码中引用 System.Text 类。

在这段代码中，为 StringBuilder 类设置的初始容量是 150。最好把容量设置为字符串可能的最大长度，确保 StringBuilder 类不需要重新分配内存，因为其容量足够用了。该容量默认设置为 16。理论上，可以设置尽可能大的数字，足够给该容量传送一个 int，但如果实际上给字符串分配 20 亿个字符的空间(这是 StringBuilder 实例理论上允许拥有的最大空间)，系统就可能会没有足够的内存。

执行上面的代码时，它首先创建一个 StringBuilder 对象，如图 9-1 所示。

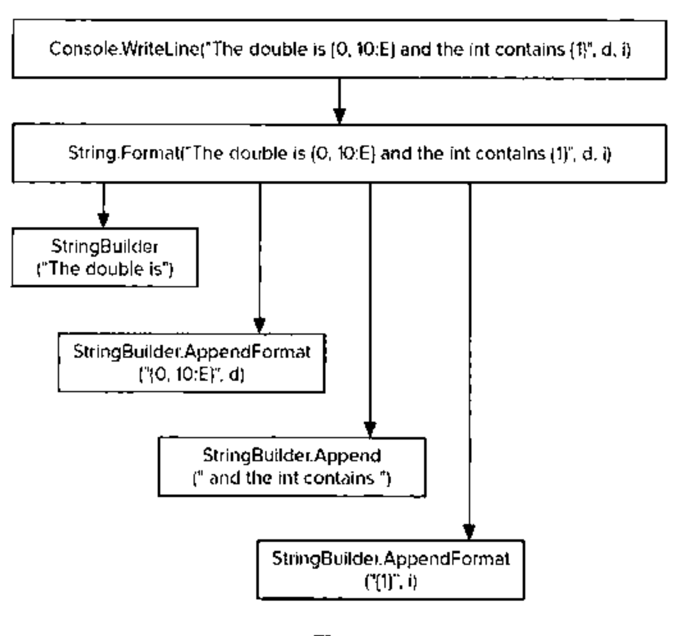


图 9-1

然后，在调用 AppendFormat()方法时，其他文本就放在空的空间中，不需要分配更多的内存。但是，多次替换文本才能获得使用 StringBuilder 类所带来的高效性能。例如，如果要以前面的方式加密文本，就可以执行整个加密过程，无须分配更多的内存：

```
StringBuilder greetingBuilder =
   new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we " +
   "enjoyed writing it");

Console.WriteLine("Not Encoded:\n" + greetingBuilder);

for(int i = 'z'; i>='a'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}

for(int i = 'Z'; i>='A'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}
```

```
Console.WriteLine("Encoded:\n" + greetingBuilder);
```

这段代码使用了 StringBuilder.Replace()方法，它的功能与 String.Replace()一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 StringBuilder 实例以及在最后一条 Console.WriteLine()语句中执行字符串操作期间分配的内存。

一般而言，使用 StringBuilder 类执行字符串的任何操作，使用 String 类存储字符串或显示最终结果。

9.1.2 StringBuilder 成员

前面介绍了 StringBuilder 类的一个构造函数，它的参数是一个初始字符串及该字符串的容量。StringBuilder 类还有几个其他的构造函数，例如，可以只提供一个字符串：

```
StringBuilder sb = new StringBuilder("Hello");
```

或者用给定的容量创建一个空的 StringBuilder 类：

```
StringBuilder sb = new StringBuilder(20);
```

除了前面介绍的 Length 和 Capacity 属性外，还有一个只读属性 MaxCapacity，它表示对给定的 StringBuilder 实例的容量限制。在默认情况下，这由 int.MaxValue 给定(大约 20 亿，如前所述)。但在构造 StringBuilder 对象时，也可以把这个值设置为较低的值：

```
// This will both set initial capacity to 100, but the max will be 500.
// Hence, this StringBuilder can never grow to more than 500 characters,
// otherwise it will raise exception if you try to do that.
StringBuilder sb = new StringBuilder(100, 500);
```

还可以随时显式地设置容量，但如果把这个值设置为小于字符串的当前长度，或者超出了最大容量的某个值，就会抛出一个异常：

```
StringBuilder sb = new StringBuilder("Hello");
sb.Capacity = 100;
```

StringBuilder 类主要的方法如表 9-2 所示。

表 9-2

方 法	说 明
Append()	给当前字符串追加一个字符串
AppendFormat()	追加特定格式的字符串
Insert()	在当前字符串中插入一个子字符串
Remove()	从当前字符串中删除字符
Replace()	在当前字符串中，用某个字符全部替换另一个字符，或者用当前字符串中的一个子字符串全部替换另一个字符串
ToString()	返回当前强制转换为 System.String 对象的字符串(在 System.Object 中被重写)

其中一些方法还有几种格式的重载方法。

AppendFormat()方法实际上会在最终调用 Console.WriteLine()方法时调用，它负责确定所有像{0:D}的格式化表达式应使用什么表达式替代。下一节讨论这个问题。

不能把 StringBuilder 强制转换为 String(隐式转换和显式转换都不行)。如果要把 StringBuilder 的内容输出为 String，唯一的方式就是使用 ToString()方法。

前面介绍了 StringBuilder 类，说明了使用它提高性能的一些方式。注意，这个类并不总能提高性能。StringBuilder 类基本上应在处理多个字符串时使用。但如果只是连接两个字符串，使用 System.String 类会比较好。

9.1.3　格式字符串

在前面的代码示例中编写了许多类和结构，对这些类和结构实现 ToString()方法，都是为了显示给定变量的内容。但是，用户常常希望以各种可能的方式显示变量的内容，在不同的文化或地区背景中有不同的格式。.NET 基类 System.DateTime 就是最明显的一个示例：可以把日期显示为 10 June 2012、10 Jun 2012、6/10/12(美国)、10/6/12(英国)或 10.06.2012(德国)。

同样，在第 7 章中编写的 Vector 结构实现了 Vector.ToString()方法，这是为了以(4, 56, 8)格式显示矢量。编写矢量的另一个非常常用的方式类似 4i + 56j + 8k。如果要使类的用户友好性比较高，就需要使用某些工具以用户希望的方式显示它们的字符串表示。.NET 运行库定义了一种标准方式：使用 IFormattable 接口。本小节的主题就是说明如何把这个重要特性添加到类和结构上。

在显示一个变量时，常常需要指定它的格式，其中经常调用 Console.WriteLine()方法。因此，本小节把这个方法作为示例，但这里的讨论适用于格式化字符串的大多数情况。例如，如果要在列表框或文本框中显示一个变量的值，一般就使用 String.Format()方法来获得该变量的适当字符串表示，但用于请求所需格式的格式说明符与传递给 Console.WriteLine()方法的格式说明符相同。因此本小节把 Console.WriteLine()方法作为一个示例来说明。首先看看在为基元类型提供格式字符串时会发生什么，再看看如何把自己的类和结构的格式说明符添加到过程中。

第 2 章在 Console.Write()和 Console.WriteLine()方法中使用了格式字符串：

```
double d = 13.45;
int i = 45;
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

格式字符串本身大都由要显示的文本组成，但只要有要格式化的变量，它在参数列表中的下标就必须放在花括号中。在花括号中还可以有与该项的格式相关的其他信息。例如，可以包含：

- 该项的字符串表示要占用的字符数，这个信息的前面应有一个逗号。负值表示该项应左对齐，正值表示该项应右对齐。如果该项占用的字符数比给定的多，其内容就会完整地显示出来。
- 格式说明符也可以显示出来。它的前面应有一个冒号，表示应如何格式化该项。例如，把一个数字格式化为货币，或者以科学计数法显示。

第 2 章简要介绍了数字类型的常见格式说明符，表 9-3 再次引用该表。

表 9-3

格式说明符	应　用	含　义	示　例
C	数字类型	特定地区的货币值	$4834.50 (USA)　£4834.50 (UK)
D	只用于整数类型	一般的整数	4834
E	数字类型	科学计数法	4.834E+003
F	数字类型	小数点后的位数固定	4384.50
G	数字类型	一般的数字	4384.5
N	数字类型	通常是特定地区的数字格式	4,384.50 (UK/USA) 4 384,50 (欧洲大陆)
P	数字类型	百分比计数法	432,000.00%
X	只用于整数类型	十六进制格式	1120 (如果要显示 0x1120，就需要写上 0x)

如果要在整数上加上前导 0，就可以将格式说明符 0 重复尽可能多的次数。例如，格式说明符 0000 会把 3 显示为 0003，99 显示为 0099。

这里不能给出完整的列表，因为其他数据类型有自己的格式说明符。本节的主要目的是说明如何为自己的类定义格式说明符。

1. 字符串的格式化

为了说明如何格式化字符串，看看执行下面的语句会得到什么结果：

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

Console.WriteLine()方法只是把参数的完整列表传送给静态方法 String.Format()。如果要在字符串中以其他方式格式化这些值，例如显示在一个文本框中，则也可以调用这个方法。实现带有 3 个参数的 WriteLine()重载方法的代码如下：

```
// Likely implementation of Console.WriteLine()

public void WriteLine(string format, object arg0, object arg1)
{
   this.WriteLine(string.Format(this.FormatProvider, format,
      new object[]{arg0, arg1}));
}
```

上面的代码调用了带有一个参数的重载方法 WriteLine()，仅显示了传递过来的字符串的内容，没有对它进行进一步的格式化。

String.Format()方法现在需要用对应对象的合适字符串表示来替换每个格式说明符，构造最终的字符串。但是，如前所述，对于这个构建字符串的过程，需要 StringBuilder 实例，而不是 String 实例。在这个示例中，StringBuilder 实例是用字符串的第一部分(即文本“The double is”)创建和初始化的。然后调用 StringBuilder.AppendFormat()方法，传递第一个格式说明符“{0,10:E}”和相应的对象 double，把这个对象的字符串表示添加到构造好的字符串对象中，这个过程会继续重复调用 StringBuilder.Append()和

StringBuilder.AppendFormat()方法，直到得到了全部格式化好的字符串为止。

下面的内容比较有趣。StringBuilder.AppendFormat()方法需要指出如何格式化对象，它首先检查对象，确定它是否实现 System 名称空间中的接口 IFormattable。只要试着把这个对象强制转换为接口，看看强制转换是否成功即可，或者使用 C#的关键字 is 实现此测试。如果测试失败，AppendFormat()方法就会调用对象的 ToString()方法，所有的对象都从 System.Object 类继承了这个方法或重写了该方法。在前面给出的编写各种类和结构的示例中，执行过程都是这样，因为目前编写的类都没有实现这个接口。这就是在前面的章节中，Object.ToString()的重写方法允许在 Console.WriteLine()语句中显示结构和类(如 Vector)的原因。

但是，所有预定义的基元数字类型都实现这个接口，对于这些类型，特别是这个示例中的 double 和 int，就不会调用继承自 System.Object 类的基本 ToString()方法。为了理解这个过程，需要了解 IFormattable 接口。

IFormattable 接口只定义了一个方法，该方法也命名为 ToString()，它带有两个参数，而 System.Object 版本的 ToString()方法不带参数。下面是 IFormattable 接口的定义：

```
interface IFormattable
{
   string ToString(string format, IFormatProvider formatProvider);
}
```

这个 ToString()重载方法的第一个参数是一个字符串，它指定要求的格式。换言之，它是字符串的说明符部分，该部分放在字符串的{}中，该参数最初传递给 Console.WriteLine()或 String. Format()方法。例如，在本例中，最初的语句如下：

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

在判断第一个说明符{0,10:E}时，在 double 变量 d 上调用这个重载方法，传递给它的第一个参数是 E。StringBuilder.AppendFormat()方法传递的总是显示在原始字符串的合适格式说明符内冒号后面的文本。

本书不讨论 ToString()方法的第 2 个参数，它是实现 IFormatProvider 接口的对象引用。这个接口提供了 ToString()接口在格式化对象时需要考虑的更多信息——一般包括文化背景信息(.NET 文化背景类似于 Windows 时区；如果格式化货币或日期，就需要这些信息)。如果直接从源代码中调用这个 ToString()重载方法，就需要提供这样一个对象。但 StringBuilder.AppendFormat()方法为这个参数传递一个空值。如果 formatProvider 为空，ToString()方法就要使用系统设置中指定的文化背景信息。

现在回过头来看看本例。第一个要格式化的项是一个 double 类，对此要求使用指数计数法，格式说明符为 E。StringBuilder.AppendFormat()方法会确保该 double 数的确实现 IFormattable 接口，因此要调用带有两个参数的 ToString()重载方法，其第一个参数是字符串“E”，第二个参数为空。现在该 double 数的这个方法在实现接口时，会考虑要求的格式和当前的文化背景，以合适的格式返回 double 的字符串表示。StringBuilder.AppendFormat()方法则按照需要在返回的字符串中添加前导空格，使之共有 10 个字符。

下一个要格式化的对象是 int，它不需要任何特殊的格式(格式说明符是{1})。由于没有格式要求，因此 StringBuilder.AppendFormat()方法会给该格式化字符串传递一个空引用，并适当地响应带有两

个参数的 int.ToString()重载方法。因为没有特殊的格式要求，所以也可以调用不带参数的 ToString()方法。

整个字符串格式化过程如图 9-2 所示。

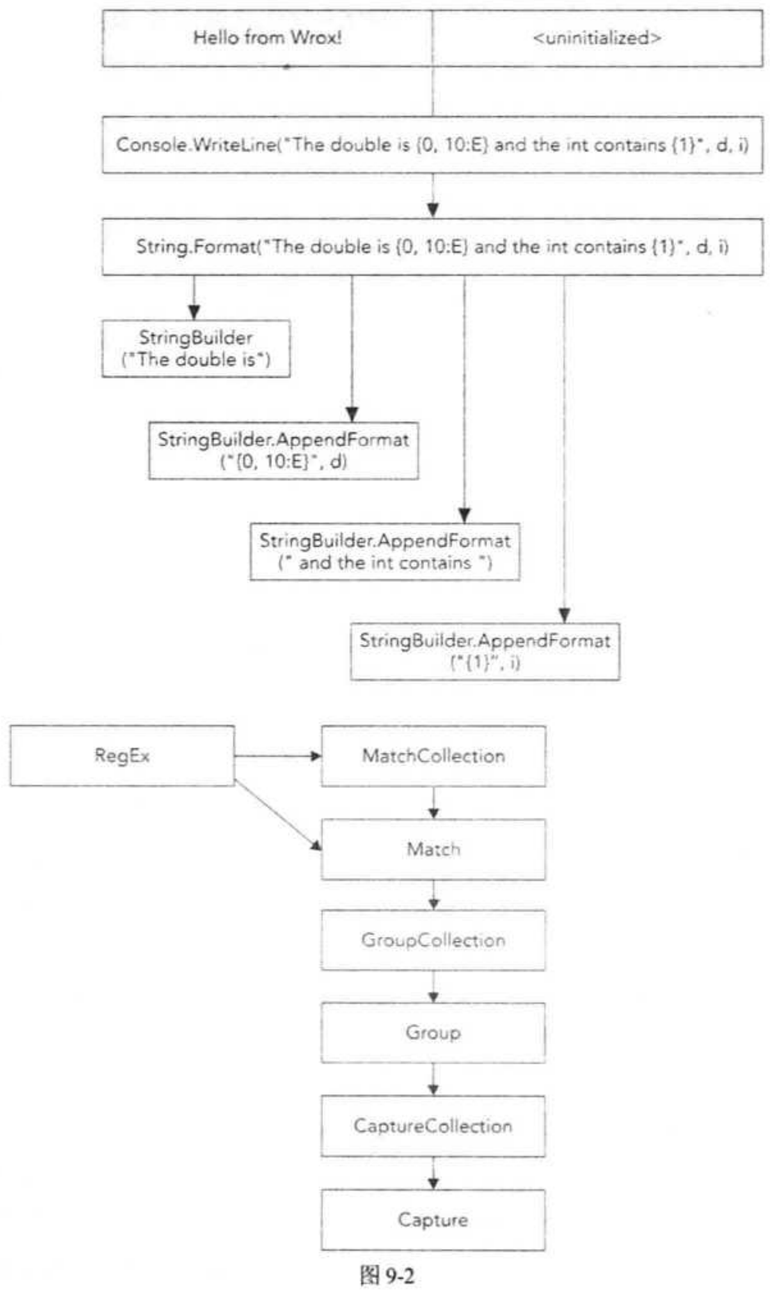


图 9-2

2. FormattableVector 示例

前面介绍了如何构造格式字符串，下面扩展第 7 章的 Vector 示例，以多种方式格式化矢量。这个示例的代码可以从 www.wrox.com 找到，文件名是 FormattableVector.cs。只要理解了所涉及的规则，实际编写代码就相当简单了。我们只需要实现 IFormattable 接口，提供由该接口定义的 ToString()重载方法的实现代码即可。

要支持的格式说明符如下：

- N——应解释为一个请求，以提供一个数字，即矢量的模，它是其成员的平方和，在数学上等于 Vector 的长度的平方，通常放在两个竖杠的中间：||34.5||。
- VE——应解释为以科学计数法显示每个成员的一个请求，如说明符 E 应用于 double，就可以表示为(2.3E+01, 4.5E+02, 1.0E+00)。
- IJK——应解释为以格式 23i + 450j + 1k 显示矢量的一个请求。
- 其他内容应仅返回 Vector 的默认表示方法(23, 450, 1.0)。

为了简单起见，显示矢量时我们不以 IJK 和科学计数法的格式实现任何选项，而是以不区分大小写的方式来测试说明符，允许使用 ijk 而不是 IJK。注意，使用什么字符串表示格式说明符完全取决于用户。

为此，首先修改 Vector 结构的声明，使之实现 IFormattable 接口：

```
struct Vector: IFormattable
{
    public double x, y, z;

    // Beginning part of Vector
```

下面添加带有两个参数的 ToString()重载方法的实现代码：

```
public string ToString(string format, IFormatProvider formatProvider)
{
  if (format == null)
  {
   return ToString();
  }

  string formatUpper = format.ToUpper();

  switch (formatUpper)
  {
    case "N":
        return "|| " + Norm().ToString() + " ||";
    case "VE":
        return String.Format("( {0:E}, {1:E}, {2:E} )", x, y, z);
    case "IJK":
        StringBuilder sb = new StringBuilder(x.ToString(), 30);
        sb.AppendFormat(" i + ");
        sb.AppendFormat(y.ToString());
        sb.AppendFormat(" j + ");
        sb.AppendFormat(z.ToString());
        sb.AppendFormat(" k");
        return sb.ToString();
    default:
       return ToString();
  }
}
```

这就是我们要编写的代码。注意在调用任何方法前，应防止使用格式字符串为空的参数。我们希望这个方法尽可能健壮。因为所有基元类型的格式说明符都不区分大小写，所以其他开发人员也

希望能以这种方式使用我们的类。对于格式说明符 VE，需要把每个成员格式化为科学计数法，所以再次使用 String.Format ()方法。字段 x、y 和 z 都是 double 类型。对于 IJK 格式说明符，把几个子字符串添加到字符串中，因此使用 StringBuilder 对象来提高性能。

为了完整起见，也可以再次使用前面开发的无参数的 ToString()重载方法：

```
public override string ToString()
{
   return "( " + x + ", " + y + ", " + z + " )";
}
```

最后，需要添加一个 Norm()方法，计算矢量的平方(模)，因为在开发 Vector 结构时，实际上没有提供这个方法：

```
public double Norm()
{
   return x*x + y*y + z*z;
}
```

下面用一些合适的测试代码测试可格式化的矢量：

```
static void Main()
{
  Vector v1 = new Vector(1,32,5);
  Vector v2 = new Vector(845.4, 54.3, -7.8);
  Console.WriteLine("\nIn IJK format,\nv1 is {0,30:IJK}\nv2 is {1,30:IJK}",
                    v1, v2);
  Console.WriteLine("\nIn default format,\nv1 is {0,30}\nv2 is {1,30}", v1, v2);
  Console.WriteLine("\nIn VE format\nv1 is {0,30:VE}\nv2 is {1,30:VE}", v1, v2);
  Console.WriteLine("\nNorms are:\nv1 is {0,20:N}\nv2 is {1,20:N}", v1, v2);
}
```

运行这个示例的结果如下所示：

```
FormattableVector
In IJK format,
v1 is          1 i + 32 j + 5 k
v2 is 845.4 i + 54.3 j + -7.8 k

In default format,
v1 is          ( 1, 32, 5 )
v2 is ( 845.4, 54.3, -7.8 )

In VE format
v1 is ( 1.000000E+000, 3.200000E+001, 5.000000E+000 )
v2 is ( 8.454000E+002, 5.430000E+001, -7.800000E+000 )

Norms are:
v1 is      || 1050 ||
v2 is  || 717710.49 ||
```

这说明了选用的自定义格式说明符是正确的。

9.2　正则表达式

正则表达式作为小型技术领域的一部分在各种程序中都有着难以置信的作用。正则表达式可以看成一种有特定功能的小型编程语言：在大的字符串表达式中定位一个子字符串。它不是一种新技术，最初它是在 UNIX 环境中开发的，与 Perl 编程语言一起使用得比较多。Microsoft 把它移植到 Windows 中，到目前为止它在脚本语言中用得比较多。但 System.Text.RegularExpressions 名称空间中的许多.NET 类都支持正则表达式。.NET Framework 的各个部分也使用正则表达式。例如，在 ASP.NET 验证服务器的控件中就使用了正则表达式。

对于不太熟悉正则表达式语言的读者，本节将主要解释正则表达式和相关的.NET 类。如果你很熟悉正则表达式，就可以浏览本节，选择学习与.NET 基类有关的内容。注意，.NET 正则表达式引擎用于兼容 Perl 5 的正则表达式，但它有一些新功能。

9.2.1　正则表达式概述

正则表达式语言是一种专门用于字符串处理的语言。它包含两个功能：

一组用于标识字符类型的转义代码。你可能很熟悉 DOS 表达式中的*字符表示任意子字符串(例如，DOS 命令 Dir Re*会列出名称以 Re 开头的所有文件)。正则表达式使用与*类似的许多序列来表示“任意一个字符”、“一个单词的中断”和“一个可选的字符”等。

- 一个系统，在搜索操作中，它把子字符串和中间结果的各个部分组合起来。

使用正则表达式，可以对字符串执行许多复杂而高级的操作，例如：

- 识别(可以是标记或删除)字符串中所有重复的单词，例如，把“The computer books books”转换为“The computer books”。
- 把所有单词都转换为标题格式，例如，把“this is a Title”转换为“This Is A Title”。
- 把长于 3 个字符的所有单词都转换为标题格式，例如，把“this is a Title”转换为“This is a Title”。
- 确保句子有正确的大写形式。
- 区分 URI 的各个元素(例如，http://www.wrox.com，提取出协议、计算机名和文件名等)。

当然，这些都是可以在 C#中用 System.String 和 System.Text.StringBuilder 的各种方法执行的任务。但是，在一些情况下，还需要编写相当多的 C#代码。如果使用正则表达式，这些代码一般可以压缩为几行。实际上，是实例化了一个对象 System.Text.RegularExpressions.RegEx(甚至更简单，调用静态的 RegEx()方法)，给它传递要处理的字符串和一个正则表达式(这是一个字符串，它包含用正则表达式语言编写的指令)，就可以了。

正则表达式字符串初看起来像是一般的字符串，但其中包含了转义序列和有特定含义的其他字符。例如，序列\b 表示一个字的开头和结尾(字的边界)，如果要表示正在查找以字符 th 开头的字，就可以编写正则表达式\bth(即字边界是序列-t -h)。如果要搜索所有以 th 结尾的单词，就可以编写 th\b(字边界是序列 t- h-)。但是，正则表达式要比这复杂得多，包括可以在搜索操作中找到存储部分文本的工具性程序。本节仅简要介绍正则表达式的功能。

正则表达式的更多信息可参阅 Andrew Watt 撰写的图书 *Beginning Regular Expressions*(John Wiley & Sons, 2005)。

假定应用程序需要把美国电话号码转换为国际格式。在美国，电话号码的格式为 314-123-1234，常常写作(314)123-1234。在把这个国家格式转换为国际格式时，必须在电话号码的前面加上+1(美国的国家代码)，并给区号加上圆括号：+1(314) 123-1234。在查找和替换时，这并不复杂，但如果要使用 String 类完成这个转换，就需要编写一些代码(这表示，必须使用 System.String 类的方法来编写代码)，而正则表达式语言可以构造一个短的字符串来表达上述含义。

所以，本节只有一个非常简单的示例，我们只考虑如何查找字符串中的某些子字符串，无须考虑如何修改它们。

9.2.2　RegularExpressionsPlayaround 示例

下面将开发一个小示例 RegularExpressionsPlayaround，通过实现并显示一些搜索的结果，说明正则表达式的一些功能，以及如何在 C#中使用.NET 正则表达式引擎。将在这个示例文档中使用的文本是对一本有关 ASP.NET 的书籍 *Professional ASP.NET 4：in C# and VB* (Wiley，2010)的简介：

```
const string myText =
@"This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entirely revised and updated for the fourth
release of .NET, this book will give you the information you need to
master ASP.NET and build a dynamic, successful, enterprise Web application.";
```

如果不考虑换行，则上面的代码是合法的 C#代码——它说明了前缀为@符号的逐字字符串的实用程序。

我们把这个文本称为输入字符串。为了说明.NET 类的正则表达式，我们先进行一次纯文本的搜索，这次搜索不带任何转义序列或正则表达式命令。假定要查找所有的字符串 ion，把这个搜索字符串称为模式。使用正则表达式和上面声明的变量 Text，编写出下面的代码：

```
const string pattern = "ion";
MatchCollection myMatches = Regex.Matches(myText, pattern,
                                          RegexOptions.IgnoreCase |
                                          RegexOptions.ExplicitCapture);

foreach (Match nextMatch in myMatches)
{
   Console.WriteLine(nextMatch.Index);
}
```

在这段代码中，使用了 System.Text.RegularExpressions 名称空间中 Regex 类的静态方法 Matches()。这个方法的参数是一些输入文本、一个模式和 RegexOptions 枚举中的一组可选标志。在本例中，指定所有的搜索都不应区分大小写。另一个标记 ExplicitCapture 改变了收集匹配的方式，对于本例，这样可以使搜索的效率更高，其原因详见后面的内容(尽管它还有这里没有探讨的其他用法)。Matches()方法返回 MatchCollections 对象的引用。匹配是一个技术术语，表示在表达式中查找模式实例的结果，用 System.Text.RegularExpressions.Match 类来表示它。因此，我们返回一个包含所有匹配的 MatchCollection，每个匹配都用一个 Match 对象来表示。在上面的代码中，只是在集合中迭代，

并使用 Match 类的 Index 属性，Match 类返回输入文本中匹配所在的索引。运行这段代码将得到 3 个匹配。表 9-4 描述了 RegexOptions 枚举的一些选项。

表 9-4

成 员 名	说 明
CultureInvariant	指定忽略字符串的文化背景
ExplicitCapture	修改收集匹配的方式，方法是确保把显式指定的匹配作为有效的搜索结果
IgnoreCase	忽略输入字符串的大小写
IgnorePatternWhitespace	在字符串中删除未转义的空白，启用通过#符号指定的注释
Multiline	修改字符^和$，把它们应用于每一行的开头和结尾，而不仅仅应用于整个字符串的开头和结尾
RightToLeft	从右到左地读取输入字符串，而不是默认地从左到右读取(适合于一些亚洲语言或其他以这种方式读取的语言)
Singleline	指定句点的含义(.)，它原来表示单行模式，现在改为匹配每个字符

到目前为止，在前面的示例中，除了一些新的.NET 基类外，其他内容都不是新的。但正则表达式的功能主要取决于模式字符串。原因是模式字符串不必仅包含纯文本。如前所述，它还可以包含元字符和转义序列，其中元字符是给出命令的特定字符，而转义序列的工作方式与 C#的转义序列相同，它们都是以反斜杠(\)开头的字符，且具有特殊的含义。

例如，假定要查找以 n 开头的字，那么可以使用转义序列\b，它表示一个字的边界(字的边界是以字母数字表中的某个字符开头，或者后面是一个空白字符或标点符号)。可以编写如下代码：

```
const string pattern = @"\bn";
MatchCollection myMatches = Regex.Matches(myText, pattern,
                                          RegexOptions.IgnoreCase |
                                          RegexOptions.ExplicitCapture);
```

注意字符串前面的符号@。要在运行时把\b 传递给.NET 正则表达式引擎，反斜杠(\)不应被 C#编译器解释为转义序列。如果要查找以序列 ion 结尾的字，就可以使用下面的代码：

```
const string pattern = @"ion\b";
```

如果要查找以字母 a 开头，以序列 ion 结尾的所有字(在本例中它仅有一个匹配的字 application)，就必须在上面的代码中添加一些内容。显然，我们需要一个以\ba 开头，以 ion\b 结尾的模式，但中间的内容怎么办？需要告诉应用程序在 a 和 ion 中间的内容可以是任意长度的字符，只要这些字符不是空白即可。实际上，正确的模式如下所示。

```
const string pattern = @"\ba\S*ion\b";
```

使用正则表达式要习惯的一点是，对像这样怪异的字符序列见怪不怪。但这个序列的工作是非常逻辑化的。转义序列\S 表示任何不是空白字符的字符。*称为限定符，其含义是前面的字符可以重复任意次，包括 0 次。序列\S*表示任意个不是空白字符的字符。因此，上面的模式匹配以 a 开头以 ion 结尾的任何单个单词。

表 9-5 是可以使用的一些主要的特定字符或转义序列，但这个表并不完整，完整的列表请参考 MSDN 文档。

表 9-5

符 号	含　义	示 例	匹配的示例
^	输入文本的开头	^B	B，但只能是文本中的第一个字符
$	输入文本的结尾	X$	X，但只能是文本中的最后一个字符
.	除了换行符(\n)以外的所有单个字符	i.ation	isation、ization
*	可以重复 0 次或多次的前导字符	ra*t	rt、rat、raat 和 raaat 等
+	可以重复 1 次或多次的前导字符	ra+t	rat、raat 和 raaat 等(但不能是 rt)
?	可以重复 0 次或 1 次的前导字符	ra?t	只有 rt 和 rat 匹配
\s	任何空白字符	\sa	[space]a、\ta、\na (\t 和\n 与 C#中的\t 和\n 含义相同)
\S	任何不是空白的字符	\SF	aF、rF、cF，但不能是\tf
\b	字边界	ion\b	以 ion 结尾的任何字
\B	不是字边界的任意位置	\BX\B	字中间的任何 X

如果要搜索其中一个元字符，就可以通过带有反斜杠的相应转义字符来表示。例如，“.” (一个句点)表示除了换行字符以外的任何单个字符，而“\.”表示一个点。

可以把替换的字符放在方括号中，请求匹配包含这些字符。例如，[1|c]表示字符可以是 l 或 c。如果要搜索 map 或 man，就可以使用序列 ma[n|p]。在方括号中，也可以指定一个范围，例如[a-z]表示所有的小写字母，[A-E]表示 A~E 之间的所有大写字母(包括字母 A 和 E)，[0-9]表示一个数字。如果要搜索一个整数(该序列只包含 0~9 的字符)，就可以编写[0-9]+。

使用“+”字符表示至少要有这样一个数字，但可以有多个数字，所以 9、 83 和 854 等都是匹配的。

9.2.3 显示结果

本节编写一个示例 RegularExpressionsPlayaround，看看正则表达式的工作方式。

该示例的核心是一个方法 WriteMatches()，它把 MatchCollection 中的所有匹配以比较详细的格式显示出来。对于每个匹配结果，它都会显示该匹配在输入字符串中的索引、匹配的字符串和一个略长的字符串，其中包含匹配结果和输入文本中至多 10 个外围字符，其中至多有 5 个字符放在匹配结果的前面，至多 5 个字符放在匹配结果的后面(如果匹配结果的位置在输入文本的开头或结尾 5 个字符内，则结果中匹配字符串前后的字符就会少于 5 个)。换言之，如果要匹配的单词是 messaging，靠近输入文本末尾的匹配结果应是“and messaging of d”，匹配结果的前后各有 5 个字符，但位于输入文本的最后一个字 data 上的匹配结果就应是"g of data" ——匹配结果的后面只有一个字符。因为在该字符的后面是字符串的结尾。这个长字符串可以更清楚地表明正则表达式是在什么地方查找到匹配结果的：

```
static void WriteMatches(string text, MatchCollection matches)
{
  Console.WriteLine("Original text was: \n\n" + text + "\n");
  Console.WriteLine("No. of matches: " + matches.Count);

  foreach (Match nextMatch in matches)
  {
    int index = nextMatch.Index;
    string result = nextMatch.ToString();
    int charsBefore = (index < 5) ? index: 5;
    int fromEnd = text.Length-index-result.Length;
    int charsAfter = (fromEnd < 5) ? fromEnd: 5;
    int charsToDisplay = charsBefore + charsAfter + result.Length;

    Console.WriteLine("Index: {0}, \tString: {1}, \t{2}",
       index, result, text.Substring(index-charsBefore, charsToDisplay));
  }
}
```

在这个方法中，处理过程是确定在较长的子字符串中有多少个字符可以显示，而无须超出输入文本的开头或结尾。注意在 Match 对象上使用了另一个属性 Value，它包含标识该匹配的字符串。而且，RegularExpressionsPlayaround 只包含名为 Find1、Find2 等的方法，这些方法根据本节中的示例执行某些搜索操作。例如，Find2 查找以 a 开头的任意字符串：

```
static void Find2()
{
  string text = @"This comprehensive compendium provides a broad and thorough
    investigation of all aspects of programming with ASP.NET. Entirely revised and
    updated for the 3.5 Release of .NET, this book will give you the information
    you need to master ASP.NET and build a dynamic, successful, enterprise Web
    application.";
  string pattern = @"\ba";
  MatchCollection matches = Regex.Matches(text, pattern,
   RegexOptions.IgnoreCase);
  WriteMatches(text, matches);
}
```

下面是一个简单的 Main()方法，可以编辑它从而选择一个 Find<n>()方法：

```
static void Main()
{
  Find1();
  Console.ReadLine();
}
```

这段代码还需要使用 RegularExpressions 名称空间：

```
using System;
using System.Text.RegularExpressions;
```

运行带有 Find 2()方法的示例，得到如下所示的结果：

```
RegularExpressionsPlayaround
Original text was:
```

```
This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entirely revised and updated for the 3.5
Release of .NET, this book will give you the information you need to master ASP.NET
and build a dynamic, successful, enterprise Web application.

No. of matches: 1
Index: 291, String: application, Web application.
```

9.2.4　匹配、组合和捕获

正则表达式的一个很好的特性是可以把字符组合起来，其工作方式与C#中的复合语句一样。在C#中，可以把任意数量的语句放在花括号中，把它们组合在一起。其结果就像一个复合语句那样。在正则表达式模式中，也可以把任何字符组合起来(包括元字符和转义序列)，像处理单个字符那样处理它们。唯一的区别是要使用圆括号，而不是花括号，得到的序列称为一组。

例如，模式(an)+定位任意重复的序列 an。限定符“+”只应用于它前面的一个字符，但因为我们把字符组合起来了，所以它现在把重复的 an 作为一个单元来对待。这意味着，如果(an)+应用到输入文本“bananas came to Europe late in the annals of history”上，就会从 bananas 中识别出 anan。另一方面，如果使用 an+，则程序将从 annals 中选择 ann，从 bananas 中选择出两个分开的 an 序列。表达式(an)+可以识别出 an、anan、ananan 等，而表达式 an+可以识别出 an、ann、annn 等。

在上面的示例中，为什么(an)+从 banana 中选择的是 anan，而没有把其中一个 an 作为一个匹配结果？因为匹配结果是不能重叠的。如果有可能重叠，在默认情况下就选择最长的匹配。

但是，组的功能要比这强大得多。在默认情况下，把模式的一部分组合为一组时，就要求正则表达式引擎按照该组来匹配，或按照整个模式来匹配。换言之，可以把组当成一个要匹配和返回的模式，如果要把字符串分解为各个部分，这种模式就非常有效。

例如，URI 的格式是<protocol>://<address>:<port>，其中端口是可选的。它的一个示例是 http://www.wrox.com:4355。假定要从一个 URI 中提取协议、地址和端口，而且不管 URI 的后面是否紧跟着空白(但没有标点符号)，那么可以使用下面的表达式：

```
\b(\S+)://([^:]+)(?::(\S+))?\b
```

该表达式的工作方式如下：首先，前导\b 序列和尾随\b 序列确保只需要考虑完全是字的文本部分。在这个文本部分中，第一组(\S+)://会识别一个或多个不是空白的字符，其后是://。在 HTTP URI 的开头会识别出 http://。花括号表示把 http 存储为一组。后面的序列([^:]+)则在上述 URI 中识别字符串 www.wrox.com，该组在遇到词的结尾(结束\b)时或标记另一组的冒号(:)时结束。

下一个组识别端口(本例是:4355)。后面的“？”表示该组在匹配过程中是可选的，如果没有:xxxx，就不会妨碍匹配的标记。这非常重要，因为端口号在 URI 中一般不指定，实际上，在大多数情况下，URI 是没有端口号的。但是，事情会比较复杂。我们希望指定冒号可以出现，也可以不出现，但不希望把这个冒号也存储在组中。为此，可以嵌套两组：内部(\S+)组识别冒号后面的内容(本例中是 4355)。外面组包含内部组，内部组的前面是一个冒号，该组又在序列“?:”的后面。这个序列表示

该组不应保存(只需要保存 4355，不需要保存:4355)。不要把这两个冒号混淆了，第一个冒号是序列“?:”的一部分，表示不保存该组，第二个冒号是要搜索的文本。

在下面的字符串上运行该模式，得到的匹配是 http://www.wrox.com。

```
Hey I've just found this amazing URI at
http:// what was it --oh yes http://www.wrox.com
```

在这个匹配过程中，找到了刚才提及的 3 组，还有第 4 组表示匹配本身。理论上，每个组都可以选择 0 次、1 次或多次匹配。单个的匹配就称为捕获。在第一组(\S+)中，有一个捕获 http，第二组也有一个捕获 www.wrox.com，但第 3 组没有捕获，因为在这个 URI 中没有端口号。

注意，该字符串包含第二个 http://。虽然它匹配第一组，但它不会被该搜索过程捕获，因为整个搜索表达式不匹配于这部分文本。

虽然前面没有介绍使用组和捕获的任何 C#示例，但下面提到的.NET 类 RegularExpressions 就通过 Group 类和 Capture 类支持组和捕获。GroupCollection 类和 CaptureCollection 类分别表示组和捕获的集合，Match 类提供一个 Groups 属性，它返回相应的 GroupCollection 对象。Group 类也相应地实现一个 Captures 属性，该属性返回 CaptureCollection 对象。这些对象之间的关系如图 9-3 所示。

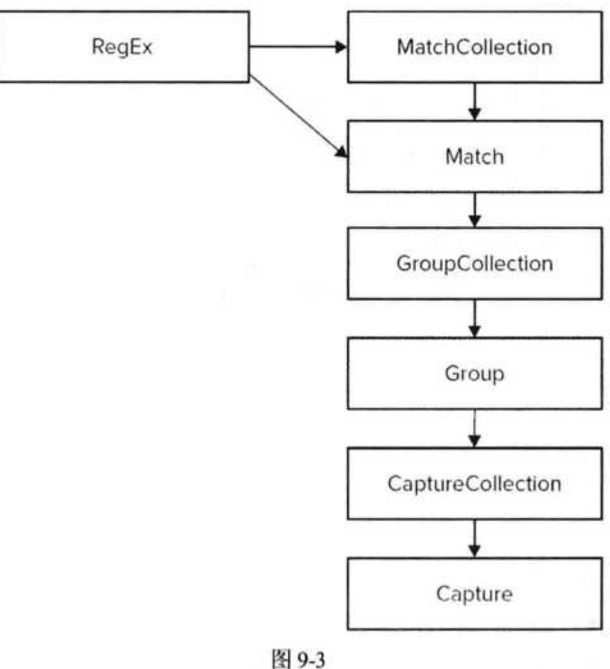


图 9-3

也许只希望把一些字符组合起来后，而不是每次都会返回一个 Group 对象。如果只是希望把一些字符组合起来，作为搜索模式的一部分，实例化对象就会浪费相当大的系统开销。对于单个组，可以以字符序列“?:”开头，禁止实例化对象，就像 URI 示例那样。而对于所有组，可以在 RegEx.Matches()方法上指定 RegExOptions.ExplicitCaptures 标志，如同前面的示例那样。

9.3 小结

在使用.NET Framework 时，可用的数据类型相当多。在应用程序(特别是关注数据提交和检索的应用程序)中，最常用的一种类型就是 String 数据类型。String 非常重要，这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理 String 数据类型的原因。

过去在使用字符串时，常常需要通过连接来分解字符串。而在.NET Framework 中，可以使用 StringBuilder 类完成许多这类任务，而且性能更好。

最后，使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种最佳工具。

第10章

集　　合

本章要点

- 理解集合接口和类型
- 使用列表、队列和栈
- 使用链表和有序列表
- 使用字典和集
- 使用位数组和位矢量
- 使用不可变和并发的集合
- 评估性能

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 列表示例(List Samples)
- 队列示例(Queue Sample)
- 链表示例(Linked List Sample)
- 有序列表示例(Sorted List Sample)
- 字典示例(Dictionary Sample)
- 集示例(Set Sample)
- 可观察的集合示例(Observable Collection Sample)
- 位数组示例(BitArray Sample)
- 不可变的集合示例(Immutable Collections Sample)
- 管道示例(Pipeline Sample)

10.1 概述

第 6 章介绍了数组和 Array 类实现的接口。数组的大小是固定的。如果元素个数是动态的，就应使用集合类。

List<T>是与数组相当的集合类。还有其他类型的集合：队列、栈、链表、字典和集。其他集合类提供的访问集合元素的 API 可能稍有不同，它们在内存中存储元素的内部结构也有区别。本章将介绍所有的集合类和它们的区别，包括性能差异。

还可以了解从多线程使用的位数组和并发集合。

.NET Framework 1.0 包含非泛型集合类，如 ArrayList 和 HashTable。CLR 2.0 添加了对泛型类和泛型集合类的支持。本章的重点是新一组的集合类，而忽略旧集合类，因为旧集合类很少需要在新应用程序中使用。

10.2 集合接口和类型

大多数集合类都可在 System.Collections 和 System.Collections.Generic 名称空间中找到。泛型集合类位于 System.Collections.Generic 名称空间中；专用于特定类型的集合类位于 System.Collections.Specialized 名称空间中。线程安全的集合类位于 System.Collections.Concurrent 名称空间中。不可变的集合类在 System.Collections.Immutable 名称空间中。

当然，组合集合类还有其他方式。集合可以根据集合类实现的接口组合为列表、集合和字典。

接口 IEnumerable 和 IEnumerator 的内容详见第 6 章。

集合和列表实现的接口如表 10-1 所示。

表 10-1

接 口	说 明
IEnumerable<T>	如果将 foreach 语句用于集合，就需要 IEnumerable 接口。这个接口定义了方法 GetEnumerator()，它返回一个实现了 IEnumerator 接口的枚举
ICollection<T>	ICollection<T>接口由泛型集合类实现。使用这个接口可以获得集合中的元素个数(Count 属性)，把集合复制到数组中(CopyTo()方法)，还可以从集合中添加和删除元素(Add()，Remove()，Clear())
IList<T>	IList<T>接口用于可通过位置访问其中的元素列表，这个接口定义了一个索引器，可以在集合的指定位置插入或删除某些项(Insert()和 RemoveAt()方法)。IList<T>接口派生自 ICollection<T>接口

(续表)

接 口	说 明
ISet<T>	ISet<T>接口由集实现。集允许合并不同的集，获得两个集的交集，检查两个集是否重叠。ISet<T>接口派生自 ICollection<T>接口
IDictionary<TKey, TValue>	IDictionary<TKey,TValue>接口由包含键和值的泛型集合类实现。使用这个接口可以访问所有的键和值，使用键类型的索引器可以访问某些项，还可以添加或删除某些项
ILookup<TKey, TValue>	ILookup<TKey，TValue>接口类似于 IDictionary<TKey,TValue>接口，实现该接口的集合有键和值，且可以通过一个键包含多个值
IComparer<T>	接口 IComparer<T>由比较器实现，通过 Compare()方法给集合中的元素排序
IEqualityComparer<T>	接口 IEqualityComparer<T>由一个比较器实现，该比较器可用于字典中的键。使用这个接口，可以对对象进行相等性比较。从.NET 4 开始，这个接口也由数组和元组实现
IProducerConsumerCollection<T>	IProducerConsumerCollection<T>接口是.NET 4 中新增的，它支持新的线程安全的集合类
IReadOnlyCollection<T> IReadOnlyList<T> IReadOnlyDictionary<TKey, TValue>	接口 IReadOnlyCollection<T>、IReadOnlyList<T>和 IReadOnlyDictionary<TKey,TValue>用于初始化后不能修改的集合。这些接口的成员只允许检索对象，不能添加或修改它们
IImmutableArray<T> IImmutableList<T> IImmutableQueue<T> IImmutableSet<T> IImmutableDictionary<TKey, TValue>	不可变接口定义了用于不可变集合的方法和属性。这些集合在初始化后不能修改

10.3 列表

.NET Framework 为动态列表提供了泛型类 List<T>。这个类实现了 IList、ICollection、IEnumerable、IList<T>、ICollection<T>和 IEnumerable<T>接口。

下面的例子将 Racer 类中的成员用作要添加到集合中的元素，以表示一级方程式的一位赛车手。这个类有 5 个属性：Id、Firstname、Lastname、Country 和 Wins 的次数。在该类的构造函数中，可以传递赛车手的姓名和获胜次数，以设置成员。重写 ToString()方法是为了返回赛车手的姓名。Racer 类也实现了泛型接口 IComparable<T>，为 Racer 类中的元素排序，还实现了 IFormattable 接口(代码文件 ListSamples/Racer.cs)。

```
[Serializable]
public class Racer: IComparable<Racer>, IFormattable
{
  public int Id { get; private set; }
  public string FirstName { get; set; }
```

```
public string LastName { get; set; }
public string Country { get; set; }
public int Wins { get; set; }
public Racer(int id, string firstName, string lastName,
             string country)
  :this(id, firstName, lastName, country, wins: 0)
{ }

public Racer(int id, string firstName, string lastName,
             string country, int wins)
{
  this.Id = id;
  this.FirstName = firstName;
  this.LastName = lastName;
  this.Country = country;
  this.Wins = wins;
}

public override string ToString()
{
  return String.Format("{0} {1}", FirstName, LastName);
}

public string ToString(string format, IFormatProvider formatProvider)
{
  if (format == null) format = "N";
  switch (format.ToUpper())
  {
    case "N": // name
      return ToString();
    case "F": // first name
      return FirstName;
    case "L": // last name
      return LastName;
    case "W": // Wins
      return String.Format("{0}, Wins: {1}", ToString(), Wins);
    case "C": // Country
      return String.Format("{0}, Country: {1}", ToString(), Country);
    case "A": // All
      return String.Format("{0}, {1} Wins: {2}", ToString(), Country,
                           Wins);
    default:
      throw new FormatException(String.Format(formatProvider,
                   "Format {0} is not supported", format));
  }
}

public string ToString(string format)
{
  return ToString(format, null);
}

public int CompareTo(Racer other)
{
  if (other == null) return -1;
```

```
        int compare = string.Compare(this.LastName, other.LastName);
        if (compare == 0)
          return string.Compare(this.FirstName, other.FirstName);
        return compare;
      }
    }
```

10.3.1 创建列表

调用默认的构造函数，就可以创建列表对象。在泛型类 List<T>中，必须为声明为列表的值指定类型。下面的代码说明了如何声明一个包含 int 的 List<T>泛型类和一个包含 Racer 元素的列表。ArrayList 是一个非泛型列表，它可以将任意 Object 类型作为其元素。

使用默认的构造函数创建一个空列表。元素添加到列表中后，列表的容量就会扩大为可接纳 4 个元素。如果添加了第 5 个元素，列表的大小就重新设置为包含 8 个元素。如果 8 个元素还不够，列表的大小就重新设置为包含 16 个元素。每次都会将列表的容量重新设置为原来的 2 倍。

```
var intList = new List<int>();
var racers = new List<Racer>();
```

如果列表的容量改变了，整个集合就要重新分配到一个新的内存块中。在 List<T>泛型类的实现代码中，使用了一个 T 类型的数组。通过重新分配内存，创建一个新数组，Array.Copy()方法将旧数组中的元素复制到新数组中。为节省时间，如果事先知道列表中元素的个数，就可以用构造函数定义其容量。下面创建了一个容量为 10 个元素的集合。如果该容量不足以容纳要添加的元素，就把集合的大小重新设置为包含 20 或 40 个元素，每次都是原来的 2 倍。

```
List<int> intList = new List<int>(10);
```

使用 Capacity 属性可以获取和设置集合的容量。

```
intList.Capacity = 20;
```

容量与集合中元素的个数不同。集合中的元素个数可以用 Count 属性读取。当然，容量总是大于或等于元素个数。只要不把元素添加到列表中，元素个数就是 0。

```
Console.WriteLine(intList.Count);
```

如果已经将元素添加到列表中，且不希望添加更多的元素，就可以调用 TrimExcess()方法，去除不需要的容量。但是，因为重新定位需要时间，所以如果元素个数超过了容量的 90%，TrimExcess()方法就什么也不做。

```
intList.TrimExcess();
```

1. 集合初始值设定项

还可以使用集合初始值设定项给集合赋值。集合初始值设定项的语法类似于第 6 章介绍的数组初始值设定项。使用集合初始值设定项，可以在初始化集合时，在花括号中给集合赋值：

```
var intList = new List<int>() {1, 2};
var stringList = new List<string>() {"one", "two"};
```

集合初始值设定项没有反映在已编译的程序集的IL代码中。编译器会把集合初始值设定项变成对初始值设定项列表中的每一项调用Add()方法。

2. 添加元素

使用Add()方法可以给列表添加元素，如下所示。实例化的泛型类型定义了Add()方法的参数类型：

```
var intList = new List<int>();
intList.Add(1);
intList.Add(2);
var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

把racers变量定义为List<Racer>类型。使用new运算符创建相同类型的一个新对象。因为类List<T>用具体类Racer来实例化，所以现在只有Racer对象可以用Add()方法添加。在下面的示例代码中，创建了5个一级方程式赛车手，并把它们添加到集合中。前3个用集合初始值设定项添加，后两个通过显式调用Add()方法来添加(代码文件ListSamples/Program.cs)。

```
var graham = new Racer(7, "Graham", "Hill", "UK", 14);
var emerson = new Racer(13, "Emerson", "Fittipaldi", "Brazil", 14);
var mario = new Racer(16, "Mario", "Andretti", "USA", 12);

var racers = new List<Racer>(20) {graham, emerson, mario};

racers.Add(new Racer(24, "Michael", "Schumacher", "Germany", 91));
racers.Add(new Racer(27, "Mika", "Hakkinen", "Finland", 20));
```

使用List<T>类的AddRange()方法，可以一次给集合添加多个元素。因为AddRange()方法的参数是IEnumerable<T>类型的对象，所以也可以传递一个数组，如下所示：

```
racers.AddRange(new Racer[] {
    new Racer(14, "Niki", "Lauda", "Austria", 25),
    new Racer(21, "Alain", "Prost", "France", 51)});
```

集合初始值设定项只能在声明集合时使用。AddRange()方法则可以在初始化集合后调用。如果在创建集合后动态获取数据，就需要调用AddRange()。

如果在实例化列表时知道集合的元素个数，就也可以将实现IEnumerable<T>类型的任意对象传递给类的构造函数。这非常类似于AddRange()方法：

```
var racers = new List<Racer>(
    new Racer[] {
        new Racer(12, "Jochen", "Rindt", "Austria", 6),
        new Racer(22, "Ayrton", "Senna", "Brazil", 41) });
```

3. 插入元素

使用 Insert()方法可以在指定位置插入元素：

```
racers.Insert(3, new Racer(6, "Phil", "Hill", "USA", 3));
```

方法 InsertRange()提供了插入大量元素的功能，类似于前面的 AddRange()方法。

如果索引集大于集合中的元素个数，就抛出 ArgumentOutOfRangeException 类型的异常。

4. 访问元素

实现了 IList 和 IList<T>接口的所有类都提供了一个索引器，所以可以使用索引器，通过传送元素号来访问元素。第一个元素可以用索引值 0 来访问。指定 racers[3]，可以访问列表中的第 4 个元素：

```
Racer r1 = racers[3];
```

可以用 Count 属性确定元素个数，再使用 for 循环遍历集合中的每个元素，并使用索引器访问每一项：

```
for (int i = 0; i < racers.Count; i++)
{
  Console.WriteLine(racers[i]);
}
```

可以通过索引访问的集合类有 ArrayList、StringCollection 和 List<T>。

因为 List<T>集合类实现了 IEnumerable 接口，所以也可以使用 foreach 语句遍历集合中的元素。

```
foreach (Racer r in racers)
{
  Console.WriteLine(r);
}
```

编译器解析 foreach 语句时，利用了 IEnumerable 和 IEnumerator 接口，参见第 6 章。

除了使用 foreach 语句之外，List<T>类还提供了 ForEach()方法，该方法用 Action<T>参数声明。

```
public void ForEach(Action<T> action);
```

实现 ForEach()方法的代码如下。ForEach()方法遍历集合中的每一项，调用作为每一项的参数传递的方法。

```
public class List<T>: IList<T>
{
```

```
    private T[] items;

    //...

    public void ForEach(Action<T> action)
    {
      if (action == null) throw new ArgumentNullException("action");

      foreach (T item in items)
      {
        action(item);
      }
    }
    //...
  }
```

为了给 ForEach()方法传递一个方法，Action<T>声明为一个委托，该委托定义了一个返回类型为 void、参数为 T 的方法。

```
public delegate void Action<T>(T obj);
```

在 Racer 项的列表中，ForEach()方法的处理程序必须声明为以 Racer 对象作为参数，返回类型是 void。

```
public void ActionHandler(Racer obj);
```

因为 Console.WriteLine()方法的一个重载版本将 Object 作为参数，所以可以将这个方法的地址传送给 ForEach()方法，把该集合的每个赛车手写入控制台：

```
racers.ForEach(Console.WriteLine);
```

也可以编写一个 lambda 表达式，它将 Racer 对象作为参数，其实现代码使用 Console. WriteLine()方法在控制台时写入一个字符串。这里，格式 A 由 IFormattable 接口的 ToString()方法用于显示赛车手的所有信息：

```
racers.ForEach(r => Console.WriteLine("{0:A}", r));
```

lambda 表达式详见第 8 章。

5. 删除元素

删除元素时，可以利用索引，也可以传递要删除的元素。下面的代码把 3 传递给 RemoveAt()方法，删除第 4 个元素：

```
racers.RemoveAt(3);
```

也可以直接将 Racer 对象传送给 Remove()方法，来删除这个元素。按索引删除比较快，因为必须在集合中搜索要删除的元素。Remove()方法先在集合中搜索，用 IndexOf()方法获取元素的索引，再使用该索引删除元素。IndexOf()方法先检查元素类型是否实现了 IEquatable<T>接口。如果是，就调用这个接口的 Equals()方法，确定集合中的元素是否等于传递给 Equals()方法的元素。如果没有实

现这个接口，就使用 Object 类的 Equals()方法比较这些元素。Object 类中 Equals()方法的默认实现代码对值类型进行按位比较，对引用类型只比较其引用。

第 7 章介绍了如何重写 Equals()方法。

这里从集合中删除了变量 graham 引用的赛车手。变量 graham 是前面在填充集合时创建的。因为 IEquatable<T>接口和 Object.Equals()方法都没有在 Racer 类中重写，所以不能用要删除元素的相同内容创建一个新对象，再把它传递给 Remove()方法。

```
if (!racers.Remove(graham))
{
  Console.WriteLine("object not found in collection");
}
```

RemoveRange()方法可以从集合中删除许多元素。它的第一个参数指定了开始删除的元素索引，第二个参数指定了要删除的元素个数。

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

要从集合中删除有指定特性的所有元素，可以使用 RemoveAll()方法。这个方法在搜索元素时使用下面将讨论的 Predicate<T>参数。要删除集合中的所有元素，可以使用 ICollection<T>接口定义的 Clear()方法。

6. 搜索

有不同的方式在集合中搜索元素。可以获得要查找的元素的索引，或者搜索元素本身。可以使用的方法有 IndexOf()、LastIndexOf()、FindIndex()、FindLastIndex()、Find()和 FindLast()。如果只检查元素是否存在，List<T>类就提供了 Exists()方法。

IndexOf()方法需要将一个对象作为参数，如果在集合中找到该元素，这个方法就返回该元素的索引。如果没有找到该元素，就返回－1。IndexOf()方法使用 IEquatable<T>接口来比较元素。

```
int index1 = racers.IndexOf(mario);
```

使用 IndexOf()方法，还可以指定不需要搜索整个集合，但必须指定从哪个索引开始搜索以及比较时要迭代的元素个数。

除了使用 IndexOf()方法搜索指定的元素之外，还可以搜索有某个特性的元素，该特性可以用 FindIndex()方法来定义。FindIndex()方法需要一个 Predicate 类型的参数：

```
public int FindIndex(Predicate<T> match);
```

Predicate<T>类型是一个委托，该委托返回一个布尔值，并且需要把类型 T 作为参数。这个委托的用法与 ForEach()方法中的 Action 委托类似。如果 Predicate<T>委托返回 true，就表示有一个匹配元素，并且找到了相应的元素。如果它返回 false，就表示没有找到元素，搜索将继续。

```
public delegate bool Predicate<T>(T obj);
```

在 List<T>类中，把 Racer 对象作为类型 T，所以可以将一个方法(该方法将类型 Racer 定义为一个参数且返回一个布尔值)的地址传递给 FindIndex()方法。查找指定国家的第一个赛车手时，可以创建如下所示的 FindCountry 类。FindCountryPredicate()方法的签名和返回类型通过 Predicate<T>委托定义。Find()方法使用变量 country 搜索用 FindCountry 类的构造函数定义的某个国家。

```
public class FindCountry
{
  public FindCountry(string country)
  {
    this.country = country;
  }
  private string country;

  public bool FindCountryPredicate(Racer racer)
  {
    Contract.Requires<ArgumentNullException>(racer != null);

    return racer.Country == country;
  }
}
```

使用 FindIndex()方法可以创建 FindCountry 类的一个新实例，把表示一个国家的字符串传递给构造函数，再传递 Find()方法的地址。FindIndex()方法成功完成后，index2 就包含集合中赛车手的 Country 属性设置为 Finland 的第一项的索引。

```
int index2 = racers.FindIndex(new FindCountry("Finland").
                              FindCountryPredicate);
```

除了用处理程序方法创建类之外，还可以在这里创建 lambda 表达式。结果与前面完全相同。现在 lambda 表达式定义了实现代码，来搜索 Country 属性设置为 Finland 的元素。

```
int index3 = racers.FindIndex(r => r.Country == "Finland");
```

与 IndexOf()方法类似，使用 FindIndex()方法也可以指定搜索开始的索引和要遍历的元素个数。为了从集合中的最后一个元素开始向前搜索某个索引，可以使用 FindLastIndex()方法。

FindIndex()方法返回所查找元素的索引。除了获得索引之外，还可以直接获得集合中的元素。Find()方法需要一个 Predicate<T>类型的参数，这与 FindIndex()方法类似。下面的 Find()方法搜索列表中 FirstName 属性设置为 Niki 的第一个赛车手。当然，也可以实现 FindLast()方法，查找与 Predicate<T>类型匹配的最后一项。

```
Racer racer = racers.Find(r => r.FirstName == "Niki");
```

要获得与 Predicate<T>类型匹配的所有项，而不是一项，可以使用 FindAll()方法。FindAll()方法使用的 Predicate<T>委托与 Find()和 FindIndex()方法相同。FindAll()方法在找到第一项后，不会停止搜索，而是继续迭代集合中的每一项，并返回 Predicate<T>类型是 true 的所有项。

这里调用了 FindAll()方法，返回 Wins 属性设置为大于 20 的整数的所有 racer 项。从 bigWinners 列表中引用所有赢得超过 20 场比赛的赛车手。

```
List<Racer> bigWinners = racers.FindAll(r => r.Wins > 20);
```

用 foreach 语句遍历 bigWinners 变量，结果如下：

```
foreach (Racer r in bigWinners)
{
  Console.WriteLine("{0:A}", r);
}
```

```
Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25
Alain Prost, France Wins: 51
```

这个结果没有排序，但这是下一步要做的工作。

7. 排序

List<T>类可以使用 Sort()方法对元素排序。Sort()方法使用快速排序算法，比较所有的元素，直到整个列表排好序为止。

Sort()方法使用了几个重载的方法。可以传递给它的参数有泛型委托 Comparison<T>和泛型接口 IComparer<T>，以及一个范围值和泛型接口 IComparer<T>。

```
public void List<T>.Sort();
public void List<T>.Sort(Comparison<T>);
public void List<T>.Sort(IComparer<T>);
public void List<T>.Sort(Int32, Int32, IComparer<T>);
```

只有集合中的元素实现了 IComparable 接口，才能使用不带参数的 Sort()方法。

Racer 类实现了 IComparable<T>接口，可以按姓氏对赛车手排序：

```
racers.Sort();
racers.ForEach(Console.WriteLine);
```

如果需要按照元素类型不默认支持的方式排序，就应使用其他技术，例如传递一个实现了 IComparer<T>接口的对象。

RacerComparer 类为 Racer 类型实现了接口 IComparer<T>。这个类允许按名字、姓氏、国籍或获胜次数排序。排序的种类用内部枚举类型 CompareType 定义。CompareType 枚举类型用 RacerComparer 类的构造函数设置。IComparer<Racer>接口定义了排序所需的 Compare()方法。在这个方法的实现代码中，使用了 string 和 int 类型的 CompareTo()方法(代码文件 ListSamples/RacerComparer.cs)。

```
public class RacerComparer: IComparer<Racer>
{
  public enum CompareType
  {
    FirstName,
    LastName,
    Country,
    Wins
  }
```

```
    private CompareType compareType;
    public RacerComparer(CompareType compareType)
    {
      this.compareType = compareType;
    }

    public int Compare(Racer x, Racer y)
    {
      if (x == null && y == null) return 0;
      if (x == null) return -1;
      if (y == null) return 1;
      int result;
      switch (compareType)
      {
        case CompareType.FirstName:
          return string.Compare(x.FirstName, y.FirstName);
        case CompareType.LastName:
          return string.Compare(x.LastName, y.LastName);
        case CompareType.Country:
          result = string.Compare(x.Country, y.Country);
          if (result == 0)
            return string.Compare(x.LastName, y.LastName);
          else
            return result;
        case CompareType.Wins:
          return x.Wins.CompareTo(y.Wins);
        default:
          throw new ArgumentException("Invalid Compare Type");
      }
    }
  }
```

如果传递给 Compare 方法的两个元素的顺序相同，该方法返回 0。如果返回值小于 0，说明第一个参数小于第二个参数；如果返回值大于 0，则第一个参数大于第二个参数。传递 null 作为参数时，Compare 方法并不会抛出一个 NullReferenceException 异常。相反，因为 null 的位置在其他任何元素之前，所以如果第一个参数为 null，该方法返回-1，如果第二个参数为 null，则返回+1。

现在，可以对 RacerComparer 类的一个实例使用 Sort()方法。传递枚举 RacerComparer.CompareType.Country，按属性 Country 对集合排序：

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
racers.ForEach(Console.WriteLine);
```

排序的另一种方式是使用重载的 Sort()方法，该方法需要一个 Comparison<T>委托：

```
public void List<T>.Sort(Comparison<T>);
```

Comparison<T>是一个方法的委托，该方法有两个T类型的参数，返回类型为int。如果参数值相等，该方法就必须返回0。如果第一个参数比第二个小，它就必须返回一个小于0的值；否则，必须返回一个大于0的值。

```
public delegate int Comparison<T>(T x, T y);
```

现在可以把一个lambda表达式传递给Sort()方法，按获胜次数排序。两个参数的类型是Racer，在其实现代码中，使用int类型的CompareTo()方法比较Wins属性。在实现代码中，因为以逆序方式使用r2和r1，所以获胜次数以降序方式排序。调用方法之后，完整的赛车手列表就按赛车手的获胜次数排序。

```
racers.Sort((r1, r2) => r2.Wins.CompareTo(r1.Wins));
```

也可以调用Reverse()方法，逆转整个集合的顺序。

8. 类型转换

使用List<T>类的ConvertAll<TOutput>()方法，可以把所有类型的集合转换为另一种类型。ConvertAll<TOutput>()方法使用一个Converter委托，该委托的定义如下：

```
public sealed delegate TOutput Converter<TInput, TOutput>(TInput from);
```

泛型类型TInput和TOutput用于转换。TInput是委托方法的参数，TOutput是返回类型。

在这个例子中，所有的Racer类型都应转换为Person类型。Racer类型包含姓氏、名字、国籍和获胜次数，而Person类型只包含名字。为了进行转换，可以忽略赛车手的国籍和获胜次数，但姓名必须转换：

```
[Serializable]
public class Person
{
  private string name;

  public Person(string name)
  {
    this.name = name;
  }

  public override string ToString()
  {
    return name;
  }
}
```

转换时调用了racers.ConvertAll<Person>()方法。这个方法的参数定义为一个lambda表达式，其参数的类型是Racer，返回类型是Person。在lambda表达式的实现代码中，创建并返回了一个新的Person对象。对于Person对象，把FirstName和LastName传递给构造函数：

```
List<Person> persons =
    racers.ConvertAll<Person>(
        r => new Person(r.FirstName + " " + r.LastName));
```

转换的结果是一个列表，其中包含转换过的 Person 对象：类型为 List<Person>的 persons 列表。

10.3.2　只读集合

创建集合后，它们就是可读写的。否则就不能给它们填充值了。但是，在填充完集合后，可以创建只读集合。List<T>集合的 AsReadOnly()方法返回 ReadOnlyCollection<T>类型的对象。ReadOnlyCollection<T>类实现的接口与 List<T>集合相同，但所有修改集合的方法和属性都抛出 NotSupportedException 异常。除了 List<T>的接口之外，ReadOnlyCollection<T>还实现了 IReadOnlyCollection<T>和 IReadOnlyList<T>接口。因为这些接口的成员，集合不能修改。

10.4　队列

队列是其元素以先进先出(FIFO)的方式来处理的集合。先放入队列中的元素会先读取。队列的例子有在机场排的队列、人力资源部中等待处理求职信的队列和打印队列中等待处理的打印任务，以及按循环方式等待 CPU 处理的线程。另外，还常常有元素根据其优先级来处理的队列。例如，在机场的队列中，商务舱乘客的处理要优先于经济舱的乘客。这里可以使用多个队列，一个队列对应一个优先级。在机场，这很常见，因为商务舱乘客和经济舱乘客有不同的登记队列。打印队列和线程也是这样。可以为一组队列建立一个数组，数组中的一项代表一个优先级。在每个数组项中都有一个队列，其中按照 FIFO 的方式进行处理。

本章的后面将使用链表的另一种实现方式，来定义优先级列表。

队列使用 System.Collections.Generic 名称空间中的泛型类 Queue<T>实现。在内部，Queue<T>类使用 T 类型的数组，这类似于 List<T>类型。它实现 ICollection 和 IEnumerable<T>接口，但没有实现 ICollection<T>接口，因为这个接口定义的 Add()和 Remove()方法不能用于队列。

因为 Queue<T>类没有实现 IList<T>接口，所以不能用索引器访问队列。队列只允许在队列中添加元素，该元素会放在队列的尾部(使用 Enqueue()方法)，从队列的头部获取元素(使用 Dequeue()方法)。

图 10-1 显示了队列的元素。Enqueue()方法在队列的一端添加元素，Dequeue()方法在队列的另一端读取和删除元素。再次调用 Dequeue()方法，会删除队列中的下一项。

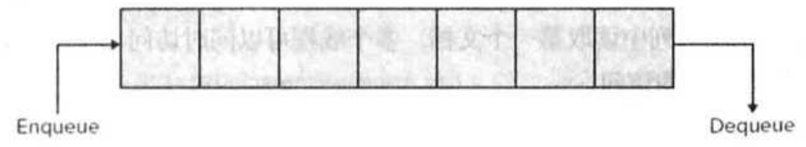


图 10-1

Queue<T>类的方法如表 10-2 所示。

表 10-2

Queue<T>类的成员	说　明
Count	Count 属性返回队列中的元素个数
Enqueue	Enqueue()方法在队列一端添加一个元素
Dequeue	Dequeue()方法在队列的头部读取和删除一个元素。如果在调用 Dequeue()方法时，队列中不再有元素，就抛出一个 InvalidOperationException 类型的异常
Peek	Peek()方法从队列的头部读取一个元素，但不删除它
TrimExcess	TrimExcess()方法重新设置队列的容量。Dequeue()方法从队列中删除元素，但它不会重新设置队列的容量。要从队列的头部去除空元素，应使用 TrimExcess()方法

在创建队列时，可以使用与 List<T>类型类似的构造函数。虽然默认的构造函数会创建一个空队列，但也可以使用构造函数指定容量。在把元素添加到队列中时，如果没有定义容量，容量就会递增，从而包含 4、8、16 和 32 个元素。类似于 List<T>类，队列的容量也总是根据需要成倍增加。非泛型类 Queue 的默认构造函数与此不同，它会创建一个包含 32 项的空数组。使用构造函数的重载版本，还可以将实现了 IEnumerable<T>接口的其他集合复制到队列中。

下面的文档管理应用程序示例说明了 Queue<T>类的用法。使用一个线程将文档添加到队列中，用另一个线程从队列中读取文档，并处理它们。

存储在队列中的项是 Document 类型。Document 类定义了标题和内容(代码文件 QueueSample/Document.cs)：

```
public class Document
{
  public string Title { get; private set; }
  public string Content { get; private set; }

  public Document(string title, string content)
  {
    this.Title = title;
    this.Content = content;
  }
}
```

DocumentManager 类是 Queue<T>类外面的一层。DocumentManager 类定义了如何处理文档：用 AddDocument()方法将文档添加到队列中，用 GetDocument()方法从队列中获得文档。

在 AddDocument()方法中，用 Enqueue()方法把文档添加到队列的尾部。在 GetDocument()方法中，用 Dequeue()方法从队列中读取第一个文档。多个线程可以同时访问 DocumentManager 类，所以用 lock 语句锁定对队列的访问。

线程和 lock 语句参见第 21 章。

IsDocumentAvailable 是一个只读类型的布尔属性，如果队列中还有文档，它就返回 true，否则返回 false(代码文件 QueueSample/DocumentManager.cs)。

```
public class DocumentManager
{
  private readonly Queue<Document> documentQueue = new Queue<Document>();

  public void AddDocument(Document doc)
  {
    lock (this)
    {
      documentQueue.Enqueue(doc);
    }
  }

  public Document GetDocument()
  {
    Document doc = null;
    lock (this)
    {
      doc = documentQueue.Dequeue();
    }
    return doc;
  }

  public bool IsDocumentAvailable
  {
    get
    {
      return documentQueue.Count > 0;
    }
  }
}
```

ProcessDocuments 类在一个单独的任务中处理队列中的文档。能从外部访问的唯一方法是 Start()。在 Start()方法中，实例化了一个新任务。创建一个 ProcessDocuments 对象，来启动任务，定义 Run()方法作为任务的启动方法。TaskFactory(通过 Task 类的静态属性 Factory 访问)的 StartNew 方法需要一个 Action 委托作为参数，用于接受 Run 方法传递的地址。TaskFactory 的 StartNew 方法会立即启动任务。

使用 ProcessDocuments 类的 Run()方法定义一个无限循环。在这个循环中，使用属性 IsDocumentAvailable 确定队列中是否还有文档。如果队列中还有文档，就从 DocumentManager 类中提取文档并处理。这里的处理仅是把信息写入控制台。在真正的应用程序中，文档可以写入文件、数据库，或通过网络发送(代码文件 QueueSample/ProcessDocuments.cs)。

```
public class ProcessDocuments
{
  public static void Start(DocumentManager dm)
  {
    Task.Factory.StartNew(new ProcessDocuments(dm).Run);
  }

  protected ProcessDocuments(DocumentManager dm)
  {
    if (dm == null)
      throw new ArgumentNullException("dm");
```

```
      documentManager = dm;
    }

    private DocumentManager documentManager;

    protected void Run()
    {
      while (true)
      {
        if (documentManager.IsDocumentAvailable)
        {
          Document doc = documentManager.GetDocument();
          Console.WriteLine("Processing document {0}", doc.Title);
        }
        Thread.Sleep(new Random().Next(20));
      }
    }
  }
```

在应用程序的Main()方法中，实例化一个DocumentManager对象，启动文档处理任务。接着创建1000个文档，并添加到DocumentManager对象中(代码文件QueueSample/Program.cs)：

```
class Program
{
  static void Main()
  {
    var dm = new DocumentManager();

    ProcessDocuments.Start(dm);

    // Create documents and add them to the DocumentManager
    for (int i = 0; i < 1000; i++)
    {
      var doc = new Document("Doc " + i.ToString(), "content");
      dm.AddDocument(doc);
      Console.WriteLine("Added document {0}", doc.Title);
      Thread.Sleep(new Random().Next(20));
    }
  }
}
```

在启动应用程序时，会在队列中添加和删除文档，输出如下所示：

```
Added document Doc 279
Processing document Doc 236
Added document Doc 280
Processing document Doc 237
Added document Doc 281
Processing document Doc 238
Processing document Doc 239
Processing document Doc 240
Processing document Doc 241
Added document Doc 282
Processing document Doc 242
Added document Doc 283
```

```
Processing document Doc 243
```

完成示例应用程序中描述的任务的真实程序可以处理用 Web 服务接收到的文档。

10.5 栈

栈是与队列非常类似的另一个容器，只是要使用不同的方法访问栈。最后添加到栈中的元素会最先读取。栈是一个后进先出(LIFO)的容器。

图 10-2 表示一个栈，用 Push()方法在栈中添加元素，用 Pop()方法获取最近添加的元素。

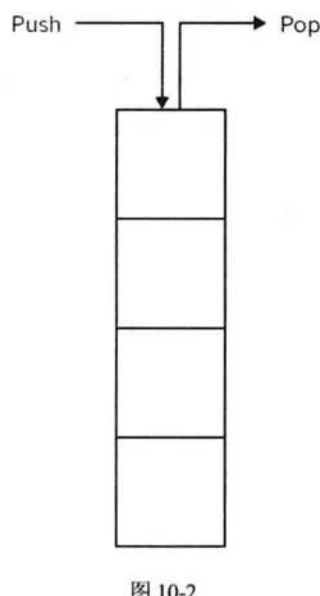


图 10-2

与 Queue<T>类相同，Stack<T>类实现 IEnumerable<T>和 ICollection 接口。

Stack<T>类的成员如表 10-3 所示。

表 10-3

Stack<T>类的成员	说 明
Count	返回栈中的元素个数
Push	在栈顶添加一个元素
Pop	从栈顶删除一个元素，并返回该元素。如果栈是空的，就抛出 InvalidOperationException 异常
Peek	返回栈顶的元素，但不删除它
Contains	确定某个元素是否在栈中，如果是，就返回 true

在下面的例子中，使用 Push()方法把 3 个元素添加到栈中。在 foreach 方法中，使用 IEnumerable 接口迭代所有的元素。栈的枚举器不会删除元素，它只会逐个返回元素(代码文件 StackSample/Program.cs)。

```
var alphabet = new Stack<char>();
```

```
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
```

因为元素的读取顺序是从最后一个添加到栈中的元素开始到第一个元素，所以得到的结果如下：

```
CBA
```

用枚举器读取元素不会改变元素的状态。使用 Pop()方法会从栈中读取每个元素，然后删除它们。这样，就可以使用 while 循环迭代集合，检查 Count 属性，确定栈中是否还有元素：

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

Console.Write("First iteration: ");
foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();

Console.Write("Second iteration: ");
while (alphabet.Count > 0)
{
  Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

结果是两个 CBA，每次迭代对应一个 CBA。在第二次迭代后，栈变空，因为第二次迭代使用了 Pop()方法：

```
First iteration: CBA
Second iteration: CBA
```

10.6 链表

LinkedList<T>是一个双向链表，其元素指向它前面和后面的元素，如图 10-3 所示。这样一来，通过移动到下一个元素可以正向遍历整个链表，通过移动到前一个元素可以反向遍历整个链表。

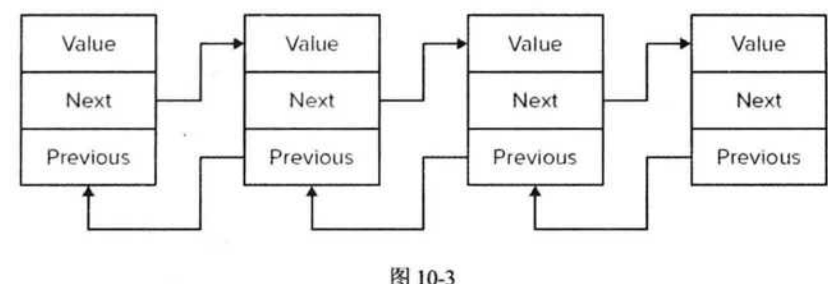


图 10-3

链表的优点是，如果将元素插入列表的中间位置，使用链表就会非常快。在插入一个元素时，只需要修改上一个元素的 Next 引用和下一个元素的 Previous 引用，使它们引用所插入的元素。在 List<T>类中，插入一个元素时，需要移动该元素后面的所有元素。

当然，链表也有缺点。链表的元素只能一个接一个地访问，这需要较长的时间来查找位于链表中间或尾部的元素。

链表不能在列表中仅存储元素。存储元素时，链表还必须存储每个元素的下一个元素和上一个元素的信息。这就是 LinkedList<T>包含 LinkedListNode<T>类型的元素的原因。使用 LinkedListNode<T>类，可以获得列表中的下一个元素和上一个元素。LinkedListNode<T>定义了属性 List、Next、Previous 和 Value。List 属性返回与节点相关的 LinkedList<T>对象，Next 和 Previous 属性用于遍历链表，访问当前节点之后和之前的节点。Value 返回与节点相关的元素，其类型是 T。

LinkedList<T>类定义的成员可以访问链表中的第一个和最后一个元素(First 和 Last)、在指定位置插入元素(AddAfter()、AddBefore()、AddFirst()和 AddLast()方法)，删除指定位置的元素(Remove()、RemoveFirst()和 RemoveLast()方法)、从链表的开头(Find()方法)或结尾(FindLast()方法)开始搜索元素。

示例应用程序使用了一个链表和一个列表。链表包含文档，这与上一个队列例子相同，但文档有一个额外的优先级。在链表中，文档按照优先级来排序。如果多个文档的优先级相同，这些元素就按照文档的插入时间来排序。

图 10-4 描述了示例应用程序中的集合。LinkedList<Document>是一个包含所有 Document 对象的链表，该图显示了文档的标题和优先级。标题指出了文档添加到链表中的时间。第一个添加的文档的标题是"One"。第二个添加的文档的标题是"Two"，依此类推。可以看出，文档 One 和 Four 有相同的优先级 8，因为 One 在 Four 之前添加，所以 One 放在链表的前面。

在链表中添加新文档时，它们应放在优先级相同的最后一个文档后面。集合 LinkedList <Document>包含 LinkedListNode<Document>类型的元素。LinkedListNode<T>类添加 Next 和 Previous 属性，使搜索过程能从一个节点移动到下一个节点上。要引用这类元素，应把 List<T>定义为 List<LinkedListNode<Document>>。为了快速访问每个优先级的最后一个文档，集合 List<LinkedList-Node>应最多包含 10 个元素，每个元素分别引用每个优先级的最后一个文档。在后面的讨论中，对每个优先级的最后一个文档的引用称为优先级节点。

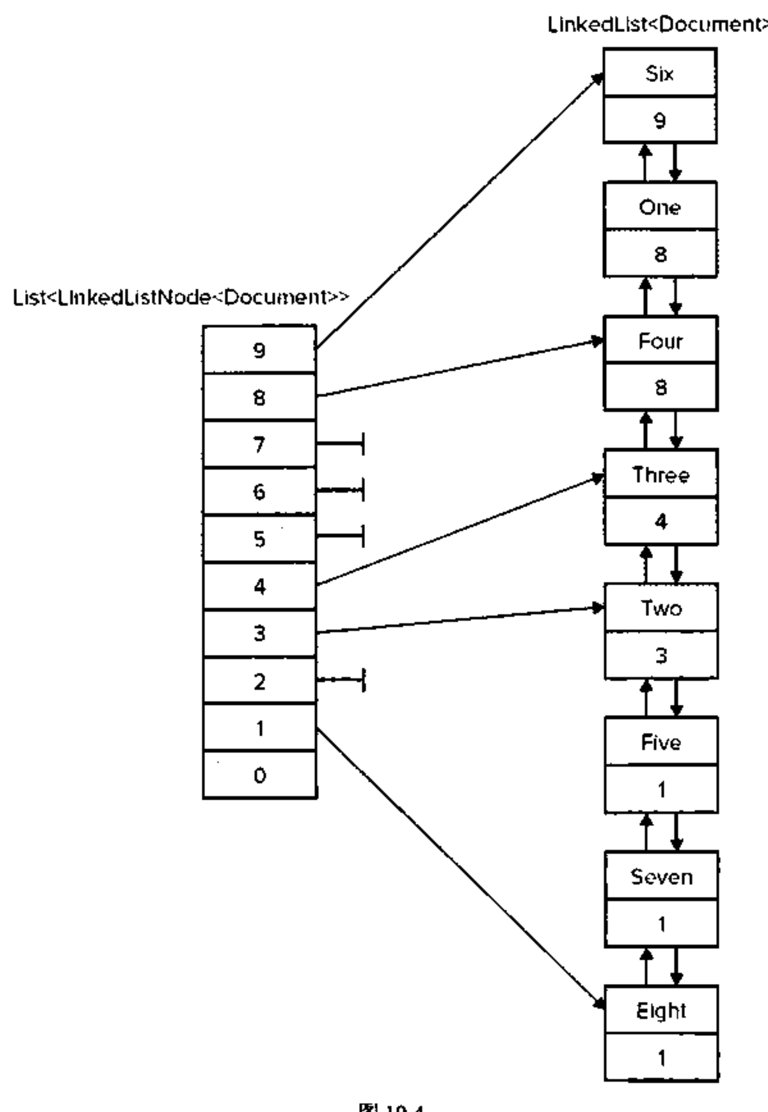


图 10-4

在上面的例子中，Document 类扩展为包含优先级。优先级用类的构造函数设置(代码文件 LinkedListSample/Document.cs)：

```
public class Document
{
  public string Title { get; private set; }
  public string Content { get; private set; }
  public byte Priority { get; private set; }

  public Document(string title, string content, byte priority)
  {
    this.Title = title;
    this.Content = content;
    this.Priority = priority;
  }
}
```

解决方案的核心是 PriorityDocumentManager 类。这个类很容易使用。在这个类的公共接口中，

可以把新的 Document 元素添加到链表中，可以检索第一个文档，为了便于测试，它还提供了一个方法，在元素链接到链表中时，该方法可以显示集合中的所有元素。

PriorityDocumentManager 类包含两个集合。LinkedList<Document>类型的集合包含所有的文档。List<LinkedListNode<Document>>类型的集合包含最多 10 个元素的引用，它们是添加指定优先级的新文档的入口点。这两个集合变量都用 PriorityDocumentManager 类的构造函数来初始化。列表集合也用 null 初始化(代码文件 LinkedListSample/PriorityDocumentManager.cs)：

```
public class PriorityDocumentManager
{
  private readonly LinkedList<Document> documentList;

  // priorities 0.9
  private readonly List<LinkedListNode<Document>> priorityNodes;

  public PriorityDocumentManager()
  {
    documentList = new LinkedList<Document>();

    priorityNodes = new List<LinkedListNode<Document>>(10);
    for (int i = 0; i < 10; i++)
    {
      priorityNodes.Add(new LinkedListNode<Document>(null));
    }
  }
```

在类的公共接口中，有一个 AddDocument()方法。AddDocument()方法只调用私有方法 AddDocumentToPriorityNode()。把实现代码放在另一个方法中的原因是，AddDocumentToPriorityNode()方法可以递归调用，如后面所示。

```
public void AddDocument(Document d)
{
  if (d == null) throw new ArgumentNullException("d");

  AddDocumentToPriorityNode(d, d.Priority);
}
```

在 AddDocumentToPriorityNode()方法的实现代码中，第一个操作是检查优先级是否在允许的优先级范围内。这里允许的范围是 0~9。如果传送了错误的值，就会抛出一个 ArgumentException 类型的异常。

接着检查是否已经有一个优先级节点与所传送的优先级相同。如果在列表集合中没有这样的优先级节点，就递归调用 AddDocumentToPriorityNode()方法，递减优先级值，检查是否有低一级的优先级节点。

如果优先级节点的优先级值与所传送的优先级值不同，也没有比该优先级值更低的优先级节点，就可以调用 AddLast()方法，将文档安全地添加到链表的末尾。另外，链表节点由负责指定文档优先级的优先级节点引用。

如果存在这样的优先级节点，就可以在链表中找到插入文档的位置。这里必须区分是存在指定优先级值的优先级节点，还是存在以较低的优先级值引用文档的优先级节点。对于第一种情况，可

以把新文档插入由优先级节点引用的位置后面。因为优先级节点总是引用指定优先级值的最后一个文档，所以必须设置优先级节点的引用。如果引用文档的优先级节点有较低的优先级值，情况就会比较复杂。这里新文档必须插入优先级值与优先级节点相同的所有文档的前面。为了找到优先级值相同的第一个文档，要通过一个 while 循环，使用 Previous 属性遍历所有的链表节点，直到找到一个优先级值不同的链表节点为止。这样，就找到了必须插入文档的位置，并可以设置优先级节点。

```
private void AddDocumentToPriorityNode(Document doc, int priority)
{
  if (priority > 9 || priority < 0)
    throw new ArgumentException("Priority must be between 0 and 9");

  if (priorityNodes[priority].Value == null)
  {
    --priority;
    if (priority >= 0)
    {
      // check for the next lower priority
      AddDocumentToPriorityNode(doc, priority);
    }
    else // now no priority node exists with the same priority or lower
         // add the new document to the end
    {
      documentList.AddLast(doc);
      priorityNodes[doc.Priority] = documentList.Last;
    }
    return;
  }
  else // a priority node exists
  {
    LinkedListNode<Document> prioNode = priorityNodes[priority];
    if (priority == doc.Priority)
       // priority node with the same priority exists
    {
      documentList.AddAfter(prioNode, doc);

      // set the priority node to the last document with the same priority
      priorityNodes[doc.Priority] = prioNode.Next;
    }
    else // only priority node with a lower priority exists
    {
      // get the first node of the lower priority
      LinkedListNode<Document> firstPrioNode = prioNode;

      while (firstPrioNode.Previous != null &&
          firstPrioNode.Previous.Value.Priority == prioNode.Value.Priority)
      {
        firstPrioNode = prioNode.Previous;
        prioNode = firstPrioNode;
      }

      documentList.AddBefore(firstPrioNode, doc);

      // set the priority node to the new value
```

```
        priorityNodes[doc.Priority] = firstPrioNode.Previous;
      }
    }
  }
```

现在还剩下几个简单的方法没有讨论。DisplayAllNodes()方法只是在一个 foreach 循环中，把每个文档的优先级和标题显示在控制台上。

GetDocument()方法从链表中返回第一个文档(优先级最高的文档)，并从链表中删除它：

```
    public void DisplayAllNodes()
    {
      foreach (Document doc in documentList)
      {
        Console.WriteLine("priority: {0}, title {1}", doc.Priority, doc.Title);
      }
    }

    // returns the document with the highest priority
    // (that's first in the linked list)
    public Document GetDocument()
    {
      Document doc = documentList.First.Value;
      documentList.RemoveFirst();
      return doc;
    }
  }
```

在 Main()方法中，PriorityDocumentManager 类用于说明其功能。在链表中添加 8 个优先级不同的新文档，再显示整个链表(代码文件 LinkedListSample/Program.cs)：

```
  static void Main()
  {
    var pdm = new PriorityDocumentManager();
    pdm.AddDocument(new Document("one", "Sample", 8));
    pdm.AddDocument(new Document("two", "Sample", 3));
    pdm.AddDocument(new Document("three", "Sample", 4));
    pdm.AddDocument(new Document("four", "Sample", 8));
    pdm.AddDocument(new Document("five", "Sample", 1));
    pdm.AddDocument(new Document("six", "Sample", 9));
    pdm.AddDocument(new Document("seven", "Sample", 1));
    pdm.AddDocument(new Document("eight", "Sample", 1));

    pdm.DisplayAllNodes();
  }
```

在处理好的结果中，文档先按优先级排序，再按添加文档的时间排序：

```
  priority: 9, title six
  priority: 8, title one
  priority: 8, title four
  priority: 4, title three
  priority: 3, title two
  priority: 1, title five
  priority: 1, title seven
```

```
priority: 1, title eight
```

10.7 有序列表

如果需要基于键对所需集合排序，就可以使用 SortedList<TKey, TValue>类。这个类按照键给元素排序。这个集合中的值和键都可以使用任意类型。

下面的例子创建了一个有序列表，其中键和值都是 string 类型。默认的构造函数创建了一个空列表，再用 Add()方法添加两本书。使用重载的构造函数，可以定义列表的容量，传递实现了 IComparer<TKey>接口的对象，该接口用于给列表中的元素排序。

Add()方法的第一个参数是键(书名)，第二个参数是值(ISBN 号)。除了使用 Add()方法之外，还可以使用索引器将元素添加到列表中。索引器需要把键作为索引参数。如果键已存在，Add()方法就抛出一个 ArgumentException 类型的异常。如果索引器使用相同的键，就用新值替代旧值(代码文件 SortedListSample/Program.cs)。

```
var books = new SortedList<string, string>();
books.Add("Professional WPF Programming", "978-0-470-04180-2");
books.Add("Professional ASP.NET MVC 3", "978-1-1180-7658-3");
books["Beginning Visual C# 2010"] = "978-0-470-50226-6";
books["Professional C# 4 and .NET 4"] = "978-0-470-50225-9";
```

SortedList<TKey, TValue>类只允许每个键有一个对应的值，如果需要每个键对应多个值，就可以使用 Lookup<TKey, TElement>类。

可以使用 foreach 语句遍历该列表。枚举器返回的元素是 KeyValuePair<TKey, TValue>类型，其中包含了键和值。键可以用 Key 属性访问，值可以用 Value 属性访问。

```
foreach (KeyValuePair<string, string> book in books)
{
  Console.WriteLine("{0}, {1}", book.Key, book.Value);
}
```

迭代语句会按键的顺序显示书名和 ISBN 号：

```
Beginning Visual C# 2010, 978-0-470-50226-6
Professional ASP.NET MVC 3, 978-1-1180-7658-3
Professional C# 4 and .NET 4, 978-0-470-50225-9
Professional WPF Programming, 978-0-470-04180-2
```

也可以使用 Values 和 Keys 属性访问值和键。因为 Values 属性返回 IList<TValue>，Keys 属性返回 IList<TKey>，所以可以通过 foreach 语句使用这些属性：

```
foreach (string isbn in books.Values)
{
  Console.WriteLine(isbn);
}

foreach (string title in books.Keys)
```

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

第一个循环显示值，第二个循环显示键：

```
978-0-470-50226-6
978-1-1180-7658-3
978-0-470-50225-9
978-0-470-04180-2
Beginning Visual C# 2010
Professional ASP.NET MVC 3
Professional C# 4 and .NET 4
Professional WPF Programming
```

如果尝试使用索引器访问一个元素，但所传递的键不存在，就会抛出一个KeyNotFoundException类型的异常。为了避免这个异常，可以使用ContainsKey()方法，如果所传递的键存在于集合中，这个方法就返回true，也可以调用TryGetValue()方法，该方法尝试获得指定键的值。如果指定键对应的值不存在，该方法就不会抛出异常。

```
string isbn;
string title = "Professional C# 7.0";
if (!books.TryGetValue(title, out isbn))
{
  Console.WriteLine("{0} not found", title);
}
```

10.8 字典

字典表示一种非常复杂的数据结构，这种数据结构允许按照某个键来访问元素。字典也称为映射或散列表。字典的主要特性是能根据键快速查找值。也可以自由添加和删除元素，这有点像List<T>类，但没有在内存中移动后续元素的性能开销。

图10-5是字典的一个简化表示。其中employee-id(如B4711)是添加到字典中的键。键会转换为一个散列。利用散列创建一个数字，它将索引和值关联起来。然后索引包含一个到值的链接。该图做了简化处理，因为一个索引项可以关联多个值，索引可以存储为一个树型结构。

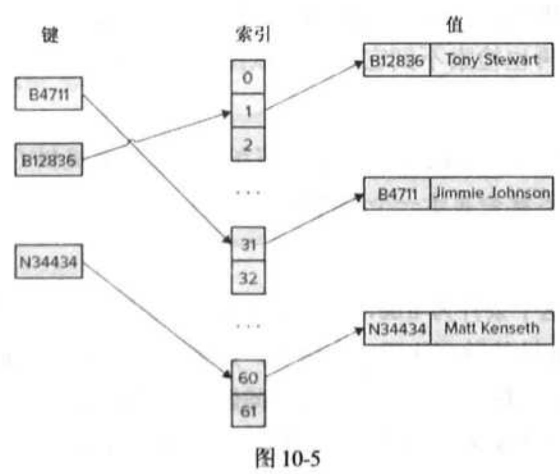


图10-5

.NET Framework 提供了几个字典类。可以使用的最主要的类是 Dictionary<TKey, TValue>。

10.8.1　键的类型

用作字典中键的类型必须重写 Object 类的 GetHashCode()方法。只要字典类需要确定元素的位置，它就要调用 GetHashCode()方法。GetHashCode()方法返回的 int 由字典用于计算在对应位置放置元素的索引。这里不介绍这个算法。我们只需要知道，它涉及素数，所以字典的容量是一个素数。

GetHashCode()方法的实现代码必须满足如下要求：

- 相同的对象应总是返回相同的值。
- 不同的对象可以返回相同的值。
- 它应执行得比较快，计算的开销不大。
- 它不能抛出异常。
- 它应至少使用一个实例字段。
- 散列代码值应平均分布在 int 可以存储的整个数字范围上。
- 散列代码最好在对象的生存期中不发生变化。

字典的性能取决于 GetHashCode()方法的实现代码。

为什么要使散列代码值平均分布在整数的取值范围内？如果两个键返回的散列代码值会得到相同的索引，字典类就必须寻找最近的可用空闲位置来存储第二个数据项，这需要进行一定的搜索，以便以后检索这一项。显然这会降低性能，如果在排序的时候许多键都有相同的索引，这类冲突就更可能出现。根据 Microsoft 的算法的工作方式，当计算出来的散列代码值平均分布在 int.MinValue 和 int.MaxValue 之间时，这种风险会降低到最小。

除了实现 GetHashCode()方法之外，键类型还必须实现 IEquatable<T>.Equals()方法，或重写 Object 类的 Equals()方法。因为不同的键对象可能返回相同的散列代码，所以字典使用 Equals()方法来比较键。字典检查两个键 A 和 B 是否相等，并调用 A.Equals(B)方法。这表示必须确保下述条件总是成立：

如果 A.Equals(B)方法返回 true，则 A.GetHashCode()和 B.GetHashCode()方法必须总是返回相同的散列代码。

这似乎有点奇怪，但它非常重要。如果设计出某种重写这些方法的方式，使上面的条件并不总是成立，把这个类的实例用作键的字典就不能正常工作，而是会发生有趣的事情。例如，把一个对象放在字典中后，就再也检索不到它，或者试图检索某项，却返回了错误的项。

因此，如果为 Equals()方法提供了重写版本，但没有提供 GetHashCode()方法的重写版本，C#编译器就会显示一个编译警告。

对于 System.Object，这个条件为 true，因为 Equals()方法只是比较引用，GetHashCode()方法实际上返回一个仅基于对象地址的散列代码。这说明，如果散列表基于一个键，而该键没有重写这些方法，这个散列表就能正常工作。但是，这么做的问题是，只有对象完全相同，键才被认为是相等的。也就是说，把一个对象放在字典中时，必须将它与该键的引用关联起来。也不能在以后用相同

的值实例化另一个键对象。如果没有重写Equals()方法和GetHashCode()方法，在字典中使用类型时就不太方便。

另外，System.String 实现了 IEquatable 接口，并重载了 GetHashCode()方法。Equals()方法提供了值的比较，GetHashCode()方法根据字符串的值返回一个散列代码。因此，在字典中把字符串用作键非常方便。

数字类型(如 Int32)也实现 IEquatable 接口，并重载 GetHashCode()方法。但是这些类型返回的散列代码只映射到值上。如果希望用作键的数字本身没有分布在可能的整数值范围内，把整数用作键就不能满足键值的平均分布规则，于是不能获得最佳的性能。Int32 并不适合在字典中使用。

如果需要使用的键类型没有实现 IEquatable 接口，并根据存储在字典中的键值重载 GetHashCode()方法，就可以创建一个实现 IEqualityComparer<T>接口的比较器。IEqualityComparer<T>接口定义了 GetHashCode()和 Equals()方法，并将传递的对象作为参数，因此可以提供与对象类型不同的实现方式。Dictionary<TKey, TValue>构造函数的一个重载版本允许传递一个实现了 IEqualityComparer <T>接口的对象。如果把这个对象赋予字典，该类就用于生成散列代码并比较键。

10.8.2 字典示例

字典示例程序建立了一个员工字典。该字典用 EmployeeId 对象来索引，存储在字典中的每个数据项都是一个 Employee 对象，该对象存储员工的详细数据。

实现 EmployeeId 结构是为了定义在字典中使用的键，该结构的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。在构造函数中填充字段。重载 ToString()方法是为了获得员工 ID 的字符串表示。与键类型的要求一样，EmployeeId 结构也要实现 IEquatable 接口，并重载 GetHashCode()方法(代码文件 DictionarySample/EmployeeId.cs)。

```
[Serializable]
public class EmployeeIdException : Exception
{
  public EmployeeIdException(string message) : base(message) { }
}

[Serializable]
public struct EmployeeId : IEquatable<EmployeeId>
{
  private readonly char prefix;
  private readonly int number;

  public EmployeeId(string id)
  {
    Contract.Requires<ArgumentNullException>(id != null);

    prefix = (id.ToUpper())[0];
    int numLength = id.Length - 1;
    try
    {
      number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
    }
    catch (FormatException)
```

```
    {
      throw new EmployeeIdException("Invalid EmployeeId format");
    }
  }

  public override string ToString()
  {
    return prefix.ToString() + string.Format("{0,6:000000}", number);
  }

  public override int GetHashCode()
  {
    return (number ^ number << 16) * 0x15051505;
  }

  public bool Equals(EmployeeId other)
  {
    if (other == null) return false;

    return (prefix == other.prefix && number == other.number);
  }

  public override bool Equals(object obj)
  {
    return Equals((EmployeeId)obj);
  }

  public static bool operator ==(EmployeeId left, EmployeeId right)
  {
    return left.Equals(right);
  }

  public static bool operator !=(EmployeeId left, EmployeeId right)
  {
    return !(left == right);
  }
}
```

由IEquatable<T>接口定义的Equals()方法比较两个EmployeeId对象的值，如果这两个值相同，它就返回true。除了实现IEquatable<T>接口中的Equals()方法之外，还可以重写Object类中的Equals()方法。

```
public bool Equals(EmployeeId other)
{
  if (other == null) return false;
  return (prefix == other.prefix && number == other.number);
}
```

由于数字是可变的，因此员工可以取1～190 000的一个值。这并没有填满整数取值范围。GetHashCode()方法使用的算法将数字向左移动16位，再与原来的数字进行异或操作，最后将结果乘以十六进制数15051505。散列代码在整数取值区域上的分布相当均匀：

```
public override int GetHashCode()
{
```

```
    return (number ^ number << 16) * 0x15051505;
  }
```

在 Internet 上，有许多更复杂的算法，它们能使散列代码在整数取值范围上更好地分布。也可以使用字符串的 GetHashCode()方法来返回一个散列。

Employee 类是一个简单的实体类，该实体类包含员工的姓名、薪水和 ID。构造函数初始化所有值，ToString()方法返回一个实例的字符串表示。ToString()方法的实现代码使用格式化字符串创建字符串表示，以提高性能(代码文件 DictionarySample/Employee.cs)。

```
[Serializable]
public class Employee
{
  private string name;
  private decimal salary;
  private readonly EmployeeId id;

  public Employee(EmployeeId id, string name, decimal salary)
  {
    this.id = id;
    this.name = name;
    this.salary = salary;
  }

  public override string ToString()
  {
    return String.Format("{0}: {1, -20} {2:C}",
      id.ToString(), name, salary);
  }
}
```

在示例程序的 Main()方法中，创建一个新的 Dictionary<TKey, TValue>实例，其中键是 EmployeeId 类型，值是 Employee 类型。构造函数指定了 31 个元素的容量。注意容量一般是素数。但如果指定了一个不是素数的值，也不需要担心。Dictionary<TKey, TValue>类会使用传递给构造函数的整数后面紧接着的一个素数，来指定容量。用 Add()方法创建员工对象和 ID，并添加到字典中。除了 Add()方法外，还可以使用索引器，将键和值添加到字典中，如员工 Matt 和 Brad 所示(代码文件 DictionarySample/Program.cs):

```
static void Main()
{
  var employees = new Dictionary<EmployeeId, Employee>(31);
  var idTony = new EmployeeId("C3755");
  var tony = new Employee(idTony, "Tony Stewart", 379025.00m);
  employees.Add(idTony, tony);
  Console.WriteLine(tony);
  var idCarl = new EmployeeId("F3547");
  var carl = new Employee(idCarl, "Carl Edwards", 403466.00m);
  employees.Add(idCarl, carl);
  Console.WriteLine(carl);
  var idKevin = new EmployeeId("C3386");
  var kevin = new Employee(idKevin, "Kevin Harwick", 415261.00m);
```

```
employees.Add(idKevin, kevin);
Console.WriteLine(kevin);
var idMatt = new EmployeeId("F3323");
var matt = new Employee(idMatt, "Matt Kenseth", 1589390.00m);
employees[idMatt] = matt;
Console.WriteLine(matt);
var idBrad = new EmployeeId("D3234");
var brad = new Employee(idBrad, "Brad Keselowski", 322295.00m);
employees[idBrad] = brad;
Console.WriteLine(brad);
```

将数据项添加到字典中后，在 while 循环中读取字典中的员工。让用户输入一个员工号，把该号码存储在变量 userInput 中。用户输入 X 即可退出应用程序。如果输入的键在字典中，就使用 Dictionary<TKey, TValue>类的 TryGetValue()方法检查它。如果找到了该键，TryGetValue()方法就返回 true；否则返回 false。如果找到了与键关联的值，该值就存储在 employee 变量中，并把该值写入控制台。

也可以使用 Dictionary<TKey, TValue>类的索引器替代 TryGetValue()方法，来访问存储在字典中的值。但是，如果没有找到键，索引器会抛出一个 KeyNotFoundException 类型的异常。

```
while (true)
{
  Console.Write("Enter employee id (X to exit)> ");
  var userInput = Console.ReadLine();
  userInput = userInput.ToUpper();
  if (userInput == "X") break;

  EmployeeId id;
  try
  {
    id = new EmployeeId(userInput);


    Employee employee;
    if (!employees.TryGetValue(id, out employee))
    {
      Console.WriteLine("Employee with id {0} does not exist",
          id);
    }
    else
    {
      Console.WriteLine(employee);
    }
  }
  catch (EmployeeIdException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

```
    }
}
```

运行应用程序，得到如下输出：

```
Enter employee id (X to exit)> C3386
C003386: Kevin Harwick       $415,261.00
Enter employee id (X to exit)> F3547
F003547: Carl Edwards        $403,466.00
Enter employee id (X to exit)> X
Press any key to continue ...
```

10.8.3 Lookup 类

Dictionary<TKey, TValue>类支持每个键关联一个值。Lookup<TKey, TElement>类非常类似于Dictionary<TKey, TValue>类，但把键映射到一个值集上。这个类在程序集 System.Core 中实现，用System.Linq 名称空间定义。

Lookup<TKey, TElement>类不能像一般的字典那样创建，而必须调用 ToLookup()方法，该方法返回一个 Lookup<TKey, TElement>对象。ToLookup()方法是一个扩展方法，它可以用于实现了IEnumerable<T>接口的所有类。在下面的例子中，填充了一个 Racer 对象列表。因为 List<T>类实现了 IEnumerable<T>接口，所以可以在赛车手列表上调用 ToLookup()方法。这个方法需要一个Func<TSource, TKey>类型的委托，Func<TSource, TKey>类型定义了键的选择器。这里使用 lambda 表达式 r => r.Country，根据国家来选择赛车手。foreach 循环只使用索引器访问来自澳大利亚的赛车手(代码文件 LookupSample/Program.cs)。

```
var racers = new List<Racer>();
racers.Add(new Racer("Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer("Alan", "Jones", "Australia", 12));
racers.Add(new Racer("Jackie", "Stewart", "United Kingdom", 27));
racers.Add(new Racer("James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham", "Australia", 14));

var lookupRacers = racers.ToLookup(r => r.Country);

foreach (Racer r in lookupRacers["Australia"])
{
   Console.WriteLine(r);
}
```

扩展方法详见第 11 章，lambda 表达式参见第 8 章。

结果显示了来自澳大利亚的赛车手：

```
Alan Jones
Jack Brabham
```

10.8.4 有序字典

SortedDictionary<TKey, TValue>类是一个二叉搜索树，其中的元素根据键来排序。该键类型必

须实现IComparable<TKey>接口。如果键的类型不能排序，则还可以创建一个实现了IComparer <TKey>接口的比较器，将比较器用作有序字典的构造函数的一个参数。

SortedDictionary<TKey, TValue>类和SortedList<TKey, TValue>类的功能类似。但因为SortedList <TKey, TValue>实现为一个基于数组的列表，而SortedDictionary<TKey, TValue>类实现为一个字典，所以它们有不同的特征。

- SortedList<TKey, TValue>类使用的内存比SortedDictionary<TKey, TValue>类少。
- SortedDictionary<TKey, TValue>类的元素插入和删除操作比较快。
- 在用已排好序的数据填充集合时，若不需要修改容量，SortedList<TKey, TValue>类就比较快。

SortedList类使用的内存比SortedDictionary类少，但SortedDictionary类在插入和删除未排序的数据时比较快。

10.9 集

包含不重复元素的集合称为“集(set)”。.NET Framework 包含两个集(HashSet<T>和SortedSet<T>)，它们都实现ISet<T>接口。HashSet<T>集包含不重复元素的无序列表，SortedSet<T>集包含不重复元素的有序列表。

ISet<T>接口提供的方法可以创建合集、交集，或者给出一个集是另一个集的超集或子集的信息。

在示例代码中，创建了3个字符串类型的新集，并用一级方程式汽车填充它们。HashSet<T>集实现ICollection<T>接口。但是在该类中，Add()方法是显式实现的，还提供了另一个Add()方法。Add()方法的区别是返回类型，它返回一个布尔值，说明是否添加了元素。如果该元素已经在集中，就不添加它，并返回false(代码文件SetSample/Program.cs)。

```
var companyTeams = new HashSet<string>()
{ "Ferrari", "McLaren", "Mercedes" };
var traditionalTeams = new HashSet<string>() { "Ferrari", "McLaren" };
var privateTeams = new HashSet<string>()
{ "Red Bull", "Toro Rosso", "Force India", "Sauber" };

if (privateTeams.Add("Williams"))
   Console.WriteLine("Williams added");
if (!companyTeams.Add("McLaren"))
   Console.WriteLine("McLaren was already in this set");
```

两个Add()方法的输出写到控制台上：

```
Williams added
McLaren was already in this set
```

IsSubsetOf()和IsSupersetOf()方法比较集和实现了IEnumerable<T>接口的集合，并返回一个布尔结果。这里，IsSubsetOf()方法验证traditionalTeams集合中的每个元素是否都包含在companyTeams集合方法中，IsSupersetOf()方法验证traditionalTeams集合是否有companyTeams集合没有的额外元素。

```
if (traditionalTeams.IsSubsetOf(companyTeams))
```

```
{
  Console.WriteLine("traditionalTeams is subset of companyTeams");
}

if (companyTeams.IsSupersetOf(traditionalTeams))
{
  Console.WriteLine("companyTeams is a superset of traditionalTeams");
}
```

这个验证的结果如下：

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
```

Williams 也是一个传统队，因此这个队添加到 traditionalTeams 集合中：

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
  Console.WriteLine("At least one team is the same with the " +
      "traditional and private teams");
}
```

因为有一个重叠，所以结果如下：

```
At least one team is the same with the traditional and private teams.
```

调用 UnionWith()方法，把引用新 SortedSet<string>的变量 allTeams 填充为 companyTeams、privateTeams 和 traditionalTeams 的合集：

```
var allTeams = new SortedSet<string>(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);

Console.WriteLine();
Console.WriteLine("all teams");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

这里返回所有队，但每个队都只列出一次，因为集只包含唯一值。因为容器是 SortedSet<string>，所以结果是有序的：

```
Ferrari
Force India
Lotus
McLaren
Mercedes
Red Bull
Sauber
Toro Rosso
Williams
```

ExceptWith()方法从 allTeams 集中删除所有私有队：

```
allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("no private team left");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

集合中的其他元素不包含私有队：

```
Ferrari
McLaren
Mercedes
```

10.10 可观察的集合

如果需要集合中的元素何时删除或添加的信息，就可以使用 ObservableCollection<T>类。这个类是为 WPF 定义的，这样 UI 就可以得知集合的变化，因此这个类在程序集 WindowsBase 中定义，用户需要引用这个程序集。这个类的名称空间是 System.Collections. ObjectModel。

ObservableCollection<T>类派生自 Collection<T>基类，该基类可用于创建自定义集合，并在内部使用 List<T>类。重写基类中的虚方法 SetItem()和 RemoveItem()，以触发 CollectionChanged 事件。这个类的用户可以使用 INotifyCollectionChanged 接口注册这个事件。

下面的示例说明了 ObservableCollection<string>()方法的用法，其中给 CollectionChanged 事件注册了 Data_CollectionChanged()方法。把两项添加到末尾，再插入一项，并删除一项(代码文件 ObservableCollectionSample/Program.cs)：

```
var data = new ObservableCollection<string>();
data.CollectionChanged += Data_CollectionChanged;
data.Add("One");
data.Add("Two");
data.Insert(1, "Three");
data.Remove("One");
```

Data_CollectionChanged()方法接收 NotifyCollectionChangedEventArgs，其中包含了集合的变化信息。Action 属性给出了是否添加或删除一项的信息。对于删除的项，会设置 OldItems 属性，列出删除的项。对于添加的项，则设置 NewItems 属性，列出新增的项。

```
static void Data_CollectionChanged(object sender,
                                   NotifyCollectionChangedEventArgs e)
{
  Console.WriteLine("action: {0}", e.Action.ToString());

  if (e.OldItems != null)
  {
    Console.WriteLine("starting index for old item(s): {0}",
                      e.OldStartingIndex);
    Console.WriteLine("old item(s):");
    foreach (var item in e.OldItems)
    {
```

```
        Console.WriteLine(item);
      }
    }
    if (e.NewItems != null)
    {
      Console.WriteLine("starting index for new item(s): {0}",
                        e.NewStartingIndex);
      Console.WriteLine("new item(s): ");
      foreach (var item in e.NewItems)
      {
        Console.WriteLine(item);
      }
    }
    Console.WriteLine();
  }
```

运行应用程序，输出如下所示。先在集合中添加One和Two项，因此显示的Add动作的索引是0和1。第3项Three插入在位置1上，所以显示的Add动作的索引是1。最后删除One项，显示的Remove动作的索引是0：

```
action: Add
starting index for new item(s): 0
new item(s):
One

action: Add
starting index for new item(s): 1
new item(s):
Two

action: Add
starting index for new item(s): 1
new item(s):
Three

action: Remove
starting index for old item(s): 0
old item(s):
One
```

10.11 位数组

如果需要处理的数字有许多位，就可以使用BitArray类和BitVector32结构。BitArray类位于名称空间System.Collections中，BitVector32结构位于名称空间System.Collections.Specialized中。这两种类型最重要的区别是，BitArray类可以重新设置大小，如果事先不知道需要的位数，就可以使用BitArray类，它可以包含非常多的位。BitVector32结构是基于栈的，因此比较快。BitVector32结构仅包含32位，它们存储在一个整数中。

10.11.1 BitArray 类

BitArray 类是一个引用类型，它包含一个 int 数组，其中每 32 位使用一个新整数。这个类的成员如表 10-4 所示。

表 10-4

BitArray 类的成员	说 明
Count Length	Count 和 Length 属性的 get 访问器返回数组中的位数。使用 Length 属性还可以定义新的数组大小，重新设置集合的大小
Item Get Set	可以使用索引器读写数组中的位。索引器是布尔类型。除了使用索引器之外，还可以使用 Get()和 Set()方法访问数组中的位
SetAll	根据传送给该方法的参数，SetAll()方法设置所有位的值
Not	Not()方法对数组中所有位的值取反
And Or Xor	使用 And()、Or()和 Xor()方法，可以合并两个 BitArray 对象。And()方法执行二元 AND，只有两个输入数组的位都设置为 1，结果位才是 1。Or()方法执行二元 OR，只要有一个输入数组的位设置为 1，结果位就是 1。Xor()方法是异或操作，只有一个输入数组的位设置为 1，结果位才是 1

辅助方法 DisplayBits()遍历 BitArray，根据位的设置情况，在控制台上显示 1 或 0(代码文件 BitArraySample/Program.cs)：

```
static void DisplayBits(BitArray bits)
{
  foreach (bool bit in bits)
  {
     Console.Write(bit ? 1: 0);
  }
}
```

说明 BitArray 类的示例创建了一个包含 8 位的数组，其索引是 0~7。SetAll()方法把这 8 位都设置为 true。接着 Set()方法把对应于 1 的位设置为 false。除了 Set()方法之外，还可以使用索引器，例如下面的第 5 个和第 7 个索引：

```
var bits1 = new BitArray(8);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.Write("initialized: ");
DisplayBits(bits1);
Console.WriteLine();
```

这是初始化位的显示结果：

```
initialized: 10111010
```

Not()方法会对 BitArray 类的位取反：

```
Console.Write(" not ");
DisplayBits(bits1);
bits1.Not();
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

Not()方法的结果是对所有的位取反。如果某位是 true，执行 Not()方法的结果就是 false，反之亦然。

```
not 10111010 = 01000101
```

这里创建了一个新的 BitArray 类。在构造函数中，因为使用变量 bits1 初始化数组，所以新数组与旧数组有相同的值。接着把第 0、1 和 4 位的值设置为不同的值。在使用 Or()方法之前，显示位数组 bits1 和 bits2。Or()方法将改变 bits1 的值：

```
var bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
DisplayBits(bits1);
Console.Write(" or ");
DisplayBits(bits2);
Console.Write(" = ");
bits1.Or(bits2);
DisplayBits(bits1);
Console.WriteLine();
```

使用 Or()方法时，从两个输入数组中提取设置位。结果是，如果某位在第一个或第二个数组中设置为 true，该位在执行 Or()方法后就是 true：

```
01000101 or 10001101 = 11001101
```

下面使用 And()方法作用于位数组 bits1 和 bits2：

```
DisplayBits(bits2);
Console.Write(" and ");
DisplayBits(bits1);
Console.Write(" = ");
bits2.And(bits1);
DisplayBits(bits2);
Console.WriteLine();
```

And()方法只把在两个输入数组中都设置为 true 的位设置为 true：

```
10001101 and 11001101 = 10001101
```

最后使用 Xor()方法进行异或操作：

```
DisplayBits(bits1);
Console.Write(" xor ");
DisplayBits(bits2);
bits1.Xor(bits2);
Console.Write(" = ");
```

```
DisplayBits(bits1);
Console.WriteLine();
```

使用 Xor()方法，只有一个(不能是两个)输入数组的位设置为 1，结果位才是 1。

```
11001101 xor 10001101 = 01000000
```

10.11.2 BitVector32 结构

如果事先知道需要的位数，就可以使用 BitVector32 结构替代 BitArray 类。BitVector32 结构效率较高，因为它是一个值类型，在整数栈上存储位。一个整数可以存储 32 位。如果需要更多的位，就可以使用多个 BitVector32 值或 BitArray 类。BitArray 类可以根据需要增大，但 BitVector32 结构不能。

表 10-5 列出了 BitVector32 结构中与 BitArray 类完全不同的成员。

表 10-5

BitVector32 结构的成员	说 明
Data	Data 属性把 BitVector32 结构中的数据返回为整数
Item	BitVector32 的值可以使用索引器设置。索引器是重载的——可以使用掩码或 BitVector32. Section 类型的片段来获取和设置值
CreateMask	这是一个静态方法，用于为访问 BitVector32 结构中的特定位创建掩码
CreateSection	这是一个静态方法，用于创建 32 位中的几个片段

示例代码用默认构造函数创建了一个 BitVector32 结构，其中所有的 32 位都初始化为 false。接着创建掩码，以访问位矢量中的位。对 CreateMask()方法的第一个调用创建了用来访问第一位的一个掩码。调用 CreateMask()方法后，bit1 被设置为 1。再次调用 CreateMask()方法，把第一个掩码作为参数传递给 CreateMask()方法，返回用来访问第二位(它是 2)的一个掩码。接着，将 bit3 设置为 4，以访问位编号 3。bit4 的值是 8，以访问位编号 4。

然后，使用掩码和索引器访问位矢量中的位，并相应地设置字段(代码文件 BitArraySample/Program.cs)：

```
var bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);

bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);
```

BitVector32 结构有一个重写的 ToString()方法，它不仅显示类名，还显示 1 或 0，来说明位是否设置了，如下所示：

```
BitVector32{00000000000000000000000000011101}
```

除了用 CreateMask()方法创建掩码之外，还可以自己定义掩码，也可以一次设置多位。十六进制值 abcdef 与二进制值 1010 1011 1100 1101 1110 1111 相同。用这个值定义的所有位都设置了：

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

在输出中可以验证设置的位：

```
BitVector32{00000000101010111100110111101111}
```

把 32 位分别放在不同的片段中非常有用。例如，IPv4 地址定义为一个 4 字节的数，该数存储在一个整数中。可以定义 4 个片段，把这个整数拆分开。在多播 IP 消息中，使用了几个 32 位的值。其中一个 32 位的值放在这些片段中：16 位表示源号，8 位表示查询器的查询内部码，3 位表示查询器的健壮变量，1 位表示抑制标志，还有 4 个保留位。也可以定义自己的位含义，以节省内存。

下面的例子模拟接收到值 0x79abcdef，把这个值传送给 BitVector32 结构的构造函数，从而相应地设置位：

```
int received = 0x79abcdef;

BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

在控制台上显示了初始化的位：

```
BitVector32{01111001101010111100110111101111}
```

接着创建 6 个片段。第一个片段需要 12 位，由十六进制值 0xfff 定义(设置了 12 位)。片段 B 需要 8 位，片段 C 需要 4 位，片段 D 和 E 需要 3 位，片段 F 需要两位。第一次调用 CreateSection()方法只是接收 0xfff，为最前面的 12 位分配内存。第二次调用 CreateSection()方法时，将第一个片段作为参数传递，从而使下一个片段从第一个片段的结尾处开始。CreateSection()方法返回一个 BitVector32. Section 类型的值，该类型包含了该片段的偏移量和掩码。

```
// sections: FF EEE DDD CCCC BBBBBBBB
// AAAAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xfff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

把一个 BitVector32.Section 类型的值传递给 BitVector32 结构的索引器，会返回一个 int，它映射到位矢量的片段上。这里使用一个帮助方法 IntToBinaryString()，获得该 int 数的字符串表示：

```
Console.WriteLine("Section A: {0}",
                  IntToBinaryString(bits2[sectionA], true));
Console.WriteLine("Section B: {0}",
                  IntToBinaryString(bits2[sectionB], true));
Console.WriteLine("Section C: {0}",
                  IntToBinaryString(bits2[sectionC], true));
Console.WriteLine("Section D: {0}",
                  IntToBinaryString(bits2[sectionD], true));
Console.WriteLine("Section E: {0}",
```

```
                        IntToBinaryString(bits2[sectionE], true));
    Console.WriteLine("Section F: {0}",
                        IntToBinaryString(bits2[sectionF], true));
```

IntToBinaryString()方法接收整数中的位，并返回一个包含 0 和 1 的字符串表示。在实现代码中遍历整数的 32 位。在迭代过程中，如果该位设置为 1，就在 StringBuilder 的后面追加 1，否则，就追加 0。在循环中，移动一位，以检查是否设置了下一位。

```
static string IntToBinaryString(int bits, bool removeTrailingZero)
{
  var sb = new StringBuilder(32);

  for (int i = 0; i < 32; i++)
  {
    if ((bits & 0x80000000) != 0)
    {
      sb.Append("1");
    }
    else
    {
      sb.Append("0");
    }
    bits = bits << 1;
  }
  string s = sb.ToString();
  if (removeTrailingZero)
  {
    return s.TrimStart('0');
  }
  else
  {
    return s;
  }
}
```

结果显示了片段 A～F 的位表示，现在可以用传递给位矢量的值来验证：

```
Section A: 110111101111
Section B: 10111100
Section C: 1010
Section D: 1
Section E: 111
Section F: 1
```

10.12　不变的集合

如果对象可以改变其状态，就很难在多个同时运行的任务中使用。这些集合必须同步。如果对象不能改变其状态，就很容易在多个线程中使用。不能改变的对象称为不变的对象。

在 Visual Studio 2013 中，Microsoft 提供了一个新的集合库：Microsoft Immutable Collections。顾名思义，它包含不变的集合类——创建后不能改变的集合类。这个库可用作 NuGet 包，在名称空间 System.Collections.Immutable 中包含新的集合类。这个库可以与.NET 4.5 和.NET 4.5.1 项目一起使用。

下面是一个简单的不变字符串数组。可以用静态的 Create 方法创建该数组，如下所示。Create 方法被重载，这个方法的其他变体允许传送任意数量的元素。在下面的代码中(代码文件 ImmutableCollectionsSample/Program.cs)，创建了一个空数组：

```
ImmutableArray<string> a1 = ImmutableArray.Create<string>();
```

空数组没有什么用。ImmutableArray<T>类型提供了添加元素的 Add 方法。但是，与其他集合类相反，Add 方法不会改变不变集合本身，而是返回一个新的不变集合。在调用 Add 方法之后，a1 仍是一个空集合，a2 是包含一个元素的不变集合。Add 方法返回新的不变集合：

```
ImmutableArray<string> a2 = a1.Add("Williams");
```

之后，就可以以流畅的方式使用这个 API，一个接一个地调用 Add 方法。变量 a3 现在引用一个不变集合，它包含 4 个元素：

```
ImmutableArray<string> a3 =
  a2.Add("Ferrari").Add("Mercedes").Add("Red Bull Racing");
```

在使用不变数组的每个阶段，都没有复制完整的集合。相反，不变类型使用了共享状态，仅在需要时复制集合。

但是，先填充集合，再将它变成不变的数组会更高效。需要进行一些处理时，可以再次使用可变的集合。此时可以使用不变集合提供的构建器类。

为了说明其操作，先创建一个 Account 类，将此类放在集合中(代码文件 ImmutableCollectionsSample/Account.cs)：

```
public class Account
{
  public string Name { get; set; }
  public decimal Amount { get; set; }
}
```

接着创建 List<Account>集合，用示例账户填充(代码文件 ImmutableCollectionsSample/Program. cs)：

```
List<Account> accounts = new List<Account>() {
  new Account {
    Name = "Scrooge McDuck",
    Amount = 667377678765m
  },
  new Account {
    Name = "Donald Duck",
    Amount = -200m
  },
new Account {
  Name = "Ludwig von Drake",
  Amount = 20000m
}};
```

有了账户集合，可以使用 ToImmutableList 扩展方法创建一个不变的集合。只要打开名称空间 System.Collections.Immutable，就可以使用这个扩展方法：

```
ImmutableList<Account> immutableAccounts = accounts.ToImmutableList();
```

变量 immutableAccounts 可以像其他集合那样枚举，它只是不能改变。

现在，要再次进行修改，可以调用 ToBuilder 方法创建一个构建器。这个方法返回一个可变的

集合。在示例代码中，删除了账号大于 0 的所有账户。最初的不变集合没有改变。用构建器进行的改变完成后，调用 Builder 的 ToImmutable 方法创建一个新的不变集合。这个集合用于输出所有透支的账户：

```
ImmutableList<Account>.Builder builder = immutableAccounts.ToBuilder();
for (int i = 0; i < builder.Count; i++)
{
  Account a = builder[i];
  if (a.Amount > 0)
  {
    builder.Remove(a);
  }
}

ImmutableList<Account> overdrawnAccounts = builder.ToImmutable();

foreach (var item in overdrawnAccounts)
{
  Console.WriteLine("{0} {1}", item.Name, item.Amount);
}
```

只读集合在本章前面的 10.3.2 节中讨论。只读集合提供了集合的只读视图。在不使用只读视图访问集合的情况下，该集合仍可以修改。而没有人能改变不变的集合。

使用多个任务线程、异步方法的编程等主题详见第 13 和第 21 章。

10.13 并发集合

从.NET 4 开始，.NET 中在名称空间 System.Collections.Concurrent 中提供了几个线程安全的集合类。线程安全的集合可防止多个线程以相互冲突的方式访问集合。

为了对集合进行线程安全的访问，定义了 IProducerConsumerCollection<T>接口。这个接口中最重要的方法是 TryAdd()和 TryTake()。TryAdd()方法尝试给集合添加一项，但如果集合禁止添加项，这个操作就可能失败。为了给出相关信息，TryAdd()方法返回一个布尔值，以说明操作是成功还是失败。TryTake()方法也以这种方式工作，以通知调用者操作是成功还是失败，并在操作成功时返回集合中的项。下面列出了 System.Collections.Concurrent 名称空间中的类及其功能。

- ConcurrentQueue<T>——这个集合类用一种免锁定的算法实现，使用在内部合并到一个链表中的 32 项数组。访问队列元素的方法有 Enqueue()、TryDequeue()和 TryPeek()。这些方法的命名非常类似于前面 Queue<T>类的方法，只是给可能调用失败的方法加上了前缀 Try。因为这个类实现了 IProducerConsumerCollection<T>接口，所以 TryAdd()和 TryTake()方法仅调用 Enqueue()和 TryDequeue()方法。

- ConcurrentStack<T>——非常类似于 ConcurrentQueue<T>类，只是带有另外的元素访问方法。ConcurrentStack<T>类定义了 Push()、PushRange()、TryPeek()、TryPop()和 TryPopRange()方法。在内部这个类使用其元素的链表。
- ConcurrentBag<T>——该类没有定义添加或提取项的任何顺序。这个类使用一个把线程映射到内部使用的数组上的概念，因此尝试减少锁定。访问元素的方法有 Add()、TryPeek()和 TryTake()。
- ConcurrentDictionary<TKey, TValue>——这是一个线程安全的键值集合。TryAdd()、TryGetValue()、TryRemove()和 TryUpdate()方法以非阻塞的方式访问成员。因为元素基于键和值，所以 ConcurrentDictionary<TKey, TValue>没有实现 IProducerConsumerCollection<T>。
- BlockingCollection<T>——这个集合在可以添加或提取元素之前，会阻塞线程并一直等待。BlockingCollection<T>集合提供了一个接口，以使用 Add()和 Take()方法来添加和删除元素。这些方法会阻塞线程，一直等到任务可以执行为止。Add()方法有一个重载版本，其中可以给该重载版本传递一个 CancellationToken 令牌。这个令牌允许取消被阻塞的调用。如果不希望线程无限期地等待下去，且不希望从外部取消调用，就可以使用 TryAdd()和 TryTake()方法，在这些方法中，也可以指定一个超时值，它表示在调用失败之前应阻塞线程和等待的最长时间。

ConcurrentXXX 集合是线程安全的，如果某个动作不适用于线程的当前状态，它们就返回 false。在继续之前，总是需要确认添加或提取元素是否成功。不能相信集合会完成任务。

BlockingCollection<T>是对实现了 IProducerConsumerCollection<T>接口的任意类的修饰器，它默认使用 ConcurrentQueue<T>类。还可以给构造函数传递任何其他实现了 IProducerConsumerCollection<T>接口的类，例如 ConcurrentBag<T>和 ConcurrentStack<T>。

10.13.1 创建管道

将这些并发集合类用于管道是一种很好的应用。一个任务向一个集合类写入一些内容，同时另一个任务从该集合中读取内容。

下面的示例应用程序演示了 BlockingCollection<T>类的用法，使用多个任务形成了一个管道。第一个管道如图 10-6 所示。第一阶段的任务读取文件名，并把它们添加到队列中。在这个任务运行的同时，第二阶段的任务已经开始从队列中读取文件名并加载它们的内容。结果被写入了另一个队列。第三阶段可以同时启动，读取并处理第二个队列的内容。结果被写入一个字典。

在这个场景中，只有第三阶段完成，并且内容已被最终处理，在字典中得到了完整的结果时，下一个阶段才会开始。图 10-7 显示了接下来的步骤。第四阶段从字典中读取内容，转换数据，然后将其写入到队列中。第五阶段在项中添加了颜色信息，然后把它们添加到另一个队列中。最后一个阶段显示了信息。第四阶段到第六阶段也可以并发运行。

看看这个示例应用程序的代码可知，完整的管道是在 StartPipeline()方法中管理的。该方法实例化了集合，并把集合传递到管道的各个阶段。第一阶段用 ReadFilenamesAsync 处理，第二和第三阶段分别由同时运行的 LoadContentAsync 和 ProcessContentAsync 处理。但是，只有当前 3 个阶段完成后，第四个阶段才能启动(代码文件 PipelineSample/Program.cs)：

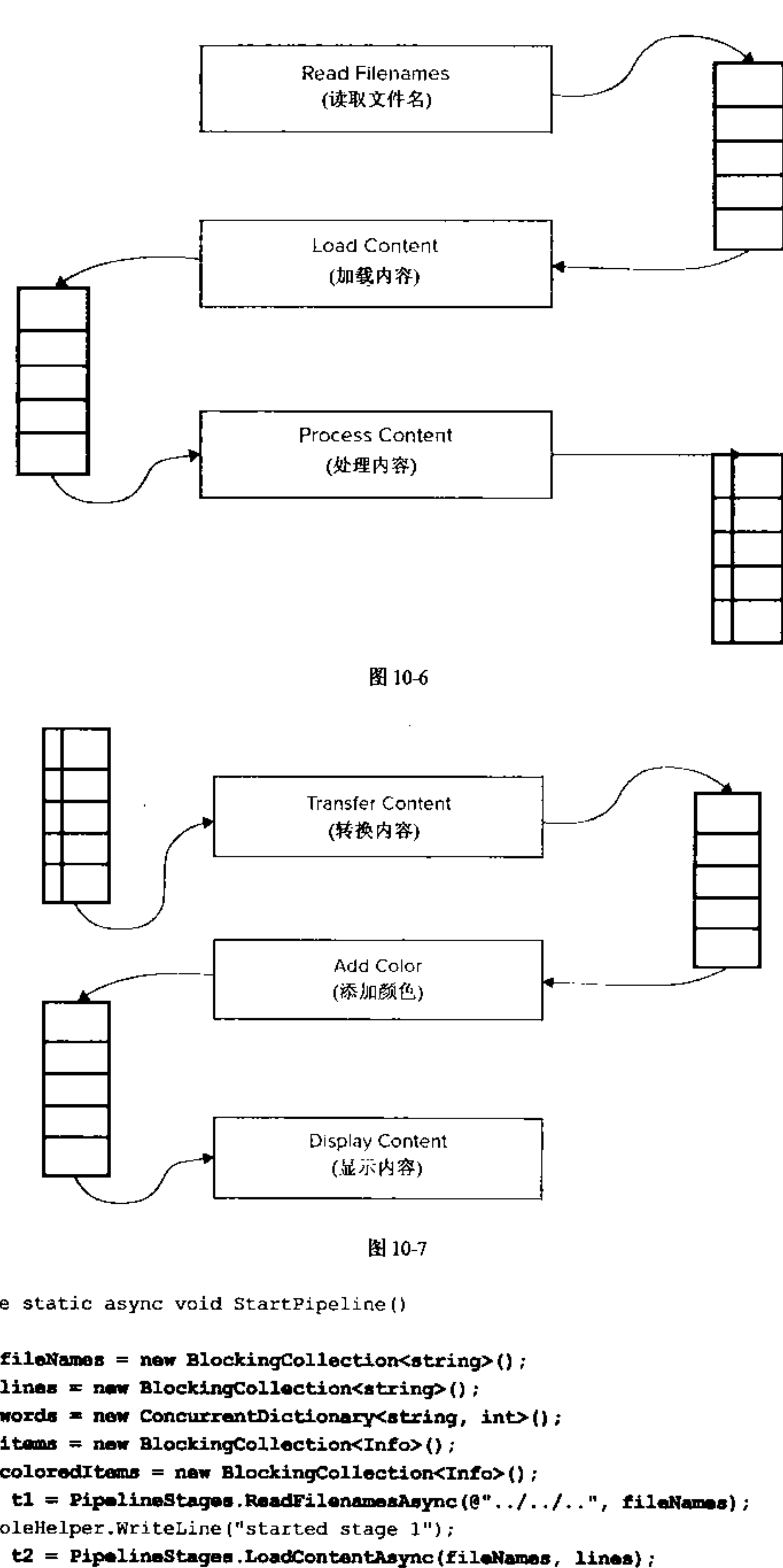


图 10-6

图 10-7

```
private static async void StartPipeline()
{
  var fileNames = new BlockingCollection<string>();
  var lines = new BlockingCollection<string>();
  var words = new ConcurrentDictionary<string, int>();
  var items = new BlockingCollection<Info>();
  var coloredItems = new BlockingCollection<Info>();
  Task t1 = PipelineStages.ReadFilenamesAsync(@"../../..", fileNames);
  ConsoleHelper.WriteLine("started stage 1");
  Task t2 = PipelineStages.LoadContentAsync(fileNames, lines);
```

```
    ConsoleHelper.WriteLine("started stage 2");
    Task t3 = PipelineStages.ProcessContentAsync(lines, words);
    await Task.WhenAll(t1, t2, t3);
    ConsoleHelper.WriteLine("stages 1, 2, 3 completed");
    Task t4 = PipelineStages.TransferContentAsync(words, items);
    Task t5 = PipelineStages.AddColorAsync(items, coloredItems);
    Task t6 = PipelineStages.ShowContentAsync(coloredItems);
    ConsoleHelper.WriteLine("stages 4, 5, 6 started");

    await Task.WhenAll(t4, t5, t6);
    ConsoleHelper.WriteLine("all stages finished");
}
```

这个示例应用程序使用了任务以及 async 和 await 关键字，第 13 章将介绍它们。第 21 章将详细介绍线程、任务和同步。第 24 章将讨论文件 I/O。

本例用 ConsoleHelper 类向控制台写入信息。该类可以方便地改变控制台输出的颜色，并使用了同步来避免返回颜色错误的输出(代码文件 PipelineSample/ConsoleHelper.cs)：

```
using System;

namespace Wrox.ProCSharp.Collections
{
  public class ConsoleHelper
  {
    private static object syncOutput = new object();
    public static void WriteLine(string message)
    {
      lock (syncOutput)
      {
        Console.WriteLine(message);
      }
    }
    public static void WriteLine(string message, string color)
    {
      lock (syncOutput)
      {
        Console.ForegroundColor = (ConsoleColor)Enum.Parse(
            typeof(ConsoleColor), color);
        Console.WriteLine(message);
        Console.ResetColor();
      }
    }
  }
}
```

10.13.2 使用 BlockingCollection

现在介绍管道的第一阶段。ReadFilenamesAsync 接收 BlockingCollection<T>为参数，在其中写

入输出。该方法的实现使用枚举器来迭代指定目录及其子目录中的 C#文件。这些文件的文件名用 Add 方法添加到 BlockingCollection<T>中。完成添加文件名的操作后，调用 CompleteAdding 方法，以通知所有读取器不应再等待集合中的任何额外项(代码文件 PipelineSample/PipelineStages.cs)：

```
using System.Collections.Concurrent;
using System.IO;
using System.Linq;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Collections
{
  public static class PipelineStages
  {
    public static Task ReadFilenamesAsync(string path,
        BlockingCollection<string> output)
    {
      return Task.Run(() =>
        {
          foreach (string filename in Directory.EnumerateFiles(path, "*.cs",
              SearchOption.AllDirectories))
          {
            output.Add(filename);
            ConsoleHelper.WriteLine(string.Format("stage 1: added {0}",
                filename));
          }
          output.CompleteAdding();
        });
    }
```

如果在写入器添加项的同时，读取器从 BlockingCollection<T>中读取，那么调用 CompleteAdding 方法是很重要的。否则，读取器会在 foreach 循环中等待更多的项被添加。

下一个阶段是读取文件并将其内容添加到另一个集合中，这由 LoadContentAsync 方法完成。该方法使用了输入集合传递的文件名，打开文件，然后把文件中的所有行添加到输出集合中。在 foreach 循环中，用输入阻塞集合调用 GetConsumingEnumerable，以迭代各项。直接使用 input 变量而不调用 GetConsumingEnumerable 是可以的，但是这只会迭代当前状态的集合，而不会迭代以后添加的项。

```
public static async Task LoadContentAsync(BlockingCollection<string> input,
    BlockingCollection<string> output)
{
  foreach (var filename in input.GetConsumingEnumerable())
  {
    using (FileStream stream = File.OpenRead(filename))
    {
      var reader = new StreamReader(stream);
      string line = null;
      while ((line = await reader.ReadLineAsync()) != null)
```

```
        {
          output.Add(line);
          ConsoleHelper.WriteLine(string.Format("stage 2: added {0}", line));
        }
      }
    }
    output.CompleteAdding();
}
```

如果在填充集合的同时，使用读取器读取集合，则需要使用 GetConsumingEnumerable 方法获取阻塞集合的枚举器，而不是直接迭代集合。

10.13.3 使用 ConcurrentDictionary

ProcessContentAsync 方法实现了第三阶段。这个方法获取输入集合中的行，然后拆分它们，将各个词筛选到输出字典中。AddOrIncrementValue 是一个辅助方法，它被实现为字典的一个扩展方法，如下所示：

```
public static Task ProcessContentAsync(BlockingCollection<string> input,
    ConcurrentDictionary<string, int> output)
{
  return Task.Run(() =>
    {
      foreach (var line in input.GetConsumingEnumerable())
      {
        string[] words = line.Split(' ', ';', '\t', '{', '}', '(', ')',
            ':', ',', '"');
        foreach (var word in words.Where(w => !string.IsNullOrEmpty(w)))
        {
          output.AddOrIncrementValue(word);
          ConsoleHelper.WriteLine(string.Format("stage 3: added {0}",
              word));
        }
      }
    });
}
```

第 3 章解释了扩展方法。

回忆一下，如果拆分得到的某个单词在字典中还不存在，那么管道的第三阶段会把该单词添加到字典中；如果该单词已经存在，则递增字典中的一个值。扩展方法 AddOrIncrementValue 实现了这个功能。因为字典不能用于 BlockingCollection<T>，所以没有阻塞方法等待添加值的操作完成。相反，当需要确认添加或更新值的操作是否成功完成时，可以使用 TryXXX 方法。如果同时另一线程也在更新值，那么更新操作可能失败。该实现代码使用了 TryGetValue 方法来检查某个单词在字典中是否已经存在，使用 TryUpdate 方法来更新值，使用 TryAdd 来添加一个值(代码文件 PipelineSample/ConcurrentDictionaryExtensions.cs)：

```
using System.Collections.Concurrent;

namespace Wrox.ProCSharp.Collections
{
  public static class ConcurrentDictionaryExtension
  {
    public static void AddOrIncrementValue(
        this ConcurrentDictionary<string, int> dict, string key)
    {
      bool success = false;
      while (!success)
      {
        int value;
        if (dict.TryGetValue(key, out value))
        {
          if (dict.TryUpdate(key, value + 1, value))
          {
            success = true;
          }
        }
        else
        {
          if (dict.TryAdd(key, 1))
          {
            success = true;
          }
        }
      }
    }
  }
}
```

运行前 3 个阶段的应用程序，得到的输出如下所示，各个阶段的操作交织在一起：

```
stage 3: added get
stage 3: added set
stage 3: added public
stage 3: added int
stage 3: added Wins
stage 2: added  public static class Pipeline
stage 2: added  {
stage 2: added    public static Task ReadFil
stage 2: added    {
stage 2: added      return Task.Run(() =>
```

10.13.4 完成管道

在完成前 3 个阶段后，接下来的 3 个阶段也可以并行运行。TransferContentAsync 从字典中获取数据，将其转换为 Info 类型，然后放到输出 BlockingCollection<T>中(代码文件 PipelineSample/PipelineStages.cs)：

```
public static Task TransferContentAsync(
    ConcurrentDictionary<string, int> input,
    BlockingCollection<Info> output)
{
```

```
    return Task.Run(() =>
      {
        foreach (var word in input.Keys)
        {
          int value;
          if (input.TryGetValue(word, out value))
          {
            var info = new Info { Word = word, Count = value };
            output.Add(info);
            ConsoleHelper.WriteLine(string.Format("stage 4: added {0}",
                info));
          }
        }
        output.CompleteAdding();
      });
}
```

管道阶段 AddColorAsync 根据 Count 属性的值设置 Info 类型的 Color 属性：

```
public static Task AddColorAsync(BlockingCollection<Info> input,
    BlockingCollection<Info> output)
{
    return Task.Run(() =>
      {
        foreach (var item in input.GetConsumingEnumerable())
        {
          if (item.Count > 40)
          {
            item.Color = "Red";
          }
          else if (item.Count > 20)
          {
            item.Color = "Yellow";
          }
          else
          {
            item.Color = "Green";
          }
          output.Add(item);
          ConsoleHelper.WriteLine(string.Format(
              "stage 5: added color {1} to {0}", item, item.Color));
        }
        output.CompleteAdding();
      });
}
```

最后一个阶段用指定的颜色在控制台中输出结果：

```
public static Task ShowContentAsync(BlockingCollection<Info> input)
{
    return Task.Run(() =>
    {
      foreach (var item in input.GetConsumingEnumerable())
      {
        ConsoleHelper.WriteLine(string.Format("stage 6: {0}", item),
```

```
                item.Color);
        }
    });
}
```

运行应用程序，得到的结果如图 10-8 所示。

```
C:\WINDOWS\system32\cmd.exe
stage 6: 1 times: seven
stage 6: 4 times: wait.Set
stage 6: 3 times: carl
stage 6: 1 times: documents
stage 6: 23 times: Revision
stage 6: 1 times: 379025.00m
stage 6: 1 times: now
stage 6: 1 times: Carl
stage 6: 10 times: await
stage 6: 2 times: Brazil
stage 6: 1 times: priorityNodes.Add
stage 6: 1 times: Hunt
stage 6: 1 times: SortedSet<string>
stage 6: 3 times: dict.TryGetValue
stage 6: 8 times: words
stage 6: 2 times: waits[i]
stage 6: 1 times: f8d631b0-3a93-4734-b5b6-9953b6cd7f34
stage 6: 6 times: started
stage 6: 3 times: HashSet<string>
all stages finished
```

图 10-8

10.14　性能

许多集合类都提供了相同的功能，例如，SortedList 类与 SortedDictionary 类的功能几乎完全相同。但是，其性能常常有很大区别。一个集合使用的内存少，另一个集合的元素检索速度快。在 MSDN 文档中，集合的方法常常有性能提示，给出了以大写 O 记号表示的操作时间：

```
O(1)
O(log n)
O(n)
```

O(1)表示无论集合中有多少数据项，这个操作需要的时间都不变。例如，ArrayList 类的 Add()方法就具有 O(1)行为。无论列表中有多少个元素，在列表末尾添加一个新元素的时间都相同。Count 属性会给出元素个数，所以很容易找到列表末尾。

O(n)表示对于集合执行一个操作需要的时间在最坏情况时是 N。如果需要重新给集合分配内存，ArrayList 类的 Add()方法就是一个 O(n)操作。改变容量，需要复制列表，复制的时间随元素的增加而线性增加。

O(log n)表示操作需要的时间随集合中元素的增加而增加，但每个元素需要增加的时间不是线性的，而是呈对数曲线。在集合中执行插入操作时，SortedDictionary<TKey,TValue>集合类具有 O(log n)行为，而 SortedList<TKey,TValue>集合类具有 O(n)行为。这里 SortedDictionary <TKey,TValue>集合类要快得多，因为它在树型结构中插入元素的效率比列表高得多。

表 10-6 列出了集合类及其执行不同操作的性能，例如添加、插入和删除元素。使用这个表可以选择性能最佳的集合类。左列是集合类，Add 列给出了在集合中添加元素所需的时间。List<T>和 HashSet<T>类把 Add 方法定义为在集合中添加元素。其他集合类用不同的方法把元素添加到集合中。例如 Stack<T>类定义了 Push()方法，Queue<T>类定义了 Enqueue()方法。这些信息也列在表中。

如果单元格中有多个大 O 值，表示若集合需要重置大小，该操作就需要一定的时间。例如，在 List<T>类中，添加元素的时间是 O(1)。如果集合的容量不够大，需要重置大小，重置大小需要的时

间长度就是 O(n)。集合越大，重置大小操作的时间就越长。最好避免重置集合的大小，而应把集合的容量设置为一个可以包含所有元素的值。

如果单元格的内容是 n/a(代表 not applicatable)，就表示这个操作不能应用于这种集合类型。

表 10-6

集 合	Add	Insert	Remove	Item	Sort	Find
List<T>	如果集合必须重置大小，就是 O(1)或 O(n)	O(n)	O(n)	O(1)	O(n log n)，最坏的情况是 O (n ^ 2)	
Stack<T>	Push()，如果栈必须重置大小，就是 O(1)或 O(n)	n/a	Pop, O(1)	n/a	n/a	n/a
Queue<T>	Enqueue()，如果队列必须重置大小，就是 O(1)或 O(n)	n/a	Dequeue, O(1)	n/a	n/a	n/a
HashSet<T>	如果集必须重置大小，就是 O(1)或 O(n)	Add() O(1)或 O(n)	O(1)	n/a	n/a	n/a
SortedSet<T>	如果集必须重置大小，就是 O(1)或 O(n)	Add() O(1)或 O(n)	O(1)	n/a	n/a	n/a
LinkedList<T>	AddLast()、O(1)	AddAfter() O(1)	O(1)	n/a	n/a	O(n)
Dictionary <TKey, TValue>	O(1)或 O(n)	n/a	O(1)	O(1)	n/a	n/a
SortedDictionary <TKey, TValue>	O(log n)	n/a	O(log n)	O(log n)	n/a	n/a
SortedList <TKey, TValue>	无序数据为 O(n)；如果必须重置大小就是 O(n)；到列表的尾部就是 O(logn)	n/a	O(n)	读写是 O(log n)；如果键在列表中，就是 O(log n)；如果键不在列表中，就是 O(n)	n/a	n/a

10.15 小结

本章介绍了如何处理不同类型的集合。数组的大小是固定的，但可以使用列表作为动态增长的集合。队列以先进先出的方式访问元素，栈以后进先出的方式访问元素。链表可以快速插入和删除元素，但搜索操作比较慢。通过键和值可以使用字典，它的搜索和插入操作比较快。集用于唯一项，可以是无序的(HashSet<T>)，也可以是有序的(SortedSet<T>)。ObservableCollection<T>类提供了在列表中的元素发生变化时触发的事件。

本章还介绍了许多接口和类及其在集合访问和排序上的用法。探讨了一些特殊的集合，如 BitArray 和 BitVector32，它们为处理带有位的集合进行了优化。

第 11 章将详细介绍 LINQ。

第11章

LINQ

本章要点

- 用列表在对象上执行传统查询
- 扩展方法
- LINQ 查询操作符
- 平行 LINQ
- 表达式树

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- LINQ Intro
- Enumerable Sample
- Parallel LINQ
- Expression Trees

11.1 LINQ 概述

语言集成查询(Language Integrated Query，LINQ)在 C#编程语言中集成了查询语法，可以用相同的语法访问不同的数据源。LINQ 提供了不同数据源的抽象层，所以可以使用相同的语法。

本章介绍 LINQ 的核心原理和 C#中支持 C# LINQ 查询的语言扩展。

读完本章后，在数据库中使用 LINQ 的内容可查阅第 33 章，查询 XML 数据的内容可参见第 34 章。

在介绍 LINQ 的特性之前，本节先介绍一个简单的 LINQ 查询。C#提供了转换为方法调用的集成查

询语言。本节会说明这个转换的过程，以便用户使用 LINQ 的全部功能。

11.1.1　列表和实体

本章的 LINQ 查询在一个包含 1950～2011 年一级方程式锦标赛的集合上进行。这些数据需要使用实体类和列表来准备。

对于实体，定义类型 Racer。Racer 定义了几个属性和一个重载的 ToString()方法，该方法以字符串格式显示赛车手。这个类实现了 IFormattable 接口，以支持格式字符串的不同变体，这个类还实现了 IComparable<Racer>接口，它根据 Lastname 为一组赛车手排序。为了执行更高级的查询，Racer 类不仅包含单值属性，如 Firstname、Lastname、Wins、Country 和 Starts，还包含多值属性，如 Cars 和 Years。Years 属性列出了赛车手获得冠军的年份。一些赛车手曾多次获得冠军。Cars 属性用于列出赛车手在获得冠军的年份中使用的所有车型(代码文件 DataLib/Racer.cs)。

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

namespace Wrox.ProCSharp.LINQ
{
  [Serializable]
  public class Racer: IComparable<Racer>, IFormattable
  {
    public Racer(string firstName, string lastName, string country,
        int starts, int wins)
      : this(firstName, lastName, country, starts, wins, null, null)
    {
    }
    public Racer(string firstName, string lastName, string country,
        int starts, int wins, IEnumerable<int> years, IEnumerable<string> cars)
    {
      this.FirstName = firstName;
      this.LastName = lastName;
      this.Country = country;
      this.Starts = starts;
      this.Wins = wins;
      this.Years = new List<int>(years);
      this.Cars = new List<string>(cars);
    }

    public string FirstName {get; set;}
    public string LastName {get; set;}
    public int Wins {get; set;}
    public string Country {get; set;}
    public int Starts {get; set;}
    public IEnumerable<string> Cars { get; private set; }
    public IEnumerable<int> Years { get; private set; }

    public override string ToString()
    {
      return String.Format("{0} {1}", FirstName, LastName);
    }

    public int CompareTo(Racer other)
```

```
      {
        if (other == null) return -1;
        return string.Compare(this.LastName, other.LastName);
      }

      public string ToString(string format)
      {
        return ToString(format, null);
      }

      public string ToString(string format, IFormatProvider formatProvider)
      {
        switch (format)
        {
          case null:
          case "N":
            return ToString();
          case "F":
            return FirstName;
          case "L":
            return LastName;
          case "C":
            return Country;
          case "S":
            return Starts.ToString();
          case "W":
            return Wins.ToString();
          case "A":
            return String.Format("{0} {1}, {2}; starts: {3}, wins: {4}",
                         FirstName, LastName, Country, Starts, Wins);
          default:
            throw new FormatException(String.Format(
                         "Format {0} not supported", format));
        }
      }
    }
  }
```

第二个实体类是 Team。这个类仅包含车队冠军的名字和获得冠军的年份。与赛车手冠军类似，针对一年中最好的车队也有一个冠军奖项(代码文件 DataLib/Team.cs)：

```
[Serializable]
public class Team
{
  public Team(string name, params int[] years)
  {
    this.Name = name;
    this.Years = new List<int>(years);
  }
  public string Name { get; private set; }
  public IEnumerable<int> Years { get; private set; }
}
```

Formula1 类在 GetChampions()方法中返回一组赛车手。这个列表包含了 1950~2011 年之间的所

有一级方程式冠军(代码文件 DataLib/Formula1.cs)。

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.LINQ
{
  public static class Formula1
  {
    private static List<Racer> racers;

    public static IList<Racer> GetChampions()
    {
      if (racers == null)
      {
        racers = new List<Racer>(40);
        racers.Add(new Racer("Nino", "Farina", "Italy", 33, 5,
           new int[] { 1950 }, new string[] { "Alfa Romeo" }));
        racers.Add(new Racer("Alberto", "Ascari", "Italy", 32, 10,
           new int[] { 1952, 1953 }, new string[] { "Ferrari" }));
        racers.Add(new Racer("Juan Manuel", "Fangio", "Argentina", 51, 24,
           new int[] { 1951, 1954, 1955, 1956, 1957 },
           new string[] { "Alfa Romeo", "Maserati", "Mercedes", "Ferrari" }));
        racers.Add(new Racer("Mike", "Hawthorn", "UK", 45, 3,
           new int[] { 1958 }, new string[] { "Ferrari" }));
        racers.Add(new Racer("Phil", "Hill", "USA", 48, 3, new int[] { 1961 },
           new string[] { "Ferrari" }));
        racers.Add(new Racer("John", "Surtees", "UK", 111, 6,
           new int[] { 1964 }, new string[] { "Ferrari" }));
        racers.Add(new Racer("Jim", "Clark", "UK", 72, 25,
           new int[] { 1963, 1965 }, new string[] { "Lotus" }));
        racers.Add(new Racer("Jack", "Brabham", "Australia", 125, 14,
           new int[] { 1959, 1960, 1966 },
           new string[] { "Cooper", "Brabham" }));
        racers.Add(new Racer("Denny", "Hulme", "New Zealand", 112, 8,
           new int[] { 1967 }, new string[] { "Brabham" }));
        racers.Add(new Racer("Graham", "Hill", "UK", 176, 14,
           new int[] { 1962, 1968 }, new string[] { "BRM", "Lotus" }));
        racers.Add(new Racer("Jochen", "Rindt", "Austria", 60, 6,
           new int[] { 1970 }, new string[] { "Lotus" }));
        racers.Add(new Racer("Jackie", "Stewart", "UK", 99, 27,
           new int[] { 1969, 1971, 1973 },
           new string[] { "Matra", "Tyrrell" }));
        //...

        return racers;
      }
    }
  }
}
```

对于后面在多个列表中执行的查询，GetConstructorChampions()方法返回所有的车队冠军的列表。车队冠军是从 1958 年开始设立的。

```
private static List<Team> teams;
public static IList<Team> GetContructorChampions()
```

```
{
  if (teams == null)
  {
    teams = new List<Team>()
    {
      new Team("Vanwall", 1958),
      new Team("Cooper", 1959, 1960),
      new Team("Ferrari", 1961, 1964, 1975, 1976, 1977, 1979, 1982,
               1983, 1999, 2000, 2001, 2002, 2003, 2004, 2007, 2008),
      new Team("BRM", 1962),
      new Team("Lotus", 1963, 1965, 1968, 1970, 1972, 1973, 1978),
      new Team("Brabham", 1966, 1967),
      new Team("Matra", 1969),
      new Team("Tyrrell", 1971),
      new Team("McLaren", 1974, 1984, 1985, 1988, 1989, 1990, 1991, 1998),
      new Team("Williams", 1980, 1981, 1986, 1987, 1992, 1993, 1994, 1996,
               1997),
      new Team("Benetton", 1995),
      new Team("Renault", 2005, 2006),
      new Team("Brawn GP", 2009),
      new Team("Red Bull Racing", 2010, 2011)
    };
  }
  return teams;
}
```

11.1.2 LINQ 查询

使用这些准备好的列表和实体，进行 LINQ 查询，例如查询出来自巴西的所有世界冠军，并按照夺冠次数排序。为此可以使用 List<T>类的方法，如 FindAll()和 Sort()方法。而使用 LINQ 的语法非常简单(代码文件 LINQIntro/Program.cs)：

```
private static void LinqQuery()
{
  var query = from r in Formula1.GetChampions()
              where r.Country == "Brazil"
              orderby r.Wins descending
              select r;

  foreach (Racer r in query)
  {
    Console.WriteLine("{0:A}", r);
  }
}
```

这个查询的结果显示了来自巴西的所有世界冠军，并排好序：

```
Ayrton Senna, Brazil; starts: 161, wins: 41
Nelson Piquet, Brazil; starts: 204, wins: 23
Emerson Fittipaldi, Brazil; starts: 143, wins: 14
```

语句

```
from r in Formula1.GetChampions()
where r.Country == "Brazil"
```

```
orderby r.Wins descending
select r;
```

是一个 LINQ 查询，子句 from、where、orderby、descending 和 select 都是这个查询中预定义的关键字。

查询表达式必须以 from 子句开头，以 select 或 group 子句结束。在这两个子句之间，可以使用 where、orderby、join、let 和其他 from 子句。

变量 query 只指定了 LINQ 查询。该查询不是通过这个赋值语句执行的，只要使用 foreach 循环访问查询，该查询就会执行。详见 11.1.4 小节的内容。

11.1.3 扩展方法

编译器会转换 LINQ 查询，以调用方法而不是 LINQ 查询。LINQ 为 IEnumerable<T>接口提供了各种扩展方法，以便用户在实现了该接口的任意集合上使用 LINQ 查询。扩展方法在静态类中声明，定义为一个静态方法，其中第一个参数定义了它扩展的类型。

扩展方法可以将方法写入最初没有提供该方法的类中。还可以把方法添加到实现某个特定接口的任何类中，这样多个类就可以使用相同的实现代码。

例如，String 类没有 Foo()方法。String 类是密封的，所以不能从这个类中继承。但可以创建一个扩展方法，如下所示：

```
public static class StringExtension
{
  public static void Foo(this string s)
  {
    Console.WriteLine("Foo invoked for {0}", s);
  }
}
```

扩展方法定义为静态方法，其第一个参数定义了它扩展的类型，扩展方法在一个静态类中声明。Foo()方法扩展了 String 类，因为它的第一个参数定义为 String 类型。为了区分扩展方法和一般的静态方法，扩展方法还需要对第一个参数使用 this 关键字。

现在就可以使用带 string 类型的 Foo()方法了：

```
string s = "Hello";
s.Foo();
```

结果在控制台上显示“Foo invoked for Hello”，因为 Hello 是传递给 Foo()方法的字符串。

也许这看起来违反了面向对象的规则，因为给一个类型定义了新方法，但没有改变该类型或派生自它的类型。但实际上并非如此。扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种新语法。对于字符串，可以用如下方式调用 Foo()方法，获得相同的结果：

```
string s = "Hello";
StringExtension.Foo(s);
```

要调用静态方法，应在类名的后面加上方法名。扩展方法是调用静态方法的另一种方式。不必提供定义了静态方法的类名，相反，编译器调用静态方法是因为它带的参数类型。只需导入包含该

类的名称空间，就可以将Foo()扩展方法放在String类的作用域中。

定义LINQ扩展方法的一个类是System.Linq名称空间中的Enumerable。只需要导入这个名称空间，就打开了这个类的扩展方法的作用域。下面列出了Where()扩展方法的实现代码。Where()扩展方法的第一个参数包含了this关键字，其类型是IEnumerable<T>。这样，Where()方法就可以用于实现了IEnumerable<T>的每个类型。例如数组和List<T>类实现了IEnumerable<T>接口。第二个参数是一个Func<T,bool>委托，它引用了一个返回布尔值、参数类型为T的方法。这个谓词在实现代码中调用，检查IEnumerable<T>源中的项是否应放在目标集合中。如果委托引用了该方法，yield return语句就将源中的项返回给目标。

```
public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, bool> predicate)
{
  foreach (TSource item in source)
    if (predicate(item))
      yield return item;
}
```

因为Where()作为一个泛型方法实现，所以它可以用于包含在集合中的任意类型。实现了IEnumerable<T>接口的任意集合都支持它。

这里的扩展方法在System.Core程序集的System.Linq名称空间中定义。

现在就可以使用Enumerable类中的扩展方法Where()、OrderByDescending()和Select()。这些方法都返回IEnumerable<TSource>，所以可以使用前面的结果依次调用这些方法。通过扩展方法的参数，使用定义了委托参数的实现代码的匿名方法(代码文件LINQIntro/Program.cs)。

```
static void ExtensionMethods()
{
  var champions = new List<Racer>(Formula1.GetChampions());
  IEnumerable<Racer> brazilChampions =
      champions.Where(r => r.Country == "Brazil").
              OrderByDescending(r => r.Wins).
              Select(r => r);

  foreach (Racer r in brazilChampions)
  {
    Console.WriteLine("{0:A}", r);
  }
}
```

11.1.4 推迟查询的执行

在运行期间定义查询表达式时，查询就不会运行。查询会在迭代数据项时运行。

再看看扩展方法Where()。它使用yield return语句返回谓词为true的元素。因为使用了yield return语句，所以编译器会创建一个枚举器，在访问枚举中的项后，就返回它们。

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                       Func<T, bool> predicate)
```

```
{
  foreach (T item in source)
  {
    if (predicate(item))
    {
      yield return item;
    }
  }
}
```

这是一个非常有趣也非常重要的结果。在下面的例子中，创建了 String 元素的一个集合，用名称填充它。接着定义一个查询，从集合中找出以字母 J 开头的所有名称。集合也应是排好序的。在定义查询时，不会进行迭代。相反，迭代在 foreach 语句中进行，在其中迭代所有的项。集合中只有一个元素 Juan 满足 where 表达式的要求，即以字母 J 开头。迭代完成后，将 Juan 写入控制台。之后在集合中添加 4 个新名称，再次进行迭代。

```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };

var namesWithJ = from n in names
                 where n.StartsWith("J")
                 orderby n
                 select n;

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

因为迭代在查询定义时不会进行，而是在执行每个 foreach 语句时进行，所以可以看到其中的变化，如应用程序的结果所示：

```
First iteration
Juan

Second iteration
Jack
Jim
John
Juan
```

当然，还必须注意，每次在迭代中使用查询时，都会调用扩展方法。在大多数情况下，这是非常有效的，因为我们可以检测出源数据中的变化。但是在一些情况下，这是不可行的。调用扩展方

法 ToArray()、ToList()等可以改变这个操作。在示例中，ToList 遍历集合，返回一个实现了 IList<string> 的集合。之后对返回的列表遍历两次，在两次迭代之间，数据源得到了新名称。

```
new List<string>
var names=
                { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names
                  where n.StartsWith("J")
                  orderby n
                  select n).ToList();

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

在结果中可以看到，在两次迭代之间输出保持不变，但集合中的值改变了：

```
First iteration
Juan

Second iteration
Juan
```

11.2 标准的查询操作符

Where、OrderByDescending 和 Select 只是 LINQ 的几个查询操作符。LINQ 查询为最常用的操作符定义了一个声明语法。还有许多查询操作符可用于 Enumerable 类。

表 11-1 列出了 Enumerable 类定义的标准查询操作符。

表 11-1

标准查询操作符	说　明
Where OfType<TResult>	筛选操作符定义了返回元素的条件。在 Where 查询操作符中，可以使用谓词，例如 lambda 表达式定义的谓词，来返回布尔值。OfType<TResult>根据类型筛选元素，只返回 TResult 类型的元素
Select SelectMany	投射操作符用于把对象转换为另一个类型的新对象。Select 和 SelectMany 定义了根据选择器函数选择结果值的投射

(续表)

标准查询操作符	说　明
OrderBy ThenBy OrderByDescending ThenByDescending Reverse	排序操作符改变所返回的元素的顺序。OrderBy 按升序排序，OrderByDescending 按降序排序。如果第一次排序的结果很类似，就可以使用 ThenBy 和 ThenBy Descending 操作符进行第二次排序。Reverse 反转集合中元素的顺序
Join GroupJoin	连接操作符用于合并不直接相关的集合。使用 Join 操作符，可以根据键选择器函数连接两个集合，这类似于 SQL 中的 JOIN。GroupJoin 操作符连接两个集合，组合其结果
GroupBy ToLookup	组合操作符把数据放在组中。GroupBy 操作符组合有公共键的元素。ToLookup 通过创建一个一对多字典，来组合元素
Any All Contains	如果元素序列满足指定的条件，限定符操作符就返回布尔值。Any、All 和 Contains 都是限定符操作符。Any 确定集合中是否有满足谓词函数的元素；All 确定集合中的所有元素是否都满足谓词函数；Contains 检查某个元素是否在集合中
Take Skip TakeWhile SkipWhile	分区操作符返回集合的一个子集。Take、Skip、TakeWhile 和 SkipWhile 都是分区操作符。使用它们可以得到部分结果。使用 Take 必须指定要从集合中提取的元素个数；Skip 跳过指定的元素个数，提取其他元素；TakeWhile 提取条件为真的元素
Distinct Union Intersect Except Zip	Set 操作符返回一个集合。Distinct 从集合中删除重复的元素。除了 Distinct 之外，其他 Set 操作符都需要两个集合。Union 返回出现在其中一个集合中的唯一元素。Intersect 返回两个集合中都有的元素。Except 返回只出现在一个集合中的元素。Zip 把两个集合合并为一个
First FirstOrDefault Last LastOrDefault ElementAt ElementAtOrDefault Single SingleOrDefault	这些元素操作符仅返回一个元素。First 返回第一个满足条件的元素。FirstOrDefault 类似于 First，但如果没有找到满足条件的元素，就返回类型的默认值。Last 返回最后一个满足条件的元素。ElementAt 指定了要返回的元素的位置。Single 只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异常
Count Sum Min Max Average Aggregate	聚合操作符计算集合的一个值。利用这些聚合操作符，可以计算所有值的总和、所有元素的个数、值最大和最小的元素，以及平均值等

(续表)

标准查询操作符	说　明
ToArray AsEnumerable ToList ToDictionary Cast<TResult>	这些转换操作符将集合转换为数组：IEnumerable、IList、IDictionary 等
Empty Range Repeat	这些生成操作符返回一个新集合。使用 Empty 时集合是空的；Range 返回一系列数字；Repeat 返回一个始终重复一个值的集合

下面是使用这些操作符的一些例子。

11.2.1 筛选

下面介绍一些查询的示例。

使用 where 子句，可以合并多个表达式。例如，找出赢得至少 15 场比赛的巴西和奥地利赛车手。传递给 where 子句的表达式的结果类型应是布尔类型：

```
var racers = from r in Formula1.GetChampions()
             where r.Wins > 15 &&
                   (r.Country == "Brazil" || r.Country == "Austria")
             select r;

foreach (var r in racers)
{
  Console.WriteLine("{0:A}", r);
}
```

用这个 LINQ 查询启动程序，会返回 Niki Lauda、Nelson Piquet 和 Ayrton Senna，如下：

```
Niki Lauda, Austria, Starts: 173, Wins: 25
Nelson Piquet, Brazil, Starts: 204, Wins: 23
Ayrton Senna, Brazil, Starts: 161, Wins: 41
```

并不是所有的查询都可以用 LINQ 查询语法完成。也不是所有的扩展方法都映射到 LINQ 查询子句上。高级查询需要使用扩展方法。为了更好地理解带扩展方法的复杂查询，最好看看简单的查询是如何映射的。使用扩展方法 Where()和 Select()，会生成与前面 LINQ 查询非常类似的结果：

```
var racers = Formula1.GetChampions().
    Where(r => r.Wins > 15 &&
          (r.Country == "Brazil" || r.Country == "Austria")).
    Select(r => r);
```

11.2.2 用索引筛选

不能使用 LINQ 查询的一个例子是 Where()方法的重载。在 Where()方法的重载中，可以传递第二个参数——索引。索引是筛选器返回的每个结果的计数器。可以在表达式中使用这个索引，执行

基于索引的计算。下面的代码由 Where()扩展方法调用，它使用索引返回姓氏以 A 开头、索引为偶数的赛车手(代码文件 EnumerableSample/Program.cs)：

```
var racers = Formula1.GetChampions().
   Where((r, index) => r.LastName.StartsWith("A") && index % 2 != 0);
foreach (var r in racers)
{
  Console.WriteLine("{0:A}", r);
}
```

姓氏以 A 开头的所有赛车手有 Alberto Ascari、Mario Andretti 和 Fernando Alonso。因为 Mario Andretti 的索引是奇数，所以他不在结果中：

```
Alberto Ascari, Italy; starts: 32, wins: 10
Fernando Alonso, Spain; starts: 177, wins: 27
```

11.2.3 类型筛选

为了进行基于类型的筛选，可以使用 OfType()扩展方法。这里数组数据包含 string 和 int 对象。使用 OfType()扩展方法，把 string 类传送给泛型参数，就从集合中仅返回字符串(代码文件 EnumerableSample/Program.cs)：

```
object[] data = { "one", 2, 3, "four", "five", 6 };
var query = data.OfType<string>();
foreach (var s in query)
{
  Console.WriteLine(s);
}
```

运行这段代码，就会显示字符串 one、four 和 five。

```
one
four
five
```

11.2.4 复合的 from 子句

如果需要根据对象的一个成员进行筛选，而该成员本身是一个系列，就可以使用复合的 from 子句。Racer 类定义了一个属性 Cars，其中 Cars 是一个字符串数组。要筛选驾驶法拉利的所有冠军，可以使用如下所示的 LINQ 查询。第一个 from 子句访问从 Formula1.Get Champions()方法返回的 Racer 对象，第二个 from 子句访问 Racer 类的 Cars 属性，以返回所有 string 类型的赛车。接着在 where 子句中使用这些赛车筛选驾驶法拉利的所有冠军(代码文件 EnumerableSample/Program.cs)。

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     fromc in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r.FirstName + " " + r.LastName;
```

这个查询的结果显示了驾驶法拉利的所有一级方程式冠军：

```
Alberto Ascari
Juan Manuel Fangio
```

```
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

C#编译器把复合的 from 子句和 LINQ 查询转换为 SelectMany()扩展方法。SelectMany()方法可用于迭代序列的序列。示例中 SelectMany()方法的重载版本如下所示：

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult> (
    this IEnumerable<TSource> source,
    Func<TSource,
    IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector);
```

第一个参数是隐式参数，它从 GetChampions()方法中接收 Racer 对象序列。第二个参数是 collectionSelector 委托，其中定义了内部序列。在 lambda 表达式 r => r.Cars 中，应返回赛车集合。第三个参数是一个委托，现在为每个赛车调用该委托，接收 Racer 和 Car 对象。lambda 表达式创建了一个匿名类型，它有 Racer 和 Car 属性。这个 SelectMany()方法的结果是摊平了赛车手和赛车的层次结构，为每辆赛车返回匿名类型的一个新对象集合。

这个新集合传递给 Where()方法，筛选出驾驶法拉利的赛车手。最后，调用 OrderBy()和 Select()方法：

```
var ferrariDrivers = Formula1.GetChampions().
    SelectMany(r => r.Cars,
        (r, c) => new { Racer = r, Car = c }).
        Where(r => r.Car == "Ferrari").
        OrderBy(r => r.Racer.LastName).
        Select(r => r.Racer.FirstName + " " + r.Racer.LastName);
```

把 SelectMany()泛型方法解析为这里使用的类型，所解析的类型如下所示。在这个例子中，数据源是 Racer 类型，所筛选的集合是一个 string 数组，当然所返回的匿名类型的名称是未知的，这里显示为 TResult：

```
public static IEnumerable<TResult> SelectMany<Racer, string, TResult> (
    this IEnumerable<Racer> source,
    Func<Racer, IEnumerable<string>> collectionSelector,
    Func<Racer, string, TResult> resultSelector);
```

查询仅从 LINQ 查询转换为扩展方法，所以结果与前面的相同。

11.2.5 排序

要对序列排序，前面使用了 orderby 子句。下面复习一下前面使用的例子，但这里使用 orderby descending 子句。其中赛车手按照赢得比赛的次数进行降序排序，赢得比赛的次数用关键字选择器指定(代码文件 EnumerableSample/Program.cs)：

```
var racers = from r in Formula1.GetChampions()
             where r.Country == "Brazil"
```

```
                orderby r.Wins descending
                select r;
```

orderby 子句解析为 OrderBy()方法，orderby descending 子句解析为 OrderByDescending()方法：

```
var racers = Formula1.GetChampions().
    Where(r => r.Country == "Brazil").
    OrderByDescending(r => r.Wins).
    Select(r => r);
```

OrderBy()和 OrderByDescending()方法返回 IOrderEnumerable<TSource>。这个接口派生自 IEnumerable<TSource>接口，但包含一个额外的方法 CreateOrderedEnumerable<TSource>()。这个方法用于进一步给序列排序。如果根据关键字选择器来排序，其中有两项相同，就可以使用 ThenBy()和 ThenByDescending ()方法继续排序。这两个方法需要 IOrderEnumerable<TSource>接口才能工作，但也返回这个接口。所以，可以添加任意多个 ThenBy()和 ThenByDescending()方法，对集合排序。

使用 LINQ 查询时，只需要把所有用于排序的不同关键字(用逗号分隔开)添加到 orderby 子句中。在下例中，所有的赛车手先按照国家排序，再按照姓氏排序，最后按照名字排序。添加到 LINQ 查询结果中的 Take()扩展方法用于提取前 10 个结果：

```
var racers = (from r in Formula1.GetChampions()
              orderby r.Country, r.LastName, r.FirstName
              select r).Take(10);
```

排序后的结果如下：

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

使用 OrderBy()和 ThenBy()扩展方法可以执行相同的操作：

```
var racers = Formula1.GetChampions().
  OrderBy(r => r.Country).
  ThenBy(r => r.LastName).
  ThenBy(r => r.FirstName).
  Take(10);
```

11.2.6 分组

要根据一个关键字值对查询结果分组，可以使用 group 子句。现在一级方程式冠军应按照国家分组，并列出一个国家的冠军数。子句 group r by r.Country into g 根据 Country 属性组合所有的赛车手，并定义一个新的标识符 g，它以后用于访问分组的结果信息。group 子句的结果根据应用到分组结果上的扩展方法 Count()来排序，如果冠军数相同，就根据关键字来排序，该关键字是国家，因为这是分组所使用的关键字。where 子句根据至少有两项的分组来筛选结果，select 子句创建一个带

Country 和 Count 属性的匿名类型(代码文件 EnumerableSample/Program.cs)。

```
var countries = from r in Formula1.GetChampions()
                groupr by r.Country into g
                orderby g.Count() descending, g.Key
                where g.Count() >= 2
                select new {
                          Country = g.Key,
                          Count = g.Count()
                        };

foreach (var item in countries)
{
  Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
}
```

结果显示了带 Country 和 Count 属性的对象集合：

```
UK        10
Brazil    3
Finland   3
Australia 2
Austria   2
Germany   2
Italy     2
USA       2
```

要用扩展方法执行相同的操作，应把 groupby 子句解析为 GroupBy()方法。在 GroupBy()方法的声明中，注意它返回实现了 IGrouping 接口的枚举对象。IGrouping 接口定义了 Key 属性，所以在定义了对这个方法的调用后，可以访问分组的关键字：

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
     this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

把子句 group r by r.Country into g 解析为 GroupBy(r => r.Country)，返回分组序列。分组序列首先用 OrderByDescending()方法排序，再用 ThenBy()方法排序。接着调用 Where()和 Select()方法。

```
var countries = Formula1.GetChampions().
    GroupBy(r => r.Country).
    OrderByDescending(g => g.Count()).
    ThenBy(g => g.Key).
    Where(g => g.Count() >= 2).
    Select(g => new { Country = g.Key,
                      Count = g.Count() });
```

11.2.7 对嵌套的对象分组

如果分组的对象应包含嵌套的序列，就可以改变 select 子句创建的匿名类型。在下面的例子中，所返回的国家不仅应包含国家名和赛车手数量这两个属性，还应包含赛车手名序列。这个序列用一个赋予 Racers 属性的 from/in 内部子句指定，内部的 from 子句使用分组标识符 g 获得该分组中的所有赛车手，用姓氏对它们排序，再根据姓名创建一个新字符串(代码文件 EnumerableSample/Program.cs)：

```
var countries = from r in Formula1.GetChampions()
                group r by r.Country into g
                orderby g.Count() descending, g.Key
                where g.Count() >= 2
                select new
                {
                  Country = g.Key,
                  Count = g.Count(),
                  Racers = from r1 in g
                           orderby r1.LastName
                           select r1.FirstName + " " + r1.LastName
                };
foreach (var item in countries)
{
  Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
  foreach (var name in item.Racers)
  {
    Console.Write("{0}; ", name);
  }
  Console.WriteLine();
}
```

结果应列出某个国家的所有冠军：

```
UK        10
Jenson Button; Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill;
Damon Hill; James Hunt; Nigel Mansell; Jackie Stewart; John Surtees;
Brazil    3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Finland   3
Mika Hakkinen; Kimi Raikkonen; Keke Rosberg;
Australia 2
Jack Brabham; Alan Jones;
Austria   2
Niki Lauda; Jochen Rindt;
Germany   2
Michael Schumacher; Sebastian Vettel;
Italy     2
Alberto Ascari; Nino Farina;
USA       2
Mario Andretti; Phil Hill;
```

11.2.8　内连接

使用 join 子句可以根据特定的条件合并两个数据源，但之前要获得两个要连接的列表。在一级方程式比赛中，有赛车手冠军和车队冠军。赛车手从 GetChampions()方法中返回，车队从 GetConstructorChampions()方法中返回。现在要获得一个年份列表，列出每年的赛车手冠军和车队冠军。

为此，先定义两个查询，用于查询赛车手和车队(代码文件 EnumerableSample/Program.cs)：

```
var racers = from r in Formula1.GetChampions()
             from y in r.Years
             select new
             {
               Year = y,
```

```
            Name = r.FirstName + " " + r.LastName
          };

var teams = from t in Formula1.GetContructorChampions()
            from y in t.Years
            select new
            {
              Year = y,
              Name = t.Name
            };
```

有了这两个查询，再通过join子句，根据赛车手获得冠军的年份和车队获得冠军的年份进行连接。select子句定义了一个新的匿名类型，它包含Year、Racer和Team属性。

```
var racersAndTeams = (from r in racers
                      joint in teams on r.Year equals t.Year
                      select new
                      {
                        r.Year,
                        Champion = r.Name,
                        Constructor = t.Name
                      }).Take(10);
Console.WriteLine("Year World Champion\t Constructor Title");
foreach (var item in racersAndTeams)
{
  Console.WriteLine("{0}: {1,-20} {2}", item.Year, item.Champion,
      item.Constructor);
}
```

当然，也可以把它们合并为一个LINQ查询，但这只是一种个人喜好的问题了：

```
var racersAndTeams =
    (from r in
      from r1 in Formula1.GetChampions()
      from yr in r1.Years
      select new
      {
        Year = yr,
        Name = r1.FirstName + " " + r1.LastName
      }
    joint in
      from t1 in Formula1.GetContructorChampions()
      from yt in t1.Years
      select new
      {
        Year = yt,
        Name = t1.Name
      }
    onr.Year equals t.Year
    orderby t.Year
    select new
    {
      Year = r.Year,
      Racer = r.Name,
      Team = t.Name
    }).Take(10);
```

结果显示了在同时有了赛车手冠军和车队冠军的前 10 年中，匿名类型中的数据：

```
Year  World Champion   Constructor Title
1958: Mike Hawthorn    Vanwall
1959: Jack Brabham     Cooper
1960: Jack Brabham     Cooper
1961: Phil Hill        Ferrari
1962: Graham Hill      BRM
1963: Jim Clark        Lotus
1964: John Surtees     Ferrari
1965: Jim Clark        Lotus
1966: Jack Brabham     Brabham
1967: Denny Hulme      Brabham
```

11.2.9 左外连接

上一个连接示例的输出从 1958 年开始，因为从这一年开始，才同时有了赛车手冠军和车队冠军。赛车手冠军出现得更早一些，是在 1950 年。使用内连接时，只有找到了匹配的记录才返回结果。为了在结果中包含所有的年份，可以使用左外连接。左外连接返回左边序列中的全部元素，即使它们在右边的序列中并没有匹配的元素。

下面修改前面的 LINQ 查询，使用左外连接。左外连接用 join 子句和 DefaultIfEmpty 方法定义。如果查询的左侧(赛车手)没有匹配的车队冠军，那么就使用 DefaultIfEmpty 方法定义其右侧的默认值(代码文件 EnumerableSample/Program.cs)：

```
var racersAndTeams =
  (from r in racers
  join t in teams on r.Year equals t.Year into rt
  from t in rt.DefaultIfEmpty()
  orderby r.Year
  select new
  {
    Year = r.Year,
    Champion = r.Name,
    Constructor = t == null ?"no constructor championship" : t.Name
  }).Take(10);
```

用这个查询运行应用程序，得到的输出将从 1950 年开始，如下所示：

```
Year  Champion                Constructor Title
1950: Nino Farina             no constructor championship
1951: Juan Manuel Fangio      no constructor championship
1952: Alberto Ascari          no constructor championship
1953: Alberto Ascari          no constructor championship
1954: Juan Manuel Fangio      no constructor championship
1955: Juan Manuel Fangio      no constructor championship
1956: Juan Manuel Fangio      no constructor championship
1957: Juan Manuel Fangio      no constructor championship
1958: Mike Hawthorn           Vanwall
1959: Jack Brabham            Cooper
```

11.2.10 组连接

左外连接使用了组连接和 into 子句。它有一部分语法与组连接相同，只不过组连接不使用

DefaultIfEmpty 方法。

使用组连接时，可以连接两个独立的序列，对于其中一个序列中的某个元素，另一个序列中存在对应的一个项列表。

下面的示例使用了两个独立的序列。一个是前面例子中已经看过的冠军列表。另一个是一个 Championship 类型的集合。下面的代码段显示了 Championship 类。该类包含冠军年份以及该年份中获得第一名、第二名和第三名的赛车手，对应的属性分别为 Year、First、Second 和 Third(代码文件 DataLib/Championship.cs)：

```
public class Championship
{
  public int Year { get; set; }
  public string First { get; set; }
  public string Second { get; set; }
  public string Third { get; set; }
}
```

GetChampionships 方法返回了冠军集合，如下面的代码段所示(代码文件 DataLib/Formula1.cs)：

```
private static List<Championship> championships;
public static IEnumerable<Championship> GetChampionships()
{
  if (championships == null)
  {
    championships = new List<Championship>();
    championships.Add(new Championship
  {
    Year = 1950,
    First = "Nino Farina",
    Second = "Juan Manuel Fangio",
    Third = "Luigi Fagioli"
  });
  championships.Add(new Championship
  {
    Year = 1951,
    First = "Juan Manuel Fangio",
    Second = "Alberto Ascari",
    Third = "Froilan Gonzalez"
  });
//...
```

冠军列表应与每个冠军年份中获得前三名的赛车手构成的列表组合起来，然后显示每一年的结果。

RacerInfo 类定义了要显示的信息，如下所示(代码文件 EnumerableSample/RacerInfo.cs)：

```
public class RacerInfo
{
  public int Year { get; set; }
  public int Position { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

使用连接语句可以把两个列表中的赛车手组合起来。

因为冠军列表中的每一项都包含 3 个赛车手，所以首先需要把这个列表摊平。一种方法是使用 SelectMany 方法。该方法使用的 lambda 表达式为冠军列表中的每一项返回包含 3 项的一个列表。在这个 lambda 表达式的实现中，因为 RacerInfo 包含 FirstName 和 LastName 属性，而收到的集合只包含带有 First、Second 和 Third 属性的一个名称，所以必须拆分字符串。这可以通过扩展方法 FirstName 和 SecondName 完成(代码文件 EnumerableSample/Program.cs)：

```
var racers = Formula1.GetChampionships()
  .SelectMany(cs => new List<RacerInfo>()
  {
  new RacerInfo {
    Year = cs.Year,
    Position = 1,
    FirstName = cs.First.FirstName(),
    LastName = cs.First.LastName()
  },
  new RacerInfo {
    Year = cs.Year,
    Position = 2,
    FirstName = cs.Second.FirstName(),
    LastName = cs.Second.LastName()
  },
  new RacerInfo {
    Year = cs.Year,
    Position = 3,
    FirstName = cs.Third.FirstName(),
    LastName = cs.Third.LastName()
  }
});
```

扩展方法 FirstName 和 SecondName 使用空格字符拆分字符串：

```
public static class StringExtension
{
  public static string FirstName(this string name)
  {
    int ix = name.LastIndexOf(' ');
    return name.Substring(0, ix);
  }
  public static string LastName(this string name)
  {
    int ix = name.LastIndexOf(' ');
    return name.Substring(ix + 1);
  }
}
```

现在就可以连接两个序列。Formula1.GetChampions 返回一个 Racers 列表，racers 变量返回包含年份、比赛结果和赛车手姓名的一个 RacerInfo 列表。仅使用姓氏比较两个集合中的项是不够的。有时候列表中可能同时包含了一个赛车手和他的父亲(如 Damon Hill 和 Graham Hill)，所以必须同时使用 FirstName 和 LastName 进行比较。这是通过为两个列表创建一个新的匿名类型实现的。通过使用 into 子句，第二个集合中的结果被添加到了变量 yearResults 中。对于第一个集合中的每一个赛车手，

都创建了一个 yearResults，它包含了在第二个集合中匹配名和姓的结果。最后，用 LINQ 查询创建了一个包含所需信息的新匿名类型：

```
var q = (from r in Formula1.GetChampions()
         joinr2 in racers on
         new
         {
           FirstName = r.FirstName,
           LastName = r.LastName
         }
         equals
         new
         {
           FirstName = r2.FirstName,
           LastName = r2.LastName
         }
         into yearResults
         select new
         {
           FirstName = r.FirstName,
           LastName = r.LastName,
           Wins = r.Wins,
           Starts = r.Starts,
           Results = yearResults
         });

foreach (var r in q)
{
  Console.WriteLine("{0} {1}", r.FirstName, r.LastName);
  foreach (var results in r.Results)
  {
    Console.WriteLine("{0} {1}.", results.Year, results.Position);
  }
}
```

下面显示了 foreach 循环得到的最终结果。Lewis Hamilton 两次进入前三：2007 年是第二名，2008 年则是冠军。Jenson Button 三次进入前三：2004、2009 和 2011。Sebastian Vettel 两次夺得冠军，并且是 2009 年的第二名：

```
Lewis Hamilton
2007 2.
2008 1.
Jenson Button
2004 3.
2009 1.
2011 2.
Sebastian Vettel
2009 2.
2010 1.
2011 1.
```

11.2.11　集合操作

扩展方法 Distinct()、Union()、Intersect()和 Except()都是集合操作。下面创建一个驾驶法拉利的

一级方程式冠军序列和驾驶迈凯轮的一级方程式冠军序列，然后确定是否有驾驶法拉利和迈凯轮的冠军。当然，这里可以使用 Intersect()扩展方法。

首先获得所有驾驶法拉利的冠军。这只是一个简单的 LINQ 查询，其中使用复合的 from 子句访问 Cars 属性，该属性返回一个字符串对象序列(代码文件 EnumerableSample/Program.cs)。

```
var ferrariDrivers = from r in
                     Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r;
```

现在建立另一个基本相同的查询，但 where 子句的参数不同，以获得所有驾驶迈凯轮的冠军。最好不要再次编写相同的查询。而可以创建一个方法，给它传递参数 car：

```
private static IEnumerable<Racer> GetRacersByCar(string car)
{
  return from r in Formula1.GetChampions()
         from c in r.Cars
         where c == car
         orderby r.LastName
         select r;
}
```

但是，因为该方法不需要在其他地方使用，所以应定义一个委托类型的变量来保存 LINQ 查询。racerByCar 变量必须是一个委托类型，该委托类型需要一个字符串参数，并返回 IEnumerable<Racer>，类似于前面实现的方法。为此，定义了几个泛型委托 Func<>，所以不需要声明自己的委托。把一个 lambda 表达式赋予 racerByCar 变量。lambda 表达式的左边定义了一个 car 变量，其类型是 Func 委托的第一个泛型参数(字符串)。右边定义了 LINQ 查询，它使用该参数和 where 子句：

```
Func<string, IEnumerable<Racer>> racersByCar =
      car => from r in Formula1.GetChampions()
             from c in r.Cars
             where c == car
             orderby r.LastName
             select r;
```

现在可以使用 Intersect()扩展方法，获得驾驶法拉利和迈凯轮的所有冠军：

```
Console.WriteLine("World champion with Ferrari and McLaren");
foreach (var racer in racersByCar("Ferrari").Intersect(
             racersByCar("McLaren")))
{
  Console.WriteLine(racer);
}
```

结果只有一个赛车手 Niki Lauda：

```
World champion with Ferrari and McLaren
Niki Lauda
```

集合操作通过调用实体类的GetHashCode()和Equals()方法来比较对象。对于自定义比较，还可以传递一个实现了IEqualityComparer<T>接口的对象。在这里的示例中，GetChampions()方法总是返回相同的对象，因此默认的比较操作是有效的。如果不是这种情况，就可以重载集合方法来自定义比较操作。

11.2.12 合并

Zip()方法是.NET 4新增的，允许用一个谓词函数把两个相关的序列合并为一个。

首先，创建两个相关的序列，它们使用相同的筛选(意大利)和排序方法。对于合并，这很重要，因为第一个集合中的第一项会与第二个集合中的第一项合并，第一个集合中的第二项会与第二个集合中的第二项合并，依此类推。如果两个序列的项数不同，Zip()方法就在到达较小集合的末尾时停止。

第一个集合中的元素有一个Name属性，第二个集合中的元素有LastName和Starts两个属性。

在racerNames集合上使用Zip()方法，需要把第二个集合(racerNamesAndStarts)作为第一个参数。第二个参数的类型是Func<TFirst, TSecond, TResult>。这个参数实现为一个lambda表达式，它通过参数first接收第一个集合的元素，通过参数second接收第二个集合的元素。其实现代码创建并返回一个字符串，该字符串包含第一个集合中元素的Name属性和第二个集合中元素的 Starts属性(代码文件EnumerableSample/Program.cs)：

```
var racerNames = from r in Formula1.GetChampions()
                 where r.Country == "Italy"
                 orderby r.Wins descending
                 select new
                 {
                   Name = r.FirstName + " " + r.LastName
                 };

var racerNamesAndStarts = from r in Formula1.GetChampions()
                          where r.Country == "Italy"
                          orderby r.Wins descending
                          select new
                          {
                            LastName = r.LastName,
                            Starts = r.Starts
                          };

var racers = racerNames.Zip(racerNamesAndStarts,
    (first, second) => first.Name + ", starts: " + second.Starts);
foreach (var r in racers)
{
  Console.WriteLine(r);
}
```

这个合并的结果是：

```
Alberto Ascari, starts: 32
```

```
Nino Farina, starts: 33
```

11.2.13 分区

扩展方法 Take()和 Skip()等的分区操作可用于分页，例如在第一个页面上只显示 5 个赛车手，在下一个页面上显示接下来的 5 个赛车手等。

在下面的 LINQ 查询中，把扩展方法 Skip()和 Take()添加到查询的最后。Skip()方法先忽略根据页面大小和实际页数计算出的项数，再使用 Take()方法根据页面大小提取一定数量的项(代码文件 EnumerableSample/Program.cs)：

```
int pageSize = 5;

int numberPages = (int)Math.Ceiling(Formula1.GetChampions().Count() /
   (double)pageSize);

for (int page = 0; page < numberPages; page++)
{
  Console.WriteLine("Page {0}", page);

  var racers =
       (from r in Formula1.GetChampions()
        orderby r.LastName, r.FirstName
        select r.FirstName + " " + r.LastName).
        Skip(page * pageSize).Take(pageSize);

  foreach (var name in racers)
  {
    Console.WriteLine(name);
  }
  Console.WriteLine();
}
```

下面输出了前 3 页：

```
Page 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jenson Button

Page 1
Jim Clark
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
Mika Hakkinen

Page 2
Lewis Hamilton
Mike Hawthorn
Damon Hill
Graham Hill
Phil Hill
```

分页在 Windows 或 Web 应用程序中非常有用，可以只给用户显示一部分数据。

这个分页机制的一个要点是，因为查询会在每个页面上执行，所以改变底层的数据会影响结果。在继续执行分页操作时，会显示新对象。根据不同的情况，这对于应用程序可能有利。如果这个操作是不需要的，就可以只对原来的数据源分页，然后使用映射到原始数据上的缓存。

使用 TakeWhile()和 SkipWhile()扩展方法，还可以传递一个谓词，根据谓词的结果提取或跳过某些项。

11.2.14　聚合操作符

聚合操作符(如 Count()、Sum()、Min()、Max()、Average()和 Aggregate())不返回一个序列，而返回一个值。

Count()扩展方法返回集合中的项数。下面的 Count()方法应用于 Racer 的 Years 属性，来筛选赛车手，只返回获得冠军次数超过 3 次的赛车手。因为同一个查询中需要使用同一个计数超过一次，所以使用 let 子句定义了一个变量 numberYears(代码文件 EnumerableSample/Program.cs)：

```
var query = from r in Formula1.GetChampions()
            let numberYears = r.Years.Count()
            where numberYears >= 3
            orderby numberYears descending, r.LastName
            select new
            {
              Name = r.FirstName + " " + r.LastName,
              TimesChampion = numberYears
            };

foreach (var r in query)
{
  Console.WriteLine("{0} {1}", r.Name, r.TimesChampion);
}
```

结果如下：

```
Michael Schumacher 7
Juan Manuel Fangio 5
Alain Prost 4
Jack Brabham 3
Niki Lauda 3
Nelson Piquet 3
Ayrton Senna 3
Jackie Stewart 3
```

Sum()方法汇总序列中的所有数字，返回这些数字的和。下面的 Sum()方法用于计算一个国家赢得比赛的总次数。首先根据国家对赛车手分组，再在新创建的匿名类型中，把 Wins 属性赋予某个国家赢得比赛的总次数。

```
var countries =
```

```
  (from c in
    from r in Formula1.GetChampions()
    group r by r.Country into c
    select new
    {
      Country = c.Key,
      Wins = (from r1 in c
              select r1.Wins).Sum()
    }
    orderby c.Wins descending, c.Country
    select c).Take(5);

foreach (var country in countries)
{
  Console.WriteLine("{0} {1}", country.Country, country.Wins);
}
```

根据获得一级方程式冠军的次数，最成功的国家是：

```
UK 167
Germany 112
Brazil 78
France 51
Finland 42
```

方法 Min()、Max()、Average()和 Aggregate()的使用方式与 Count()和 Sum()相同。Min()方法返回集合中的最小值，Max()方法返回集合中的最大值，Average()方法计算集合中的平均值。对于 Aggregate()方法，可以传递一个 lambda 表达式，该表达式对所有的值进行聚合。

11.2.15 转换操作符

本章前面提到，查询可以推迟到访问数据项时再执行。在迭代中使用查询时，查询会执行。而使用转换操作符会立即执行查询，把查询结果放在数组、列表或字典中。

在下面的例子中，调用 ToList()扩展方法，立即执行查询，得到的结果放在 List<T>类中(代码文件 EnumerableSample/Program.cs)：

```
List<Racer> racers = (from r in Formula1.GetChampions()
                      where r.Starts > 150
                      orderby r.Starts descending
                      select r).ToList();
foreach (var racer in racers)
{
  Console.WriteLine("{0} {0:S}", racer);
}
```

把返回的对象放在列表中并没有这么简单。例如，对于集合类中从赛车到赛车手的快速访问，可以使用新类 Lookup<TKey, TElement>。

Dictionary<TKey, TValue>类只支持一个键对应一个值。在 System.Linq 名称空间的类 Lookup<TKey, TElement>类中，一个键可以对应多个值。这些类详见第 10 章。

使用复合的 from 查询，可以摊平赛车手和赛车序列，创建带有 Car 和 Racer 属性的匿名类型。在返回的 Lookup 对象中，键的类型应是表示汽车的 string，值的类型应是 Racer。为了进行这个选择，可以给 ToLookup()方法的一个重载版本传递一个键和一个元素选择器。键选择器引用 Car 属性，元素选择器引用 Racer 属性。

```
var racers = (from r in Formula1.GetChampions()
              from c in r.Cars
              select new
              {
                Car = c,
                Racer = r
              }).ToLookup(cr => cr.Car, cr => cr.Racer);
if (racers.Contains("Williams"))
{
  foreach (var williamsRacer in racers["Williams"])
  {
    Console.WriteLine(williamsRacer);
  }
}
```

用 Lookup 类的索引器访问的所有 Williams 冠军如下：

```
Alan Jones
Keke Rosberg
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve
```

如果需要在非类型化的集合上(如 ArrayList)使用 LINQ 查询，就可以使用 Cast()方法。在下面的例子中，基于 Object 类型的 ArrayList 集合用 Racer 对象填充。为了定义强类型化的查询，可以使用 Cast()方法。

```
var list = new System.Collections.ArrayList(Formula1.GetChampions()
    as System.Collections.ICollection);

var query = from r in list.Cast<Racer>()
            where r.Country == "USA"
            orderby r.Wins descending
            select r;
foreach (var racer in query)
{
  Console.WriteLine("{0:A}", racer);
}
```

11.2.16 生成操作符

生成操作符 Range()、Empty()和 Repear()不是扩展方法，而是返回序列的正常静态方法。在 LINQ to Objects 中，这些方法可用于 Enumerable 类。

有时需要填充一个范围的数字，此时就应使用 Range()方法。这个方法把第一个参数作为起始值，把第二个参数作为要填充的项数。

```
var values = Enumerable.Range(1, 20);
foreach (var item in values)
{
  Console.Write("{0} ", item);
}
Console.WriteLine();
```

当然，结果如下所示：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Range()方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查询，并返回一个 RangeEnumerator，其中只有一条 yield return 语句，来递增值。

可以把该结果与其他扩展方法合并起来，获得另一个结果，例如，使用 Select()扩展方法：

```
var values = Enumerable.Range(1, 20).Select(n => n * 3);
```

Empty()方法返回一个不返回值的迭代器，它可以用于需要一个集合的参数，其中可以给参数传递空集合。

Repeat()方法返回一个迭代器，该迭代器把同一个值重复特定的次数。

11.3　并行 LINQ

System.Linq 名称空间中包含的类 ParallelEnumerable 可以分解查询的工作，使其分布在多个线程上。尽管 Enumerable 类给 IEnumerable<T>接口定义了扩展方法，但 ParallelEnumerable 类的大多数扩展方法是 ParallelQuery<TSource>类的扩展。一个重要的例外是 AsParallel()方法，它扩展了 IEnumerable<TSource>接口，返回 ParallelQuery<TSource>类，所以正常的集合类可以以平行方式查询。

11.3.1　并行查询

为了说明并行 LINQ(Parallel LINQ，PLINQ)，需要一个大型集合。对于可以放在 CPU 的缓存中的小集合，并行 LINQ 看不出效果。在下面的代码中，用随机值填充一个大型的 int 集合(代码文件 ParallelLinqSample/Program.cs)：

```
static IEnumerable<int> SampleData()
{
  const int arraySize = 100000000;
  var r = new Random();
  return Enumerable.Range(0, arraySize).Select(x => r.Next(140)).ToList();
}
```

现在可以使用 LINQ 查询筛选数据，进行一些计算，获取所筛选数据的平均数。该查询用 where 子句定义了一个筛选器，仅汇总对应值小于 20 的项，接着调用聚合函数 Sum()方法。与前面的 LINQ 查询的唯一区别是，这次调用了 AsParallel()方法。

```
var res = (from x in data.AsParallel()
          where Math.Log(x) < 4
          select x).Average();
```

与前面的 LINQ 查询一样，编译器会修改语法，以调用 AsParallel()、Where()、Select()和 Average()方法。AsParallel()方法用 ParallelEnumerable 类定义，以扩展 IEnumerable<T>接口，所以可以对简单的数组调用它。AsParallel()方法返回 ParallelQuery<TSource>。因为返回的类型，所以编译器选择的 Where()方法是 ParallelEnumerable.Where()，而不是 Enumerable.Where()。在下面的代码中，Select()和 Average()方法也来自 ParallelEnumerable 类。与 Enumerable 类的实现代码相反，对于 ParallelEnumerable 类，查询是分区的，以便多个线程可以同时处理该查询。集合可以分为多个部分，其中每个部分由不同的线程处理，以筛选其余项。完成分区的工作后，就需要合并，获得所有部分的总和。

```
var res = data.AsParallel().Where(x => Math.Log(x) < 4).
                            Select(x => x).Average();
```

运行这行代码会启动任务管理器，这样就可以看出系统的所有 CPU 都在忙碌。如果删除 AsParallel()方法，就不可能使用多个 CPU。当然，如果系统上没有多个 CPU，就不会看到并行版本带来的改进。

11.3.2 分区器

AsParallel()方法不仅扩展了 IEnumerable<T>接口，还扩展了 Partitioner 类。通过它，可以影响要创建的分区。

Partitioner 类用 System.Collection.Concurrent 名称空间定义，并且有不同的变体。Create()方法接受实现了 IList<T>类的数组或对象。根据这一点，以及 Boolean 类型的参数 loadBalance 和该方法的一些重载版本，会返回一个不同的 Partitioner 类型。对于数组，.NET 4 包含派生自抽象基类 OrderablePartitioner<TSource>的 DynamicPartitionerForArray<TSource>类和 StaticPartitionerForArray<TSource>类。

修改 11.3.1 小节中的代码，手工创建一个分区器，而不是使用默认的分区器：

```
var result = (from x in Partitioner.Create(data, true).AsParallel()
          where Math.Log(x) < 4
          select x).Average();
```

也可以调用 WithExecutionMode()和 WithDegreeOfParallelism()方法，来影响并行机制。对于 WithExecutionMode()方法可以传递 ParallelExecutionMode 的一个 Default 值或者 ForceParallelism 值。默认情况下，并行 LINQ 避免使用系统开销很高的并行机制。对于 WithDegreeOf Parallelism()方法，可以传递一个整数值，以指定应并行运行的最大任务数。查询不应使用全部 CPU，这个方法会很有用。

11.3.3 取消

.NET 提供了一种标准方式，来取消长时间运行的任务，这也适用于并行 LINQ。

要取消长时间运行的查询，可以给查询添加 WithCancellation()方法，并传递一个 CancellationToken 令牌作为参数。CancellationToken 令牌从 CancellationTokenSource 类中创建。该查询在单独的线程中

运行，在该线程中，捕获一个 OperationCanceledException 类型的异常。如果取消了查询，就触发这个异常。在主线程中，调用 CancellationTokenSource 类的 Cancel()方法可以取消任务。

```
var cts = new CancellationTokenSource();

Task.Factory.StartNew(() =>
  {
    try
    {
      var res = (from x in data.AsParallel().WithCancellation(cts.Token)
                 where Math.Log(x) < 4
                 select x).Average();
      Console.WriteLine("query finished, sum: {0}", res);
    }
    catch (OperationCanceledException ex)
    {
       Console.WriteLine(ex.Message);
    }
  });

  Console.WriteLine("query started");
  Console.Write("cancel? ");
  string input = Console.ReadLine();
  if (input.ToLower().Equals("y"))
  {
    // cancel!
    cts.Cancel();
  }
```

关于取消和 CancellationToken 令牌的内容详见第 21 章。

11.4 表达式树

在 LINQ to Objects 中，扩展方法需要将一个委托类型作为参数，这样就可以将 lambda 表达式赋予参数。lambda 表达式也可以赋予 Expression<T>类型的参数。C#编译器根据类型给 lambda 表达式定义不同的行为。如果类型是 Expression<T>，编译器就从 lambda 表达式中创建一个表达式树，并存储在程序集中。这样，就可以在运行期间分析表达式树，并进行优化，以便于查询数据源。

下面看看一个前面使用的查询表达式(代码文件 ExpressionTreeSample/Program.cs)：

```
var brazilRacers = from r in racers
                   where r.Country == "Brazil"
                   orderby r.Wins
                   select r;
```

这个查询表达式使用了扩展方法 Where()、OrderBy()和 Select()。Enumerable 类定义了 Where()扩展方法，并将委托类型 Func<T,bool>作为参数谓词。

```
public static IEnumerable<TSource> Where<TSource>(
```

```
    this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

这样，就把 lambda 表达式赋予谓词。这里 lambda 表达式类似于前面介绍的匿名方法。

```
Func<Racer, bool> predicate = r => r.Country == "Brazil";
```

Enumerable 类不是唯一一个定义了扩展方法 Where()的类。Queryable<T>类也定义了 Where()扩展方法。这个类对 Where()扩展方法的定义是不同的：

```
public static IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, bool>> predicate);
```

其中，把 lambda 表达式赋予类型 Expression<T>，该类型的操作是不同的：

```
Expression<Func<Racer, bool>> predicate = r => r.Country == "Brazil";
```

除了使用委托之外，编译器还会把表达式树放在程序集中。表达式树可以在运行期间读取。表达式树从派生自抽象基类 Expression 的类中构建。Expression 类与 Expression<T>不同。继承自 Expression 类的表达式类有 BinaryExpression、ConstantExpression、InvocationExpression、lambdaExpression、NewExpression、NewArrayExpression、TernaryExpression 和 Unary Expression 等。编译器会从 lambda 表达式中创建表达式树。

例如，lambda 表达式 r.Country =="Brazil"使用了 ParameterExpression、MemberExpression、ConstantExpression 和 MethodCallExpression，来创建一个表达式树，并将该树存储在程序集中。之后在运行期间使用这个树，创建一个用于底层数据源的优化查询。

DisplayTree()方法在控制台上图形化地显示表达式树。其中传递了一个 Expression 对象，并根据表达式的类型，把表达式的一些信息写到控制台上。根据表达式的类型，递归地调用 DisplayTree()方法。

在这个方法中，没有处理所有的表达式类型，只处理了在下一个示例表达式中使用的类型。

```
private static void DisplayTree(int indent, string message,
                                Expression expression)
{
  string output = String.Format("{0} {1} ! NodeType: {2}; Expr: {3} ",
     "".PadLeft(indent, '>'), message, expression.NodeType, expression);

  indent++;
  switch (expression.NodeType)
  {
    case ExpressionType.lambda:
      Console.WriteLine(output);
      lambdaExpression lambdaExpr = (lambdaExpression)expression;
      foreach (var parameter in lambdaExpr.Parameters)
      {
        DisplayTree(indent, "Parameter", parameter);
```

```
        }
        DisplayTree(indent, "Body", lambdaExpr.Body);
        break;
      case ExpressionType.Constant:
        ConstantExpression constExpr = (ConstantExpression)expression;
        Console.WriteLine("{0} Const Value: {1}", output, constExpr.Value);
        break;
      case ExpressionType.Parameter:
        ParameterExpression paramExpr = (ParameterExpression)expression;
        Console.WriteLine("{0} Param Type: {1}", output,
                          paramExpr.Type.Name);
        break;
      case ExpressionType.Equal:
      case ExpressionType.AndAlso:
      case ExpressionType.GreaterThan:
        BinaryExpression binExpr = (BinaryExpression)expression;
        if (binExpr.Method != null)
        {
          Console.WriteLine("{0} Method: {1}", output,
                            binExpr.Method.Name);
        }
        else
        {
          Console.WriteLine(output);
        }
        DisplayTree(indent, "Left", binExpr.Left);
        DisplayTree(indent, "Right", binExpr.Right);
        break;
      case ExpressionType.MemberAccess:
        MemberExpression memberExpr = (MemberExpression)expression;
        Console.WriteLine("{0} Member Name: {1}, Type: {2}", output,
          memberExpr.Member.Name, memberExpr.Type.Name);
        DisplayTree(indent, "Member Expr", memberExpr.Expression);
        break;
      default:
        Console.WriteLine();
        Console.WriteLine("{0} {1}", expression.NodeType,
                          expression.Type.Name);
        break;
    }
  }
```

前面已经介绍了用于显示表达式树的表达式。这是一个 lambda 表达式，它有一个 Racer 参数，表达式体提取赢得比赛次数超过 6 次的巴西赛车手：

```
Expression<Func<Racer, bool>> expression =
   r => r.Country == "Brazil" && r.Wins > 6;

DisplayTree(0, "lambda", expression);
```

下面看看结果。lambda 表达式包含一个 Parameter 和一个 AndAlso 节点类型。AndAlso 节点类型的左边是一个 Equal 节点类型，右边是一个 GreaterThan 节点类型。Equal 节点类型的左边是 MemberAccess 节点类型，右边是 Constant 节点类型。

```
lambda! NodeType: lambda; Expr: r => ((r.Country == "Brazil")
   AndAlso (r.Wins > 6))
> Parameter! NodeType: Parameter; Expr: r Param Type: Racer
> Body! NodeType: AndAlso; Expr: ((r.Country == "Brazil")
   AndAlso (r.Wins > 6))
>> Left! NodeType: Equal; Expr: (r.Country == "Brazil") Method: op_Equality
>>> Left! NodeType: MemberAccess; Expr: r.Country
    Member Name: Country, Type: String
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil
>> Right! NodeType: GreaterThan; Expr: (r.Wins > 6)
>>> Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: 6 Const Value: 6
```

使用 Expression<T>类型的一个例子是 ADO.NET Entity Framework 和 WCF 数据服务的客户端提供程序。这些技术用 Expression<T>参数定义了扩展方法。这样，访问数据库的 LINQ 提供程序就可以读取表达式，创建一个运行期间优化的查询，从数据库中获取数据。

11.5 LINQ 提供程序

.NET 包含几个 LINQ 提供程序。LINQ 提供程序为特定的数据源实现了标准的查询操作符。LINQ 提供程序也许会实现比 LINQ 定义的更多扩展方法，但至少要实现标准操作符。LINQ to XML 实现了一些专门用于 XML 的方法，例如，System.Xml.Linq 名称空间中的 Extensions 类定义的 Elements()、Descendants()和 Ancestors()方法。

LINQ 提供程序的实现方案是根据名称空间和第一个参数的类型来选择的。实现扩展方法的类的名称空间必须是开放的，否则扩展类就不在作用域内。在 LINQ to Objects 中定义的 Where()方法的参数和在 LINQ to Entities 中定义的 Where()的方法参数不同。

LINQ to Objects 中的 Where()方法用 Enumerable 类定义：

```
public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

在 System.Linq 名称空间中，还有另一个类实现了操作符 Where。这个实现代码由 LINQ to Entities 使用。这些实现代码在 Queryable 类中可以找到：

```
public static IQueryable<TSource> Where<TSource>(
  this IQueryable<TSource> source,
  Expression<Func<TSource, bool>> predicate);
```

这两个类都在 System.Linq 名称空间的 System.Core 程序集中实现。那么，编译器如何选择使用哪个方法？表达式类型有什么用途？无论是用 Func<TSource, bool>参数传递，还是用 Expression<Func<TSource, bool>>参数传递，lambda 表达式都相同。只是编译器的行为不同，它根据 source 参数来选择。编译器根据其参数选择最匹配的方法。在 ADO.NET Entity Framework 中定义的 ObjectContext 类的 CreateQuery<T>()方法返回一个实现了 IQueryable<TSource>接口的 ObjectQuery<T>对象，因此

Entity Framework 使用 Queryable 类的 Where()方法。

11.6　小结

本章讨论了 LINQ 查询和查询所基于的语言结构，如扩展方法和 lambda 表达式，还列出了各种 LINQ 查询操作符，它们不仅用于筛选数据源，给数据源排序，还用于执行分区、分组、转换、连接等操作。

使用并行 LINQ 可以轻松地并行化运行时间较长的查询。

另一个重要的概念是表达式树。表达式树允许在运行期间构建对数据源的查询，因为表达式树存储在程序集中。表达式树的用法详见第 33 章。LINQ 是一个非常深奥的主题，更多的信息可查阅第 33 章和第 34 章。还可以下载其他第三方提供程序，例如，LINQ to MySQL、LINQ to Amazon、LINQ to Flickr、LINQ to LDAP 和 LINQ to SharePoint。无论使用什么数据源，都可以通过 LINQ 使用相同的查询语法。

第12章

动态语言扩展

本章要点

- 理解 Dynamic Language Runtime
- dynamic 类型
- DLR ScriptRuntime
- 使用 DynamicObject 和 ExpandoObject 创建动态对象

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- DLRHost
- Dynamic
- DynamicFileReader
- ErrorExample

随着 Ruby、Python 等语言的成长，以及 JavaScript 的使用更加广泛，动态编程引起了人们越来越多的兴趣。在.NET Framework 的以前版本中，var 关键字和匿名方法开辟出 C#的“动态编程”路径。在版本 4 中，增加了 dynamic 类型。尽管 C#仍是一种静态的类型化语言，但这些新增内容给它提供了一些开发人员期望的动态功能。

本章介绍 dynamic 类型及其使用规则，并讨论 DynamicObject 的实现方式和使用方式。另外，还将介绍 DynamicObject 的框架实现方式，即 ExpandoObject。

12.1 DLR

C# 4 的动态功能是 Dynamic Language Runtime(动态语言运行时，DLR)的一部分。DLR 是添加到 CLR 的一系列服务，它允许添加动态语言，如 Ruby 和 Python，并使 C#具备和这些动态语言相同的某些动态功能。

在 CodePlex 网站上有一个开源的 DLR 版本，这个版本也包含在.NET 4.5 Framework 中，还增加了对语言实现程序的一些支持。

在.NET Framework 中，DLR 位于 System.Dynamic 名称空间和 System.Runtime.ComplierServices 名称空间的几个类中。

IronRuby 和 IronPython 是 Ruby 和 Python 语言的开源版本，它们使用 DLR。Silverlight 也使用 DLR。通过包含 DLR，可以给应用程序添加脚本编辑功能。脚本运行库允许给脚本传入变量和从脚本传出变量。

12.2 dynamic 类型

dynamic 类型允许编写忽略编译期间的类型检查的代码。编译器假定，给 dynamic 类型的对象定义的任何操作都是有效的。如果该操作无效，则在代码运行之前不会检测该错误，如下面的示例所示：

```
class Program
{
  static void Main(string[] args)
  {
    varstaticPerson = new Person();
    dynamicdynamicPerson = new Person();
    staticPerson.GetFullName("John", "Smith");
    dynamicPerson.GetFullName("John", "Smith");
  }
}

class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string GetFullName()
  {
    returnstring.Concat(FirstName, " ", LastName);
  }
}
```

这个示例没有编译，因为它调用了 staticPerson.GetFullName()方法。因为 Person 对象上的方法不接受两个参数，所以编译器会提示出错。如果注释掉该行代码，这个示例就会编译。如果执行它，就会发生一个运行错误。所抛出的异常是 RuntimeBinderException 异常。RuntimeBinder 对象会在运行时判断该调用，确定 Person 类是否支持被调用的方法。这将在本章后面讨论。

与 var 关键字不同，定义为 dynamic 的对象可以在运行期间改变其类型。注意在使用 var 关键字时，对象类型的确定会延迟。类型一旦确定，就不能改变。动态对象的类型可以改变，而且可以改变多次，这不同于把对象的类型强制转换为另一种类型。在强制转换对象的类型时，是用另一种兼容的类型创建一个新对象。例如，不能把 int 强制转换为 Person 对象。在下面的示例中，如果对象是动态对象，就可以把它从 int 变成 Person 类型：

```
dynamicdyn;
```

```
dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = "This is a string";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);

dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine("{0} {1}", dyn.FirstName, dyn.LastName);
```

执行这段代码可以看出，dyn 对象的类型实际上从 System.Int32 变成 System.String，再变成 Person。如果 dyn 声明为 int 或 string，这段代码就不会编译。

对于 dynamic 类型有两个限制。动态对象不支持扩展方法，匿名函数(lambda 表达式)也不能用作动态方法调用的参数，因此 LINQ 不能用于动态对象。大多数 LINQ 调用都是扩展方法，而 lambda 表达式用作这些扩展方法的参数。

后台上的动态操作

在后台，这些是如何发生的？C#仍是一种静态的类型化语言，这一点没有改变。看看使用 dynamic 类型生成的 IL(中间语言)。

首先，看看下面的示例 C#代码：

```
using System;

namespaceDeCompile
{
    class Program
    {
        static void Main(string[] args)
        {
            StaticClassstaticObject = new StaticClass();
            DynamicClassdynamicObject = new DynamicClass();
            Console.WriteLine(staticObject.IntValue);
            Console.WriteLine(dynamicObject.DynValue);
            Console.ReadLine();
        }
    }

    classStaticClass
    {
       publicintIntValue = 100;
    }

    classDynamicClass
    {
       public dynamic DynValue = 100;
    }
}
```

其中有两个类 StaticClass 和 DynamicClass。StaticClass 类有唯一一个返回 int 的字段。

DynamicClass 有唯一一个返回 dynamic 对象的字段。Main()方法仅创建了这些对象，并输出方法返回的值。该示例非常简单。

现在注释掉 Main()方法中对 DynamicClass 类的引用：

```
static void Main(string[] args)
{
    StaticClas sstaticObject = new StaticClass();
    //DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

使用 ildasm 工具(参见第 19 章)，可以看到给 Main()方法生成的 IL：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size 26 (0x1a)
  .maxstack 1
  .locals init ([0] class DeCompile.StaticClassstaticObject)
  IL_0000: nop
  IL_0001: newobj     instance void DeCompile.StaticClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldloc.0
  IL_0008: ldfld      int32 DeCompile.StaticClass::IntValue
  IL_000d: call       void [mscorlib]System.Console::WriteLine(int32)
  IL_0012: nop
  IL_0013: call       string [mscorlib]System.Console::ReadLine()
  IL_0018: pop
  IL_0019: ret
} // end of method Program::Main
```

这里不讨论 IL 的细节，只看看这段代码，就可以看出其作用。第 0001 行调用了 StaticClass 构造函数，第 0008 行调用了 StaticClass 类的 IntValue 字段。下一行输出了其值。

现在注释掉对 StaticClass 类的引用，取消 DynamicClass 引用的注释：

```
static void Main(string[] args)
{
    //StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

再次编译应用程序，下面是生成的 IL：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size 121 (0x79)
  .maxstack 9
  .locals init ([0] class DeCompile.DynamicClass dynamicObject,
```

```
        [1] class
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo[]
                CS$0$0000)
  IL_0000: nop
  IL_0001: newobj instance void DeCompile.DynamicClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite'1
                 <class [mscorlib]
System.Action'3<class
[System.Core]System.Runtime.CompilerServices.CallSite,class [mscorlib]
System.Type,object>>DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_000c: brtrue.s IL_004d
  IL_000e: ldc.i4.0
  IL_000f: ldstr "WriteLine"
  IL_0014: ldtokenDeCompile.Program
  IL_0019: call class [mscorlib]System.Type
[mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001e: ldnull
  IL_001f: ldc.i4.2
  IL_0020: newarr
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
  IL_0025: stloc.1
  IL_0026: ldloc.1
  IL_0027: ldc.i4.0
  IL_0028: ldc.i4.s 33
  IL_002a: ldnull
  IL_002b: newobj instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpArgumentInfoFlags, string)
  IL_0030: stelem.ref
  IL_0031: ldloc.1
  IL_0032: ldc.i4.1
  IL_0033: ldc.i4.0
  IL_0034: ldnull
  IL_0035: newobj instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpArgumentInfoFlags, string)
  IL_003a: stelem.ref
  IL_003b: ldloc.1
  IL_003c: newobj instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpInvokeMemberBinder::.ctor(valuetypeMicrosoft.CSharp]Microsoft.CSharp
.RuntimeBinder.CSharpCallFlags, string)
class [mscorlib]System.Type,
class [mscorlib]System.Collections.Generic.IEnumerable'1
<class [mscorlib]System.Type>,
class [mscorlib]System.Collections.Generic.IEnumerable'1
<class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo>)
  IL_0041: call class [System.Core]System.Runtime.CompilerServices.CallSite'1
<!0> class [System.Core]System.Runtime.CompilerServices.CallSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Create(class
[System.Core]System.Runtime.CompilerServices.CallSiteBinder)
  IL_0046: stsfld class [System.Core]System.Runtime.CompilerServices.CallSite'1
```

```
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_004b: br.s IL_004d
  IL_004d: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_0052: ldfld !0 class [System.Core]System.Runtime.CompilerServices.CallSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Target
  IL_0057: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_005c: ldtoken [mscorlib]System.Console
  IL_0061: call class [mscorlib]System.Type
[mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0066: ldloc.0
  IL_0067: ldfld object DeCompile.DynamicClass::DynValue
  IL_006c: callvirt instance void class [mscorlib]System.Action'3
          <class [System.Core]System.Runtime.CompilerServices.CallSite, class
          [mscorlib]System.Type,object>::Invoke(!0,!1,!2)
  IL_0071: nop
  IL_0072: call string [mscorlib]System.Console::ReadLine()
  IL_0077: pop
  IL_0078: ret
} // end of method Program::Main
```

显然，C#编译器做了许多工作，以支持动态类型。在生成的代码中，会看到对 System.Runtime.CompilerServices.CallSite 类和 System.Runtime.CompilerServices.CallSiteBinder 类的引用。

CallSite 是在运行期间处理查找操作的类型。在运行期间调用动态对象时，必须找到该对象，看看其成员是否存在。CallSite 会缓存这个信息，这样查找操作就不需要重复执行。没有这个过程，循环结构的性能就有问题。

CallSite 完成了成员查找操作后，就调用 CallSiteBinder()方法。它从 CallSite 中提取信息，并生成表达式树，来表示绑定器绑定的操作。

显然这需要做许多工作。优化非常复杂的操作时要格外小心。显然，使用 dynamic 类型是有用的，但它是有代价的。

12.3　包含 DLR ScriptRuntime

假定能给应用程序添加脚本编辑功能，并给脚本传入数值和从脚本传出数值，使应用程序可以利用脚本完成工作。这些都是在应用程序中包含 DLR 的 ScriptRuntime 而提供的功能。目前，

IronPython、IronRuby 和 JavaScript 都支持包含在应用程序中的脚本语言。

有了 ScriptRuntime，就可以执行存储在文件中的代码段或完整的脚本。可以选择合适的语言引擎，或者让 DLR 确定使用什么引擎。脚本可以在自己的应用程序域或者在当前的应用程序域中创建。不仅可以给脚本传入数值并从脚本中传出数值，还可以在脚本中调用在动态对象上创建的方法。

这种灵活性为包含 ScriptRuntime 提供了无数种用法。下面的示例说明了使用 ScriptRuntime 的一种方式。假定有一个购物车应用程序，它的一个要求是根据某种标准计算折扣。这些折扣常常随着新销售策略的启动和完成而变化。处理这个要求有许多方式，本例将说明如何使用 ScriptRuntime 和少量 Python 脚本达到这个要求。

为了简单起见，本例是一个 Windows 客户端应用程序。它也可以是一个大型 Web 应用程序或任何其他应用程序的一部分。图 12-1 显示了这个应用程序的样例屏幕。

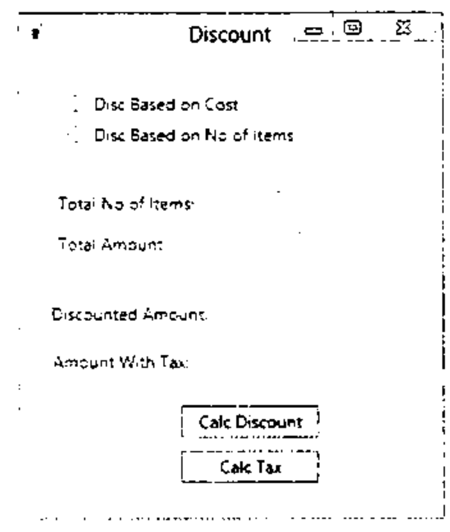


图 12-1

该应用程序提取所购买的物品数量和物品的总价，并根据所选的单选按钮使用某个折扣。在实际的应用程序中，系统使用略微复杂的方式确定要使用的折扣，但对于本例，单选按钮就足够了。

下面是计算折扣的代码：

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    stringscriptToUse;
    if (CostRadioButton.IsChecked.Value)
    {
       scriptToUse = "AmountDisc.py";
    }
    else
    {
       scriptToUse = "CountDisc.py";
    }
    ScriptRuntimescriptRuntime = ScriptRuntime.CreateFromConfiguration();
    ScriptEnginepythEng = scriptRuntime.GetEngine("Python");
    ScriptSource source = pythEng.CreateScriptSourceFromFile(scriptToUse);
    ScriptScope scope = pythEng.CreateScope();
    scope.SetVariable("prodCount", Convert.ToInt32(totalItems.Text));
    scope.SetVariable("amt", Convert.ToDecimal(totalAmt.Text));
    source.Execute(scope);
    label5.Content = scope.GetVariable("retAmt").ToString();
}
```

第一部分仅确定要应用折扣的脚本 AmountDisc.py 或 CountDisc.py。AmountDisc.py 根据购买的金额计算折扣。

```
discAmt = .25
retAmt = amt
if amt > 25.00:
```

```
  retAmt = amt-(amt*discAmt)
```

能打折的最低购买金额是$25。如果购买金额小于这个值，就不计算折扣，否则就使用 2.5%的折扣率。

CountDisc.py 根据购买的物品数量计算折扣：

```
discCount = 5
discAmt = .1
retAmt = amt
ifprodCount > discCount:
  retAmt = amt-(amt*discAmt)
```

在这个 Python 脚本中，购买的物品数量必须大于 5，才能给总价应用 10%的折扣率。

下一部分是启动 ScriptRuntime 环境。这需要执行 4 个特定的步骤：创建 ScriptRuntime 对象、设置合适的 ScriptEngine、创建 ScriptSource 以及创建 ScriptScope。

ScriptRuntime 对象是起点，也是包含 ScriptRuntime 的基础。它拥有包含环境的全局状态。ScriptRuntime 对象使用 CreateFromConfiguration()静态方法创建。app.config 文件如下所示：

```
<configuration>
  <configSections>
    <section
    name="microsoft.scripting"
    type="Microsoft.Scripting.Hosting.Configuration.Section,
        Microsoft.Scripting,
        Version=0.9.6.10,
        Culture=neutral,
        PublicKeyToken=null"
        requirePermission="false" />
  </configSections>

  <microsoft.scripting>
    <languages>
      <language
        names="IronPython;Python;py"
        extensions=".py"
        displayName="IronPython 2.6 Alpha"
        type="IronPython.Runtime.PythonContext,
              IronPython,
              Version=2.6.0.1,
              Culture=neutral,
              PublicKeyToken=null" />
    </languages>
  </microsoft.scripting>
</configuration>
```

这段代码定义了“microsoft.scripting”的一部分，设置了 IronPython 语言引擎的几个属性。

接着，从 ScriptRuntime 中获取一个对 ScriptEngine 的引用。在本例中，指定需要 Python 引擎，但 ScriptRuntime 可以自己确定这一点，因为脚本的扩展名是 py。

ScriptEngine 完成了执行脚本代码的工作。执行文件或代码段中的脚本有几种方法。ScriptEngine 还提供了 ScriptSource 和 ScriptScope。

ScriptSource 对象允许访问脚本，它表示脚本的源代码。有了它，就可以操作脚本的源代码。从

磁盘上加载它，逐行解析它，甚至把脚本编译到 CompiledCode 对象中。如果多次执行同一个脚本，这就很方便。

ScriptScope 对象实际上是一个名称空间。要给脚本传入值或从脚本传出值，应把一个变量绑定到 ScriptScope 上。本例调用 SetVariable 方法给 Python 脚本传入 prodCount 变量和 amt 变量。它们是 totalItems 文本框和 totalAmt 文本框中的值。计算出来的折扣使用 GetVariable()方法从脚本中检索。在本例中，retAmt 变量包含了我们需要的值。

在 CalcTax 按钮中，调用了 Python 对象上的方法。CalcTax.py 脚本是一个非常简单的方法，它接受一个输入值，加上 7.5%的税，再返回新值。代码如下：

```
defCalcTax(amount):
return amount*1.075
```

下面是调用 CalcTax()方法的 C#代码：

```
private void button2_Click(object sender, RoutedEventArgs e)
{
  ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
  dynami ccalcRate = scriptRuntime.UseFile("CalcTax.py");
  label6.Content = calcRate.CalcTax(Convert.ToDecimal(label5.Content)).ToString();
}
```

这是一个非常简单的过程。这里再次使用与前面相同的配置设置创建了 ScriptRuntime 对象。calRate 是一个 ScriptScope 对象，它定义为动态对象，以便轻松地调用 CalcTax()方法。这是使用动态类型简化编程工作的一个示例。

12.4 DynamicObject 和 ExpandoObject

如果要创建自己的动态对象，该怎么办？这有两种方法：从 DynamicObject 中派生，或者使用 ExpandoObject。使用 DynamicObject 需要做的工作较多，因为必须重写几个方法。ExpandoObject 是一个可立即使用的密封类。

12.4.1 DynamicObject

考虑一个表示人的对象。一般应定义名字、中间名和姓氏等属性。现在假定要在运行期间构建这个对象，且系统事先不知道该对象有什么属性或该对象可能支持什么方法。此时就可以使用基于 DynamicObject 的对象。需要这类功能的场合几乎没有，但到目前为止，C#语言还没有提供该功能。

先看看 DynamicObject：

```
classWroxDynamicObject : DynamicObject
{
    Dictionary<string, object> _dynamicData = new Dictionary<string, object>();

    public override boolTryGetMember(GetMemberBinder binder, out object result)
    {
        bool success = false;
        result = null;
```

```
        if (_dynamicData.ContainsKey(binder.Name))
        {
            result = _dynamicData[binder.Name];
            success = true;
        }
        else
        {
            result = "Property Not Found!";
            success = false;
        }
        return success;
    }

    public override boolTrySetMember(SetMemberBinder binder, object value)
    {
        _dynamicData[binder.Name] = value;
        return true;
    }

    public override boolTryInvokeMember(InvokeMemberBinder binder,
                                        object[] args,
                                        out object result)
    {
        dynamic method = _dynamicData[binder.Name];
        result = method((DateTime)args[0]);
        return result != null;
    }
}
```

在这个示例中，重写了 3 个方法 TrySetMember()、TryGetMember()和 TryInvokeMember()。

TrySetMember()方法给对象添加了新方法、属性或字段。本例把成员信息存储在一个 Dictionary 对象中。传送给 TrySetMember()方法的 SetMemberBinder 对象包含 Name 属性，它用于标识 Dictionary 中的元素。

TryGetMember()方法根据 GetMemberBinder 对象的 Name 属性检索存储在 Dictionary 中的对象。

下面的代码使用了刚才新建的动态对象：

```
dynamicwroxDyn = new WroxDynamicObject();
wroxDyn.FirstName = "Bugs";
wroxDyn.LastName = "Bunny";
Console.WriteLine(wroxDyn.GetType());
Console.WriteLine("{0} {1}", wroxDyn.FirstName, wroxDyn.LastName);
```

看起来很简单，但在哪里调用了重写的方法？正是.NET Framework 帮助完成了调用。DynamicObject 处理了绑定，我们只需引用 FirstName 和 LastName 属性即可，就好像它们一直存在一样。

添加方法很简单。可以使用上例中的 WroxDynamicObject，给它添加 GetTomorrowDate()方法，该方法接受一个 DateTime 对象为参数，返回表示第二天的日期字符串。代码如下：

```
dynamicwroxDyn = new WroxDynamicObject();
Func<DateTime, string>GetTomorrow = today =>today.AddDays(1).ToShortDateString();
wroxDyn.GetTomorrowDate = GetTomorrow;
```

```
Console.WriteLine("Tomorrow is {0}", wroxDyn.GetTomorrowDate(DateTime.Now));
```

这段代码使用 Func<T, TResult>创建了委托 GetTomorrow。该委托表示的方法调用了 AddDays，给传入的 Date 加上一天，返回得到的日期字符串。接着把委托设置为 wroxDyn 对象上的 GetTomorrowDate()方法。最后一行调用新方法，并传递今天的日期。动态功能再次发挥了作用，对象上有了一个有效的方法。

12.4.2 ExpandoObject

ExpandoObject 的工作方式类似于上一节创建的 WroxDynamicObject，区别是不必重写方法，如下面的代码示例所示：

```
static void DoExpando()
{
    dynamicexpObj = new ExpandoObject();
    expObj.FirstName = "Daffy";
    expObj.LastName = "Duck";
    Console.WriteLine(expObj.FirstName + " " + expObj.LastName);
    Func<DateTime, string>GetTomorrow = today =>today.AddDays(1).ToShortDateString();
    expObj.GetTomorrowDate = GetTomorrow;
    Console.WriteLine("Tomorrow is {0}", expObj.GetTomorrowDate(DateTime.Now));

    expObj.Friends = new List<Person>();
    expObj.Friends.Add(new Person() { FirstName = "Bob", LastName = "Jones" });
    expObj.Friends.Add(new Person() { FirstName = "Robert", LastName = "Jones" });
    expObj.Friends.Add(new Person() { FirstName = "Bobby", LastName = "Jones" });

    foreach (Person friend in expObj.Friends)
    {
       Console.WriteLine(friend.FirstName + " " + friend.LastName);
    }
}
```

注意这段代码与前面的代码几乎完全相同，也添加了 FirstName 和 LastName 属性，以及 GetTomorrow 函数，但它还多做了一件事——把一个 Person 对象集合添加为对象的一个属性。

初看起来，这似乎与使用 dynamic 类型没有区别。但其中有两个微妙的区别非常重要。第一，不能仅创建 dynamic 类型的空对象。必须把 dynamic 类型赋予某个对象，例如，下面的代码是无效的：

```
dynamicdynObj;
dynObj.FirstName = "Joe";
```

与前面的示例一样，此时可以使用 ExpandoObject。

第二，因为 dynamic 类型必须赋予某个对象，所以如果执行 GetType 调用，它就会报告赋予了 dynamic 类型的对象类型。所以如果把它赋予 int，GetType 就报告它是一个 int。这不适用于 ExpandoObject 或派生自 DynamicObject 的对象。

如果需要控制动态对象中属性的添加和访问，则使该对象派生自 DynamicObject 是最佳选择。使用 DynamicObject，可以重写几个方法，准确地控制对象与运行库的交互方式。而对于其他情况，就应使用 dynamic 类型或 ExpandoObject。

下面是使用 dynamic 类型和 ExpandoObject 的另一个例子。假设需求是开发一个通用的逗号分

隔值(CSV)文件的解析工具。从一个扩展到另一个扩展时，不知道文件中将包含什么数据，只知道值之间是用逗号分隔的，并且第一行包含字段名。

首先，打开文件并读入数据流。这可以用一个简单的辅助方法完成：

```
privateStreamReader OpenFile(string fileName)
{
  if(File.Exists(fileName))
  {
    return new StreamReader(fileName);
  }
  return null;
}
```

这段代码打开文件，并创建一个新的 StreamReader 来读取文件内容。

接下来要获取字段名。方法很简单：读取文件的第一行，使用 Split 函数创建字段名的一个字符串数组。

```
string[] headerLine = fileStream.ReadLine().Split(',');
```

接下来的部分很有趣：读入文件的下一行，就像处理字段名那样创建一个字符串数组，然后创建动态对象。具体代码如下所示：

```
varretList = new List<dynamic>();
while (fileStream.Peek() > 0)
{
  string[] dataLine = fileStream.ReadLine().Split(',');
  dynamicdynamicEntity = new ExpandoObject();
  for(inti=0;i<headerLine.Length;i++)
  {
    ((IDictionary<string,object>)dynamicEntity).Add(headerLine[i],dataLine[i]);
  }
  retList.Add(dynamicEntity);
}
```

有了字段名和数据元素的字符串数组后，创建一个新的 ExpandoObject，在其中添加数据。注意，代码中将 ExpandoObject 强制转换为 Dictionary 对象。用字段名作为键，数据作为值。然后，把这个新对象添加到所创建的 retList 对象中，返回给调用该方法的代码。

这样做的好处是有了一段可以处理传递给它的任何数据的代码。这里唯一的要求是确保第一行是字段名，并且所有的值是用逗号分隔的。可以把这个概念扩展到其他文件类型，甚至 DataReader。

12.5 小结

本章介绍了 dynamic 类型，它改变了我们看待 C#编程的方式。通过使用 ExpandoObject 代替多个对象，代码量会显著减少。另外，通过使用 DLR 及添加 Python 或 Ruby 等脚本语言，可以创建多态性更好的应用程序，改变它们十分简单，并且不需要重新编译。

动态开发越来越流行，它允许执行在静态的类型化语言中很难实现的操作。dynamic 类型和 DLR 允许 C#程序员使用某些动态功能。

第13章

异 步 编 程

本章要点

- 异步编程的重要性
- 异步模式
- async 和 await 关键字基础
- 创建和使用异步方法
- 异步方法的错误处理

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- AsyncPatterns(异步模式)
- Foundations(async 和 await 关键字)
- ErrorHandling(异步方法的错误处理)

13.1 异步编程的重要性

C# 5.0 最重要的改进，就是提供了更强大的异步编程。C# 5.0 仅增加两个新的关键字：async 和 await。这两个关键字将是本章的重点。

使用异步编程，方法调用是在后台运行(通常在线程或任务的帮助下)，并且不会阻塞调用线程。

本章将学习 3 种不同模式的异步编程：异步模式、基于事件的异步模式和新增加的基于任务的异步模式(TAP)。TAP 是利用 async 和 await 关键字来实现的。通过这里的比较，将认识到新增加的基于任务的异步模式的真正优势。

讨论过不同的模式之后，通过创建任务和使用异步方法，来介绍异步编程的基础知识。还会论述延续任务和同步上下文的相关内容。

与异步任务一样，错误处理也需要特别重视。有些错误要采用不同的处理方式。

在本章的最后，讨论了如何取消正在执行的任务。如果后台任务执行时间较长，就有可能需要取消任务。对于如何取消，也将在本章学习到相关内容。

第21章介绍了并行编程的相关内容。

如果应用程序没有立刻响应用户的请求，会让用户反感。用鼠标操作，我们习惯了出现延迟，过去几十年都是这样操作的。有了触摸UI，应用程序要求立刻响应用户的请求。否则，用户就会不断重复同一个动作。

因为在旧版本的.NET Framework中用异步编程非常不方便，所以并没有总是这样做。Visual Studio 2010是经常阻塞UI线程的应用程序之一。例如，在Visual Studio 2010中，打开一个包含数百个项目的解决方案，这意味可能需要等待很长的时间。自从Visual Studio 2012以来，情况就不一样了，因为项目都是在后台异步加载的，并且选中的项目会优先加载。这种加载行为是异步编程内置到Visual Studio 2012中带来的重要变化之一。同样，在Visual Studio 2010中，对话框经常会不响应，这在Visual Studio 2012和2013中也不太可能发生了。

很多.NET Framework的API都提供了同步版本和异步版本。因为同步版本的API用起来更为简单，所以常常在不适合使用时也用了同步版本的API。在新的Windows运行库(WinRT)中，如果一个API调用时间超过40ms，就只能使用其异步版本。现在，在.NET 4.5中，异步编程和同步编程一样简单，所以用异步API应该不会有任何的障碍。

13.2 异步模式

在学习新的async和await关键字之前，先看看.NET Framework的异步模式。从.NET 1.0开始就提供了异步特性，而且.NET Framework的许多类都实现了一个或者多个异步模式。委托类型也实现了异步模式。

因为在Windows Forms和WPF中，用异步模式更新界面非常复杂，所以.NET 2.0推出了基于事件的异步模式。在这种模式中，事件处理程序是被拥有同步上下文的线程调用，所以更新界面很容易用这种模式处理。在此之前，这种模式也称为异步组件模式。

现在，在.NET 4.5中，推出了另外一种新的方式来实现异步编程：基于任务的异步模式(TAP)。这种模式是基于.NET 4.0中新增的Task类型，并通过async和await关键字来使用编译器功能。

为了了解async和await关键字的优势，第一个示例应用程序利用Windows Presentation Foundation(WPF)和网络编程来阐述异步编程的概况。如果没有WPF和网络编程的经验，也不用失望。你同样能够按照这里的要领，掌握异步编程是如何实现的。下面的示例演示了异步模式之间的差异。看完这些之后，通过一些简单的控制台应用程序，将学会异步编程的基础知识。

第35章和第36章详细介绍了WPF。第26章讨论了网络编程。

下面的示例WPF应用程序演示了异步模式之间的差异，它利用了类库中的类型。该应用程序用Bing和Flickr的服务在网络上寻找图片。用户可以输入一个搜索关键词来找到图片，该搜索关键词通过一个简单HTTP请求发送到Bing和Flickr服务。

界面的设计来自 Visual Studio 设计器，如图 13-1 所示。在屏幕上方是一个文本输入框，紧接着是几个开始搜索按钮或清除结果列表的按钮。左下方的控制区是一个 ListBox 控件，用于显示所有找到的图片。右侧是一个 Image 控件，用更高的分辨率显示 ListBox 控件中被选择的图片。

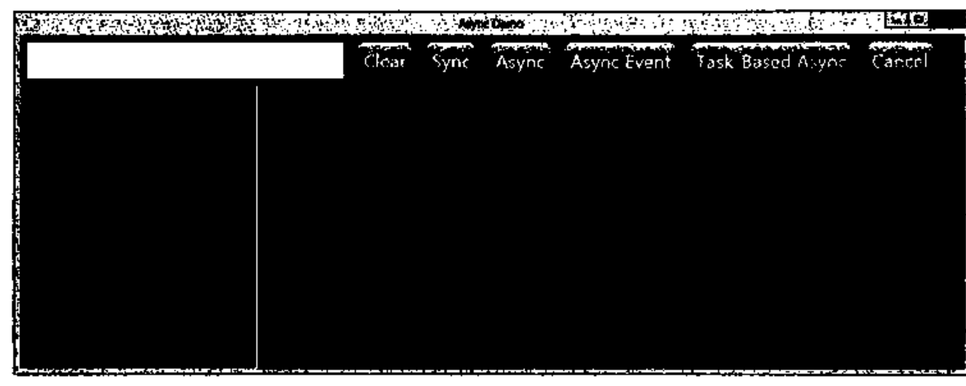


图 13-1

为了能够理解示例应用程序，先从包含几个辅助类的类库 AsyncLib 开始。这些类用于该 WPF 应用程序。

SearchItemResult 类表示结果集合中的一项，用于显示图片、标题和图片来源。该类仅定义了简单属性：Title、Url、ThumbnailUrl 和 Source。ThumbnailUrl 属性用于引用缩略图片，Url 属性包含到更大尺寸图片的链接。Title 属性包含描述图片的文本。BindableBase 是 SearchItemResult 的基类。该基类通过实现 INotifyPropertyChanged 接口实现通知机制，WPF 用其通过数据绑定进行更新(代码文件 AsyncLib/SearchItemResult.cs)：

```
namespace Wrox.ProCSharp.Async
{
  public class SearchItemResult : BindableBase
  {
    private string title;
    public string Title
    {
      get { return title; }
      set { SetProperty(ref title, value); }
    }

    private string url;
    public string Url
    {
      get { return url; }
      set { SetProperty(ref url, value); }
    }

    private string thumbnailUrl;
    public string ThumbnailUrl
    {
      get { return thumbnailUrl; }
      set { SetProperty(ref thumbnailUrl, value); }
    }
```

```
      private string source;
      public string Source
      {
        get { return source; }
        set { SetProperty(ref source, value); }
      }
    }
  }
```

SearchInfo 类是另外一个用于数据绑定的类。SearchTerm 属性包含用于搜索该类型图片的用户输入。List 属性返回所有找到的图片列表，其类型为 SearchItemResult(代码文件 AsyncLib/SearchInfo.cs)：

```
using System.Collections.ObjectModel;

namespace Wrox.ProCSharp.Async
{
  public class SearchInfo : BindableBase
  {
    public SearchInfo()
    {
      list = new ObservableCollection<SearchItemResult>();
      list.CollectionChanged += delegate { OnPropertyChanged("List"); };
    }

    private string searchTerm;
    public string SearchTerm
    {
      get { return searchTerm; }
      set { SetProperty(ref searchTerm, value); }
    }

    privateObservableCollection<SearchItemResult> list;
    public ObservableCollection<SearchItemResult> List
    {
      get
      {
        return list;
      }
    }
  }
}
```

在 XAML 代码中，TextBox 控件用于输入搜索关键词。该控件绑定到 SearchInfo 类型的 SearchTerm 属性。几个按钮控件用于激活事件处理程序。例如，Sync 按钮调用 OnSearchSync 方法(XAML 文件 AsyncPatterns/MainWindow.xaml)：

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
  <StackPanel.LayoutTransform>
    <ScaleTransform ScaleX="2" ScaleY="2" />
  </StackPanel.LayoutTransform>
  <TextBox Text="{Binding SearchTerm}" Width="200" Margin="4" />
  <Button Click="OnClear">Clear</Button>
  <Button Click="OnSearchSync">Sync</Button>
```

```
    <Button Click="OnSeachAsyncPattern">Async</Button>
    <Button Click="OnAsyncEventPattern">Async Event</Button>
    <Button Click="OnTaskBasedAsyncPattern">Task Based Async</Button>
  </StackPanel>
```

在 XAML 代码的第二部分包含一个 ListBox 控件。为了在 ListBox 控件中进行特殊显示，使用了 ItemTemplate。每个 ItemTemplate 包含两个 TextBlock 控件和一个 Image 控件。该 ListBox 控件绑定到 SearchInfo 类的 List 属性，ItemTemplate 中控件的属性分别绑定到 SearchItemResult 类型的属性：

```
<Grid Grid.Row="1">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="3*" />
  </Grid.ColumnDefinitions>
  <ListBox Grid.IsSharedSizeScope="True" ItemsSource="{Binding List}"
      Grid.Column="0" IsSynchronizedWithCurrentItem="True"
      Background="Black">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="ItemTemplateGroup" />
          </Grid.ColumnDefinitions>
          <StackPanel HorizontalAlignment="Stretch" Orientation="Vertical"
              Background="{StaticResource linearBackgroundBrush}">
            <TextBlock Text="{Binding Source}"Foreground="White" />
            <TextBlock Text="{Binding Title}"Foreground="White" />
            <Image HorizontalAlignment="Center"
                Source="{Binding ThumbnailUrl}"Width="100" />
          </StackPanel>
        </Grid>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
  <GridSplitter Grid.Column="1" Width="3" HorizontalAlignment="Left" />
  <Image Grid.Column="1" Source="{Binding List/Url}"/>
</Grid>
```

现在来看看 BingRequest 类。该类包含如何向 Bing 服务发出请求的一些信息。该类的 Url 属性返回一个用于请求图片的 URL 字符串。请求字符串由搜索关键词、请求图片的数量(Count)和跳过图片的数量(Offset)构成。Bing 是需要身份认证的。用户 ID 用 AppId 来定义，并使用返回 NetworkCredential 对象的 Credentials 属性。要运行应用程序，需要使用 Windows Azure Marketplace 注册，并申请一个 Bing Search API。编写本书时，Bing 提供的免费事务每月多达 5000 次，这足够运行示例应用程序。每次搜索是一个事务。注册 Bing Search API 的链接为 https://datamarket.azure.com/dataset/bing/search。注册获取 AppID 后，需要复制 AppID，将其添加到 BingRequest 类中。

用创建的 URL 将请求发送到 Bing 之后，Bing 会返回一个 XML 字符串。BingRequest 类的 Parse 方法会解析该 XML 字符串，并返回 SearchItemResult 对象的集合(代码文件 AsyncLib/BingRequest.cs)：

BingRequest类和FlickrRequest类的Parse方法利用了LINQ to XML。第34章将讨论如何使用LINQ to XML。

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

namespace Wrox.ProCSharp.Async
{
  public class BingRequest : IImageRequest
  {
    private const string AppId = "enter your Bing AppId here";

    public BingRequest()
    {
      Count = 50;
      Offset = 0;
    }
    private string searchTerm;
    public string SearchTerm
    {
      get { return searchTerm; }
      set { searchTerm = value; }
    }

    public ICredentials Credentials
    {
      get
      {
        return new NetworkCredentials(AppId, AppId);
      }
    }

    public string Url
    {
      get
      {
        return string.Format("https://api.datamarket.azure.com/" +
          "Data.ashx/Bing/Search/v1/Image?Query=%27{0}%27&" +
          "$top={1}&$skip={2}&$format=Atom",
          SearchTerm, Count, Offset);
      }
    }

    public int Count { get; set; }
    public int Offset { get; set; }

    public IEnumerable<SearchItemResult> Parse(string xml)
    {
      XElement respXml = XElement.Parse(xml);
      // XNamespace atom = XNamespace.Get("http://www.w3.org/2005/Atom");
```

```
        XNamespace d = XNamespace.Get(
          "http://schemas.microsoft.com/ado/2007/08/dataservices");
        XNamespace m = XNamespace.Get(
          "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");

        return (from item in respXml.Descendants(m + "properties")
                select new SearchItemResult
                {
                  Title = new string(item.Element(d +
                    "Title").Value.Take(50).ToArray()),
                  Url = item.Element(d + "MediaUrl").Value,
                  ThumbnailUrl = item.Element(d + "Thumbnail").
                    Element(d + "MediaUrl").Value,
                  Source = "Bing"
                  }).ToList();
      }
    }
  }
```

BingRequest 类和 FlickrRequest 类都实现了 IImageRequest 接口。该接口定义了 SearchTerm 属性、Url 属性和 Parse 方法，Parse 方法很容易迭代两个图片服务提供商返回的结果(代码文件 AsyncLib/IImageRequest.cs)：

```
using System;
using System.Collections.Generic;
using System.Net;

namespace Wrox.ProCSharp.Async
{
  public interface IImageRequest
  {
    string SearchTerm { get; set; }
    string Url { get; }

    IEnumerable<SearchItemResult>Parse(string xml);

    ICredentials Credentials { get; }
  }
}
```

FlickrRequest 和 BingRequest 类非常相似。它仅是用搜索关键词创建了不同的 URL 来请求图片，Parse 方法的实现也不同，因为从 Flickr 返回的 XML 与从 Bing 返回的 XML 不同。和 Bing 一样，也需要为 Flickr 注册一个 AppId，注册链接为 http://www.flickr.com/services/apps/create/ apply/。

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

namespace Wrox.ProCSharp.Async
{
  public class FlickrRequest : IImageRequest
  {
    private const string AppId = "Enter your Flickr AppId here";
```

```
    public FlickrRequest()
    {
      Count = 50;
      Page = 1;
    }

    private string searchTerm;
    public string SearchTerm
    {
      get { return searchTerm; }
      set { searchTerm = value; }
    }

    public string Url
    {
      get
      {
        return string.Format("http://api.flickr.com/services/rest?" +
            "api_key={0}&method=flickr.photos.search&content_type=1&" +
            "text={1}&per_page={2}&page={3}", AppId, SearchTerm, Count, Page);
      }
    }

    public ICredentials Credentials
    {
     get { return null; }
    }

    public int Count { get; set; }
    public int Page { get; set; }

    public IEnumerable<SearchItemResult> Parse(string xml)
    {
      XElement respXml = XElement.Parse(xml);
      return (from item in respXml.Descendants("photo")
              select new SearchItemResult
              {
                Title = new string(item.Attribute("title").Value.
                    Take(50).ToArray()),
                Url = string.Format("http://farm{0}.staticflickr.com/" +
                    "{1}/{2}_{3}_z.jpg",
                  item.Attribute("farm").Value, item.Attribute("server").Value,
                  item.Attribute("id").Value, item.Attribute("secret").Value),
                    ThumbnailUrl = string.Format("http://farm{0}." +
                    "staticflickr.com/{1}/{2}_{3}_t.jpg",
                  item.Attribute("farm").Value,
                  item.Attribute("server").Value,
                  item.Attribute("id").Value,
                  item.Attribute("secret").Value),
                Source = "Flickr"
              }).ToList();
    }
  }
}
```

现在，只需要连接到 WPF 应用程序和类库中的类型。在 MainWindow 类的构造函数中，创建了 SearchInfo 实例，并将窗口的 DataContext 属性设置为这个实例。现在，可以用之前的 XAML 代码进行数据绑定，如下所示(代码文件 AsyncPatterns/MainWindow.xaml.cs)：

```
public partial class MainWindow : Window
{
  private SearchInfo searchInfo;

  public MainWindow()
  {
    InitializeComponent();
    searchInfo = new SearchInfo();
    this.DataContext = searchInfo;
  }
```

MainWindow 类也包含一个辅助方法 GetSearchRequests，它返回一个由 BingRequest 类型和 FlickrRequest 类型构成的 IImageRequest 对象集合。如果仅注册一个服务，则可以修改代码，仅返回注册的那个服务。当然，也可以创建 IImageRequest 类型的其他服务，例如，使用 Google 或 Yahoo。然后，把这些请求类型添加到返回的集合中：

```
private IEnumerable<IImageRequest> GetSearchRequests()
{
  return new List<IImageRequest>
  {
    new BingRequest { SearchTerm = searchInfo.SearchTerm },
    new FlickrRequest { SearchTerm = searchInfo.SearchTerm}
  };
}
```

13.2.1　同步调用

现在，一切准备就绪，开始同步调用这些服务。Sync 按钮的单击处理程序 OnSearchSync，遍历 GetSearchRequests 方法返回的所有搜索请求，并且用 Url 属性发出 WebClient 类的 HTTP 请求。DownloadString 方法会阻塞，直到接收到结果。得到的 XML 赋给 resp 变量。通过 Parse 方法分析 XML 内容，返回一个 SearchItemResult 对象的集合。然后，集合的每一项添加到 searchInfo 中包含的列表(代码文件 AsyncPatterns/MainWindow.xaml.cs)：

```
private void OnSearchSync(object sender, RoutedEventArgs e)
{
  foreach (var req in GetSearchRequests())
  {
    var client = new WebClient();
    client.Credentials = req.Credentials;
    string resp = client.DownloadString(req.Url);
    IEnumerable<SearchItemResult> images = req.Parse(resp);
    foreach (var image in images)
    {
      searchInfo.List.Add(image);
    }
  }
}
```

运行该应用程序(如图13-2所示)，用户界面被阻塞，直到OnSearchSync方法完成对Bing和Filckr的网络调用，并完成结果分析。完成这些调用所需的时间取决于网络速度，以及Bing与Flickr当前的工作量。但是，对于用户来说，等待都是不愉快的。

图13-2

因此，有必要使用异步调用。

13.2.2 异步模式

进行异步调用的方式之一是使用异步模式。异步模式定义了BeginXXX方法和EndXXX方法。例如，如果有一个同步方法DownloadString，其异步版本就是BeginDownloadString和EndDownloadString方法。BeginXXX方法接受其同步方法的所有输入参数，EndXXX方法使用同步方法的所有输出参数，并按照同步方法的返回类型来返回结果。使用异步模式时，BeginXXX方法还定义了一个AsyncCallback参数，用于接受在异步方法执行完成后调用的委托。BeginXXX方法返回IAsyncResult，用于验证调用是否已经完成，并且一直等到方法的执行结束。

WebClient类没有提供异步模式的实现方式，但是可以用HttpWebRequest类来替代，因为该类通过BeginGetResponse和EndGetResponse方法提供了这种模式。下面的示例没有体现这一点，而是使用了委托，委托类型定义了Invoke方法用于调用同步方法，还定义了BeginInvoke方法和EndInvolve方法，用于使用异步模式。在这里，声明了Func<string, string>类型的委托downloadString，来引用有一个string参数和一个string返回值的方法。downloadString变量引用的方法是用lambda表达式实现的，并且调用WebClient类型的同步方法DownloadString。这个委托通过调用BeginInvolve方法来异步调用。这个方法是使用线程池中的一个线程来进行异步调用的。

BeginInvoke方法的第一个参数是Func委托的第一个字符串泛型参数，用于传递URL。第二个参数的类型是AsyncCallback。AsyncCallback是一个委托，需要IAsyncResult作为参数。当异步方法执行完毕后，将调用这个委托引用的方法。之后，会调用downloadString.EndInvoke来检索结果，其方式与以前解析XML内容和获得集合项的方式相同。但是，这里不可能直接把结果返回给UI，因为UI绑定到一个单独的线程，而回调方法在一个后台线程中运行。所以，必须使用窗口的Dispatcher属性切换回UI线程。Dispatcher的Invoke方法需要一个委托作为参数；这就是指定Action<SearchItemResult>委托的原因，它会在绑定到UI的集合中添加项(代码文件AsyncPatterns/MainWindow.xaml.cs)：

```
private void OnSeachAsyncPattern(object sender, RoutedEventArgs e)
{
```

```
    Func<string, ICredentials, string> downloadString = (address, cred) =>
      {
        var client = new WebClient();
        client.Credentials = cred;
        return client.DownloadString(address);
      };

    Action<SearchItemResult> addItem = item =>searchInfo.List.Add(item);

    foreach (var req in GetSearchRequests())
    {
      downloadString.BeginInvoke(req.Url, req.Credentials, ar =>
        {
          string resp = downloadString.EndInvoke(ar);
          IEnumerable<SearchItemResult> images = req.Parse(resp);
          foreach (var image in images)
          {
            this.Dispatcher.Invoke(addItem, image);
          }
        }, null);
    }
  }
```

异步模式的优势是使用委托功能很容易地实现异步编程。程序现在运行正常，也不会阻塞 UI。但是，使用异步模式是非常复杂的。幸运的是，.NET 2.0 推出了基于事件的异步模式，更便于处理 UI 的更新。这个模式将会在 13.2.3 小节介绍。

第 8 章介绍了委托类型和 lambda 表达式。第 21 章介绍了线程和线程池。

13.2.3 基于事件的异步模式

OnAsyncEventPattern 方法使用了基于事件的异步模式。这个模式由 WebClient 类实现，因此可以直接使用。

基于事件的异步模式定义了一个带有“Async”后缀的方法。例如，对于同步方法 DownloadString，WebClient 类提供了一个异步变体方法 DownloadStringAsync。异步方法完成时，不是定义要调用的委托，而是定义一个事件。当异步方法 DownloadStringAsync 完成后，会直接调用 DownloadStringCompleted 事件。赋给该事件处理程序的方法，在 lambda 表达式中实现。实现方式和之前差不多，但是现在，可以直接访问 UI 元素了，因为事件处理程序是从拥有同步上下文的线程中调用，在 WindowsForm 和 WPF 应用程序中，拥有同步上下文的线程就是 UI 线程(代码文件 AsyncPatterns/MainWindow.xaml.cs)：

```
private void OnAsyncEventPattern(object sender, RoutedEventArgs e)
{
  foreach (var req in GetSearchRequests())
  {
    var client = new WebClient();
    client.Credentials = req.Credentials;
```

```
        client.DownloadStringCompleted += (sender1, e1) =>
          {
            string resp = e1.Result;
            IEnumerable<SearchItemResult> images = req.Parse(resp);
            foreach (var image in images)
            {
              searchInfo.List.Add(image);
            }
          };
        client.DownloadStringAsync(new Uri(req.Url));
      }
    }
```

基于事件的异步模式的优势在于易于使用。但是，在自定义类中实现这个模式就没有那么简单了。一种方式是使用 BackgroundWorker 类来实现异步调用同步方法。BackgroundWorker 类实现了基于事件的异步模式。

这使代码更加简单了。但是，与同步方法调用相比，顺序颠倒了。调用异步方法之前，需要定义这个方法完成时发生什么。13.2.4 小节将进入异步编程的新世界：利用 async 和 await 关键字。

13.2.4 基于任务的异步模式

在.NET 4.5 中，更新了 WebClient 类，还提供了基于任务的异步模式(TAP)。该模式定义了一个带有“Async”后缀的方法，并返回一个 Task 类型。由于 WebClient 类已经提供了一个带 Async 后缀的方法来实现基于任务的异步模式，因此新方法名为 DownloadStringTaskAsync。

DownloadStringTaskAsync 方法声明为返回 Task<string>。但是，不需要声明一个 Task<string>类型的变量来设置 DownloadStringTaskAsync 方法返回的结果。只要声明一个 String 类型的变量，并使用 await 关键字。await 关键字会解除线程（这里是 UI 线程）的阻塞，完成其他任务。当 DownloadStringTaskAsync 方法完成其后台处理后，UI 线程就可以继续，从后台线程中获得结果，赋值给字符串变量 resp。然后执行 await 关键字后面的代码(代码文件 AsyncPatterns/MainWindow.xaml.cs)：

```
private async void OnTaskBasedAsyncPattern(object sender,
    RoutedEventArgs e)
{
  foreach (var req in GetSearchRequests())
  {
    var client = new WebClient();
    client.Credentials = req.Credentials;
    string resp = await client.DownloadStringTaskAsync(req.Url);

    IEnumerable<SearchItemResult> images = req.Parse(resp);
    foreach (var image in images)
    {
      searchInfo.List.Add(image);
    }
  }
}
```

async 关键字创建了一个状态机，类似于 yield return 语句。参见第 6 章。

现在，代码就简单多了。没有阻塞，也不需要切换回 UI 线程，这些都是自动实现的。代码顺序也和惯用的同步编程一样。

接下来，将代码改为使用与 WebClient 不同的类，该类以更加直接的方式实现基于任务的异步模式，并且没有提供同步方法。该类是在.NET 4.5 中新添加的 HttpClient 类。使用 GetAsync 方法发出一个异步 GET 请求。然后，要读取内容，需要另一个异步方法。ReadAsStringAsync 方法返回字符串格式的内容。

```
private async void OnTaskBasedAsyncPattern(object sender,
    RoutedEventArgs e)
{
  foreach (var req in GetSearchRequests())
  {
    var clientHandler = new HttpClientHandler
    {
      Credentials = req.Credentials
    };
    var client = new HttpClient(clientHandler);
    var response = await client.GetAsync(req.Url);
    string resp = await response.Content.ReadAsStringAsync();

    IEnumerable<SearchItemResult> images = req.Parse(resp);
    foreach (var image in images)
    {
      searchInfo.List.Add(image);
    }
  }
}
```

解析 XML 字符串可能需要一段时间。因为解析代码在 UI 线程上运行，这时 UI 线程就不能响应用户的其他请求了。要利用同步功能创建后台任务，可以使用 Task.Run 方法。在下面的示例中，Task.Run 打包 XML 字符串的解析，返回一个 SearchItemResult 集合：

```
private async void OnTaskBasedAsyncPattern(object sender,
    RoutedEventArgs e)
{
  foreach (var req in GetSearchRequests())
  {
    var clientHandler = new HttpClientHandler
    {
      Credentials = req.Credentials
    };
    var client = new HttpClient(clientHandler);
    var response = await client.GetAsync(req.Url, cts.Token);
    string resp = await response.Content.ReadAsStringAsync();

    await Task.Run(() =>
```

```
        {
          IEnumerable<SearchItemResult> images = req.Parse(resp);
          foreach (var image in images)
          {
            searchInfo.List.Add(image);
          }
        }
      }
    }
```

因为传递给 Task.Run 方法的代码块在后台线程上运行，所以这里的问题和以前引用 UI 代码相同。一个解决方案是在 Task.Run 方法内只执行 req.Parse 方法，在任务外执行 foreach 循环，把结果添加到 UI 线程的列表中。现在，在.NET 4.5 中，WPF 提供了更好的解决方案，可以在后台线程上填充已绑定 UI 的集合。这只需要使用 BindingOperations.EnableCollectionSynchronization 属性，启用集合的同步访问功能。如下面的代码段所示：

```
public partial class MainWindow : Window
{
  private SearchInfo searchInfo;
  private object lockList = new object();

  public MainWindow()
  {
    InitializeComponent();
    searchInfo = new SearchInfo();
    this.DataContext = searchInfo;

    BindingOperations.EnableCollectionSynchronization(
        searchInfo.List, lockList);
  }
```

认识到 async 和 await 关键字的优势后，13.3 节将讨论异步编程的基础。

13.3 异步编程的基础

async 和 await 关键字只是编译器功能。编译器会用 Task 类创建代码。如果不使用这两个关键字，也可以用 C# 4.0 和 Task 类的方法来实现同样的功能，只是没有那么方便。

本节介绍了编译器用 async 和 await 关键字能做什么，如何采用简单的方式创建异步方法，如何并行调用多个异步方法，以及如何修改已经实现异步模式的类，以使用新的关键字。

13.3.1 创建任务

下面从同步方法 Greeting 开始，该方法等待一段时间后，返回一个字符串(代码文件 Foundations/Program.cs)：

```
static string Greeting(string name)
{
  Thread.Sleep(3000);
  return string.Format("Hello, {0}", name);
}
```

定义方法 GreetingAsync，可以使方法异步化。基于任务的异步模式指定，在异步方法名后加上 Async 后缀，并返回一个任务。异步方法 GreetingAsync 和同步方法 Greeting 具有相同的输入参数，但是它返回的是 Task<string>。Task<string>定义了一个返回字符串的任务。一个比较简单的做法是用 Task.Run 方法返回一个任务。泛型版本的 Task.Run<string>创建一个返回字符串的任务：

```
staticTask<string>GreetingAsync(string name)
{
  return Task.Run<string>(() =>
    {
     return Greeting(name);
    });
}
```

13.3.2 调用异步方法

可以使用 await 关键字来调用返回任务的异步方法 GreetingAsync。使用 await 关键字需要有用 async 修饰符声明的方法。在 GreetingAsync 方法完成前，该方法内的其他代码不会继续执行。但是，启动 CallerWithAsync 方法的线程可以被重用。该线程没有阻塞：

```
private async static void CallerWithAsync()
{
  string result = await GreetingAsync("Stephanie");
  Console.WriteLine(result);
}
```

如果异步方法的结果不传递给变量，也可以直接在参数中使用 await 关键字。在这里，GreetingAsync 方法返回的结果将像前面的代码片段一样等待，但是这一次的结果会直接传给 Console.WriteLine 方法：

```
private async static void CallerWithAsync2()
{
 Console.WriteLine(await GreetingAsync("Stephanie"));
}
```

async 修饰符只能用于返回 Task 或 viod 的方法。它不能用于程序的入口点，即 Main 方法不能使用 async 修饰符。await 只能用于返回 Task 的方法。

13.3.3 小节中，会介绍是什么驱动了这个 await 关键字，在后台使用了延续任务。

13.3.3 延续任务

GreetingAsync 方法返回一个 Task<string>对象。该 Task<string>对象包含任务创建的信息，并保存到任务完成。Task 类的 ContinueWith 方法定义了任务完成后就调用的代码。指派给 ContinueWith 方法的委托接收将已完成的任务作为参数传入，使用 Result 属性可以访问任务返回的结果：

```
private static void CallerWithContinuationTask()
{
  Task<string> t1 = GreetingAsync("Stephanie");
```

```
    t1.ContinueWith(t =>
      {
        string result = t.Result;
        Console.WriteLine(result);
      });
  }
```

编译器把await关键字后的所有代码放进ContinueWith方法的代码块中来转换await关键字。

13.3.4 同步上下文

如果验证一下方法中使用的线程，会发现CallerWithAsync方法和CallerWithContinuationTask方法，在方法的不同生命阶段使用了不同的线程。一个线程用于调用GreetingAsync方法，另外一个线程执行await关键字后面的代码，或者继续执行ContinueWith方法内的代码块。

使用一个控制台应用程序，通常不会有什么问题。但是，必须保证在所有应该完成的后台任务完成之前，至少有一个前台线程仍然在运行。示例应用程序调用Console.ReadLine来保证主线程一直在运行，直到按下返回键。

为了执行某些动作，有些应用程序会绑定到指定的线程上(例如，在WPF应用程序中，只有UI线程才能访问UI元素)，这将会是一个问题。

如果使用async和await关键字，当await完成之后，不需要进行任何特别处理，就能访问UI线程。默认情况下，生成的代码就会把线程转换到拥有同步上下文的线程中。WPF应用程序设置了DispatcherSynchronizationContext属性，WindowsForm应用程序设置了WindowsFormsSynchronization-Context属性。如果调用异步方法的线程分配给了同步上下文，await完成之后将继续执行。默认情况下，使用了同步上下文。如果不使用相同的同步上下文，必须调用Task类的ConfigureAwait(continueOnCapturedContext：false)。例如，一个WPF应用程序，其await后面的代码没有用到任何的UI元素。在这种情况下，避免切换到同步上下文会执行得更快。

13.3.5 使用多个异步方法

在一个异步方法里，可以调用一个或多个异步方法。如何编写代码，取决于一个异步方法的结果是否依赖于另一个异步方法。

1. 按顺序调用异步方法

使用await关键字可以调用每个异步方法。在有些情况下，如果一个异步方法依赖另一个异步方法的结果，await关键字就非常有用。在这里，GreetingAsync异步方法的第二次调用完全独立于其第一次调用的结果。这样，如果每个异步方法都不使用await，整个MultipleAsyncMethods异步方法将更快地返回结果，如下所示：

```
private async static void MultipleAsyncMethods()
{
  string s1 = await GreetingAsync("Stephanie");
  string s2 = await GreetingAsync("Matthias");
  Console.WriteLine("Finished both methods.\n " +
     "Result 1: {0}\n Result 2: {1}", s1, s2);
}
```

2. 使用组合器

如果异步方法不依赖于其他异步方法，每个异步方法都不使用 await，而是把每个异步方法的返回结果赋值给 Task 变量，就会运行得更快。GreetingAsync 方法返回 Task<string>。这些方法现在可以并行运行了。组合器可以帮助实现这一点。一个组合器可以接受多个同一类型的参数，并返回同一类型的值。多个同一类型的参数被组合成一个参数来传递。Task 组合器接受多个 Task 对象作为参数，并返回一个 Task。

示例代码调用 Task.WhenAll 组合器方法，它可以等待，直到两个任务都完成。

```
private async static void MultipleAsyncMethodsWithCombinators1()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  await Task.WhenAll(t1, t2);
  Console.WriteLine("Finished both methods.\n " +
      "Result 1: {0}\n Result 2: {1}", t1.Result, t2.Result);
}
```

Task 类定义了 WhenAll 和 WhenAny 组合器。从 WhenAll 方法返回的 Task，是在所有传入方法的任务都完成了才会返回 Task。从 WhenAny 方法返回的 Task，是在其中一个传入方法的任务完成了就会返回 Task。

Task 类型的 WhenAll 方法定义了几个重载版本。如果所有的任务返回相同的类型，那么该类型的数组可用于 await 返回的结果。GreetingAsync 方法返回一个 Task<string>，等待返回的结果是一个字符串(string)形式。因此，Task.WhenAll 可用于返回一个字符串数组：

```
private async static void MultipleAsyncMethodsWithCombinators2()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  string[] result = await Task.WhenAll(t1, t2);
  Console.WriteLine("Finished both methods.\n " +
      "Result 1: {0}\n Result 2: {1}", result[0], result[1]);
}
```

13.3.6 转换异步模式

并非.NET Framework 的所有类在.NET 4.5 中都引入了新的异步方法。在使用框架中的不同类时会发现，还有许多类只提供了 BeginXXX 方法和 EndXXX 方法的异步模式，没有提供基于任务的异步模式。

首先，从前面定义的同步方法 Greeting 中，借助于委托，创建一个异步方法。Greeting 方法接收一个字符串作为参数，并返回一个字符串。因此，Func<string, string>委托的变量可用于引用 Greeting 方法。按照异步模式，BeginGreeting 方法接收一个 string 参数，一个 AsyncCallback 参数和一个 object 参数，返回 IAsyncResult。EndGreeting 方法返回来自 Greeting 方法的结果——一个字符串——并接收一个 IAsyncResult 参数。在实现代码中，该委托仅用于异步执行任务。

```
private static Func<string, string> greetingInvoker = Greeting;

static IAsyncResult BeginGreeting(string name, AsyncCallback callback,
```

```
  object state)
{
  return greetingInvoker.BeginInvoke(name, callback, state);
}

static string EndGreeting(IAsyncResult ar)
{
  return greetingInvoker.EndInvoke(ar);
}
```

现在，BeginGreeting方法和EndGreeting方法都是可用的，它们都应转换为使用async和await关键字来获取结果。TaskFactory类定义了FromAsync方法，它可以把使用异步模式的方法转换为基于任务的异步模式的方法(TAP)。

示例代码中，Task类型的第一个泛型参数Task<string>，定义了调用方法的返回值类型。FromAsync方法的泛型参数定义了方法的输入类型。这样，输入参数又是字符串类型。FromAsync方法的前两个参数是委托类型，传入BeginGreeting和EndGreeting方法的地址。紧跟这两个参数后面的，是输入参数和对象状态参数。因对象状态没有用到，所以给它分配null值。因为FromAsync方法返回Task类型，即示例代码中的Task<string>，可以使用await，如下所示：

```
private static async void ConvertingAsyncPattern()
{
  string s = await Task<string>.Factory.FromAsync<string>(
    BeginGreeting, EndGreeting, "Angela", null);
  Console.WriteLine(s);
}
```

13.4 错误处理

第16章详细介绍了错误和异常处理。

但是，在使用异步方法时，应该知道错误的一些特殊处理方式。

从一个简单的方法开始，它在延迟后抛出一个异常(代码文件ErrorHandling/Program.cs)：

```
static async Task ThrowAfter(int ms, string message)
{
  await Task.Delay(ms);
  throw new Exception(message);
}
```

如果调用异步方法，并且没有等待，可以将异步方法放在try/catch块中，就会捕获不到异常。这是因为DontHandle方法在ThrowAfter抛出异常之前，已经执行完毕。需要等待ThrowAfter方法(用await关键字)，如下所示：

```
private static void DontHandle()
{
  try
  {
    ThrowAfter(200, "first");
    // exception is not caught because this method is finished
    // before the exception is thrown
```

```
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
}
```

返回 void 的异步方法不会等待。这是因为从 async void 方法抛出的异常无法捕获。因此，异步方法最好返回一个 Task 类型。处理程序方法或重写的基类方法不受此规则限制。

13.4.1 异步方法的异常处理

异步方法异常的一个较好处理方式，就是使用 await 关键字，将其放在 try/catch 语句中，如以下代码块所示。异步调用 ThrowAfter 方法后，HandleOneError 方法就会释放线程，但它会在任务完成时保持任务的引用。此时(2s 后，抛出异常)，会调用匹配的 catch 块内的代码。

```
private static async void HandleOneError()
{
  try
  {
    await ThrowAfter(2000, "first");
  }
  catch (Exception ex)
  {
    Console.WriteLine("handled {0}", ex.Message);
  }
}
```

13.4.2 多个异步方法的异常处理

如果调用两个异步方法，每个都会抛出异常，该如何处理呢？在下面的示例中，第一个 ThrowAfter 方法被调用，2s 后抛出异常(含消息 first)。该方法结束后，另一个 ThrowAfter 方法也被调用，1s 后也抛出异常。事实并非如此，因为对第一个 ThrowAfter 方法的调用已经抛出了异常，try 块内的代码没有继续调用第二个 ThrowAfter 方法，而是在 catch 块内对第一个异常进行处理。

```
private static async void StartTwoTasks()
{
  try
  {
    await ThrowAfter(2000, "first");
    await ThrowAfter(1000, "second"); // the second call is not invoked
                                      // because the first method throws
                                      // an exception
  }
  catch (Exception ex)
  {
    Console.WriteLine("handled {0}", ex.Message);
  }
}
```

现在，并行调用这两个 ThrowAfter 方法。第一个 ThrowAfter 方法 2s 后抛出异常，1s 后第二个 ThrowAfter 方法也抛出异常。使用 Task.WhenAll，不管任务是否抛出异常，都会等到两个任务完成。因此，等待 2s 后，Task.WhenAll 结束，异常被 catch 语句捕获到。但是，只能看见传递给 WhenAll 方法的第一个任务的异常信息。没有显示先抛出异常的任务(第二个任务)，但该任务也在列表中：

```
private async static void StartTwoTasksParallel()
{
  try
  {
    Task t1 = ThrowAfter(2000, "first");
    Task t2 = ThrowAfter(1000, "second");
    await Task.WhenAll(t1, t2);
  }
  catch (Exception ex)
  {
    // just display the exception information of the first task
    // that is awaited within WhenAll
    Console.WriteLine("handled {0}", ex.Message);
  }
}
```

有一种方式可以获取所有任务的异常信息，就是在 try 块外声明任务变量 t1 和 t2，使它们可以在 catch 块内访问。在这里，可以使用 IsFaulted 属性检查任务的状态，以确认它们是否为出错状态。若出现异常，IsFaulted 属性会返回 true。可以使用 Task 类的 Exception.InnerException 访问异常信息本身。另一种获取所有任务的异常信息的更好方式如下所述。

13.4.3 使用 AggregateException 信息

为了得到所有失败任务的异常信息，可以将 Task.WhenAll 返回的结果写到一个 Task 变量中。这个任务会一直等到所有任务都结束。否则，仍然可能错过抛出的异常。13.4.2 小节中，catch 语句只检索到第一个任务的异常。不过，现在可以访问外部任务的 Exception 属性了。Exception 属性是 AggregateException 类型的。这个异常类型定义了 InnerExceptions 属性(不只是 InnerException)，它个包含了等待中的所有异常的列表。现在，可以轻松遍历所有异常了。

```
private static async void ShowAggregatedException()
{
  Task taskResult = null;
  try
  {
    Task t1 = ThrowAfter(2000, "first");
    Task t2 = ThrowAfter(1000, "second");
    await (taskResult = Task.WhenAll(t1, t2));
  }
  catch (Exception ex)
  {
    Console.WriteLine("handled {0}", ex.Message);
    foreach (var ex1 in taskResult.Exception.InnerExceptions)
    {
      Console.WriteLine("inner exception {0}", ex1.Message);
    }
  }
}
```

13.5 取消

在一些情况下，后台任务可能运行很长时间，取消任务就非常有用了。对于取消任务，从.NET 4.0开始就提供了一种标准的机制。这种机制可用于基于任务的异步模式。

取消框架基于协助行为，不是强制性的。一个运行时间很长的任务需要检查自己是否被取消，在这种情况下，它的工作就是清理所有已打开的资源，并结束相关工作。

取消基于 CancellationTokenSource 类，该类可用于发送取消请求。请求发送给引用 CancellationToken 类的任务，其中 CancellationToken 类与 CancellationTokenSource 类相关联。13.5.1 小节将修改本章前面创建的 AsyncPatterns 示例，来阐述取消任务的相关内容。

13.5.1 开始取消任务

首先，使用 MainWindow 类的私有字段成员定义一个 CancellationTokenSource 类型的变量 cts。该成员用于取消任务，并将令牌传递给应取消的方法(代码文件 AsyncPatterns/MainWindow.xaml.cs)：

```
public partial class MainWindow : Window
{
  private SearchInfo searchInfo;
  private object lockList = new object();
  private CancellationTokenSource cts;
```

新添加一个按钮，用于取消正在运行的任务，添加事件处理程序 OnCancel 方法。在这个方法中，变量 cts 用 Cancel 方法取消任务：

```
private void OnCancel(object sender, RoutedEventArgs e)
{
  if (cts != null)
    cts.Cancel();
}
```

CancellationTokenSource 类还支持在指定时间后才取消任务。CancelAfter 方法传入一个时间值，单位是 ms，在该时间过后，就取消任务。

13.5.2 使用框架特性取消任务

现在，将 CancellationToken 传入异步方法。框架中的某些异步方法提供可以传入 CancellationToken 的重载版本，来支持取消任务。例如 HttpClient 类的 GetAsync 方法。除了 URI 字符串，重载的 GetAsync 方法还接受 CancellationToken 参数。可以使用 Token 属性检索 CancellationTokenSource 类的令牌。

GetAsync 方法的实现会定期检查是否应取消操作。如果取消，就清理资源，之后抛出 OperationCanceledException 异常。如下面的代码片段所示，catch 处理程序捕获到了该异常：

```
private async void OnTaskBasedAsyncPattern(object sender,
    RoutedEventArgs e)
{
  cts = new CancellationTokenSource();
  try
  {
```

```
        foreach (var req in GetSearchRequests())
        {
          var client = new HttpClient();
          var response = await client.GetAsync(req.Url, cts.Token);
          string resp = await response.Content.ReadAsStringAsync();

          //...
        }
      }
      catch (OperationCanceledException ex)
      {
        MessageBox.Show(ex.Message);
      }
    }
```

13.5.3 取消自定义任务

如何取消自定义任务？Task 类的 Run 方法提供了重载版本，它也传递 CancellationToken 参数。但是，对于自定义任务，需要检查是否请求了取消操作。下例中，这是在 foreach 循环中实现的，可以使用 IsCancellationRequsted 属性检查令牌。在抛出异常之前，如果需要做一些清理工作，最好验证一下是否请求取消操作。如果不需要做清理工作，检查之后，会立即用 ThrowIfCancellationRequested 方法触发异常。

```
await Task.Run(() =>
{
  varimages = req.Parse(resp);
  foreach (var image in images)
  {
    cts.Token.ThrowIfCancellationRequested();
    searchInfo.List.Add(image);
  }
}, cts.Token);
```

现在，用户可以取消运行时间长的任务了。

13.6 小结

本章介绍了 C#5.0 中新增的 async 和 await 关键字。通过几个示例，介绍了基于任务的异步模式，比.NET 早期版本中的异步模式和基于事件的异步模式更具优势。

本章也讨论了在 Task 类的辅助下，创建异步方法是非常容易的。同时，学会了如何使用 async 和 await 关键字等待这些方法，而不会阻塞线程。最后，介绍了异步方法的错误处理。

若想了解更多关于并行编程、线程和任务的详细信息，参考第 21 章。

第 14 章将继续关注 C#和.NET 的核心功能，详细介绍了内存和资源管理。

第14章

内存管理和指针

本章要点

- 运行库在栈和堆上分配空间
- 垃圾回收
- 使用析构函数和 System.IDisposable 接口来释放非托管的资源
- C#中使用指针的语法
- 使用指针实现基于栈的高性能数组

本章源代码下载地址(wrox.com):

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- PointerPlayground
- PointerPlayground2
- QuickArray

14.1 内存管理

本章介绍内存管理和内存访问的各个方面。尽管运行库负责为程序员处理大部分内存管理工作，但程序员仍必须理解内存管理的工作原理，了解如何高效地处理非托管的资源。

如果很好地理解了内存管理和 C#提供的指针功能，也就能很好地集成 C#代码和原来的代码，并能在非常注重性能的系统中高效地处理内存。

14.2 后台内存管理

C#编程的一个优点是程序员不需要担心具体的内存管理，垃圾回收器会自动处理所有的内存清理工作。用户可以得到像 C++语言那样的效率，而不需要考虑像在 C++中那样内存管理工作的复杂

性。虽然不必手动管理内存，但仍需理解后台发生的事情。理解程序在后台如何处理内存有助于提高应用程序的速度和性能。本节要介绍给变量分配内存时在计算机的内存中发生的情况。

本节不详细介绍许多主题的相关内容。应把这一节看作是一般过程的简化向导，而不是实现的确切说明。

14.2.1　值数据类型

Windows 使用一个虚拟寻址系统，该系统把程序可用的内存地址映射到硬件内存中的实际地址上，这些任务完全由 Windows 在后台管理。其实际结果是 32 位处理器上的每个进程都可以使用 4GB 的内存——无论计算机上实际有多少物理内存(在 64 位处理器上，这个数字会更大)。这个 4GB 的内存实际上包含了程序的所有部分，包括可执行代码、代码加载的所有 DLL，以及程序运行时使用的所有变量的内容。这个 4GB 的内存称为虚拟地址空间，或虚拟内存。为了方便起见，本章将它简称为内存。

4GB 中的每个存储单元都是从 0 开始往上排序的。要访问存储在内存的某个空间中的一个值，就需要提供表示该存储单元的数字。在任何复杂的高级语言中，如 C#、VB、C++和 Java，编译器负责把人们可以理解的变量名转换为处理器可以理解的内存地址。

在处理器的虚拟内存中，有一个区域称为栈。栈存储不是对象成员的值数据类型。另外，在调用一个方法时，也使用栈存储传递给方法的所有参数的副本。为了理解栈的工作原理，需要注意在 C#中变量的作用域。如果变量 a 在变量 b 之前进入作用域，b 就会首先超出作用域。考虑下面的代码：

```
{
  int a;
  // do something
  {
    int b;
    // do something else
  }
}
```

首先声明 a。在内部代码块中声明了 b。然后内部代码块终止，b 就超出作用域，最后 a 超出作用域。所以 b 的生存期完全包含在 a 的生存期中。在释放变量时，其顺序总是与给它们分配内存的顺序相反，这就是栈的工作方式。

还要注意，b 在另一个代码块中(通过另一对嵌套的花括号来定义)。因此，它包含在另一个作用域中。这称为块作用域或结构作用域。

我们不知道栈具体在地址空间的什么地方，这些信息在进行 C#开发是不需要知道的。栈指针(操作系统维护的一个变量)表示栈中下一个空闲存储单元的地址。程序第一次开始运行时，栈指针指向为栈保留的内存块末尾。栈实际上是向下填充的，即从高内存地址向低内存地址填充。当数据入栈后，栈指针就会随之调整，以始终指向下一个空闲存储单元。这种情况如图 14-1 所示。在该图中，显示了栈指针 800 000(十六进制的 0xC3500)，下一个空闲存储单元是地址 799 999。

下面的代码会告诉编译器，需要一些存储空间以存储一个整数和一个双精度浮点数，这些存储

单元分别称为 nRacingCars 和 engineSize，声明每个变量的代码行表示开始请求访问这个变量，闭合花括号标识这两个变量超出作用域的地方。

```
{
  int nRacingCars = 10;
  double engineSize = 3000.0;
  // do calculations;
}
```

图 14-1

假定使用如图 14-1 所示的栈。nRacingCars 变量进入作用域，赋值为 10，这个值放在存储单元 799 996~799 999 上，这 4 个字节就在栈指针所指空间的下面。有 4 个字节是因为存储 int 要使用 4 个字节。为了容纳该 int，应从栈指针对应的值中减去 4，所以它现在指向位置 799 996，即下一个空闲单元(799 995)。

下一行代码声明变量 engineSize(这是一个 double)，把它初始化为 3 000.0。一个 double 数要占用 8 个字节，所以值 3 000.0 放在栈上的存储单元 799 988~799 995 上，栈指针对应的值减去 8，再次指向栈上的下一个空闲单元。

当 engineSize 超出作用域时，运行库就知道不再需要这个变量了。因为变量的生存期总是嵌套的，当 engineSize 在作用域中时，无论发生什么情况，都可以保证栈指针总是会指向存储 engineSize 的空间。为了从内存中删除这个变量，应给栈指针对应的值递增 8，现在它指向 engineSize 末尾紧接着的空间。此处就是放置闭合花括号的地方。当 nRacingCars 也超出作用域时，栈指针对应的值就再次递增 4。从栈中删除 enginesize 和 nRacingCars 之后，此时如果在作用域中又放入另一个变量，从 799 999 开始的存储单元就会被覆盖，这些空间以前是存储 nRacingCars 的。

如果编译器遇到 int i、j 这样的代码行，则这两个变量进入作用域的顺序是不确定的。两个变量是同时声明的，也是同时超出作用域的。此时，变量以什么顺序从内存中删除就不重要了。编译器在内部会确保先放在内存中的那个变量后删除，这样就能保证该规则不会与变量的生存期冲突。

14.2.2　引用数据类型

尽管栈有非常高的性能，但它还没有灵活到可以用于所有的变量。变量的生存期必须嵌套，在许多情况下，这种要求都过于苛刻。通常我们希望使用一个方法分配内存，来存储一些数据，并在方法退出后的很长一段时间内数据仍是可用的。只要是用 new 运算符来请求分配存储空间，就存在这种可能性——例如对于所有的引用类型。此时就要使用托管堆。

如果读者以前编写过需要管理低级内存的 C++代码，就会很熟悉堆(heap)。托管堆和 C++使用的堆不同，它在垃圾回收器的控制下工作，与传统的堆相比有很显著的优势。

托管堆(简称为堆)是处理器的可用 4GB 中的另一个内存区域。要了解堆的工作原理和如何为引用数据类型分配内存，看看下面的代码：

```
void DoWork()
{
  Customer arabel;
  arabel = new Customer();
  Customer otherCustomer2 = new EnhancedCustomer();
}
```

在这段代码中，假定存在两个类 Customer 和 EnhancedCustomer。EnhancedCustomer 类扩展了 Customer 类。

首先，声明一个 Customer 引用 arabel，在栈上给这个引用分配存储空间，但这仅是一个引用，而不是实际的 Customer 对象。arabel 引用占用 4 个字节的空间，足够包含 Customer 对象的存储地址(需要 4 个字节把 0～4GB 之间的内存地址表示为一个整数值)。

然后看下一行代码：

```
arabel = new Customer();
```

这行代码完成了以下操作：首先，它分配堆上的内存，以存储 Customer 对象(一个真正的对象，不只是一个地址)。然后把变量 arabel 的值设置为分配给新 Customer 对象的内存地址(它还调用合适的 Customer()构造函数初始化类实例中的字段，但此处我们不必担心这部分)。

Customer 实例没有放在栈中，而是放在堆中。在这个例子中，现在还不知道一个 Customer 对象占用多少字节，但为了讨论方便，假定是 32 个字节。这 32 个字节包含了 Customer 实例的字段，和.NET 用于识别和管理其类实例的一些信息。

为了在堆上找到存储新 Customer 对象的一个存储位置，.NET 运行库在堆中搜索，选取第一个未使用的且包含 32 个字节的连续块。为了讨论方便，假定其地址是 200 000，arabel 引用占用栈中的 799 996~799 999 位置。这表示在实例化 arabel 对象前，内存的内容应如图 14-2 所示。

给 Customer 对象分配空间后，内存的内容应如图 14-3 所示。注意，与栈不同，堆上的内存是向上分配的，所以空闲空间在已用空间的上面。

图 14-2　　　　图 14-3

下一行代码声明了一个 Customer 引用，并实例化一个 Customer 对象。在这个例子中，用一行代码在栈上为 otherCustomer2 引用分配空间，同时在堆上为 mrJones 对象分配空间：

```
Customer otherCustomer2 = new EnhancedCustomer();
```

该行把栈上的 4 个字节分配给 otherCustomer2 引用，它存储在 799 992~799 995 位置上，而 otherCustomer2 对象在堆上从 200 032 开始向上分配空间。

从这个例子可以看出，建立引用变量的过程要比建立值变量的过程更复杂，且不能避免性能的系统开销。实际上，我们对这个过程进行了过分的简化，因为.NET 运行库需要保存堆的状态信息，在堆中添加新数据时，这些信息也需要更新。尽管有这些性能开销，但仍有一种机制，在给变量分配内存时，不会受到栈的限制。把一个引用变量的值赋予另一个相同类型的变量，就有两个变量引用内存中的同一对象了。当一个引用变量超出作用域时，它会从栈中删除，如上一节所述，但引用对象的数据仍保留在堆中，一直到程序终止，或垃圾回收器删除它为止，而只有在该数据不再被任何变量引用时，它才会被删除。

这就是引用数据类型的强大之处，在 C#代码中广泛使用了这个特性。这说明，我们可以对数据的生存期进行非常强大的控制，因为只要保持对数据的引用，该数据就肯定存在于堆上。

14.2.3 垃圾回收

由上面的讨论和图 14-2 和图 14-3 可以看出，托管堆的工作方式非常类似于栈，对象会在内存中一个挨一个地放置，这样就很容易使用指向下一个空闲存储单元的堆指针，来确定下一个对象的位置。在堆上添加更多的对象时，也容易调整。但这比较复杂，因为基于堆的对象的生存期与引用它们的基于栈的变量的作用域不匹配。

在垃圾回收器运行时，它会从堆中删除不再引用的所有对象。在完成删除操作后，堆会立即把对象分散开来，与已经释放的内存混合在一起，如图 14-4 所示。

图 14-4

如果托管的堆也是这样，在其上给新对象分配内存就成为一个很难处理的过程，运行库必须搜索整个堆，才能找到足够大的内存块来存储每个新对象。但是，垃圾回收器不会让堆处于这种状态。只要它释放了能释放的所有对象，就会把其他对象移动回堆的端部，再次形成一个连续的内存块。因此，堆可以继续像栈那样确定在什么地方存储新对象。当然，在移动对象时，这些对象的所有引用都需要用正确的新地址来更新，但垃圾回收器也会处理更新问题。

垃圾回收器的这个压缩操作是托管的堆与非托管的旧堆的区别所在。使用托管的堆，就只需要读取堆指针的值即可，而不需遍历地址的链表，来查找一个地方来放置新数据。因此，在.NET 下实例化对象要快得多。有趣的是，访问它们也比较快，因为对象会压缩到堆上相同的内存区域，这样需要交换的页面较少。Microsoft 相信，尽管垃圾回收器需要做一些工作，压缩堆，修改它移动的所有对象引用，致使性能降低，但这些性能会得到更多弥补。

一般情况下，垃圾回收器在.NET 运行库认为需要进行垃圾回收时运行。可以调用 System.GC.Collect()方法，强迫垃圾回收器在代码的某个地方运行，System.GC 类是一个表示垃圾回收器的.NET 类，Collect()方法启动一个垃圾回收过程。但是，GC 类适用的场合很少，例如，代码中有大量的对象刚刚取消引用，就适合调用垃圾回收器。但是，垃圾回收器的逻辑不能保证在一次垃圾收集过程中，所有未引用的对象都从堆中删除。

创建对象时，会把这些对象放在托管堆上。堆的第一部分称为第 0 代。创建新对象时，会把它们移动到堆的这个部分中。因此，这里驻留了最新的对象。

对象会继续放在这个部分，直到垃圾回收过程第一次进行回收。这个清理过程之后仍保留的对象会被压缩，然后移动到堆的下一部分上或世代部分——第 1 代对应的部分。

此时，第 0 代对应的部分为空，所有的新对象都再次放在这一部分上。在垃圾回收过程中遗留下来的旧对象放在第 1 代对应的部分上。老对象的这种移动会再次发生。接着重复下一次回收过程。这意味着，第 1 代中在垃圾回收过程中遗留下来的对象会移动到堆的第 2 代，位于第 0 代的对象会移动到第 1 代，第 0 代仍用于放置新对象。

有趣的是，在给对象分配内存空间时，如果超出了第 0 代对应的部分的容量，或者调用了 GC.Collect()方法，就会进行垃圾回收。

这个过程极大地提高了应用程序的性能。一般而言，最新的对象通常是可以回收的对象，而且可能也会回收大量比较新的对象。如果这些对象在堆中的位置是相邻的，垃圾回收过程就会更快。另外，相关的对象相邻放置也会使程序执行得更快。

在.NET 中，垃圾回收提高性能的另一个领域是架构处理堆上较大对象的方式。在.NET 下，较大对象有自己的托管堆，称为大对象堆。使用大于 85 000 个字节的对象时，它们就会放在这个特殊的堆上，而不是主堆上。.NET 应用程序不知道两者的区别，因为这是自动完成的。其原因是在堆上压缩大对象是比较昂贵的，因此驻留在大对象堆上的对象不执行压缩过程。

在进一步改进垃圾回收过程后，第二代和大对象堆上的回收现在放在后台线程上进行。这表示，应用程序线程仅会为第 0 代和第 1 代的回收而阻塞，减少了总暂停时间，对于大型服务器应用程序尤其如此。服务器和工作站默认打开这个功能。要关闭该功能，可以在配置文件中把<gcConcurrent>元素设置为 false。

有助于提高应用程序性能的另一个优化是垃圾回收的平衡，它专用于服务器的垃圾回收。服务器一般有一个线程池，执行大致相同的工作。内存分配在所有线程上都是类似的。对于服务器，每个逻辑服务器都有一个垃圾回收堆。其中一个堆用尽了内存，触发了垃圾回收过程时，所有其他堆也可能会得益于垃圾的回收。如果一个线程使用的内存远远多于其他线程，导致垃圾回收，其他线程可能不需要垃圾回收，这就不是很高效。垃圾回收过程会平衡这些堆——小对象堆和大对象堆。进行这个平衡过程，可以减少不必要的回收。

为了利用包含大量内存的硬件，垃圾回收过程添加了 GCSettings.LatencyMode 属性。把这个属性设置为 GCLatencyMode 枚举的一个值，可以控制垃圾回收器进行回收的方式。表 14-1 列出了可用的值。

表 14-1　GCLatencyMode 的设置

成　员	说　明
Batch	禁用并发设置，把垃圾回收设置为最大吞吐量。这会重写配置设置
Interactive	默认行为
LowLatency	保守的垃圾回收。只有系统存在内存压力时，才进行完整的回收。只应用于较短时间，执行特定的操作
SustainedLowLatency	只有系统存在内存压力时，才进行完整的内存块回收

LowLatency 设置使用的时间应为最小值，分配的内存量应尽可能小。如果不小心，就可能出现溢出内存错误。

为了使用 64 位机器的高内存量，添加了<gcAllorVeryLargeObjects>配置设置。它允许创建大于 2GB 的对象。这对 32 位机器没有影响，32 位机器仍有 2GB 的限制。

14.3 释放非托管的资源

垃圾回收器的出现意味着，通常不需要担心不再需要的对象，只要让这些对象的所有引用都超出作用域，并允许垃圾回收器在需要时释放内存即可。但是，垃圾回收器不知道如何释放非托管的资源(例如文件句柄、网络连接和数据库连接)。托管类在封装对非托管资源的直接或间接引用时，需要制定专门的规则，确保非托管的资源在回收类的一个实例时释放。

在定义一个类时，可以使用两种机制来自动释放非托管的资源。这些机制常常放在一起实现，因为每种机制都为问题提供了略为不同的解决方法。这两种机制是：

- 声明一个析构函数(或终结器)，作为类的一个成员
- 在类中实现 System.IDisposable 接口

下面依次讨论这两种机制，然后介绍如何同时实现它们，以获得最佳的效果。

14.3.1 析构函数

前面介绍了构造函数可以指定必须在创建类的实例时进行的某些操作。相反，在垃圾回收器销毁对象之前，也可以调用析构函数。由于执行这个操作，因此析构函数初看起来似乎是放置释放非托管资源、执行一般清理操作的代码的最佳地方。但是，事情并不是如此简单。

在讨论 C#中的析构函数时，在底层的.NET 体系结构中，这些函数称为终结器(finalizer)。在 C#中定义析构函数时，编译器发送给程序集的实际上是 Finalize()方法。它不会影响源代码，但如果需要查看程序集的内容，就应知道这个事实。

C++开发人员应很熟悉析构函数的语法。它看起来类似于一个方法，与包含的类同名，但有一个前缀波形符(~)。它没有返回类型，不带参数，没有访问修饰符。下面是一个例子：

```
class MyClass
{
  ~MyClass()
  {
    // destructor implementation
  }
}
```

C#编译器在编译析构函数时，它会隐式地把析构函数的代码编译为等价于 Finalize()方法的代码，从而确保执行父类的 Finalize()方法。下面列出了等价于编译器为~MyClass()析构函数生成的 IL 的 C#代码：

```
protected override void Finalize()
{
  try
  {
    // destructor implementation
  }
  finally
  {
```

```
        base.Finalize();
      }
    }
```

如上所示，在~MyClass()析构函数中实现的代码封装在 Finalize()方法的一个 try 块中。对父类的 Finalize()方法的调用放在 finally 块中，确保该调用的执行。第 16 章会讨论 try 块和 finally 块。

有经验的 C++开发人员大量使用了析构函数，有时不仅用于清理资源，还提供调试信息或执行其他任务。C#析构函数要比 C++析构函数的使用少得多。与 C++析构函数相比，C#析构函数的问题是它们的不确定性。在销毁 C++对象时，其析构函数会立即运行。但由于使用 C#时垃圾回收器的工作方式，无法确定 C#对象的析构函数何时执行。所以，不能在析构函数中放置需要在某一时刻运行的代码，也不应寄望于析构函数会以特定顺序对不同类的实例调用。如果对象占用了宝贵而重要的资源，应尽快释放这些资源，此时就不能等待垃圾回收器来释放了。

另一个问题是 C#析构函数的实现会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾回收器的一次处理中从内存中删除，但有析构函数的对象需要两次处理才能销毁：第一次调用析构函数时，没有删除对象，第二次调用才真正删除对象。另外，运行库使用一个线程来执行所有对象的 Finalize()方法。如果频繁使用析构函数，而且使用它们执行长时间的清理任务，对性能的影响就会非常显著。

14.3.2　IDisposable 接口

在 C#中，推荐使用 System.IDisposable 接口替代析构函数。IDisposable 接口定义了一种模式(具有语言级的支持)，该模式为释放非托管的资源提供了确定的机制，并避免产生析构函数固有的与垃圾回收器相关的问题。IDisposable 接口声明了一个 Dispose()方法，它不带参数，返回 void，MyClass 类的 Dispose()方法的实现代码如下：

```
class MyClass: IDisposable
{
  public void Dispose()
  {
     // implementation
  }
}
```

Dispose()方法的实现代码显式地释放由对象直接使用的所有非托管资源，并在所有也实现 IDisposable 接口的封装对象上调用 Dispose()方法。这样，Dispose()方法为何时释放非托管资源提供了精确的控制。

假定有一个 ResourceGobbler 类，它需要使用某些外部资源，且实现 IDisposable 接口。如果要实例化这个类的实例，使用它，然后释放它，就可以使用下面的代码：

```
ResourceGobbler theInstance = new ResourceGobbler();

// do your processing

theInstance.Dispose();
```

但是，如果在处理过程中出现异常，这段代码就没有释放 theInstance 使用的资源，所以应使用 try 块(详见第 16 章)，编写下面的代码：

```
ResourceGobbler theInstance = null;

try
{
  theInstance = new ResourceGobbler();

  // do your processing
}
finally
{
  if (theInstance != null)
  {
      theInstance.Dispose();
  }
}
```

即使在处理过程中出现了异常，这个版本也可以确保总是在 theInstance 上调用 Dispose()方法，总是释放 theInstance 使用的任意资源。但是，如果总是要重复这样的结构，代码就很容易被混淆。C#提供了一种语法，可以确保在实现 IDisposable 接口的对象的引用超出作用域时，在该对象上自动调用 Dispose()方法。该语法使用了 using 关键字来完成这一工作——该关键字在完全不同的环境下，它与名称空间没有关系。下面的代码生成与 try 块等价的 IL 代码：

```
using (ResourceGobbler theInstance = new ResourceGobbler())
{
   // do your processing
}
```

using 语句的后面是一对圆括号，其中是引用变量的声明和实例化，该语句使变量的作用域限定在随后的语句块中。另外，在变量超出作用域时，即使出现异常，也会自动调用其 Dispose()方法。然而，如果已经使用 try 块来捕获其他异常，就会比较清晰，如果避免使用 using 语句，仅在已有的 try 块的 finally 子句中调用 Dispose()方法，还可以避免进行额外的代码缩进。

对于某些类，使用 Close()方法要比 Dispose()方法更富有逻辑性，例如，在处理文件或数据库连接时就是这样。在这些情况下，常常实现 IDisposable 接口，再实现一个独立的 Close()方法，Close()方法只调用 Dispose()方法。这种方法在类的使用上比较清晰，还支持 C#提供的 using 语句。

14.3.3 实现 IDisposable 接口和析构函数

前面的章节讨论了自定义类所使用的释放非托管资源的两种方式：

- 利用运行库强制执行的析构函数，但析构函数的执行是不确定的，而且，由于垃圾回收器的工作方式，它会给运行库增加不可接受的系统开销。
- IDisposable 接口提供了一种机制，该机制允许类的用户控制释放资源的时间，但需要确保调用 Dispose()方法。

一般情况下，最好的方法是实现这两种机制，获得这两种机制的优点，克服其缺点。假定大多数程序员都能正确调用 Dispose()方法，同时把实现析构函数作为一种安全机制，以防没有调用

Dispose()方法。下面是一个双重实现的例子：

```
using System;

public class ResourceHolder: IDisposable
{

  private bool isDisposed = false;

  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (!isDisposed)
    {
      if (disposing)
      {
        // Cleanup managed objects by calling their
        // Dispose() methods.
      }
      // Cleanup unmanaged objects
    }
    isDisposed = true;
  }

  ~ResourceHolder()
  {
    Dispose (false);
  }

  public void SomeMethod()
  {
    // Ensure object not already disposed before execution of any method
    if(isDisposed)
    {
       throw new ObjectDisposedException("ResourceHolder");
    }

    // method implementation...
  }
}
```

从上述代码可以看出，Dispose()方法有第二个 protected 重载方法，它带一个布尔参数，这是真正完成清理工作的方法。Dispose(bool)方法由析构函数和 IDisposable.Dispose()方法调用。这种方式的重点是确保所有的清理代码都放在一个地方。

传递给 Dispose(bool) 方法的参数表示 Dispose(bool) 方法是由析构函数调用，还是由 IDisposable.Dispose()方法调用——Dispose(bool)方法不应从代码的其他地方调用，其原因是：

- 如果使用者调用 IDisposable.Dispose()方法，该使用者就指定应清理所有与该对象相关的资源，包括托管和非托管的资源。
- 如果调用了析构函数，原则上所有的资源仍需要清理。但是在这种情况下，析构函数必须由垃圾回收器调用，而且用户不应试图访问其他托管的对象，因为我们不再能确定它们的状态了。在这种情况下，最好清理已知的非托管资源，希望任何引用的托管对象还有析构函数，这些析构函数执行自己的清理过程。

isDisposed 成员变量表示对象是否已被清理，并确保不试图多次清理成员变量。它还允许在执行实例方法之前测试对象是否已清理，如 SomeMethod()方法所示。这个简单的方法不是线程安全的，需要调用者确保在同一时刻只有一个线程调用方法。要求使用者进行同步是一个合理的假定，在整个.NET 类库中(例如，在集合类中)反复使用了这个假定。第 21 章将讨论线程和同步。

最后，IDisposable.Dispose()方法包含一个对 System.GC.SuppressFinalize()方法的调用。GC 类表示垃圾回收器，SuppressFinalize()方法则告诉垃圾回收器有一个类不再需要调用其析构函数了。因为 Dispose()方法已经完成了所有需要的清理工作，所以析构函数不需要做任何工作。调用 SuppressFinalize()方法就意味着垃圾回收器认为这个对象根本没有析构函数。

14.4　不安全的代码

如前面的章节所述，C#非常擅长于对开发人员隐藏大部分基本内存管理，因为它使用了垃圾回收器和引用。但是，有时需要直接访问内存。例如，由于性能问题，要在外部(非.NET 环境)的 DLL 中访问一个函数，该函数需要把一个指针当作参数来传递(许多 Windows API 函数就是这样)。本节将论述 C#直接访问内存的内容的功能。

14.4.1　用指针直接访问内存

下面把指针当作一个新论题来介绍，而实际上，指针并不是新东西。因为在代码中可以自由使用引用，而引用就是一个类型安全的指针。前面已经介绍了表示对象和数组的变量实际上存储相应数据(被引用者)的内存地址。指针只是一个以与引用相同的方式存储地址的变量。其区别是 C#不允许直接访问在引用变量中包含的地址。有了引用后，从语法上看，变量就可以存储引用的实际内容。

C#引用主要用于使 C#语言易于使用，防止用户无意中执行某些破坏内存中内容的操作。另一方面，使用指针，就可以访问实际内存地址，执行新类型的操作。例如，给地址加上 4 个字节，就可以查看甚至修改存储在新地址中的数据。

下面是使用指针的两个主要原因：

- **向后兼容性**——尽管.NET 运行库提供了许多工具，但仍可以调用本地的 Windows API 函数。对于某些操作这可能是完成任务的唯一方式。这些 API 函数都是用 C++或 C#语言编写的，通常要求把指针作为其参数。但在许多情况下，还可以使用 DllImport 声明，以避免使用指针，例如，使用 System.IntPtr 类。
- **性能**——在一些情况下，速度是最重要的，而指针可以提供最优性能。假定用户知道自己在做什么，就可以确保以最高效的方式访问或处理数据。但是，注意在代码的其他区域中，

不使用指针，也可以对性能进行必要的改进。请使用代码配置文件，查找代码中的瓶颈，VS 中就包含一个代码配置文件。

但是，这种低级的内存访问也是有代价的。使用指针的语法比引用类型的语法更复杂。而且，指针使用起来比较困难，需要非常高的编程技巧和很强的能力，仔细考虑代码所完成的逻辑操作，才能成功地使用指针。如果不仔细，使用指针就很容易在程序中引入细微的、难以查找的错误。例如，很容易重写其他变量，导致栈溢出，访问某些没有存储变量的内存区域，甚至重写.NET 运行库所需要的代码信息，因而使程序崩溃。

另外，如果使用指针，就必须授予代码运行库的代码访问安全机制的高级别信任，否则就不能执行它。在默认的代码访问安全策略中，只有代码运行在本地计算机上，这才是可能的。如果代码必须运行在远程地点，如 Internet，用户就必须给代码授予额外的许可，代码才能工作。除非用户信任你和代码，否则他们不会授予这些许可，第 22 章将讨论代码访问安全性。

尽管有这些问题，但指针在编写高效的代码时是一种非常强大和灵活的工具。

这里强烈建议不要轻易使用指针，否则代码不仅难以编写和调试，而且无法通过 CLR 施加的内存类型安全检查。

1. 用 unsafe 关键字编写不安全的代码

因为使用指针会带来相关的风险，所以 C#只允许在特别标记的代码块中使用指针。标记代码所用的关键字是 unsafe。下面的代码把一个方法标记为 unsafe：

```
unsafe int GetSomeNumber()
{
   // code that can use pointers
}
```

任何方法都可以标记为 unsafe——无论该方法是否应用了其他修饰符(例如，静态方法、虚方法等)。在这种方法中，unsafe 修饰符还会应用到方法的参数上，允许把指针用作参数。还可以把整个类或结构标记为 unsafe，这表示假设所有的成员都是不安全的：

```
unsafe class MyClass
{
   // any method in this class can now use pointers
}
```

同样，可以把成员标记为 unsafe：

```
class MyClass
{
   unsafe int* pX; // declaration of a pointer field in a class
}
```

也可以把方法中的一块代码标记为 unsafe：

```
void MyMethod()
{
```

```
    // code that doesn't use pointers
    unsafe
    {
        // unsafe code that uses pointers here
    }
    // more 'safe' code that doesn't use pointers
}
```

但要注意，不能把局部变量本身标记为 unsafe：

```
int MyMethod()
{
   unsafe int *pX; // WRONG
}
```

如果要使用不安全的局部变量，就需要在不安全的方法或语句块中声明和使用它。在使用指针前还有一步要完成。C#编译器会拒绝不安全的代码，除非告诉编译器代码包含不安全的代码块。标记所用的关键字是 unsafe。因此，要编译包含不安全代码块的文件 MySource.cs(假定没有其他编译器选项)，就要使用下述命令：

```
csc /unsafe MySource.cs
```

或者

```
csc -unsafe MySource.cs
```

如果使用 Visual Studio 2005、2008、2010、2012 或 2013，就可以在项目属性窗口的 Build 选项卡中找到编译不安全代码的选项。

2. 指针的语法

把代码块标记为 unsafe 后，就可以使用下面的语法声明指针：

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

这段代码声明了 4 个变量，pWidth 和 pHeight 是整数指针，pResult 是 double 型指针，pFlags 是字节型的数组指针。我们常常在指针变量名的前面使用前缀 p 来表示这些变量是指针。在变量声明中，符号*表示声明一个指针，换言之，就是存储特定类型的变量的地址。

C++开发人员应注意 C++与 C#中的语法差异。C#语句中的“int*pX, pY;”对应于 C++语句中的“int *pX, *pY;”。在 C#中，*符号与类型相关，而与变量名无关。

声明了指针类型的变量后，就可以用与一般变量相同的方式使用它们，但首先需要学习另外两个运算符：

- &表示“取地址”，并把一个值数据类型转换为指针，例如，int 转换为*int。这个运算符称为寻址运算符。

- *表示“获取地址的内容”，把一个指针转换为值数据类型(例如，*float 转换为 float)。这个运算符称为“间接寻址运算符”(有时称为“取消引用运算符”)。

从这些定义中可以看出，&和*的作用是相反的。

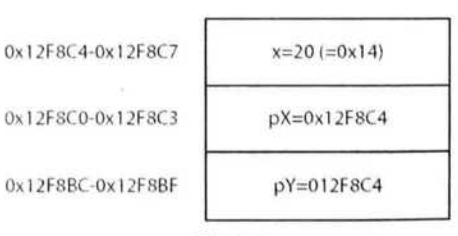
符号&和*也表示按位 AND(&)和乘法(*)运算符，为什么还可以以这种方式使用它们？答案是在实际使用时它们是不会混淆的，用户和编译器总是知道在什么情况下这两个符号有什么含义，因为按照指针的定义，这些符号总是以一元运算符的形式出现——它们只作用于一个变量，并出现在代码中该变量的前面。另一方面，按位 AND 和乘法运算符是二元运算符，它们需要两个操作数。

下面的代码说明了如何使用这些运算符：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
```

首先声明一个整数 x，其值是 10。接着声明两个整数指针 pX 和 pY。然后把 pX 设置为指向 x(换言之，把 pX 的内容设置为 x 的地址)。然后把 pX 的值赋予 pY，所以 pY 也指向 x。最后，在语句 *pY = 20 中，把值 20 赋予 pY 指向的地址包含的内容。实际上是把 x 的内容改为 20，因为 pY 指向 x。注意在这里，变量 pY 和 x 之间没有任何关系。只是此时 pY 碰巧指向存储 x 的存储单元而已。

要进一步理解这个过程，假定 x 存储在栈的存储单元 0x12F8C4～0x12F8C7 中(十进制就是 1 243 332～12 433 35，即有 4 个存储单元，因为一个 int 占用 4 个字节)。因为栈向下分配内存，所以变量 pX 存储在 0x12F8C0～0x12F8C3 的位置上，pY 存储在 0x12F8BC～0x12F8BF 的位置上。注意，pX 和 pY 也分别占用 4 个字节。这不是因为一个 int 占用 4 个字节，而是因为在 32 位处理器上，需要用 4 个字节存储一个地址。利用这些地址，在执行完上述代码后，栈应如图 14-5 所示。



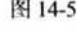


图 14-5

这个示例使用 int 来说明该过程，其中 int 存储在 32 位处理器中栈的连续空间上，但并不是所有的数据类型都会存储在连续的空间中。原因是 32 位处理器最擅长于在 4 个字节的内存块中检索数据。这种计算机上的内存会分解为 4 个字节的块，在 Windows 上，每个块有时称为 DWORD，因为这是 32 位无符号 int 数在.NET 出现之前的名字。这是从内存中获取 DWORD 的最高效的方式——跨越 DWORD 边界存储数据通常会降低硬件的性能。因此，.NET 运行库通常会给某些数据类型填充一些空间，使它们占用的内存是 4 的倍数。例如，short 数据占用两个字节，但如果把一个 short 放在栈中，栈指针仍会向下移动 4 个字节，而不是两个字节，这样，下一个存储在栈中的变量就仍从 DWORD 的边界开始存储。

可以把指针声明为任意一种值类型——即任何预定义的类型 uint、int 和 byte 等，也可以声明为一个结构。但是不能把指针声明为一个类或数组，因为这么做会使垃圾回收器出现问题。为了正常工作，垃圾回收器需要知道在堆上创建了什么类的实例，它们在什么地方。但如果代码开始使用指针处理类，就很容易破坏堆中.NET 运行库为垃圾回收器维护的与类相关的信息。在这里，垃圾回收器可以访问的任何数据类型称为托管类型，而指针只能声明为非托管类型，因为垃圾回收器不能处理它们。

3. 将指针强制转换为整数类型

由于指针实际上存储了一个表示地址的整数，因此任何指针中的地址都可以和任何整数类型之间相互转换。指针到整数类型的转换必须是显式指定的，隐式的转换是不允许的。例如，编写下面的代码是合法的：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

把指针 pX 中包含的地址强制转换为一个 uint，存储在变量 y 中。接着把 y 强制转换回一个 int*，存储在新变量 pD 中。因此 pD 也指向 x 的值。

把指针的值强制转换为整数类型的主要目的是显示它。虽然 Console.Write()和 Console.WriteLine()方法没有带指针的重载方法，但是必须把强制指针的值转换为整数类型，这两个方法才能接受和显示它们：

```
Console.WriteLine("Address is " + pX); // wrong -- will give a
                                       // compilation error
Console.WriteLine("Address is " + (uint)pX); // OK
```

可以把一个指针强制转换为任何整数类型，但是，因为在 32 位系统上，一个地址占用 4 个字节，把指针强制转换为除了 uint、long 或 ulong 之外的数据类型，肯定会导致溢出错误(int 数也可能导致这个问题，因为它的取值范围是–20 亿~20 亿，而地址的取值范围是 0~40 亿)。C#用于 64 位处理器时，一个地址占用 8 个字节。因此在这样的系统上，把指针强制转换为非 ulong 类型，就可能导致溢出错误。

还要注意，checked 关键字不能用于涉及指针的转换。对于这种转换，即使在设置 checked 的情况下，发生溢出时也不会抛出异常。.NET 运行库假定，如果使用指针，就知道自己要做什么，不必担心可能出现的溢出。

4. 指针类型之间的强制转换

也可以在指向不同类型的指针之间进行显式的转换。例如：

```
byte aByte = 8;
byte* pByte= &aByte;
double* pDouble = (double*)pByte;
```

这是一段合法的代码，但如果要执行这段代码，就要小心了。在上面的示例中，如果要查找指针 pDouble 指向的 double 值，就会查找包含 1 个 byte(aByte)的内存，和一些其他内存，并把它当作包含一个 double 值的内存区域来对待——这不会得到一个有意义的值。但是，可以在类型之间转换，实现 C union 类型的等价形式，或者把指针强制转换为其他类型，例如把指针转换为 sbyte，检查内存的单个字节。

5. void 指针

如果要维护一个指针，但不希望指定它指向的数据类型，就可以把指针声明为 void：

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

void 指针的主要用途是调用需要 void*参数的 API 函数。在 C#语言中，使用 void 指针的情况并不是很多。特殊情况下，如果试图使用*运算符取消引用 void 指针，编译器就会标记一个错误。

6. 指针算术的运算

可以给指针加减整数。但是，编译器很智能，知道如何执行这个操作。例如，假定有一个 int 指针，要在其值上加 1。编译器会假定我们要查找 int 后面的存储单元，因此会给该值加上 4 个字节，即加上一个 int 占用的字节数。如果这是一个 double 指针，加 1 就表示在指针的值上加 8 个字节，即一个 double 占用的字节数。只有指针指向 byte 或 sbyte(都是 1 个字节)时，才会给该指针的值加上 1。

可以对指针使用运算符+、–、+=、–=、++和––，这些运算符右边的变量必须是 long 或 ulong 类型。

不允许对 void 指针执行算术运算。

例如，假定有如下定义：

```
uint u = 3;
byte b = 8;
double d = 10.0;
uint* pUint= &u;        // size of a uint is 4
byte* pByte = &b;       // size of a byte is 1
double* pDouble = &d;   // size of a double is 8
```

下面假定这些指针指向的地址是：

- pUint: 1243332
- pByte: 1243328
- pDouble: 1243320

执行这段代码后：

```
++pUint;                    // adds (1*4) = 4 bytes to pUint
pByte -= 3;                 // subtracts (3*1) = 3 bytes from pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

指针应包含的内容是：

- pUint: 1243336
- pByte: 1243325
- pDouble2: 1243352

一般规则是，给类型为 T 的指针加上数值 X，其中指针的值为 P，则得到的结果是 P+ X*(sizeof(T))。使用这条规则时要小心。如果给定类型的连续值存储在连续的存储单元中，指针加法就允许在存储单元之间移动指针。但如果类型是 byte 或 char，其总字节数不是 4 的倍数，连续值就不是默认地存储在连续的存储单元中。

如果两个指针都指向相同的数据类型，则也可以把一个指针从另一个指针中减去。此时，结果是一个 long，其值是指针值的差被该数据类型所占用的字节数整除的结果：

```
double* pD1 = (double*)1243324; // note that it is perfectly valid to
                                // initialize a pointer like this.
double* pD2 = (double*)1243300;
long L = pD1-pD2; // gives the result 3 (=24/sizeof(double))
```

7. sizeof 运算符

这一节将介绍如何确定各种数据类型的大小。如果需要在代码中使用某种类型的大小，就可以使用 sizeof 运算符，它的参数是数据类型的名称，返回该类型占用的字节数。例如：

```
int x = sizeof(double);
```

这将设置 x 的值为 8。

使用 sizeof 的优点是不必在代码中硬编码数据类型的大小，使代码的移植性更强。对于预定义的数据类型，sizeof 返回下面的值。

```
sizeof(sbyte) = 1; sizeof(byte) = 1;
sizeof(short) = 2; sizeof(ushort) = 2;
sizeof(int) = 4; sizeof(uint) = 4;
sizeof(long) = 8; sizeof(ulong) = 8;
sizeof(char) = 2; sizeof(float) = 4;
sizeof(double) = 8; sizeof(bool) = 1;
```

也可以对自己定义的结构使用 sizeof，但此时得到的结果取决于结构中的字段类型。不能对类使用 sizeof。

8. 结构指针：指针成员访问运算符

结构指针的工作方式与预定义值类型的指针的工作方式完全相同。但是这有一个条件：结构不能包含任何引用类型，这是因为前面介绍的一个限制——指针不能指向任何引用类型。为了避免这种情况，如果创建一个指针，它指向包含任何引用类型的任何结构，编译器就会标记一个错误。

假定定义了如下结构：

```
struct MyStruct
```

```
{
  public long X;
  public float F;
}
```

就可以给它定义一个指针：

```
MyStruct* pStruct;
```

然后对其进行初始化：

```
MyStruct Struct = new MyStruct();
pStruct = &Struct;
```

也可以通过指针访问结构的成员值：

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

但是，这个语法有点复杂。因此，C#定义了另一个运算符，用一种比较简单的语法，通过指针访问结构的成员，它称为指针成员访问运算符，其符号是一个短划线，后跟一个大于号，它看起来像一个箭头：->。

C++开发人员认识指针成员访问运算符。因为 C++使用这个符号完成相同的任务。

使用这个指针成员访问运算符，上述代码可以重写为：

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

也可以直接把合适类型的指针设置为指向结构中的一个字段：

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

或者

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

9. 类成员指针

前面说过，不能创建指向类的指针，这是因为垃圾回收器不维护关于指针的任何信息，只维护关于引用的信息，因此创建指向类的指针会使垃圾回收器不能正常工作。

但是，大多数类都包含值类型的成员，可以为这些值类型成员创建指针，但这需要一种特殊的语法。例如，假定把上面示例中的结构重写为类：

```
class MyClass
{
  public long X;
```

```
    public float F;
}
```

然后就可以为它的字段X和F创建指针了，方法与前面一样。但这么做会产生一个编译错误：

```
MyClass myObject = new MyClass();
long* pL = &(myObject.X); // wrong -- compilation error
float* pF = &(myObject.F); // wrong -- compilation error
```

尽管X和F都是非托管类型，但它们嵌入在一个对象中，这个对象存储在堆上。在垃圾回收的过程中，垃圾回收器会把MyObject移动到内存的一个新单元上，这样，pL和pF就会指向错误的存储地址。由于存在这个问题，因此编译器不允许以这种方式把托管类型的成员的地址分配给指针。

解决这个问题的方法是使用 fixed 关键字，它会告诉垃圾回收器，可能有引用某些对象的成员的指针，所以这些对象不能移动。如果要声明一个指针，则使用fixed的语法如下所示：

```
MyClass myObject = new MyClass();
fixed (long* pObject = &(myObject.X))
{
   // do something
}
```

在关键字 fixed 后面的圆括号中，定义和初始化指针变量。这个指针变量(在本例中是 pObject)的作用域是花括号标识的fixed块。这样，垃圾回收器就知道，在执行fixed块中的代码时，不能移动myObject对象。

如果要声明多个这样的指针，就可以在同一个代码块前放置多条fixed语句：

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
fixed (float* pF = &(myObject.F))
{
   // do something
}
```

如果要在不同的阶段固定几个指针，就可以嵌套整个fixed块：

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
{
   // do something with pX
   fixed (float* pF = &(myObject.F))
   {
      // do something else with pF
   }
}
```

如果这些变量的类型相同，就也可以在同一条fixed块中初始化多个变量：

```
MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X))
{
   // etc.
}
```

在上述情况中，是否声明不同的指针，让它们指向相同或不同对象中的字段，或者指向与类实例无关的静态字段，这一点并不重要。

14.4.2 指针示例：PointerPlayground

下面给出一个使用指针的示例：PointerPlayground。它执行一些简单的指针操作，显示结果，还允许查看内存中发生的情况，并确定变量存储在什么地方：

```
using System;

namespace PointerPlayground
{
  class MainEntryPoint
  {
    static unsafe void Main()
    {
      int x=10;
      short y = -1;
      byte y2 = 4;
      double z = 1.5;
      int* pX = &x;
      short* pY = &y;
      double* pZ = &z;

      Console.WriteLine(
        "Address of x is 0x{0:X}, size is {1}, value is {2}",
        (uint)&x, sizeof(int), x);
      Console.WriteLine(
        "Address of y is 0x{0:X}, size is {1}, value is {2}",
        (uint)&y, sizeof(short), y);
      Console.WriteLine(
        "Address of y2 is 0x{0:X}, size is {1}, value is {2}",
        (uint)&y2, sizeof(byte), y2);
      Console.WriteLine(
        "Address of z is 0x{0:X}, size is {1}, value is {2}",
        (uint)&z, sizeof(double), z);
      Console.WriteLine(
        "Address of pX=&x is 0x{0:X}, size is {1}, value is 0x{2:X}",
        (uint)&pX, sizeof(int*), (uint)pX);
      Console.WriteLine(
        "Address of pY=&y is 0x{0:X}, size is {1}, value is 0x{2:X}",
        (uint)&pY, sizeof(short*), (uint)pY);
      Console.WriteLine(
        "Address of pZ=&z is 0x{0:X}, size is {1}, value is 0x{2:X}",
        (uint)&pZ, sizeof(double*), (uint)pZ);

      *pX = 20;
      Console.WriteLine("After setting *pX, x = {0}", x);
      Console.WriteLine("*pX = {0}", *pX);

      pZ = (double*)pX;
      Console.WriteLine("x treated as a double = {0}", *pZ);

      Console.ReadLine();
```

```
        }
    }
}
```

这段代码声明了4个值变量：

- int x
- short y
- byte y2
- double z

它还声明了指向其中3个值的指针：pX、pY和pZ。

然后显示这3个变量的值，以及它们的大小和地址。注意在获取pX、pY和pZ的地址时，我们查看的是指针的指针，即值的地址的地址！还要注意，与显示地址的常见方式一致，在Console.WriteLine()命令中使用{0:X}格式说明符，确保该内存地址以十六进制格式显示。

最后，使用指针pX把x的值改为20，执行一些指针类型强制转换，如果把x的内容当作double类型，就会得到无意义的结果。

编译并运行这段代码，得到下面的结果，其中列出了用/unsafe标志进行编译和不用/unsafe标志进行编译的结果：

```
csc PointerPlayground.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17379
for Microsoft(R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground.cs(7,26): error CS0227: Unsafe code may only appear if
        compiling with /unsafe

csc /unsafe PointerPlayground.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17379
for Microsoft(R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground
Address of x is 0x12F4B0, size is 4, value is 10
Address of y is 0x12F4AC, size is 2, value is -1
Address of y2 is 0x12F4A8, size is 1, value is 4
Address of z is 0x12F4A0, size is 8, value is 1.5
Address of pX=&x is 0x12F49C, size is 4, value is 0x12F4B0
Address of pY=&y is 0x12F498, size is 4, value is 0x12F4AC
Address of pZ=&z is 0x12F494, size is 4, value is 0x12F4A0
After setting *pX, x = 20
*pX = 20
x treated as a double = 2.86965129997082E-308
```

检查这些结果，可以证实"后台内存管理"一节描述的栈操作，即栈向下给变量分配内存。注意，这还证实了栈中的内存块总是按照4个字节的倍数进行分配。例如，y是一个short数(其大小为2字节)，其地址是1 242 284(十进制)，表示为该变量分配的存储单元是1 242 284~1 242 287。如果.NET运行库严格地逐个排列变量，则y应只占用两个存储单元，即1 242 284和1 242 285。

下一个示例PointerPlayground2介绍指针的算术，以及结构指针和类成员指针。开始时，定义一个

结构 CurrencyStruct，它把货币值表示为美元和美分，再定义一个等价的类 CurrencyClass:

```
internal struct CurrencyStruct
{
  public long Dollars;
  public byte Cents;

  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}

internal class CurrencyClass
{
  public long Dollars;
  public byte Cents;

  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}
```

定义好结构和类后，就可以对它们应用指针了。下面的代码是一个新的示例。这段代码比较长，我们对此将做详细讲解。首先显示 CurrencyStruct 结构的字节数，创建它的两个实例和一些指针，然后使用 pAmount 指针初始化一个 CurrencyStruct 结构 amount1 的成员，显示变量的地址：

```
public static unsafe void Main()
{
  Console.WriteLine(
     "Size of CurrencyStruct struct is " + sizeof(CurrencyStruct));
  CurrencyStruct amount1, amount2;
  CurrencyStruct* pAmount = &amount1;
  long* pDollars = &(pAmount->Dollars);
  byte* pCents = &(pAmount->Cents);

  Console.WriteLine("Address of amount1 is 0x{0:X}", (uint)&amount1);
  Console.WriteLine("Address of amount2 is 0x{0:X}", (uint)&amount2);
  Console.WriteLine("Address of pAmount is 0x{0:X}", (uint)&pAmount);
  Console.WriteLine("Address of pDollars is 0x{0:X}", (uint)&pDollars);
  Console.WriteLine("Address of pCents is 0x{0:X}", (uint)&pCents);
  pAmount->Dollars = 20;
  *pCents = 50;
  Console.WriteLine("amount1 contains " + amount1);
```

现在根据栈的工作方式，执行一些指针操作。因为变量是按顺序声明的，所以 amount2 存储在 amount1 后面的地址中，sizeof(CurrencyStruct)运算符返回 16(见后面的屏幕输出)，所以 CurrencyStruct 结构占用的字节数是 4 的倍数。在递减了 Currency 指针后，它就指向 amount2：

```
--pAmount; // this should get it to point to amount2
Console.WriteLine("amount2 has address 0x{0:X} and contains {1}",
   (uint)pAmount, *pAmount);
```

在调用 Console.WriteLine()语句时，它显示了 amount2 的内容，但还没有对它进行初始化。显示出来的东西就是随机的垃圾——在执行该示例前内存中存储在该单元中的内容。但这有一个要点：一般情况下，C#编译器会禁止使用未初始化的变量，但在开始使用指针时，就很容易绕过许多通常的编译检查。此时我们这么做，是因为编译器无法知道我们实际上要显示的是 amount2 的内容。因为知道了栈的工作方式，所以可以说出递减 pAmount 的结果是什么。使用指针算法，可以访问编译器通常禁止访问的各种变量和存储单元，因此指针算法是不安全的。

接下来在 pCents 指针上进行指针运算。pCents 指针目前指向 amount1.Cents，但此处的目的是使用指针算法让它指向 amount2.Cents，而不是直接告诉编译器我们要做什么。为此，需要从 pCents 指针所包含的地址中减去 sizeof(Currency)：

```
// do some clever casting to get pCents to point to cents
// inside amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Address of pCents is now 0x{0:X}", (uint)&pCents);
```

最后，使用 fixed 关键字创建一些指向类实例中字段的指针，使用这些指针设置这个实例的值。注意，这也是我们第一次查看存储在堆中(而不是栈)的项的地址：

```
Console.WriteLine("\nNow with classes");
// now try it out with classes
CurrencyClass amount3 = new CurrencyClass();

fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
  Console.WriteLine(
    "amount3.Dollars has address 0x{0:X}", (uint)pDollars2);
  Console.WriteLine(
    "amount3.Cents has address 0x{0:X}", (uint) pCents2);
  *pDollars2 = -100;
  Console.WriteLine("amount3 contains " + amount3);
}
```

编译并运行这段代码，得到如下所示的结果：

```
csc /unsafe PointerPlayground2.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground2
Size of CurrencyStruct struct is 16
Address of amount1 is 0x12F4A4
Address of amount2 is 0x12F494
Address of pAmount is 0x12F490
Address of pDollars is 0x12F48C
Address of pCents is 0x12F488
amount1 contains $20.50
amount2 has address 0x12F494 and contains $0.0
Address of pCents is now 0x12F488
```

```
Now with classes
amount3.Dollars has address 0xA64414
amount3.Cents has address 0xA6441C
amount3 contains $-100.0
```

注意在这个结果中，显示了未初始化的 amount2 的值，CurrencyStruct 结构的字节数是 16，大于其字段的字节数(一个 long 数占用 8 个字节，加上 1 个字节等于 9 个字节)。

14.4.3　使用指针优化性能

前面用许多篇幅介绍了使用指针可以完成的各种任务，但在前面的示例中，仅是处理内存，让有兴趣的人们了解实际上发生了什么事，并没有帮助人们编写出更好的代码！本节将应用我们对指针的理解，用一个示例来说明使用指针可以大大提高性能。

1. 创建基于栈的数组

本节将探讨指针的一个主要应用领域：在栈中创建高性能、低系统开销的数组。第 2 章介绍了 C#如何支持数组的处理。C#很容易使用一维数组和矩形或锯齿形多维数组，但有一个缺点：这些数组实际上都是对象，它们是 System.Array 的实例。因此数组存储在堆上，这会增加系统开销。有时，我们希望创建一个使用时间比较短的高性能数组，不希望有引用对象的系统开销。而使用指针就可以做到，但指针只对于一维数组比较简单。

为了创建一个高性能的数组，需要使用另一个关键字：stackalloc。stackalloc 命令指示.NET 运行库在栈上分配一定量的内存。在调用 stackalloc 命令时，需要为它提供两条信息：

- 要存储的数据类型
- 需要存储的数据项数

例如，要分配足够的内存，以存储 10 个 decimal 数据项，可以编写下面的代码：

```
decimal* pDecimals = stackalloc decimal[10];
```

注意，这条命令只分配栈内存。它不会试图把内存初始化为任何默认值，这正好符合我们的目的。因为要创建一个高性能的数组，给它不必要地初始化相应值会降低性能。

同样，要存储 20 个 double 数据项，可以编写下面的代码：

```
double* pDoubles = stackalloc double[20];
```

虽然这行代码指定把变量的个数存储为一个常数，但它等于在运行时计算的一个数字。所以可以把上面的示例写为：

```
int size;
size = 20; // or some other value calculated at runtime
double* pDoubles = stackalloc double[size];
```

从这些代码段中可以看出，stackalloc 的语法有点不寻常。它的后面紧跟要存储的数据类型名(该数据类型必须是一个值类型)，之后把需要的项数放在方括号中。分配的字节数是项数乘以 sizeof(数据类型)。在这里，使用方括号表示这是一个数组。如果给 20 个 double 数分配存储单元，就得到了一个有 20 个元素的 double 数组，最简单的数组类型是逐个存储元素的内存块，如图 14-6 所示。

在图 14-6 中，显示了 stackalloc 返回的指针，stackalloc 总是返回分配数据类型的指针，它指向新分配内存块的顶部。要使用这个内存块，可以取消对已返回指针的引用。例如，给 20 个 double 数分配内存后，把第一个元素(数组的元素 0)设置为 3.0，可以编写下面的代码：

```
double* pDoubles = stackalloc double[20];
*pDoubles = 3.0;
```

stackalloc 返回的指针

在栈上分配的连续存储单元

数组的第 0 个元素

数组的第 1 个元素

数组的第 2 个元素

等

图 14-6

要访问数组的下一个元素，可以使用指针算法。如前所述，如果给一个指针加 1，它的值就会增加它指向的数据类型的字节数。在本例中，就会把指针指向已分配的内存块中的下一个空闲存储单元。因此可以把数组的第二个元素(元素编号为 1)设置为 8.4：

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

同样，可以用表达式*(pDoubles+X)访问数组中下标为 X 的元素。

这样，就得到一种访问数组中元素的方式，但对于一般目的，使用这种语法过于复杂。C#为此定义了另一种语法。对指针应用方括号时，C#为方括号提供了一种非常精确的含义。如果变量 p 是任意指针类型，X 是一个整数，表达式 p[X]就被编译器解释为*(p+X)，这适用于所有的指针，不仅仅是用 stackalloc 初始化的指针。利用这个简捷的记号，就可以用一种非常方便的语法访问数组。实际上，访问基于栈的一维数组所使用的语法与访问基于堆的、由 System.Array 类表示的数组完全相同：

```
double* pDoubles = stackalloc double [20];
pDoubles[0] = 3.0; // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4; // pDoubles[1] is the same as *(pDoubles+1)
```

把数组的语法应用于指针并不是新东西。自从开发出 C 和 C++语言以来，它就是这两种语言的基础部分。实际上，C++开发人员会把这里用 stackalloc 获得的、基于栈的数组完全等同于传统的基于栈的 C 和 C++数组。这种语法和指针与数组的链接方式是 C 语言在 20 世纪 70 年代后期流行起来的原因之一，也是指针的使用成为 C 和 C++中一种流行的编程技巧的主要原因。

尽管高性能的数组可以用与一般 C#数组相同的方式访问，但需要注意：在 C#中，下面的代码会抛出一个异常：

```
double[] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

抛出异常的原因是：使用越界的下标来访问数组：下标是 50，而允许的最大下标是 19。但是，如果使用 stackalloc 声明了一个等价的数组，对数组进行边界检查时，这个数组中就没有封装任何对

象，因此下面的代码不会抛出异常：

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

在这段代码中，我们分配了足够的内存来存储 20 个 double 类型的数。接着把 sizeof(double)存储单元的起始位置设置为该存储单元的起始位置加上 50*sizeof(double)个存储单元，来保存双精度值 3.0。但这个存储单元超出了刚才为 double 数分配的内存区域。谁也不知道这个地址存储了什么数据。最好是只使用某个当前未使用的内存，但所重写的存储单元也有可能是在栈上用于存储其他变量，或者是某个正在执行的方法的返回地址。因此，使用指针获得高性能的同时，也会付出一些代价：需要确保自己知道在做什么，否则就会抛出非常古怪的运行错误。

2. QuickArray 示例

下面用一个 stackalloc 示例 QuickArray 来结束关于指针的讨论。在这个示例中，程序仅要求用户提供为数组分配的元素数。然后代码使用 stackalloc 给 long 型数组分配一定的存储单元。这个数组的元素是从 0 开始的整数的平方，结果显示在控制台上：

```
using System;

namespace QuickArray
{
  internal class Program
  {
    private static unsafe void Main()
    {
      Console.Write("How big an array do you want? \n> ");
      string userInput = Console.ReadLine();
      uint size = uint.Parse(userInput);

      long* pArray = stackalloc long[(int) size];
      for (int i = 0; i < size; i++)
      {
        pArray[i] = i*i;
      }

      for (int i = 0; i < size; i++)
      {
        Console.WriteLine("Element {0} = {1}", i, *(pArray + i));
      }

      Console.ReadLine();
    }
  }
}
```

运行这个示例，得到如下所示的结果：

```
How big an array do you want?
> 15
Element 0 = 0
Element 1 = 1
Element 2 = 4
Element 3 = 9
```

```
Element 4 = 16
Element 5 = 25
Element 6 = 36
Element 7 = 49
Element 8 = 64
Element 9 = 81
Element 10 = 100
Element 11 = 121
Element 12 = 144
Element 13 = 169
Element 14 = 196
_
```

14.5 小结

要想成为真正优秀的 C#程序员，必须牢固掌握存储单元和垃圾回收的工作原理。本章描述了 CLR 管理以及在堆和栈上分配内存的方式，讨论了如何编写正确地释放非托管资源的类，并介绍如何在 C#中使用指针，这些都是很难理解的高级主题，初学者常常不能正确实现。

本章为第 16 章的错误处理和第 21 章的使用线程打下了基础。第 15 章介绍 C#中的反射技术。

第15章

反　　射

本章要点

- 使用自定义特性
- 在运行期间使用反射检查元数据
- 从支持反射的类中构建访问点

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- LookupWhatsNew
- TypeView
- VectorClass
- WhatsNewAttributes

15.1　在运行期间处理和检查代码

本章讨论自定义特性和反射。自定义特性允许把自定义元数据与程序元素关联起来。这些元数据是在编译过程中创建的，并嵌入到程序集中。反射是一个普通术语，它描述了在运行过程中检查和处理程序元素的功能。例如，反射允许完成以下任务：

- 枚举类型的成员
- 实例化新对象
- 执行对象的成员
- 查找类型的信息
- 查找程序集的信息
- 检查应用于某种类型的自定义特性
- 创建和编译新程序集

这个列表列出了许多功能，包括.NET Framework 类库提供的一些最强大、最复杂的功能。但本章不可能介绍反射的所有功能，仅讨论最常用的功能。

为了说明自定义特性和反射，我们将开发一个示例，说明公司如何定期升级软件，自动记录升级的信息。在这个示例中，要定义几个自定义特性，表示程序元素最后修改的日期，以及发生了什么变化。然后使用反射开发一个应用程序，它在程序集中查找这些特性，自动显示软件自某个给定日期以来升级的所有信息。

本章要讨论的另一个示例是一个应用程序，该程序从数据库中读取信息或把信息写入数据库，并使用自定义特性，把类和属性标记为对应的数据库表和列。然后在运行期间从程序集中读取这些特性，使程序可以自动从数据库的相应位置检索或写入数据，无须为每个表或每一列编写特定的逻辑。

15.2 自定义特性

前面介绍了如何在程序的各个数据项上定义特性。这些特性都是 Microsoft 定义好的，作为.NET Framework 类库的一部分，许多特性都得到了 C#编译器的支持。对于这些特殊的特性，编译器可以以特殊的方式定制编译过程，例如，可以根据 StructLayout 特性中的信息在内存中布置结构。

.NET Framework 也允许用户定义自己的特性。显然，这些特性不会影响编译过程，因为编译器不能识别它们，但这些特性在应用于程序元素时，可以在编译好的程序集中用作元数据。

这些元数据在文档说明中非常有用。但是，使自定义特性非常强大的因素是使用反射，代码可以读取这些元数据，使用它们在运行期间作出决策。也就是说，自定义特性可以直接影响代码运行的方式。例如，自定义特性可以用于支持对自定义许可类进行声明性的代码访问安全检查，把信息与程序元素关联起来，程序元素由测试工具使用，或者在开发可扩展的架构时，允许加载插件或模块。

15.2.1 编写自定义特性

为了理解编写自定义特性的方式，应了解一下在编译器遇到代码中某个应用了自定义特性的元素时，该如何处理。以数据库为例，假定有一个 C#属性声明，如下所示。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
   get {
      // etc.
```

当 C#编译器发现这个属性(property)应用了一个 FieldName 特性时，首先会把字符串 Attribute 追加到这个名称的后面，形成一个组合名称 FieldNameAttribute，然后在其搜索路径的所有名称空间(即在 using 语句中提及的名称空间)中搜索有指定名称的类。但要注意，如果用一个特性标记数据项，而该特性的名称以字符串 Attribute 结尾，编译器就不会把该字符串加到组合名称中，而是不修改该特性名。因此，上面的代码等价于：

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
   get {
      // etc.
```

编译器会找到含有该名称的类，且这个类直接或间接派生自 System.Attribute。编译器还认为这个类包含控制特性用法的信息。特别是属性类需要指定：

- 特性可以应用到哪些类型的程序元素上(类、结构、属性和方法等)
- 它是否可以多次应用到同一个程序元素上
- 特性在应用到类或接口上时，是否由派生类和接口继承
- 这个特性有哪些必选和可选参数

如果编译器找不到对应的特性类，或者找到一个这样的特性类，但使用特性的方式与特性类中的信息不匹配，编译器就会产生一个编译错误。例如，如果特性类指定该特性只能应用于类，但我们把它应用到结构定义上，就会产生一个编译错误。

继续上面的示例，假定定义了一个 FieldName 特性：

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false,
  Inherited=false)]
public class FieldNameAttribute: Attribute
{
  private string name;
  publicFieldNameAttribute(string name)
  {
    this.name = name;
  }
}
```

下面几节讨论这个定义中的每个元素。

1. AttributeUsage 特性

要注意的第一个问题是特性(attribute)类本身用一个特性——System.AttributeUsage 特性来标记。这是 Microsoft 定义的一个特性，C#编译器为它提供了特殊的支持(你可能认为 AttributeUsage 根本不是一个特性，它更像一个元特性，因为它只能应用到其他特性上，不能应用到类上)。AttributeUsage 主要用于标识自定义特性可以应用到哪些类型的程序元素上。这些信息由它的第一个参数给出，该参数是必选的，其类型是枚举类型 AttributeTargets。在上面的示例中，指定 FieldName 特性只能应用到属性(property)上——这是因为我们在前面的代码段中把它应用到属性上。AttributeTargets 枚举的成员如下：

- All
- Assembly
- Class
- Constructor
- Delegate
- Enum
- GenericParameter(仅.NET 2.0 及更高版本提供)
- Interface
- Method
- Module

- Parameter
- Property
- ReturnValue
- Struct

这个列表列出了可以应用该特性的所有程序元素。注意在把特性应用到程序元素上时，应把特性放在元素前面的方括号中。但是，在上面的列表中，有两个值不对应于任何程序元素：Assembly 和 Module。特性可以应用到整个程序集或模块中，而不是应用到代码中的一个元素上，在这种情况下，这个特性可以放在源代码的任何地方，但需要用关键字 Assembly 或 Module 作为前缀：

```
[assembly:SomeAssemblyAttribute(Parameters)]
[module:SomeAssemblyAttribute(Parameters)]
```

在指定自定义特性的有效目标元素时，可以使用按位 OR 运算符把这些值组合起来。例如，如果指定 FieldName 特性可以同时应用到属性和字段上，可以编写下面的代码：

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
  AllowMultiple=false,
  Inherited=false)]
public class FieldNameAttribute: Attribute
```

也可以使用 AttributeTargets.All 指定自定义特性可以应用到所有类型的程序元素上。AttributesUsage 特性还包含另外两个参数：AllowMultiple 和 Inherited。它们用不同的语法来指定：<ParameterName>=<ParameterValue>，而不是只给出这些参数的值。这些参数是可选的，根据需要，可以忽略它们。

AllowMultiple 参数表示一个特性是否可以多次应用到同一项上，这里把它设置为 false，表示如果编译器遇到下述代码，就会产生一个错误：

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{

  // etc.
```

如果把 Inherited 参数设置为 true，就表示应用到类或接口上的特性也可以自动应用到所有派生的类或接口上。如果特性应用到方法或属性上，它就可以自动应用到该方法或属性等的重写版本上。

2. 指定特性参数

下面介绍如何指定自定义特性接受的参数。在编译器遇到下述语句时：

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{

  // etc.
```

编译器会检查传递给特性的参数(在本例中，是一个字符串)，并查找该特性中带这些参数的

构造函数。如果编译器找到一个这样的构造函数，编译器就会把指定的元数据传递给程序集。如果编译器找不到，就生成一个编译错误。如后面所述，反射会从程序集中读取元数据(特性)，并实例化它们表示的特性类。因此，编译器需要确保存在这样的构造函数，才能在运行期间实例化指定的特性。

在本例中，仅为 FieldNameAttribute 类提供了一个构造函数，而这个构造函数有一个字符串参数。因此，在把 FieldName 特性应用到一个属性上时，必须为它提供一个字符串参数，如上面的代码所示。

如果可以选择特性提供的参数类型，就可以提供构造函数的不同重载方法，尽管一般是仅提供一个构造函数，使用属性来定义任何其他可选参数，下面将介绍可选参数。

3. 指定特性的可选参数

在 AttributeUsage 特性中，可以使用另一种语法，把可选参数添加到特性中。这种语法指定可选参数的名称和值，它通过特性类中的公共属性或字段起作用。例如，假定修改 SocialSecurityNumber 属性的定义，如下所示：

```
[FieldName("SocialSecurityNumber", Comment="This is the primary key field")]
public string SocialSecurityNumber
{

   // etc.
```

在本例中，编译器识别第二个参数的语法<ParameterName>=<ParameterValue>，并且不会把这个参数传递给 FieldNameAttribute 类的构造函数，而是查找一个有该名称的公共属性或字段(最好不要使用公共字段，所以一般情况下要使用特性)，编译器可以用这个属性设置第二个参数的值。如果希望上面的代码工作，就必须给 FieldNameAttribute 类添加一些代码：

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false,
  Inherited=false)]
public class FieldNameAttribute: Attribute
{
  private string comment;
  public string Comment
  {
    get
    {
      return comment;
    }
    set
    {
      comment = value;
    }
  }

    // etc
}
```

15.2.2　自定义特性示例：WhatsNewAttributes

本节开始编写前面描述过的示例 WhatsNewAttributes，该示例提供了一个特性，表示最后一次修改程序元素的时间。这个示例比前面所有的示例都复杂，因为它包含 3 个不同的程序集：

- WhatsNewAttributes 程序集，它包含特性的定义。
- VectorClass 程序集，它包含所应用的特性的代码。
- LookUpWhatsNew 程序集，它包含显示已改变的数据项详细信息的项目。

其中，只有 LookUpWhatsNew 程序集是目前为止使用的一个控制台应用程序，其余两个程序集都是库，它们都包含类的定义，但都没有程序的入口点。对于 VectorClass 程序集，我们使用了 VectorAsCollection 示例，但从中删除了入口点和测试代码类，只剩下 Vector 类。这些类详见本章后面的内容。

在命令行上编译，以此管理 3 个相关的程序集要求较高的技巧。尽管分别给出编译这 3 个源文件的命令，但也可以编辑代码示例(可以从 Wrox 网站 www.wrox.com 上下载)，作为一个组合的 Visual Studio 解决方案，详见第 17 章。下载的文件包含所需的 Visual Studio 2013 解决方案文件。

1. WhatsNewAttributes 库程序集

首先从核心的 WhatsNewAttributes 程序集开始。其源代码包含在 WhatsNewAttributes.cs 文件中，该文件位于本章示例代码中 WhatsNewAttributes 解决方案的 WhatsNewAttributes 项目中。编译的语法非常简单：在命令行上，给编译器提供 target:library 标记即可。要编译 WhatsNewAttributes 程序集，输入：

```
csc /target:libraryWhatsNewAttributes.cs
```

WhatsNewAttributes.cs 文件定义了两个特性类 LastModifiedAttribute 和 SupportsWhatsNew-Attribute。LastModifiedAttribute 特性可以用于标记最后一次修改数据项的时间，它有两个必选参数(这两个参数传递给构造函数)：修改的日期和包含描述修改信息的字符串。它还有一个可选参数 issues (表示存在一个公共属性)，它可以用来描述该数据项的任何重要问题。

在现实生活中，或许想把特性应用到任何对象上。为了使代码比较简单，这里仅允许将它应用于类和方法，并允许它多次应用到同一项上(AllowMultiple=true)，因为可以多次修改某一项，每次修改都需要用一个不同的特性实例来标记。

SupportsWhatsNew 是一个较小的类，它表示不带任何参数的特性。这个特性是一个程序集的特性，它用于把程序集标记为通过 LastModifiedAttribute 维护的文档。这样，以后查看这个程序集的程序会知道，它读取的程序集是我们使用自动文档过程生成的那个程序集。这部分示例的完整源代码如下所示(代码文件 WhatsNewAttributes.cs)：

```
using System;

namespace WhatsNewAttributes
{
  [AttributeUsage(
    AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple=true, Inherited=false)]
  public class LastModifiedAttribute: Attribute
  {
```

```
        private readonly DateTime _dateModified;
        private readonly string _changes;

        public LastModifiedAttribute(string dateModified, string changes)
        {
            dateModified = DateTime.Parse(dateModified);
            _changes = changes;
        }

        public DateTime DateModified
        {
           get { return _dateModified; }
        }

        public string Changes
        {
           get { return _changes; }
        }

        public string Issues { get; set; }
    }

    [AttributeUsage(AttributeTargets.Assembly)]
    public class SupportsWhatsNewAttribute: Attribute
    {
    }
}
```

从上面的描述可以看出，上面的代码非常简单。但要注意，不必提供 Changes 和 DateModified 属性的 set 访问器，不需要这些访问器是因为，在构造函数中这些参数都设置为必选参数。需要 get 访问器，是为了以后可以读取这些特性的值。

2. VectorClass 程序集

本节就使用这些特性，我们用前面的 VectorAsCollection 示例的修订版本来说明。注意这里需要引用刚才创建的 WhatsNewAttributes 库，还需要使用 using 语句指定相应的名称空间，这样编译器才能识别这些特性：

```
using System;
using System.Collections;
using System.Text;
using WhatsNewAttributes;

[assembly: SupportsWhatsNew]
```

在这段代码中，添加了一行用 SupportsWhatsNew 特性标记程序集本身的代码。

下面考虑 Vector 类的代码。我们并不是真的要修改这个类中的某些主要内容，只是添加两个 LastModified 特性，以标记出本章对 Vector 类进行的操作。把 Vector 定义为一个类，而不是结构，以简化后面显示特性所编写的代码(在 VectorAsCollection 示例中，Vector 是一个结构，但其枚举器是一个类。于是，在查看程序集时，这个示例的下一次迭代必须同时考虑类和结构。这会使例子比较复杂)。

```
namespace VectorClass
{
  [LastModified("14 Feb 2010", "IEnumerable interface implemented " +
    "So Vector can now be treated as a collection")]
  [LastModified("10 Feb 2010", "IFormattable interface implemented " +
    "So Vector now responds to format specifiers N and VE")]
  class Vector: IFormattable, IEnumerable
  {
    public double x, y, z;

    public Vector(double x, double y, double z)
    {
      this.x = x;
      this.y = y;
      this.z = z;
    }

    [LastModified("10 Feb 2010",
                  "Method added in order to provide formatting support")]
    public string ToString(string format, IFormatProviderformatProvider)
    {
      if (format == null)
      {
        returnToString();
      }
```

再把包含的 VectorEnumerator 类标记为 new：

```
[LastModified("14 Feb 2010",
              "Class created as part of collection support for Vector")]
private class VectorEnumerator: IEnumerator
{
```

为了在命令行上编译这段代码，应输入下面的命令：

```
csc /target:library /reference:WhatsNewAttributes.dll VectorClass.cs
```

上面是这个示例的代码。目前还不能运行它，因为我们只有两个库。在描述了反射的工作原理后，就介绍这个示例的最后一部分，从中可以查看和显示这些特性。

15.3　反射

本节先介绍 System.Type 类，通过这个类可以访问关于任何数据类型的信息。然后简要介绍 System.Reflection.Assembly 类，它可以用于访问给定程序集的相关信息，或者把这个程序集加载到程序中。最后把本节的代码和上一节的代码结合起来，完成 WhatsNewAttributes 示例。

15.3.1　System.Type 类

这里使用 Type 类只为了存储类型的引用：

```
Typet = typeof(double);
```

我们以前把 Type 看作一个类，但它实际上是一个抽象的基类。只要实例化了一个 Type 对象，实际上就实例化了 Type 的一个派生类。尽管一般情况下派生类只提供各种 Type 方法和属性的不同重载，但是这些方法和属性返回对应数据类型的正确数据，Type 有与每种数据类型对应的派生类。它们一般不添加新的方法或属性。通常，获取指向任何给定类型的 Type 引用有 3 种常用方式：

- 使用C#的typeof运算符，如上述代码所示。这个运算符的参数是类型的名称(但不放在引号中)。
- 使用 GetType()方法，所有的类都会从 System.Object 继承这个方法。

```
double d = 10;
Type t = d.GetType();
```

在一个变量上调用 GetType()方法，而不是把类型的名称作为其参数。但要注意，返回的 Type 对象仍只与该数据类型相关：它不包含与该类型的实例相关的任何信息。如果引用了一个对象，但不能确保该对象实际上是哪个类的实例，这个方法就很有用。

- 还可以调用 Type 类的静态方法 GetType()：

```
Type t = Type.GetType("System.Double");
```

Type 是许多反射功能的入口。它实现许多方法和属性，这里不可能列出所有的方法和属性，而主要介绍如何使用这个类。注意，可用的属性都是只读的：可以使用 Type 确定数据的类型，但不能使用它修改该类型！

1. Type 的属性

由 Type 实现的属性可以分为下述 3 类。首先，许多属性都可以获取包含与类相关的各种名称的字符串，如表 15-1 所示。

表 15-1

属　性	返　回　值
Name	数据类型名
FullName	数据类型的完全限定名(包括名称空间名)
Namespace	在其中定义数据类型的名称空间名

其次，属性还可以进一步获取 Type 对象的引用，这些引用表示相关的类，如表 15-2 所示。

表 15-2

属　性	返回对应的 Type 引用
BaseType	该 Type 的直接基本类型
UnderlyingSystemType	该 Type 在.NET 运行库中映射到的类型(某些.NET 基类实际上映射到由 IL 识别的特定预定义类型)

许多布尔属性表示这种类型是一个类，还是一个枚举等。这些特性包括 IsAbstract、IsArray、IsClass、IsEnum、IsInterface、IsPointer、IsPrimitive(一种预定义的基元数据类型)、IsPublic、IsSealed 和 IsValueType。例如，使用一种基元数据类型：

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract);  // writes false
Console.WriteLine(intType.IsClass);     // writes false
Console.WriteLine(intType.IsEnum);      // writes false
Console.WriteLine(intType.IsPrimitive); // writes true
Console.WriteLine(intType.IsValueType); // writes true
```

或者使用 Vector 类:

```
Type vecType = typeof(Vector);
Console.WriteLine(vecType.IsAbstract);  // writes false
Console.WriteLine(vecType.IsClass);     // writes true
Console.WriteLine(vecType.IsEnum);      // writes false
Console.WriteLine(vecType.IsPrimitive); // writes false
Console.WriteLine(vecType.IsValueType); // writes false
```

也可以获取在其中定义该类型的程序集的引用,该引用作为 System.Reflection.Assembly 类的实例的一个引用来返回:

```
Type t = typeof (Vector);
Assembly contai6ningAssembly = new Assembly(t);
```

2. 方法

System.Type 的大多数方法都用于获取对应数据类型的成员信息:构造函数、属性、方法和事件等。它有许多方法,但它们都有相同的模式。例如,有两个方法可以获取数据类型的方法的细节信息:GetMethod()和 GetMethods()。GetMethod()方法返回 System.Reflection.MethodInfo 对象的一个引用,其中包含一个方法的细节信息。GetMethods()返回这种引用的一个数组。其区别是 GetMethods()方法返回所有方法的细节信息;而 GetMethod()方法返回一个方法的细节信息,其中该方法包含特定的参数列表。这两个方法都有重载方法,重载方法有一个附加的参数,即 BindingFlags 枚举值,该值表示应返回哪些成员,例如,返回公有成员、实例成员和静态成员等。

例如,GetMethods()最简单的一个重载方法不带参数,返回数据类型所有公共方法的信息:

```
Type t = typeof(double);
MethodInfo[] methods = t.GetMethods();
foreach (MethodInfonextMethod in methods)
{
  // etc.
    }
```

Type 的成员方法如表 15-3 所示,遵循同一个模式。注意名称为复数形式的方法返回一个数组。

表 15-3

返回的对象类型	方　法
ConstructorInfo	GetConstructor(),GetConstructors()
EventInfo	GetEvent(),GetEvents()
FieldInfo	GetField(),GetFields()
MemberInfo	GetMember(),GetMembers(),GetDefaultMembers()
MethodInfo	GetMethod(),GetMethods()
PropertyInfo	GetProperty(),GetProperties()

GetMember()和 GetMembers()方法返回数据类型的任何成员或所有成员的详细信息，不管这些成员是构造函数、属性和方法等。

15.3.2 TypeView 示例

下面用一个短小的示例 TypeView 来说明 Type 类的一些功能，这个示例可以用来列出数据类型的所有成员。本例主要说明对于 double 型 TypeView 的用法，也可以修改该样例中的一行代码，使用其他的数据类型。TypeView 提供的信息要比在控制台窗口中显示的信息多得多，所以我们将打破常规，在一个消息框中显示这些信息。对于一个 double 数运行 TypeView 示例，结果如图 15-1 所示。

```
Analysis of type Double

Type Name: Double
Full Name: System.Double
Namespace: System
Base Type:ValueType
UnderlyingSystem Type:Double

PUBLIC MEMBERS:
System.Double Method IsInfinity
System.Double Method IsPositiveInfinity
System.Double Method IsNegativeInfinity
System.Double Method IsNaN
System.Double Method CompareTo
System.Double Method CompareTo
System.Double Method Equals
System.Double Method Equals
System.Double Method GetHashCode
System.Double Method ToString
System.Double Method ToString
System.Double Method ToString
System.Double Method ToString
System.Double Method Parse
System.Double Method Parse
System.Double Method Parse
System.Double Method Parse
System.Double Method TryParse
System.Double Method TryParse
System.Double Method GetTypeCode
System.Object Method GetType
System.Double Field MinValue
System.Double Field MaxValue
System.Double Field Epsilon
System.Double Field NegativeInfinity
System.Double Field PositiveInfinity
System.Double Field NaN

OK
```

图 15-1

该消息框显示了数据类型的名称、全名和名称空间，以及底层类型和基类的名称。然后，它迭代该数据类型的所有公有实例成员，显示所声明类型的每个成员、成员的类型(方法、字段等)以及成员的名称。声明类型是实际声明类型成员的类的名称(例如，如果在 System.Double 中定义或重载它，该声明类型就是 System.Double，如果成员继承自某个基类，该声明类型就是相关基类的名称)。

TypeView 不会显示方法的签名，因为我们是通过 MemberInfo 对象获取所有公有实例成员的详细信息，参数的相关信息不能通过 MemberInfo 对象来获得。为了获取该信息，需要引用 MemberInfo 和其他更特殊的对象，即需要分别获取每一种类型的成员的详细信息。

TypeView 会显示所有公有实例成员的详细信息，但对于 double 类型，仅定义了字段和方法。对于本示例，把 TypeView 编译为一个控制台应用程序，可以从控制台应用程序中显示消息框。但是，使用消息框就意味着需要引用基类程序集 System.Windows.Forms.dll，它包含 System.Windows.Forms 名称空间中的类，在这个名称空间中，定义了我们需要的 MessageBox 类。下面列出 TypeView 的代码。开始时需要添加几条 using 语句：

```
using System;
usingSystem.Reflection;
usingSystem.Text;
usingSystem.Windows.Forms;
```

需要 System.Text 的原因是我们要使用 StringBuilder 对象构建在消息框中显示的文本，以及消息框本身的 System.Windows.Forms。全部代码都放在 MainClass 一个类中，这个类包含两个静态方法和一个静态字段，StringBuilder 的一个实例叫作 OutputText，OutputText 用于创建在消息框中显示的文本。Main()方法和类的声明如下所示：

```
classMainClass
{
  staticStringBuilderOutputText = new StringBuilder();

  static void Main()
  {
    // modify this line to retrieve details of any
```

```
        // other data type
        Type t = typeof(double);

        AnalyzeType(t);
        MessageBox.Show(OutputText.ToString(), "Analysis of type "
                                                + t.Name);
        Console.ReadLine();
    }
```

实现的 Main()方法首先声明一个 Type 对象，来表示我们选择的数据类型，再调用方法 AnalyzeType()，AnalyzeType()方法从 Type 对象中提取信息，并使用该信息构建输出文本。最后在消息框中显示输出。使用 MessageBox 类非常直观：只需要调用其静态方法 Show()，给它传递两个字符串，两个字符串分别为消息框中的文本和标题。这些都由 AnalyzeType()方法来完成：

```
static void AnalyzeType(Type t)
{
  AddToOutput("Type Name: " + t.Name);
  AddToOutput("Full Name: " + t.FullName);
  AddToOutput("Namespace: " + t.Namespace);

  Type tBase = t.BaseType;

  if (tBase != null)
  {
    AddToOutput("Base Type:" + tBase.Name);
  }

  Type tUnderlyingSystem = t.UnderlyingSystemType;

  if (tUnderlyingSystem != null)
  {
    AddToOutput("UnderlyingSystem Type:" + tUnderlyingSystem.Name);
  }

  AddToOutput("\nPUBLIC MEMBERS:");
  MemberInfo [] Members = t.GetMembers();

  foreach (MemberInfoNextMember in Members)
  {
    AddToOutput(NextMember.DeclaringType + " " +
    NextMember.MemberType + " " + NextMember.Name);
  }
}
```

实现 AnalyzeType()方法，仅需要调用 Type 对象的各种属性，就可以获得我们需要的类型名称的相关信息，再调用 GetMembers()方法，获得一个 MemberInfo 对象数组，该数组用于显示每个成员的信息。注意这里使用了一个辅助方法 AddToOutput()，该方法创建要在消息框中显示的文本：

```
static void AddToOutput(string Text)
{
  OutputText.Append("\n" + Text);
}
```

使用下面的命令编译 TypeView 程序集：

```
csc /reference:System.Windows.Forms.dll Program.cs
```

15.3.3　Assembly 类

Assembly 类在 System.Reflection 名称空间中定义，它允许访问给定程序集的元数据，它也包含可以加载和执行程序集(假定该程序集是可执行的)的方法。与 Type 类一样，Assembly 类包含非常多的方法和属性，这里不可能逐一论述。下面仅介绍完成 WhatsNewAttributes 示例所需要的方法和属性。

在使用 Assembly 实例做一些工作前，需要把相应的程序集加载到正在运行的进程中。为此，可以使用静态成员 Assembly.Load()或 Assembly.LoadFrom()。这两个方法的区别是 Load()方法的参数是程序集的名称，运行库会在各个位置上搜索该程序集，试图找到该程序集，这些位置包括本地目录和全局程序集缓存。而 LoadFrom()方法的参数是程序集的完整路径名，它不会在其他位置搜索该程序集：

```
Assembly assembly1 = Assembly.Load("SomeAssembly");
Assembly assembly2 = Assembly.LoadFrom
   (@"C:\My Projects\Software\SomeOtherAssembly");
```

这两个方法都有许多其他重载版本，它们提供了其他安全信息。加载了一个程序集后，就可以使用它的各种属性进行查询，例如，查找它的全名：

```
string name = assembly1.FullName;
```

1. 获取在程序集中定义的类型的详细信息

Assembly 类的一个功能是它可以获得在相应程序集中定义的所有类型的详细信息，只要调用 Assembly.GetTypes()方法，它就可以返回一个包含所有类型的详细信息的 System.Type 引用数组，然后就可以按照上一节的方式处理这些 Type 引用了：

```
Type[] types = theAssembly.GetTypes();

foreach(Type definedType in types)
{
   DoSomethingWith(definedType);
}
```

2. 获取自定义特性的详细信息

用于查找在程序集或类型中定义了什么自定义特性的方法取决于与该特性相关的对象类型。如果要确定程序集从整体上关联了什么自定义特性，就需要调用 Attribute 类的一个静态方法 GetCustomAttributes()，给它传递程序集的引用：

```
Attribute[] definedAttributes =
            Attribute.GetCustomAttributes(assembly1);
            // assembly1 is an Assembly object
```

这是相当重要的。以前你可能想知道，在定义自定义特性时，为什么必须费尽周折为它们编写类，以及为什么 Microsoft 没有更简单的语法。答案就在于此。自定义特性确实与对象一样，加载了程序集后，就可以读取这些特性对象，查看它们的属性，调用它们的方法。

GetCustomAttributes()方法用于获取程序集的特性，它有两个重载方法：如果在调用它时，除了程序集的引用外，没有指定其他参数，该方法就会返回为这个程序集定义的所有自定义特性。当然，也可以通过指定第二个参数来调用它，第二个参数是表示感兴趣的特性类的一个 Type 对象，在这种情况下，GetCustomAttributes()方法就返回一个数组，该数组包含指定类型的所有特性。

注意，所有特性都作为一般的 Attribute 引用来获取。如果要调用为自定义特性定义的任何方法或属性，就需要把这些引用显式转换为相关的自定义特性类。调用 Assembly.GetCustomAttributes()的另一个重载方法，可以获得与给定数据类型相关的自定义特性的详细信息，这次传递的是一个 Type 引用，它描述了要获取的任何相关特性的类型。另一方面，如果要获得与方法、构造函数和字段等相关的特性，就需要调用 GetCustomAttributes()方法，该方法是 MethodInfo、ConstructorInfo 和 FieldInfo 等类的一个成员。

如果只需要给定类型的一个特性，就可以调用 GetCustomAttribute()方法，它返回一个 Attribute 对象。在 WhatsNewAttributes 示例中使用 GetCustomAttribute()方法，是为了确定程序集中是否有 SupportsWhatsNew 特性。为此，调用 GetCustomAttributes()方法，传递对 WhatsNewAttributes 程序集的一个引用和 SupportWhatsNewAttribute 特性的类型。如果有这个特性，就返回一个 Attribute 实例。如果在程序集中没有定义任何实例，就返回 null。如果找到两个或多个实例，GetCustomAttribute()方法就抛出一个 System.Reflection. AmbiguousMatchException 异常。该调用如下所示：

```
Attribute supportsAttribute =
          Attribute.GetCustomAttributes(assembly1,
          typeof(SupportsWhatsNewAttribute));
```

15.3.4 完成 WhatsNewAttributes 示例

现在已经有足够的知识来完成 WhatsNewAttributes 示例了。为该示例中的最后一个程序集 LookUpWhatsNew 编写源代码，这部分应用程序是一个控制台应用程序，它需要引用其他两个程序集 WhatsNewAttributes 和 VectorClass。这是一个命令行应用程序，但仍可以像前面的 TypeView 示例那样在消息框中显示结果，因为结果是许多文本，所以不能显示在一个控制台窗口屏幕截图中。

这个文件的名称为 LookUpWhatsNew.cs，编译它的命令是：

```
csc /reference:WhatsNewAttributes.dll /reference:VectorClass.dll LookUpWhatsNew.cs
```

在这个文件的源代码中，首先指定要使用的名称空间 System.Text，因为需要再次使用一个 StringBuilder 对象：

```
using System;
using System.Reflection;
using System.Windows.Forms;
using System.Text;
using WhatsNewAttributes;

namespace LookUpWhatsNew
{
```

WhatsNewChecker 类包含主程序入口点和其他方法。我们定义的所有方法都在这个类中，它还有两个静态字段：outputText 和 backDateTo。outputText 字段包含在准备阶段创建的文本，这个文本要写到消息框中，backDateTo 字段存储了选择的日期——自从该日期以来进行的所有修改都要显示出来。一般情况下，需要显示一个对话框，让用户选择这个日期，但我们不想编写这种代码，以免

转移读者的注意力。因此，把 backDateTo 字段硬编码为日期 2010 年 2 月 1 日。在下载这段代码时，很容易修改这个日期：

```
internal class WhatsNewChecker
{
  private static readonly StringBuilder outputText = new StringBuilder(1000);
  private static DateTime backDateTo = new DateTime(2010, 2, 1);

  static void Main()
  {
    Assembly theAssembly = Assembly.Load("VectorClass");
    Attribute supportsAttribute =
      Attribute.GetCustomAttribute(
        theAssembly, typeof(SupportsWhatsNewAttribute));
    string name = theAssembly.FullName;

    AddToMessage("Assembly: " + name);

    if (supportsAttribute == null)
    {
      AddToMessage(
        "This assembly does not support WhatsNew attributes");
      return;
    }
    else
    {
      AddToMessage("Defined Types:");
    }

    Type[] types = theAssembly.GetTypes();

    foreach(Type definedType in types)
      DisplayTypeInfo(definedType);

    MessageBox.Show(outputText.ToString(),
      "What\'s New since " + backDateTo.ToLongDateString());
    Console.ReadLine();
  }
```

Main()方法首先加载 VectorClass 程序集，验证它是否真的用 SupportsWhatsNew 特性标记。我们知道，VectorClass 程序集应用了 SupportsWhatsNew 特性，虽然才编译了该程序集，但进行这种检查还是必要的，因为用户可能希望检查这个程序集。

验证了这个程序集后，使用 Assembly.GetTypes()方法获得一个数组，其中包括在该程序集中定义的所有类型，然后在这个数组中遍历它们。对每种类型调用一个方法——DisplayTypeInfo()，它给 outputText 字段添加相关的文本，包括 LastModifiedAttribute 类的任何实例的详细信息。最后，显示带有完整文本的消息框。DisplayTypeInfo()方法如下所示：

```
private static void DisplayTypeInfo(Type type)
{
  // make sure we only pick out classes
  if (!(type.IsClass))
  {
    return;
  }
```

```
    AddToMessage("\nclass " + type.Name);

    Attribute [] attribs = Attribute.GetCustomAttributes(type);

    if (attribs.Length == 0)
    {
      AddToMessage("No changes to this class\n");
    }
    else
    {
      foreach (Attribute attrib in attribs)
      {
        WriteAttributeInfo(attrib);
      }
    }

    MethodInfo [] methods = type.GetMethods();
    AddToMessage("CHANGES TO METHODS OF THIS CLASS:");

    foreach (MethodInfo nextMethod in methods)
    {
      object [] attribs2 =
        nextMethod.GetCustomAttributes(
          typeof(LastModifiedAttribute), false);

      if (attribs2 != null)
      {
        AddToMessage(
          nextMethod.ReturnType + " " + nextMethod.Name + "()");
        foreach (Attribute nextAttrib in attribs2)
        {
          WriteAttributeInfo(nextAttrib);
        }
      }
    }
  }
```

注意，在这个方法中，首先应检查所传递的 Type 引用是否表示一个类。因为，为了简化代码，指定 LastModified 特性只能应用于类或成员方法，如果该引用不是类(它可能是一个结构、委托或枚举)，那么进行任何处理都是浪费时间。

接着使用 Attribute.GetCustomAttributes()方法确定这个类是否有相关的 LastModifiedAttribute 实例。如果有，就使用辅助方法 WriteAttributeInfo()把它们的详细信息添加到输出文本中。

最后使用 Type.GetMethods()方法遍历这种数据类型的所有成员方法，然后对每个方法进行相同的处理(类似于对类执行的操作)：检查每个方法是否有相关的 LastModifiedAttribute 实例，如果有，就用 WriteAttributeInfo()方法显示它们。

下面的代码显示了 WriteAttributeInfo()方法，它负责确定为给定的 LastModifiedAttribute 实例显示什么文本，注意因为这个方法的参数是一个 Attribute 引用，所以需要先把该引用强制转换为 LastModifiedAttribute 引用。之后，就可以使用最初为这个特性定义的属性获取其参数。在把该特性添加到要显示的文本中之前，应检查特性的日期是否是最近的：

```
private static void WriteAttributeInfo(Attribute attrib)
{

  LastModifiedAttributelastModifiedAttrib =
```

```
            attrib as LastModifiedAttribute;

        if (lastModifiedAttrib == null)
        {
            return;
        }
        // check that date is in range
        DateTimemodifiedDate = lastModifiedAttrib.DateModified;

        if (modifiedDate<backDateTo)
        {
            return;
        }

        AddToMessage(" MODIFIED: " +
            modifiedDate.ToLongDateString() + ":");
        AddToMessage(" " + lastModifiedAttrib.Changes);

        if (lastModifiedAttrib.Issues != null)
        {
            AddToMessage(" Outstanding issues:" +
                lastModifiedAttrib.Issues);
        }
    }
```

最后，是辅助方法 AddToMessage():

```
        static void AddToMessage(string message)
        {
            outputText.Append("\n" + message);
        }
    }
}
```

运行这段代码，得到如图 15-2 所示的结果。

注意，在列出 VectorClass 程序集中定义的类型时，实际上选择了两个类：Vector 类和内嵌的 VectorEnumerator 类。还要注意，这段代码把 backDateTo 日期硬编码为 2 月 1 日，实际上选择的是日期为 2 月 14 日的特性(添加集合支持的时间)，而不是 2 月 10 日(添加 IFormattable 接口的时间)。

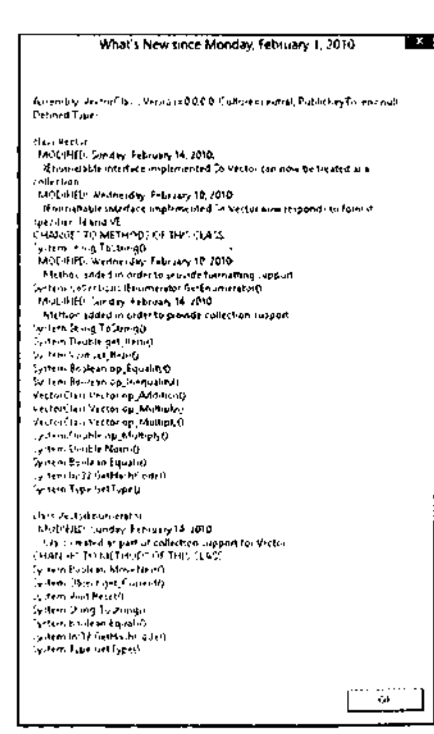


图 15-2

15.4 小结

本章没有介绍反射的全部内容，反射需要一整本书来讨论。我们只介绍了 Type 和 Assembly 类，它们是访问反射所提供的扩展功能的主要入口点。

另外，本章还探讨了反射的一个常用方面：自定义特性，它比其他方面更常用。介绍了如何定义和应用自己的自定义特性，以及如何在运行期间检索自定义特性的信息。

第16章

错误和异常

本章要点

- 异常类
- 使用 try…catch…finally 捕获异常
- 创建用户定义的异常
- 获取调用者的信息

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Simple Exceptions
- Solicit Cold Call
- Caller Information

16.1 简介

错误的出现并不总是编写应用程序的人的原因，有时应用程序会因为应用程序的最终用户引发或运行代码的环境而发生错误。无论如何，我们都应预测应用程序中出现的错误，并相应地进行编码。

.NET Framework 改进了处理错误的方式。C#处理错误的机制可以为每种错误提供自定义处理方式，并把识别错误的代码与处理错误的代码分离开来。

无论编码技术有多好，程序都必须能处理可能出现的任何错误。例如，在一些复杂的代码处理过程中，代码没有读取文件的许可，或者在发送网络请求时，网络可能会中断。在这种异常情况下，方法只返回相应的错误代码通常是不够的——可能方法调用嵌套了 15 级或者 20 级，此时，程序需要跳过所有的 15 或 20 级方法调用，才能完全退出任务，并采取相应的应对措施。C#语言提供了处理这种情形的最佳工具，称为异常处理机制。

本章介绍了在多种不同的场景中捕获和抛出异常的方式。讨论不同名称空间中定义的异常类型

及其层次结构，并学习如何创建自定义异常类型。还将学到捕获异常的不同方式，例如捕获特定类型的异常或者捕获基类的异常。本章还会介绍如何处理嵌套的 try 块，以及如何以这种方式捕获异常。对于无论如何都要调用的代码——即使发生了异常或者代码带错运行，可以使用本章介绍的 try/finally 块。

C# 5.0 中引入了一种新的处理错误的功能，允许获取调用者的信息，例如文件路径、行号和成员名。本章也会介绍这种新功能。

学习完本章后，你将很好地掌握 C#应用程序中的高级异常处理技术。

16.2　异常类

在 C#中，当出现某个特殊的异常错误条件时，就会创建(或抛出)一个异常对象。这个对象包含有助于跟踪问题的信息。我们可以创建自己的异常类(详见后面的内容)，但.NET 提供了许多预定义的异常类，多到这里不可能提供详尽的列表。在图 16-1 类的层次结构图中显示了其中的一些类，它们给出了大致的模式。本节将简要介绍在.NET 基类库中可用的一些异常。

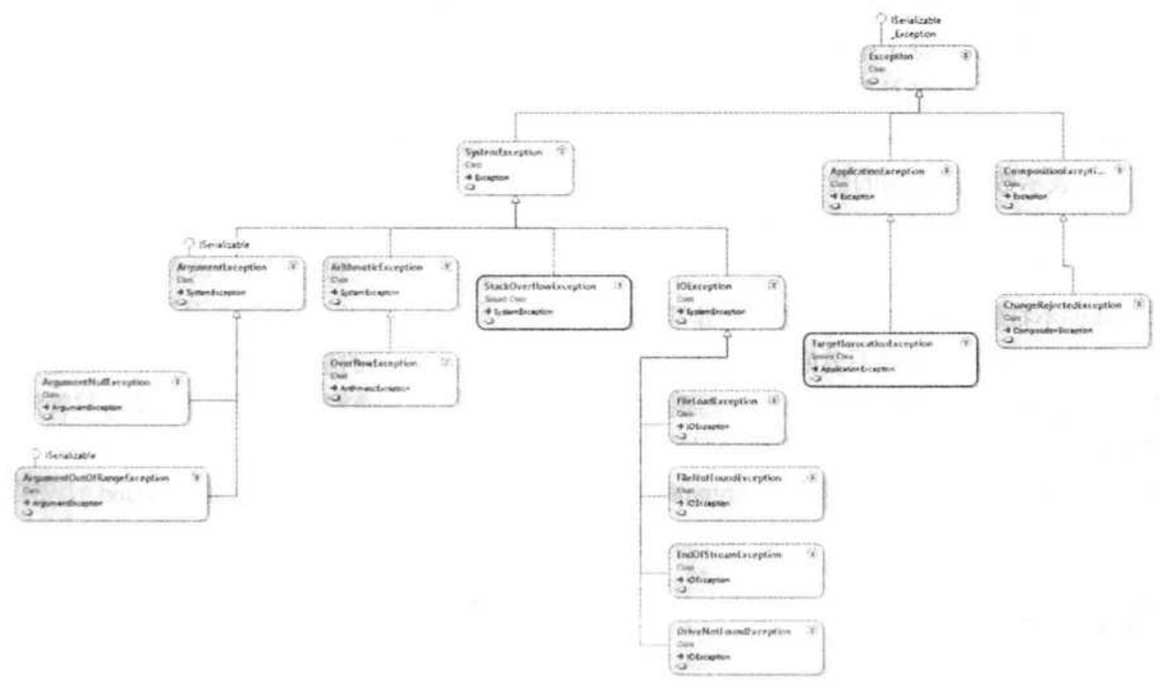


图 16-1

图 16-1 中的所有类都在 System 名称空间中，但 IOException 类、CompositionException 类和派生于这两个类的类除外。IOException 类及其派生类在 System.IO 名称空间中。System.IO 名称空间处理文件数据的读写。CompositionException 及其派生类在 System.ComponentModel.Composition 名称空间中。该名称空间处理部件和组件的动态加载。一般情况下，异常没有特定的名称空间，异常类应放在生成异常的类所在的名称空间中，因此与 IO 相关的异常就在 System.IO 名称空间中。在许多基类名称空间中都有异常类。

对于.NET 类，一般的异常类 System.Exception 派生自 System.Object，通常不在代码中抛出 System.Exception 泛型对象，因为它们无法确定错误情况的本质。

在该层次结构中有两个重要的类，它们派生自 System.Exception 类：

- SystemException——该类用于通常由.NET 运行库抛出的异常，或者有着非常一般的本质并且可以由几乎所有的应用程序抛出的异常。例如，如果.NET 运行库检测到栈已满，它就会抛出 StackOverflowException 异常。另一方面，如果检测到调用方法时参数不正确，就可以在自己的代码中选择抛出 ArgumentException 异常或其子类异常。SystemException 异常的子类包括表示致命错误和非致命错误的异常。
- ApplicationException——在.NET Framework 最初的设计中，是打算把这个类作为自定义应用程序异常类的基类的。不过，CLR 抛出的一些异常类也派生自这个类(例如 TargetInvocationException)，应用程序抛出的异常则派生自 SystemException(例如 ArgumentException)。因此从 ApplicationException 派生自定义异常类型没有提供任何好处，所以不再是一种好做法。取而代之的是，可以直接从 Exception 基类派生自定义异常类。.NET Framework 中的许多异常类直接派生自 Exception。

其他可能用到的异常类包括：

- StackOverflowException——如果分配给栈的内存区域已满，就会抛出这个异常。如果一个方法连续地递归调用它自己，就可能发生栈溢出。这一般是一个致命错误，因为它禁止应用程序执行除了中断以外的其他任务。在这种情况下，甚至也不可能执行 finally 块。通常用户自己不能处理像这样的错误，而应退出应用程序。
- EndOfStreamException——这个异常通常是因为读到文件末尾而抛出的。第 26 章解释流，流表示数据源之间的数据流。
- OverflowException——如果要在 checked 环境下把包含值–40 的 int 类型数据强制转换为 uint 数据，就会抛出这个异常。

我们不打算讨论图 16-1 中的其他异常类。

异常类的层次结构并不多见，因为其中的大多数类并没有给它们的基类添加任何功能。但是在处理异常时，添加继承类的一般原因是更准确地指定错误，所以不需要重写方法或添加新方法(尽管常常要添加额外的属性，以包含有关错误情况的额外信息)。例如，当传递了不正确的参数值时，可给方法调用使用 ArgumentException 基类，ArgumentNullException 异常类派生于 ArgumentException 异常类，它专门用于处理所传递的参数值是 Null 的情况。

16.3 捕获异常

.NET Framework 提供了大量的预定义基类异常对象，本节就介绍如何在代码中使用它们捕获错误情况。为了在 C#代码中处理可能的错误情况，一般要把程序的相关部分分成 3 种不同类型的代码块：

- try 块包含的代码组成了程序的正常操作部分，但这部分程序可能遇到某些严重的错误。
- catch 块包含的代码处理各种错误情况，这些错误是执行 try 块中的代码时遇到的。这个块还可以用于记录错误。
- finally 块包含的代码清理资源或执行通常要在 try 块或 catch 块末尾执行的其他操作。无论是否抛出异常，都会执行 finally 块，理解这一点非常重要。因为 finally 块包含了应总是执行的清理代码，如果在 finally 块中放置了 return 语句，编译器就会标记一个错误。例如，使用 finally 块时，可以关闭在 try 块中打开的连接。finally 块是完全可选的。如果不需要清

理代码(如删除对象或关闭已打开的对象)，就不需要包含此块。

下面的步骤说明了这些块是如何组合在一起捕获错误情况的：

(1) 执行的程序流进入 try 块。

(2) 如果在 try 块中没有错误发生，在块中就会正常执行操作。当程序流到达 try 块末尾后，如果存在一个 finally 块，程序流就会自动进入 finally 块(第(5)步)。但如果在 try 块中程序流检测到一个错误，程序流就会跳转到 catch 块(第(3)步)。

(3) 在 catch 块中处理错误。

(4) 在 catch 块执行完后，如果存在一个 finally 块，程序流就会自动进入 finally 块：

(5) 执行 finally 块(如果存在)。

用于完成这些任务的 C#语法如下所示：

```
try
{
  // code for normal execution
}
catch
{
  // error handling
}
finally
{
  // clean up
}
```

实际上，上面的代码还有几种变体：

- 可以省略 finally 块，因为它是可选的。
- 可以提供任意多个 catch 块，处理不同类型的错误。但不应包含过多的 catch 块，以防降低应用程序的性能。
- 可以省略 catch 块——此时，该语法不是标识异常，而是一种确保程序流在离开 try 块后执行 finally 块中的代码的方式。如果在 try 块中有几个出口点，这很有用。

这看起来很不错，实际上是有问题的。如果运行 try 块中的代码，则程序流如何在错误发生时切换到 catch 块？如果检测到一个错误，代码就执行一定的操作，称为"抛出一个异常"；换言之，它实例化一个异常对象类，并抛出这个异常：

```
throw new OverflowException();
```

这里实例化了 OverflowException 类的一个异常对象。只要应用程序在 try 块中遇到一条 throw 语句，就会立即查找与这个 try 块对应的 catch 块。如果有多个与 try 块对应的 catch 块，应用程序就会查找与 catch 块对应的异常类，确定正确的 catch 块。例如，当抛出一个 OverflowException 异常对象时，执行的程序流就会跳转到下面的 catch 块：

```
catch (OverflowException ex)
{
  // exception handling here
}
```

换言之，应用程序查找的 catch 块应表示同一个类(或基类)中匹配的异常类实例。

有了这些额外的信息，就可以扩展刚才介绍的 try 块。为了讨论方便，假定可能在 try 块中发生

两个严重错误：溢出和数组超出范围。假定代码包含两个布尔变量 Overflow 和 OutOfBounds，它们分别表示这两种错误情况是否存在。我们知道，存在表示溢出的预定义溢出异常类 Overflow-Exception；同样，存在预定义的 IndexOutOfRangeException 异常类，用于处理超出范围的数组。

现在，try 块如下所示：

```
try
{
  // code for normal execution

  if (Overflow == true)
  {
    throw new OverflowException();
  }

  // more processing

  if (OutOfBounds == true)
  {
    throw new IndexOutOfRangeException();
  }

  // otherwise continue normal execution
}
catch (OverflowException ex)
{
  // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
  // error handling for the index out of range error condition
}
finally
{
  // clean up
}
```

我们得到的 try 块看起来并不比 Visual Basic 6 中的 On Error GoTo 语句强多少，但可以更清晰地将不同的代码段分开。实际上，C#的错误处理有一个更强大、更灵活的机制。

这是因为 throw 语句可以嵌套在 try 块的几个方法调用中，甚至在程序流进入其他方法时，也会继续执行同一个 try 块。如果应用程序遇到一条 throw 语句，就会立即退出栈上所有的方法调用，查找 try 块的结尾和合适的 catch 块的开头，此时，中间方法调用中的所有局部变量都会超出作用域。try...catch 结构最适合于本节开头描述的场合：错误发生在一个方法调用中，而该方法调用可能嵌套了 15 或 20 级，这些处理操作会立即停止。

从上面的论述可以看出，try 块在控制执行的程序流上有重要的作用。但是，异常是用于处理异常情况的，这是其名称的由来。不应该用异常来控制退出 do...while 循环的时间。

16.3.1 实现多个 catch 块

要了解 try...catch...finally 块是如何工作的，最简单的方式是用两个示例来说明。第一个示例是 SimpleExceptions。它多次要求用户输入一个数字，然后显示这个数字。为了便于解释这个示例，假定该数字必须在 0～5 之间，否则程序就不能对该数字进行正确的处理。所以，如果用户输入超出该

范围的数字，程序就抛出一个异常。程序会继续要求用户输入更多数字，直到用户不再输入任何内容，按回车键为止。

这段代码没有说明何时使用异常处理，但是它显示了使用异常处理的好方法。顾名思义，异常用于处理异常情况。用户经常输入一些无聊的东西，所以这种情况不会真正发生。正常情况下，程序会处理不正确的用户输入，方法是进行即时检查，如果有问题，就要求用户重新输入。但是，在一个要求几分钟内读懂的小示例中生成异常是比较困难的，为了描述异常是如何工作的，后面将使用更真实的示例。

SimpleExceptions 的代码如下所示(代码文件 SimpleExceptions/Program.cs)：

```
using System;

namespaceWrox.ProCSharp.ErrorsAndExceptions
{
  public class Program
  {
    public static void Main()
    {
      while (true)
      {
        try
        {
          stringuserInput;

          Console.Write("Input a number between 0 and 5 " +
              "(or just hit return to exit)> ");
          userInput = Console.ReadLine();

          if (userInput == "")
          {
            break;
          }

          int index = Convert.ToInt32(userInput);

          if (index < 0 || index > 5)
          {
            throw new IndexOutOfRangeException("You typed in " + userInput);
          }

          Console.WriteLine("Your number was " + index);
        }
        catch (IndexOutOfRangeException ex)
        {
          Console.WriteLine("Exception: " +
              "Number should be between 0 and 5. {0}", ex.Message);
        }
        catch (Exception ex)
        {
          Console.WriteLine(
              "An exception was thrown. Message was: {0}", ex.Message);
        }
```

```
                finally
                {
                    Console.WriteLine("Thank you");
                }
            }
        }
    }
}
```

这段代码的核心是一个 while 循环，它连续使用 Console.ReadLine()方法以请求用户输入。ReadLine()方法返回一个字符串，所以程序首先要用 System.Convert.ToInt32()方法把它转换为 int 型。System.Convert 类包含执行数据转换的各种有用方法，并提供了 int.Parse()方法的一个替代方法。一般情况下，System.Convert 类包含执行各种类型转换的方法，C#编译器把 int 解析为 System.Int32 基类的实例。

值得注意的是，传递给 catch 块的参数只能用于该 catch 块。这就是为什么在上面的代码中，能在后续的 catch 块中使用相同的参数名 ex 的原因。

在上面的代码中，我们也检查一个空字符串，因为该空字符串是退出 while 循环的条件。注意这里用 break 语句退出 try 块和 while 循环——这是有效的。当然，当程序流退出 try 块时，会执行 finally 块中的 Console.WriteLine()语句。尽管这里仅显示一句问候，但一般在这里可以关闭文件句柄，调用各种对象的 Dispose()方法，以执行清理工作。一旦计算机退出了 finally 块，它就会继续执行下一条语句，如果没有 finally 块，该语句也会执行。在本例中，我们返回 while 循环的开头，再次进入 try 块(除非执行 while 循环中 break 语句的结果是进入 finally 块，此时就会退出 while 循环)。

下面看看异常情况：

```
if (index < 0 || index > 5)
{
    throw new IndexOutOfRangeException("You typed in " + userInput);
}
```

在抛出一个异常时，需要选择要抛出的异常类型。可以使用 System.Exception 异常类，但这个类是一个基类，最好不要把这个类的实例当作一个异常抛出，因为它没有包含关于错误的任何信息。而.NET Framework 包含了许多派生自 System.Exception 异常类的其他异常类，每个类都对应于一种特定类型的异常情况，也可以定义自己的异常类。在抛出一个匹配特定错误情况的类的实例时，应提供尽可能多的异常信息。在本例中，System.IndexOutOfRangeException 异常类是最佳选择。IndexOutOfRangeException 异常类有几个重载的构造函数，我们选择的一个重载，其参数是一个描述错误的字符串。另外，也可以选择派生自己的自定义异常对象，它描述该应用程序环境中的错误情况。

假定用户这次输入了一个不在 0～5 范围内的数字，if 语句就会检测到一个错误，并实例化和抛出一个 IndexOutOfRangeException 异常对象。应用程序会立即退出 try 块，并查找处理 IndexOutOf-RangeException 异常的 catch 块。它遇到的第一个 catch 块如下所示：

```
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(
        "Exception: Number should be between 0 and 5. {0}", ex.Message);
}
```

由于这个 catch 块带合适类的一个参数，因此它就会传递给异常实例，并执行。在本例中，是显示错误信息和 Exception.Message 属性(它对应于给 IndexOutRangeException 异常类的构造函数传递的字符串)。执行了这个 catch 块后，控制权就切换到 finally 块，就好像没有发生过任何异常。

注意，本例还提供了另一个 catch 块：

```
catch (Exception ex)
{
  Console.WriteLine("An exception was thrown. Message was: {0}",
      ex.Message);
}
```

如果没有在前面的 catch 块中捕获到这类异常，则这个 catch 块也能处理 IndexOutOfRangeException 异常。基类的一个引用也可以指向派生自它的类的所有实例，所有的异常都派生自 System.Exception 异常类。这个 catch 块没有执行，因为应用程序只执行它在可用的 catch 块列表中找到的第一个合适的 catch 块。还有第二个 catch 块的原因是，不仅 try 块包含这段代码，还有另外 3 个方法调用 Console.ReadLine()、Console.Write()和 Convert.ToInt32()也包含这段代码，它们是 System 名称空间中的方法。这 3 个方法都可能抛出异常。

如果输入的内容不是数字，如 a 或 hello，Convert.ToInt32()方法就会抛出 System.FormatException 类的一个异常，表示传递给 ToInt32()方法的字符串对应的格式不能转换为 int。此时，应用程序会跟踪这个方法调用，查找可以处理该异常的处理程序。第一个 catch 块带一个 IndexOutOfRangeException 异常，不能处理这种异常。应用程序接着查看第二个 catch 块，显然它可以处理这类异常，因为 FormatException 异常类派生于 Exception 异常类，所以把 FormatException 异常类的实例作为参数传递给它。

该示例的这种结构是非常典型的多 catch 块结构。最先编写的 catch 块用于处理非常特殊的错误情况，接着是比较一般的块，它们可以处理任何错误，我们没有为它们编写特定的错误处理程序。实际上，catch 块的顺序很重要，如果以相反的顺序编写这两个块，代码就不会编译，因为第二个 catch 块是不会执行的(Exception catch 块会捕获所有异常)。因此，最上面的 catch 块应用于最特殊的异常情况，最后是最一般的 catch 块。

前面分析了该示例的代码，现在可以运行它。下面的输出说明了不同的输入会得到不同的结果，并说明抛出了 IndexOutOfRangeException 异常和 FormatException 异常：

```
SimpleExceptions
Input a number between 0 and 5 (or just hit return to exit)> 4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 0
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Message was: Input string was not in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
Thank you
```

16.3.2 在其他代码中捕获异常

上面的示例说明了两个异常的处理。一个是 IndexOutOfRangeException 异常，它由我们自己的代码抛出，另一个是 FormatException 异常，它由一个基类抛出。如果检测到错误，或者某个方法因传递的参数有误而被错误调用，库中的代码就常常会抛出一个异常。但库中的代码很少捕获这样的异常。应由客户端代码来决定如何处理这些问题。

在调试时，异常经常从基类库中抛出，调试的过程在某种程度上是确定异常抛出的原因，并消除导致错误发生的缘由。主要目标是确保代码在发布后，异常只发生在非常少见的情况下，如果可能，应在代码中以适当的方式处理它。

16.3.3 System.Exception 属性

本示例只使用了异常对象的一个 Message 属性。在 System.Exception 异常类中还有许多其他属性，如表 16-1 所示。

表 16-1

属　性	说　明
Data	这个属性可以给异常添加键/值语句，以提供关于异常的额外信息
HelpLink	链接到一个帮助文件上，以提供关于该异常的更多信息
InnerException	如果此异常是在 catch 块中抛出的，它就会包含把代码发送到 catch 块中的异常对象
Message	描述错误情况的文本
Source	导致异常的应用程序名或对象名
StackTrace	栈上方法调用的详细信息，它有助于跟踪抛出异常的方法
TargetSite	描述抛出异常的方法的.NET 反射对象

在这些属性中，如果可以进行栈跟踪，StackTrace 和 TargetSite 属性由.NET 运行库自动提供。Source 属性总是由.NET 运行库填充为抛出异常的程序集的名称(但可以在代码中修改该属性，提供更具体的信息)，Data、Message、HelpLink 和 InnerException 属性必须在抛出异常的代码中填充，方法是在抛出异常前设置这些属性。例如，抛出异常的代码如下所示：

```
if (ErrorCondition == true)
{
  varmyException = new ClassMyException("Help!!!!");
  myException.Source = "My Application Name";
  myException.HelpLink = "MyHelpFile.txt";
  myException.Data["ErrorDate"] = DateTime.Now;
  myException.Data.Add("AdditionalInfo",
     "Contact Bill from the Blue Team");
  throwmyException;
}
```

其中 ClassMyException 是抛出的异常类的名称。注意所有异常类的名称通常以 Exception 结尾。Data 属性可以用两种方式设置。

16.3.4 没有处理异常时所发生的情况

有时抛出了一个异常后，代码中没有catch块能处理这类异常。前面的SimpleExceptions示例就说明

了这种情况。例如，假定忽略FormatException异常和通用的catch块，只有捕获IndexOutOfRangeException异常的块。此时，如果抛出一个FormatException异常，会发生什么情况呢？

答案是.NET 运行库会捕获它。本节后面将介绍如何嵌套 try 块——实际上在本示例中，就有一个在后台处理的嵌套的 try 块。.NET 运行库把整个程序放在另一个更大的 try 块中，对于每个.NET 程序它都会这么做。这个 try 块有一个 catch 处理程序，它可以捕获任何类型的异常。如果出现代码没有处理的异常，程序流就会退出程序，由.NET 运行库中的 catch 块捕获它。但是，事与愿违。代码的执行会立即终止，并给用户显示一个对话框，说明代码没有处理异常，并给出.NET 运行库能检索到的关于异常的详细信息。至少异常会被捕获，这就是第 2 章在 Vector 示例程序抛出一个异常时发生的情况。

一般情况下，如果编写一个可执行程序，就应捕获尽可能多的异常，并以合理的方式处理它们。如果编写一个库，最好不要捕获异常(除非某个特殊异常表示在代码中可以处理的情况)，但要假定调用代码可以处理它们。当然，用户可能想捕获 Microsoft 预定义的任何异常，以便抛出自己的异常对象，该异常对象给客户端代码提供更特定的信息。

16.3.5　嵌套的 try 块

异常的一个特性是 try 块可以嵌套，如下所示：

```
try
{
  // Point A
  try
  {
    // Point B
  }
  catch
  {
    // Point C
  }
  finally
  {
    // clean up
  }
  // Point D
}
catch
{
  // error handling
}
finally
{
  // clean up
}
```

在上面的代码中，每个 try 块都只有一个 catch 块，但可以把多个 catch 块连接在一起。下面详细讨论嵌套的 try 块如何工作。

如果抛出的异常在外层的 try 块中，但在内层 try 块的外部(标记为 A 点和 D 点的代码块)，这种情况就与前面介绍的情况没有任何区别：异常由外层的 catch 块捕获，并执行外层的 finally 块，或者执行 finally 块，由.NET 运行库处理异常。

如果异常是在内层 try 块(标记为 B 点的代码块)中抛出的，且有一个合适的内层 catch 块处理该异常，这又是我们熟悉的情况：在内层处理异常，执行内层的 finally 块，之后继续执行外层的 try 块(标记为 D 点的代码块)。

现在假定异常是在代码块的内层 try 块中抛出的，但内层的 catch 块中没有合适的处理程序。这时会像通常一样执行内层的 finally 块，但接着.NET 运行库必须退出内层的 try 块，才能搜索到合适的处理程序。下一个要搜索的区域显然是外层的 catch 块。如果系统在这里找到一个处理程序，就会执行该处理程序，再执行外层的 finally 块。如果没有找到合适的处理程序，就会继续搜索。在这里，执行的是外层的 finally 块，因为没有更多的 catch 块，所以控制权会转移到.NET 运行库。注意，任何时候不会执行外层 try 块中 D 点后面的代码。

如果异常是在 C 点抛出的，就更有趣了。如果程序执行到 C 点中，它就必须处理在 B 点抛出的异常。在 catch 块中抛出另一个异常很正常。此时，异常的处理就跟它是在外层 try 块中抛出的一样，程序流会立即退出内层的 catch 块，执行内层的 finally 块，系统在外层的 catch 块中搜索处理程序。同样，如果在内层的 finally 块中抛出一个异常，搜索会在外层的 catch 块开始，控制权会立即转移到最匹配的处理程序。

在 catch 块和 finally 块中抛出异常是完全合理的。此时既可以使用 throw 关键字再次抛出相同的异常，而不传递任何异常信息，也可以抛出一个新的异常对象。抛出新异常时，可以将新对象的构造函数作为内部异常来分配给原异常。接下来的“修改异常的类型”一节将详细介绍这方面的内容。

尽管本例只有两个 try 块，但无论嵌套了多少个 try 块，规则都是一样的。在每个阶段中，.NET 运行库顺序执行 try 块，查找合适的处理程序。在每个阶段中，当退出 catch 块后，就会执行对应 finally 块中的任何清理代码，但不执行 finally 块外部的代码，直到找到合适的 catch 处理程序，并执行为止。

try 块的嵌套也可能发生在方法之间。例如，如果方法 A 调用了 try 块中的方法 B，那么方法 B 也包含一个 try 块。

前面说明了嵌套的 try 块的工作方式，它在以下场景中是很有用的：

- 修改所抛出的异常的类型。
- 能够在代码的不同地方处理不同类型的异常。

1. 修改异常的类型

当最初抛出的异常不足以说明问题时，修改异常的类型就非常重要。通常的情况是抛出的异常(可能由.NET 运行库抛出)是一种相当低级的异常，说明发生溢出(OverflowException 异常)或传递给方法的参数不正确(派生于 ArgumentException 异常类的一个类)。但是，由于抛出异常的环境，我们知道这暴露了一些底层的问题(例如，因为刚才读取的文件包含了不正确的数据，才发生了溢出异常)。此时处理程序对于第一个异常所能做的最佳处理就是抛出另一个异常，以便更准确地说明这个问题，从而让另一个 catch 块更恰当地处理它。也可以通过一个由 Exception 异常类实现的 InnerException 属性来处理最初的异常。InnerException 属性只包含另一个抛出的相关异常的引用——最终的处理程序例程需要这些额外信息。

当然，还应指出，异常可能在 catch 块中抛出。例如，可以从某个配置文件中读取数据，这个

文件包含处理错误的详细指令——结果可能是这个文件不存在。

2. 在不同的地方处理不同的异常

嵌套 try 块的第二个原因是不同类型的异常可以在代码的不同地方处理。例如，在循环中，可能会发生各种异常。其中一些异常比较严重，需要退出整个循环，而另外一些则不太严重，只需要退出这次迭代，进入循环的下一次迭代即可。在循环的内部有一个 try 块就可以处理不太严重的异常，再在循环外面用一个外层的 try 块来处理比较严重的错误。在下面的异常示例中，将解释具体的操作情况。

16.4 用户定义的异常类

下面介绍有关异常的第二个示例，这个示例称为 SolicitColdCall，它包含两个嵌套的 try 块，说明了如何定义自定义异常类，再从 try 块中抛出另一个异常。

这个示例假定一家销售公司希望有更多的客户。该公司的销售部门打算给一些人打电话，希望他们成为自己的客户。用销售行业的行话来讲，就是“陌生电话”(cold-calling)。为此，应有一个文本文件存储这些陌生人的姓名，该文件应有良好的格式，其中第一行包含文件中的人数，后面的行包含这些人的姓名。换言之，正确的格式如下所示。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

这个示例的目的是在屏幕上显示这些人的姓名(由销售人员读取)，这就是为什么只把姓名放在文件中，但没有电话号码的原因。

程序要求用户输入文件的名称，然后读取文件，并显示其中的人名。这听起来是一个很简单的任务，但也会出现两个错误，需要退出整个过程：

- 用户可能输入不存在的文件名。这作为 FileNotFound 异常来捕获。
- 文件的格式可能不正确，这里可能有两个问题。首先，文件的第一行不是整数。第二，文件中可能没有第一行指定的那么多人名。这两种情况都需要在一个自定义异常中处理，我们已经专门为此编写了 ColdCallFileFormatException 异常。

还会有其他问题，虽然不至于退出整个过程，但需要删除某个人名，继续处理文件中的下一个人名(因此这需要在内层的 try 块中处理)。一些人是商业间谍，为销售公司的竞争对手工作，显然，我们不希望不小心打电话给他们，让这些人知道我们要做的工作。为简单起见，假设姓名以 B 开头的那些人是商业间谍。这些人应在第一次准备数据文件时从文件中删除，但为防止有商业间谍混入，需要检查文件中的每个姓名，如果检测到一个商业间谍，就应抛出一个 SalesSpyFoundException 异常，当然，这是另一个自定义异常对象。

最后，编写一个类 ColdCallFileReader 来实现这个示例，该类维护与 cold-call 文件的连接，并从中检索数据。我们将以非常安全的方式编写这个类，如果其方法调用不正确，就会抛出异常。例如，如果在文件打开前，调用了读取文件的方法，就会抛出一个异常。为此，我们编写了另一个异常类 UnexpectedException。

16.4.1 捕获用户定义的异常

首先是 SolicitColdCall 示例的 Main()方法，它捕获用户定义的异常。注意，下面要调用 System.IO 名称空间和 System 名称空间中的文件处理类(代码文件 SolicitColdCall/Program.cs)。

```
using System;
using System.IO;

namespace Wrox.ProCSharp.ErrorsAndExceptions
{
  class Program
  {
    static void Main()
    {
      Console.Write("Please type in the name of the file " +
          "containing the names of the people to be cold called > ");
      string fileName = Console.ReadLine();
      var peopleToRing = new ColdCallFileReader();

      try
      {
        peopleToRing.Open(fileName);
        for (inti = 0; i<peopleToRing.NPeopleToRing; i++)
        {
          peopleToRing.ProcessNextPerson();
        }
          Console.WriteLine("All callers processed correctly");
      }
        catch(FileNotFoundException)
        {
          Console.WriteLine("The file {0} does not exist", fileName);
        }
        catch(ColdCallFileFormatException ex)
        {
          Console.WriteLine("The file {0} appears to have been corrupted",
              fileName);
          Console.WriteLine("Details of problem are: {0}", ex.Message);
          if (ex.InnerException != null)
          {
            Console.WriteLine(
                "Inner exception was: {0}", ex.InnerException.Message);
          }
        }
        catch(Exception ex)
        {
          Console.WriteLine("Exception occurred:\n" + ex.Message);
        }
        finally
        {
          peopleToRing.Dispose();
        }
        Console.ReadLine();
      }
    }
```

这段代码基本上只是一个循环，用来处理文件中的人名。开始时，先让用户输入文件名，再实例化ColdCallFileReader类的一个对象，这个类稍后定义，正是这个类负责处理文件中数据的读取。

注意是在第一个try块的外部读取文件——这是因为这里实例化的变量需要在后面的catch 块和finally块中使用，如果在try块中声明它们，它们在try块的闭合花括号处就超出了作用域，这会导致异常。

在 try 块中打开文件(使用 ColdCallFileReader.Open()方法)，并循环处理其中的所有人名。ColdCallFileReader.ProcessNextPerson()方法会读取并显示文件中的下一个人名，而 ColdCallFileReader.NpeopleToRing 属性则说明文件中应有多少个人名(通过读取文件的第一行来获得)。有 3 个 catch 块，其中两个分别用于处理 FileNotFoundException 和 ColdCallFileFormatException 异常，第 3 个则用于处理任何其他.NET 异常。

在 FileNotFoundException 异常中，我们会为它显示一条消息，注意在这个 catch 块中，根本不会使用异常实例，原因是这个 catch 块用于说明应用程序的用户友好性。异常对象一般会包含技术信息，这些技术信息对开发人员很有用，但对于最终用户来说则没有什么用，所以本例将创建一条更简单的消息。

对于 ColdCallFileFormatException 异常的处理程序，则执行相反的操作，说明了如何获得更完整的技术信息，包括内层异常的细节(如果存在内层异常)。

最后，如果捕获到其他一般异常，就显示一条用户友好消息，而不是让这些异常由.NET 运行库处理。注意我们选择不处理没有派生自 System.Exception 异常类的异常，因为不直接调用非.NET 的代码。

finally 块清理资源。在本例中，这是指关闭已打开的任何文件。ColdCallFileReader.Dispose()方法完成了这个任务。

C#提供了一个 using 语句，编译器自己会在使用该语句的地方创建一个 try/finally 块，该块调用 finally 块中的 Dispose 方法。实现了一个 Dispose 方法的对象就可以使用 using 语句。第 14 章详细介绍了 using 语句。

16.4.2 抛出用户定义的异常

下面看看处理文件读取，以及(可能)抛出用户定义的异常类 ColdCallFileReader 的定义。因为这个类维护一个外部文件连接，所以需要确保它根据第 4 章有关释放对象的规则，正确地释放它。这个类派生自 IDisposable 类。

首先声明一些私有字段(代码文件 SolicitColdCall/ColdCallFileReader.cs)：

```
public class ColdCallFileReader: IDisposable
{
  privateFileStream fs;
  privateStreamReadersr;
  privateuintnPeopleToRing;
  privateboolisDisposed = false;
  privateboolisOpen = false;
```

FileStream 和 StreamReader 都在 System.IO 名称空间中，它们都是用于读取文件的基类。FileStream 基类主要用于连接文件，StreamReader 基类则专门用于读取文本文件，并实现 Readline()方法，该方法读取文件中的一行文本。第 24 章在深入讨论文件处理时将讨论 StreamReader 基类。

isDisposed 字段表示是否调用了 Dispose()方法，我们选择实现 ColdCallFileReader 异常，这样，一旦调用了 Dispose()方法，就不能重新打开文件连接，重新使用对象了。isOpen 字段也用于错误检查——在本例中，检查 StreamReader 基类是否连接到打开的文件上。

打开文件和读取第一行的过程——告诉我们文件中有多少个人名——由 Open()方法来处理：

```
public void Open(string fileName)
{
  if (isDisposed)
   throw new ObjectDisposedException("peopleToRing");

  fs = new FileStream(fileName, FileMode.Open);
  sr = new StreamReader(fs);

  try
  {
    stringfirstLine = sr.ReadLine();
    nPeopleToRing = uint.Parse(firstLine);
    isOpen = true;
  }
  catch (FormatException ex)
  {
    throw new ColdCallFileFormatException(
      "First line isn\'t an integer", ex);
  }
}
```

与 ColdCallFileReader 异常类的所有其他方法一样，该方法首先检查在删除对象后，客户端代码是否不正确地调用了它，如果是，就抛出一个预定义的 ObjectDisposedException 异常对象。Open()方法也会检查 isDisposed 字段，看看是否已调用 Dispose()方法。因为调用 Dispose()方法会告诉调用者现在已经处理完对象，所以，如果已经调用了 Dispose()方法，就说明有一个试图打开新文件连接的错误。

接着，这个方法包含前两个内层的 try 块，其目的是捕获因为文件的第一行没有包含一个整数而抛出的任何错误。如果出现这个问题，.NET 运行库就抛出一个 FormatException 异常，该异常捕获并转换为一个更有意义的异常，这个更有意义的异常表示 cold-call 文件的格式有问题。注意 System.FormatException 异常表示与基本数据类型相关的格式问题，而不是与文件有关，所以在本例中它不是传递回主调例程的一个特别有用的异常。新抛出的异常会被最外层的 try 块捕获。因为这里不需要清理资源，所以不需要 finally 块。

如果一切正常，就把 isOpen 字段设置为 true，表示现在有一个有效的文件连接，可以从中读取数据。

ProcessNextPerson()方法也包含一个内层 try 块：

```
public void ProcessNextPerson()
{
  if (isDisposed)
  {
    throw new ObjectDisposedException("peopleToRing");
  }
```

```
    if (!isOpen)
    {
      throw new UnexpectedException(
        "Attempted to access coldcall file that is not open");
    }

    try
    {
      string name;
      name = sr.ReadLine();
      if(name == null)
      {
        throw new ColdCallFileFormatException("Not enough names");
      }
      if (name[0] == 'B')
      {
        throw new SalesSpyFoundException(name);
      }
      Console.WriteLine(name);
    }
    catch(SalesSpyFoundException ex)
    {
      Console.WriteLine(ex.Message);
    }
    finally
    {
    }
  }
```

这里可能存在两个与文件相关的错误(假定实际上有一个打开的文件连接，ProcessNextPerson()方法会先进行检查)。第一，读取下一个人名时，可能发现这是一个商业间谍。如果发生这种情况，在这个方法中就使用第一个 catch 块捕获异常。因为这个异常已经在循环中被捕获，所以程序流会继续在程序的 Main()方法中执行，处理文件中的下一个人名。

如果读取下一个人名，发现已经到达文件的末尾，就会发生错误。StreamReader 对象的 ReadLine()方法的工作方式是：如果到达文件末尾，它就会返回一个 null，而不是抛出一个异常。所以，如果找到一个 null 字符串，就说明文件的格式不正确，因为文件的第一行中的数字要比文件中的实际人数多。如果发生这种错误，就抛出一个 ColdCallFileFormatException 异常，它由外层的异常处理程序捕获(使程序终止执行)。

同样，这里不需要 finally 块，因为没有要清理的资源，但这次要放置一个空的 finally 块，表示在这里可以完成用户希望完成的任务。

这个示例就要完成了。ColdCallFileReader 异常类还有另外两个成员：NPeopleToRing 属性返回文件中应有的人数，Dispose()方法可以关闭已打开的文件。注意 Dispose()方法仅返回它是否被调用——这是实现该方法的推荐方式。它还检查在关闭前是否有一个文件流要关闭。这个例子说明了防御编码技术：

```
public uint NPeopleToRing
{
  get
  {
```

```
      if (isDisposed)
      {
        throw new ObjectDisposedException("peopleToRing");
      }

      if (!isOpen)
      {
        throw new UnexpectedException(
            "Attempted to access cold-call file that is not open");
      }

      return nPeopleToRing;
    }
  }

  public void Dispose()
  {
    if (isDisposed)
    {
      return;
    }

    isDisposed = true;
    isOpen = false;

    if (fs != null)
    {
      fs.Close();
      fs = null;
    }
  }
```

16.4.3　定义用户定义的异常类

最后，需要定义 3 个异常类。定义自己的异常非常简单，因为几乎不需要添加任何额外的方法。只需要实现构造函数，确保基类的构造函数正确调用即可。下面是实现 SalesSpyFoundException 异常类的完整代码(代码文件 SolicitColdCall/SalesSpyFoundException.cs)：

```
public class SalesSpyFoundException: Exception
{
  public SalesSpyFoundException(string spyName)
    : base("Sales spy found, with name " + spyName)
  {
  }

  public SalesSpyFoundException(string spyName, Exception innerException)
    : base("Sales spy found with name " + spyName, innerException)
  {
  }
}
```

注意，这个类派生自 Exception 异常类，正是我们期望的自定义异常。实际上，如果要更正式地创建它，可以把它放在一个中间类中，例如 ColdCallFileException 异常类，让它派生于 Exception 异常类，再从这个类派生出两个异常类，并确保处理代码可以很好地控制哪个异常处理程序处理哪个异常即可。但为了使这个示例比较简单，就不这么做了。

在 SalesSpyFoundException 异常类中，处理的内容要多一些。假定传送给它的构造函数的信息仅是找到的间谍名，从而把这个字符串转换为含义更明确的错误信息。我们还提供了两个构造函数，其中一个构造函数的参数只是一条消息，另一个构造函数的参数是一个内层异常。在定义自己的异常类时，至少把这两个构造函数都包括进来(尽管以后将不能在示例中使用 SalesSpyFoundException 异常类的第 2 个构造函数)。

对于 ColdCallFileFormatException 异常类，规则是一样的，但不必对消息进行任何处理(代码文件 SolicitColdCall/ColdCallFileFormatException.cs)：

```
public class ColdCallFileFormatException: Exception
{
  public ColdCallFileFormatException(string message)
     : base(message)
  {
  }

  public ColdCallFileFormatException(string message, Exception innerException)
     : base(message, innerException)
  {
  }
}
```

最后是 UnexpectedException 异常类，它看起来与 ColdCallFileFormatException 异常类是一样的(代码文件 SolicitColdCall/UnexpectedException.cs)：

```
public class UnexpectedException: Exception
{
  public UnexpectedException(string message)
     : base(message)
  {
  }
  public UnexpectedException(string message, Exception innerException)
     : base(message, innerException)
  {
  }
}
```

下面准备测试该程序。首先，使用 people.txt 文件，其内容已经在前面列出了。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

它有 4 个名字(与文件中第一行给出的数字匹配)，包括一个间谍。接着，使用下面的 people2.txt 文件，它有一个明显的格式错误：

```
49
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

最后，尝试执行该例子，但指定一个不存在的文件名 people3.txt，对这 3 个文件名运行程序 3 次，得到的结果如下：

```
SolicitColdCall
```

```
Please type in the name of the file containing the names of the people to be cold
   called> people.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
All callers processed correctly


SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
  called> people2.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
The file people2.txt appears to have been corrupted.
Details of the problem are: Not enough names


SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
   called> people3.txt
The file people3.txt does not exist.
```

最后，这个应用程序说明了处理程序中可能存在的错误和异常的许多不同方式。

16.5 调用者信息

在处理错误时，获得错误发生位置的信息。C# 5.0 提供了一种新功能，可以用特性和可选参数获得这些信息。这些特性包括 CallerLineNumber、CallerFilePath 和 CallerMemberName，它们定义在 System.Runtime.CompilerServices 名称空间中，可以应用到参数上。对于可选参数，当没有提供调用信息时，编译器会在调用方法时为它们使用默认值。有了调用者信息特性，编译器不会填入默认值，而是填入行号、文件路径和成员名称。

下面代码段中的 Log 方法演示了这些特性的用法。这段代码将信息写入到控制台中(代码文件 CallerInformation/Program.cs)：

```
public void Log([CallerLineNumber] int line = -1,
  [CallerFilePath] string path = null,
  [CallerMemberName] string name = null)
{
  Console.WriteLine((line < 0) ? "No line" : "Line " + line);
  Console.WriteLine((path == null) ? "No file path" : path);
  Console.WriteLine((name == null) ? "No member name" : name);
  Console.WriteLine();
}
```

下面在几种不同的场景中调用该方法。在下面的 Main 方法中，分别使用 Program 类的一个实例来调用 Log 方法，在属性的 set 访问器中调用 Log 方法，以及在一个 lambda 表达式中调用 Log 方法。这里没有为该方法提供参数值，所以编译器会为其填入值：

```
static void Main()
{
  var p = new Program();
```

```
    p.Log();
    p.SomeProperty = 33;

    Action a1 = () =>p.Log();
    a1();
  }

  private int someProperty;
  public int SomeProperty
  {
    get { return someProperty; }
    set
    {
      this.Log();
      someProperty = value;
    }
  }
```

运行此程序的结果如下所示。在调用 Log 方法的地方，可以看到行号、文件名和调用者的成员名。对于 Main 方法中调用的 Log 方法，成员名为 Main。对于属性 SomeProperty 的 set 访问器中调用的 Log 方法，成员名为 SomeProperty。lambda 表达式中的 Log 方法没有显示生成的方法名，而是显示了调用该 lambda 表达式的方法的名称(Main)，这当然更加有用。

```
Line 11
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main

Line 24
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
SomeProperty

Line 14
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main
```

在构造函数中使用 Log 方法时，调用者成员名显示为 ctor。在析构函数中，调用者成员名为 Finalize，因为它是生成的方法的名称。

CallerMemberName 的一个很好的用途是用在 INotifyPropertyChanged 接口的实现中。该接口要求在方法的实现中传递属性的名称。在本书中的几个章节中都可以看到这个接口的实现，例如第 36 章。

16.6 小结

本章介绍了 C#通过异常处理错误情况的多种机制，我们不仅可以输出代码中的一般错误代码，还可以用指定的方式处理最特殊的错误情况。有时一些错误情况是通过.NET Framework 本身提供的，有时则需要编写自己的错误情况，如本章的例子所示。在这两种情况下，都可以采用许多方式来保护应用程序的工作流，使之不出现不必要和危险的错误。

第 17 章将利用前面学习的许多内容，在.NET 开发人员的 IDE——Visual Studio 2013 中实践这些内容。

第 II 部分

Visual Studio

- 第 17 章　Visual Studio 2013
- 第 18 章　部　　署

第17章

Visual Studio 2013

本章要点

- 使用 Visual Studio 2013
- 架构工具
- 分析应用程序
- 测试
- 用 Visual Studio 进行重构
- Visual Studio 2013 的多目标框架功能
- 用不同技术进行工作(WPF、WCF 和 WF 等)

本章源代码下载地址(wrox.com)：
本章没有可供下载的代码。

17.1 使用 Visual Studio 2013

到目前为止，你应该已经对 C#语言比较熟悉，并准备开始学习本书的应用部分。在这些章节中会介绍如何使用 C#编写各种应用程序。但在学习之前，需要理解如何使用 Visual Studio 和.NET 环境提供的一些功能使程序达到最佳效果。

本章讲解在实际工作中，如何在.NET 环境中编程。介绍主要的开发环境 Visual Studio，该环境用于编写、编译、调试和优化 C#程序，并且为编写优秀的应用程序提供指导。Visual Studio 是主要的 IDE，用于多种目的，包括编写 ASP.NET 应用程序、Windows Forms 应用程序和 Windows Presentation Foundation(WPF)应用程序，也可用于编写访问 WCF 服务或者 Web API 的 Windows Store 应用等。

本章还探讨如何构建目标框架为.NET Framework 4.5 或 4.5.1 的应用程序。使用 Visual Studio 2013，可以直接使用最新的应用程序类型进行工作，例如 WPF、Windows Communication Foundation(WCF)和 Windows Workflow Foundation(WF)。

Visual Studio 2013 是一个全面集成的开发环境。编写、调试、编译代码以生成一个程序集的整个过程被设计得尽可能容易。这意味着 Visual Studio 是一个非常全面的多文档界面应用程序，在该环境中可以完成所有代码开发的相关事情。它具有以下特性：

- **文本编辑器**　使用这个编辑器，可以编写 C#(还有 Visual Basic 2013、C++、F#、JavaScript、XAML 和 SQL)代码。这个文本编辑器是非常先进的。例如，当用户输入时，它会用缩进代码行自动布局代码，匹配代码块的开始和结束括号，以及使用颜色编码关键字。它还会在用户输入时检查语法，并用下划线标识导致编译错误的代码，这也称为设计时的调试。另外，它具有 IntelliSense 功能，当开始输入时它会自动显示类、字段或方法的名称。开始输入方法参数时，它也会显示可用重载的参数列表。图 17-1 用.NET 的一个基类 ListBox 展示了 IntelliSense 功能。

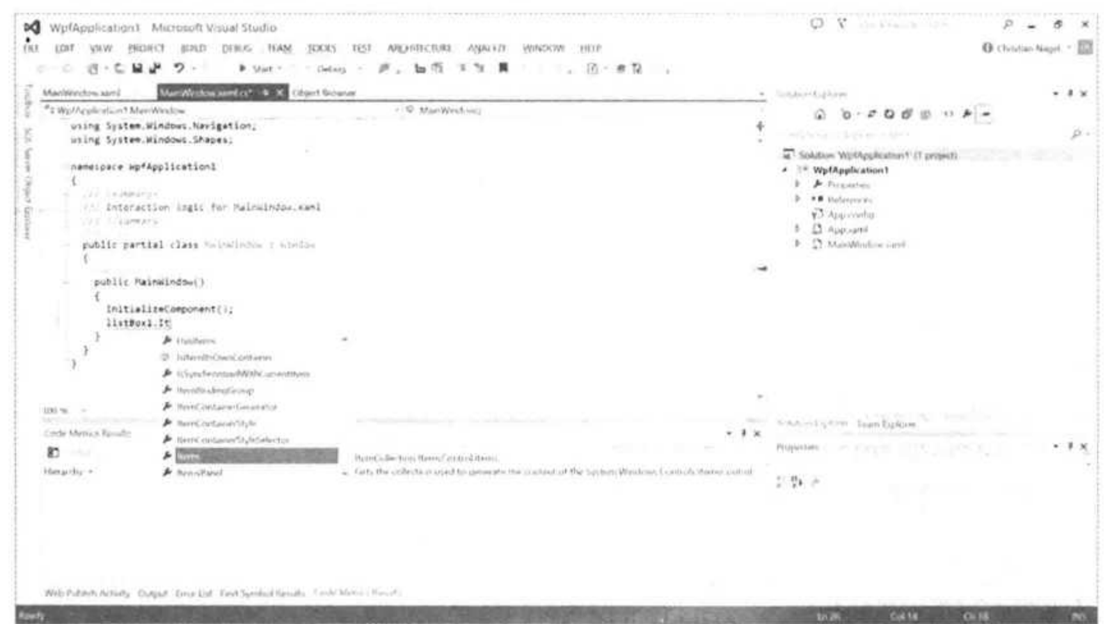


图 17-1

如果需要 IntelliSense 的列表框，或者因为其他原因该列表框不见了，可以按下 Ctrl+Space 组合键找回该列表框。如果希望看到 IntelliSense 框下面的代码，可以按住 Ctrl 按钮。

- **设计视图编辑器**　这个编辑器允许在项目中放置用户界面控件和数据绑定控件；Visual Studio 会在项目中自动将必需的 C#代码添加到源文件中，来实例化这些控件(这是可能的，因为所有.NET 控件都是具体基类的实例)。
- **支持窗口**　这些窗口允许查看和修改项目的各个方面，例如源代码中的类、Windows Forms 和 Web Forms 类的可用属性(以及它们的启动值)。也可以使用这些窗口来指定编译选项，例如代码需要引用的程序集。
- **集成调试器**　从编程的本质上讲，第一次试运行时，代码可能会无法正常运行。可能第二次或者第三次都无法正常运行。Visual Studio 无缝地链接到一个调试器中，允许设置断点，监视集成环境中的变量。

- **集成 MSDN 帮助** Visual Studio 允许在 IDE 中访问 MSDN 文档。例如，如果使用文本编辑器时不太确定一个关键字的含义，只需要选择该关键字并按 F1 键，Visual Studio 将会访问 MDSN 并向展示相关主题。同样，如果不确定某个编译错误是什么意思，可以选择错误消息并按 F1 键，调出 MSDN 文档，查看该错误的演示。
- **访问其他程序** Visual Studio 也可以访问一些其他实用程序，在不退出集成开发环境的情况下，就可以检查和修改计算机或网络的相关方面。可以用这些实用工具来检查运行的服务和数据库连接，直接查看 SQL Server 表，甚至用一个 Internet Explorer 窗口来浏览 Web。

Visual Studio 2010 基于 WPF 重新设计了外壳，而不是基于原生的 Windows 控件。Visual Studio 2012 的界面在此基础上又有了一些变化，尤其是增强了现代用户界面风格的展现方式。现代用户界面风格的核心是内容，而不是镶边。当然，像 Visual Studio 这样的工具，不太可能移除所有镶边；但考虑到用代码编辑器工作的重要性，Visual Studio 2012 提供了更多的空间。菜单和工具栏的尺寸被减小；默认情况下，只有一个工具栏处于打开状态。消除菜单和工具栏的边框也为编辑器提供了更多的空间。此外，Visual Studio 2010 经常打开一些其他工具的窗口，而现在很多功能都被集成到新的解决方案资源管理器中。

Microsoft 现代风格的外观改变了色彩的运用。如果用以前版本的 Visual Studio 工作，可能偶尔会发现自己不能编辑代码，过了一会儿，才发现原来运行了调试器。现在，可以用状态栏的颜色清楚地识别项目的状态。缺少色彩是社区提出的一个主要缺陷，所以 Visual Studio 22013 提供了色彩更丰富的主题。

Visual Studio 2012 和 2013 的一个重要目标是具有更快的响应速度。在以前的版本中，如果打开一个包含许多项目的解决方案，工作之前可能需要等待很长一段时间。现在，所有的项目都是异步加载的；已打开进行编辑的文件优先加载，其他已打开的文件会在后台继续加载。用这种方式，在完成加载之前，就可以做一些工作。新的异步功能在很多地方都已用到。例如，当 IntelliSense 线程开始并加载信息时，就可以在编辑器中输入知道的方法了。Add Reference 对话框也是异步搜索程序集的。因为很多操作都是在后台进行的，所以 Visual Studio 2012 比以前的版本反应更加灵敏。

对于 XAML 代码编辑，Visual Studio 2010 和 Expression Blend 4 使用了不同的编辑器引擎。现在，微软公司的团队已经合并，Visual Studio 和 Blend for Visual Studio 的编辑器是一样的。如果想使用这两个工具工作，这是好消息，因为它们现在的工作方式很相似。模板编辑也被集成到 Visual Studio 中。Visual Studio 2013 还提供了更简单的导航样式和 IntelliSense 数据绑定，增强了这个集成。

Visual Studio 的另一项改进是搜索。有许多地方可以使用搜索，在以前版本的 Visual Studio 中，不太常用的菜单项不好找。现在可以使用位于窗口右上方的 Quick Launch 来搜索菜单栏、工具栏和选项(如图 17-2 所示)。搜索功能还可以从其他地方找到，如工具栏、解决方案资源管理器、代码编辑器(可以按 Ctrl+F 组合键来调用)、引用管理器上的程序集等。

17.1.1 项目文件的改进

当使用 Visual Studio 2010 打开一个在 Visual Studio 2008 中创建的项目时，项目文件被转换，再也不能用 Visual Studio 2008 打开该项目了。这种行为在 Visual Studio 2013 中得到了改进。如果使用 Visual Studio 2013 打开一个由 Visual Studio 2010 创建的项目，仍可以使用 Visual Studio 2010 打开该项目。Visual Studio 2013 使用相同的文件格式。这样就允许一个团队的成员用不同版本的 Visual Studio 对同一个项目进行工作。但是，如果将一个项目改为使用.NET Framework 4.5.1，该项目就不

能再使用 Visual Studio 2010 打开了。因为 Visual Studio 2010 仅支持.NET 2.0 到.NET 4.0 的程序。而使用 Visual Studio 2012 可以打开.NET 4.5.1 项目。如图 17-2 所示。

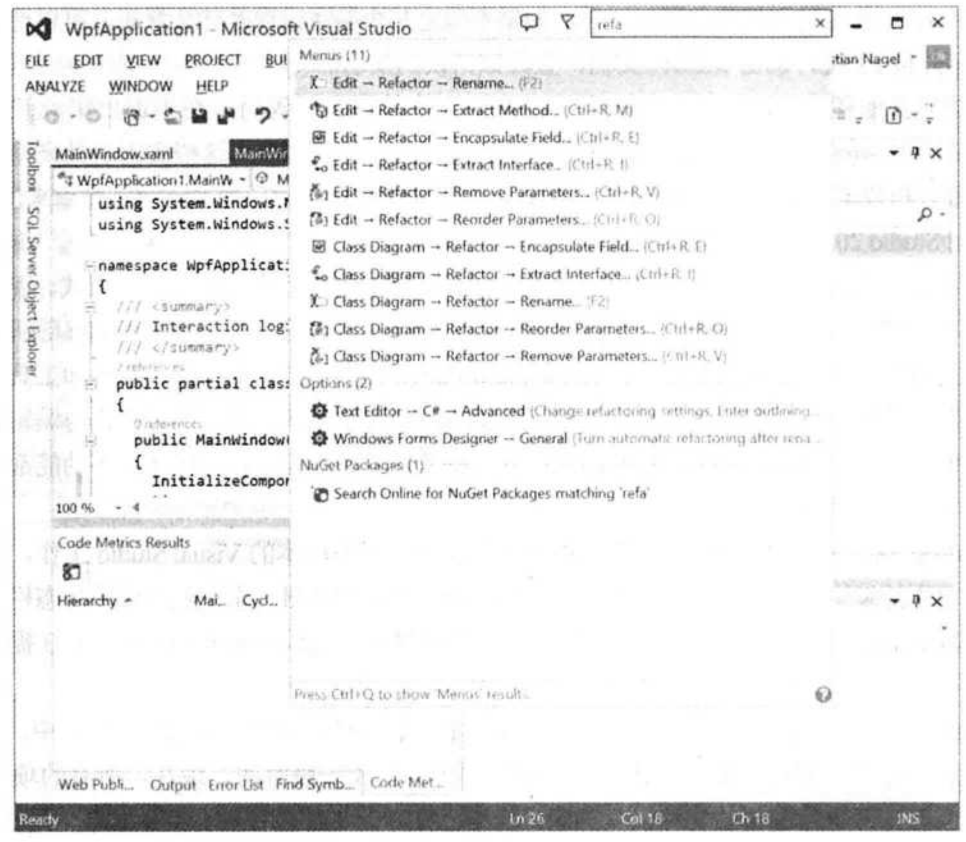


图 17-2

如果在 Windows 8 或 Windows 8.1 系统上安装 Visual Studio 2012，可以创建一个全新的应用序类别：Windows Store 应用。可以使用 C#和 XAML 创建这些应用程序，并使用新的 Windows Runtime 和.NET Framework 的一个子集。这些应用程序可以运行在 Windows 8 和 Windows 的以后版本上。

与纯.NET 应用程序相比，对于 Windows Store 应用，Visual Studio 或不同.NET 版本提供的兼容性不大相同。使用 Visual Studio 2013 可以为 Windows 8.1 创建新的 Windows Store 应用，但不能为 Windows 8 创建 Windows Store 应用。要创建 Windows 8 应用程序，需要使用 Visual Studio 2012。但是，可以使用 Visual Studio 2013 编辑已有的 Windows 8 应用程序。

17.1.2　Visual Studio 的版本

Visual Studio 2013 提供了多个版本。最便宜的是 Visual Studio 2013 Express 版，这个版本是免费的！可供购买的是 Professional、Premium 和 Ultimate 版。只有 Ultimate 版包含所有功能。Visual Studio 2013 Professional 版不包含代码度量、大量的测试工具、检查重复代码、架构和建模工具。Ultimate 版独享的功能有 IntelliTrace(智能跟踪)、负载测试和一些架构工具。微软的 Fakes 框架(隔离单元测试)只能用于 Visual Studio Premium 和 Ultimate 版。本章介绍 Visual Studio 2013 包含的一些功能，这些功能仅适用于特定版本。有关 Visual Studio 2013 各个版本中功能的详细信息，请参考 http://www.microsoft.com/visualstudio/en-us/products/compare。

17.1.3 Visual Studio 设置

当第一次运行 Visual Studio 时，需要选择一个符合环境的设置集，例如 General Development、Visual Basic、Visual C#、Visual C++或 Web Development。这些不同的设置反映了过去用于这些语言的不同工具。在微软平台上编写应用程序时，可以使用不同的工具来创建 Visual Basic、C++和 Web 应用程序。同样，Visual Basic、Visual C++和 Visual InterDev 具有完全不同的编程环境、设置和工具选项。

选择主设置类别后就确定了键盘快捷键、菜单和工具窗口的位置。通过命令 Tools | Customize... (工具栏和命令)和 Tools | Options...(在此可以找到所有工具的设置)，可以改变主设置类别。也可以重置设置集，通过命令 Tools | Import and Export Settings...可以调用一个向导，允许选择一个新的默认设置集(如图 17-3 所示)。

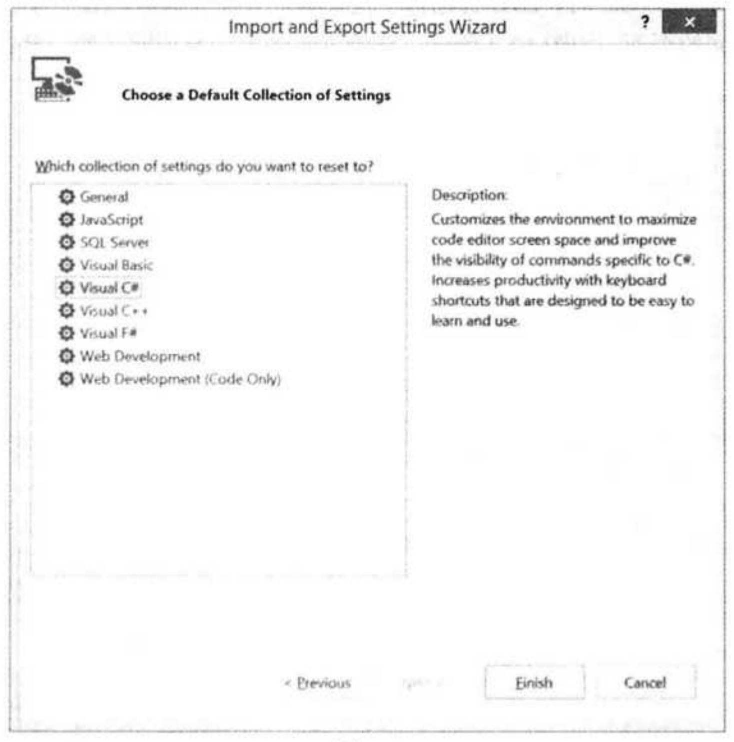


图 17-3

接下来的小节贯穿一个项目的创建、编码和调试过程，展示 Visual Studio 在各个阶段能够帮助完成什么工作。

17.2 创建项目

安装 Visual Studio 2013 之后，会希望开始自己的第一个项目。使用 Visual Studio，很少会启动一个空白文件，然后添加 C#代码。但在本书前面的章节中，一直是按这种方式做的(当然，如果真的想从头开始编写代码，或者打算创建一个包含多个项目的解决方案，则可以选择一个空白解决方案的项目模板)。

在此，告诉 Visual Studio 想创建什么类型的项目，它会生成文件和 C#代码，为该类型的项目提

供了一个框架。之后在这个基础上继续添加代码即可。例如，如果想创建一个 Windows 客户端应用程序(一个 WPF 应用程序)，Visual Studio 将生成一个 XAML 文件和一个包含 C#源代码的文件，它创建了一个基本的窗体。这个窗体能够与 Windows 通信和接收事件。它能够最大化、最小化或调整大小；用户需要做的仅是添加控件和想要的功能。如果的应用程序是一个命令行应用程序(一个控制台应用程序)，Visual Studio 将提供一个基本的名称空间、一个类和一个 Main 方法，用户可以从这里开始。

最后一点也很重要：在创建项目时，Visual Studio 会根据项目是编译为命令行应用程序、库还是 WPF 应用程序，为 C#编译器设置编译选项。它还会告诉编译器，应用程序需要引用哪些基类库。例如，WPF GUI 应用程序需要引用许多与 WPF 相关的库，控制台应用程序则不需要引用这些库。当然，在编辑代码的过程中，可以根据需要修改这些设置。

第一次启动 Visual Studio 时，IDE 会包含一些菜单、一个工具栏以及一个包含入门信息、操作方法视频和最新新闻的页面，如图 17-4 所示。起始页包含指向有用网站的一些链接，可以打开现有项目或者新建项目。

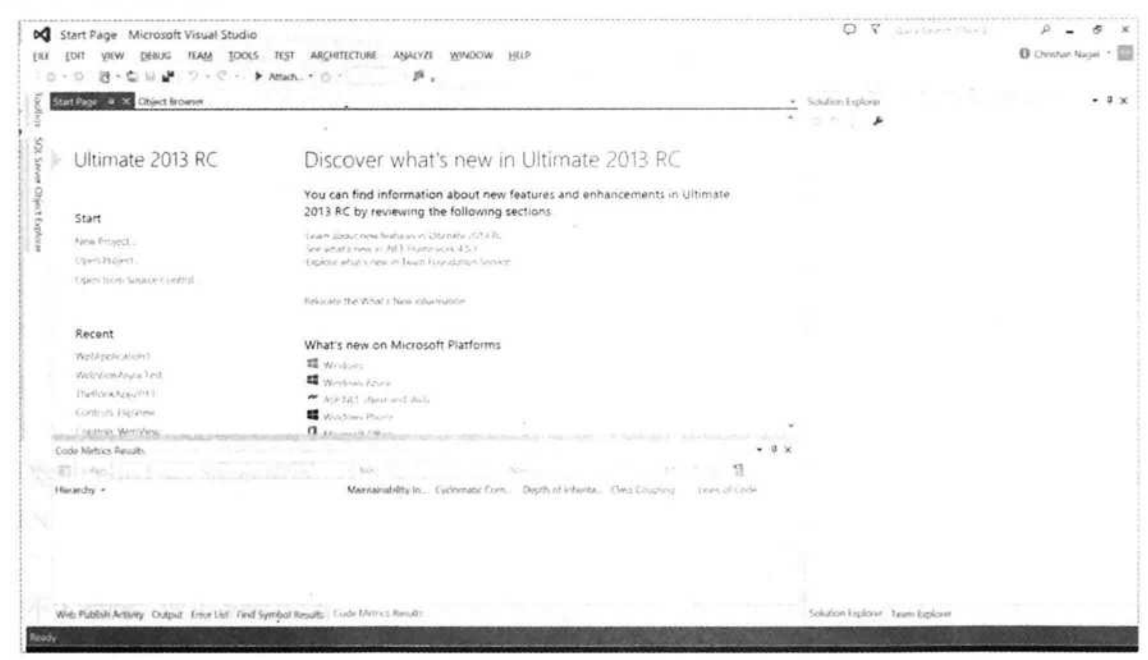


图 17-4

图 17-4 是使用 Visual Studio 2013 后显示的起始页，其中包含最近编辑过的项目列表。单击其中的某个项目可以打开该项目。

17.2.1　面向多个版本的.NET Framework

Visual Studio 2013 允许设置想用于工作的.NET Framework 版本。当打开 New Project 对话框时，如图 17-5 所示，对话框顶部的一个下拉列表显示了可用的选项。

在这种情况下，该下拉列表允许设置的.NET Framework 版本有 2.0、3.0、3.5、4.0、4.5 和 4.5.1。也可以通过单击 MoreFrameworks 链接来安装.NET Framework 的其他版本。该链接打开一个网站，从这个网站中可以下载.NET Framework 的其他版本，例如 4.01、4.02 和 4.03。

当使用 Upgrade 对话框将一个 Visual Studio 2012 的解决方案升级到 Visual Studio 2013 时，重点

是要理解只升级了解决方案以使用 Visual Studio 2013；并没有将项目升级为.NET Framework 4.5.1。你的项目仍然会使用以前的框架版本，但现在能够使用新的 Visual Studio 2013 在项目中工作。

如果想改变解决方案正在使用的框架版本，右击项目并选择该解决方案的属性。如果正在处理一个 ASP.NET 项目，将会看到如图 17-6 所示的对话框。

在此对话框中，Application 选项卡允许改变应用程序正在使用的框架版本。

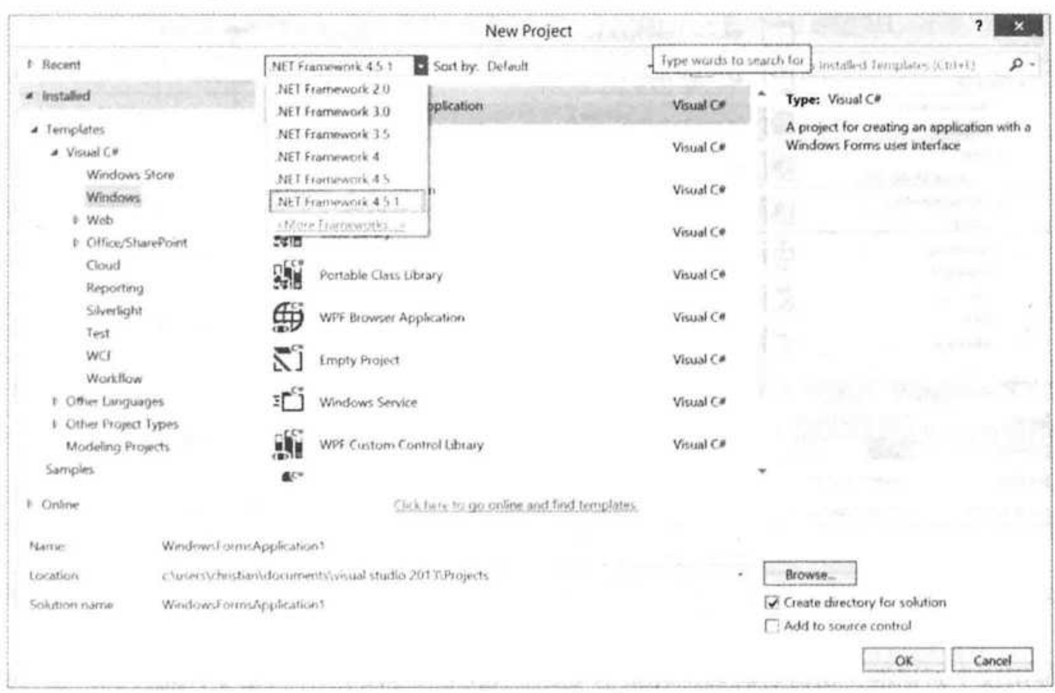


图 17-5

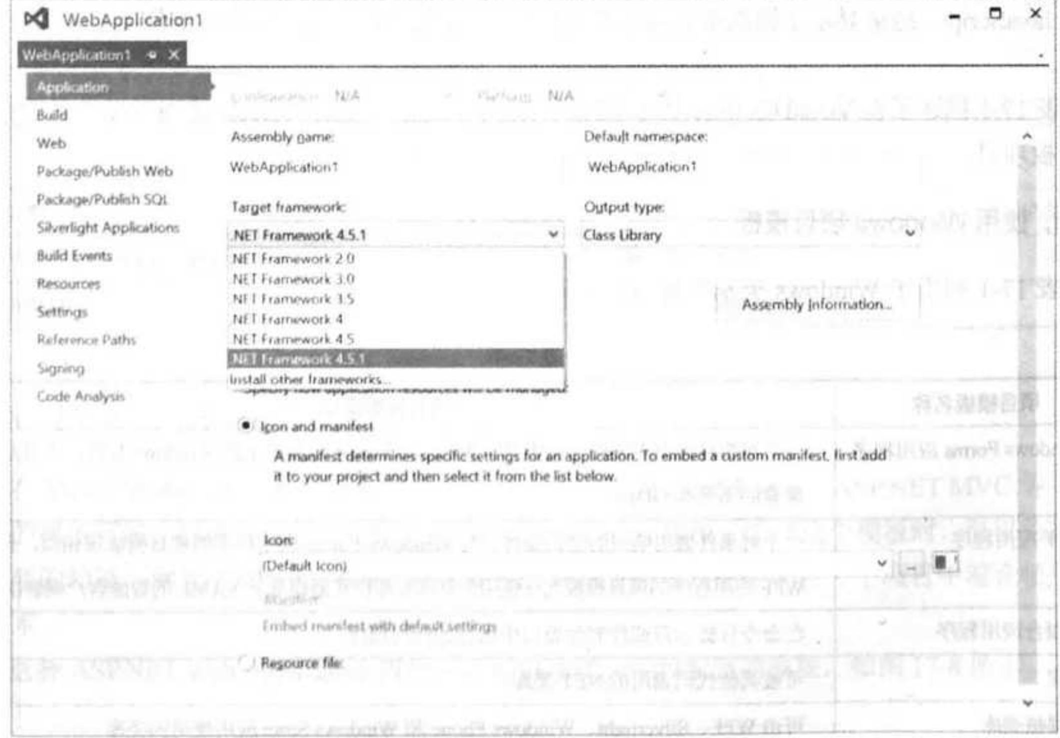


图 17-6

17.2.2 选择项目类型

要新建一个项目，从 Visual Studio 菜单中选择 File | New Project。New Project 对话框如图 17-7 所示，通过该对话框可以大致了解能够创建的不同项目。

图 17-7

使用这个对话框，可以有效地选择希望 Visual Studio 生成的初始框架文件和代码、想要的编译选项类型以及想要用于编译代码的编译器：Visual C#、LightSwitch、Visual Basic、Visual C++、Visual F#或 JavaScript。这里显示了微软承诺可以在.NET 上工作的各种语言。这个示例使用 C#控制台应用程序。

表 17-1 描述了在 Visual C#项目中可用的所有选项。注意，Other Projects 选项下提供了更专用的 C#模板项目。

1. 使用 Windows 项目模板

表 17-1 列出了 Windows 类别中所有的项目模板。

表 17-1

项目模板名称	项目模板描述
Windows Forms 应用程序	一个对事件做出响应的空白窗体。Windows Forms 包装原生的 Windows 控件，并使用基于像素的图形和 GDI+
WPF 应用程序	一个对事件做出响应的空白窗体。与 Windows Forms 应用程序的项目模板很相似，但是该 WPF 应用程序的项目模板允许使用矢量图形和样式创建基于 XAML 的智能客户端解决方案
控制台应用程序	在命令行提示符或控制台窗口中运行的应用程序
类库	可被其他代码调用的.NET 类库
可移植类库	可由 WPF、Silverlight、Windows Phone 和 Windows Store 应用使用的类库

(续表)

项目模板名称	项目模板描述
WPF 浏览器应用程序	与 WPF Windows 应用程序很相似，这种变体允许针对浏览器创建一个基于 XAML 的应用程序。如今，应该考虑使用不同技术实现这一点，例如具备 ClickOnce 的 WPF 应用程序、Silverlight 项目或者 HTML5
空项目	只包含一个应用程序配置文件和一个控制台应用程序设置的空项目
Windows Service	自动随 Windows 一起启动并以特权本地系统账户身份执行操作的 Windows Service 项目
WPF 自定义控件库	在 WPF 应用程序中使用的自定义控件
WPF 用户控件库	使用 WPF 构建的用户控件库
Windows Forms 控件库	创建在 Windows Forms 应用程序中使用的控件的项目

2. 使用 Windows Store 项目模板

表 17-2 列出了 Windows Store 应用的项目模板。如果 Visual Studio 安装在 Windows 8.1 上，就可以使用这些项目模板。这些项目模板用于创建运行在 Windows 8.1 上的新现代 UI 中的应用程序。

表 17-2

项目模板名称	项目模板描述
空白应用程序(XAML)	一个使用 XAML 的空白 Windows Store 应用，没有样式和其他基类。样式和基类很容易地在以后添加进来
网格应用程序(XAML)	一个 Windows Store 应用，用三个页面显示组和项的详细信息
Hub 应用程序(XAML)	一个 Windows Store 应用，通过三个页面使用新的 Hub 控件
拆分应用程序(XAML)	一个 Windows Store 应用，用两个页面显示组和项的详细信息
类库(Windows Store 应用)	一个.NET 类库，其他用.NET 编写的 Windows Store 应用
Windows 运行库组件	一个 Windows 运行库类库，其他用不同编程语言(C#、C++、JavaScript)开发的 Windows Store 应用调用可以调用它
移动类库	一个可由 WPF、Silverlight、Windows Phone 和 Windows Store 应用使用的类库
单元测试库(Windows Store 应用)	一个包含 Windows Store 应用单元测试的库
编码的 UI 测试项目	这个项目定义了编码的 UI 测试，以自动测试 UI

3. 使用 Web 项目模板

在 Visual Studio 2013 中，ASP.NET 的一个主要变化是 Web Forms、ASP.NET MVC 等不再有不同的模板可选。目前只有一个 ASP.NET Web Application 模板。选择这个模板时，可以为模板选择需要的功能，例如 ASP.NET MVC 或 ASP.NETWeb Forms，更便于在一个项目中混合使用不同的技术。

选择 ASP.NET Web Application 模板后，可以选择一些预配置的模板，如图 17-8 所示。

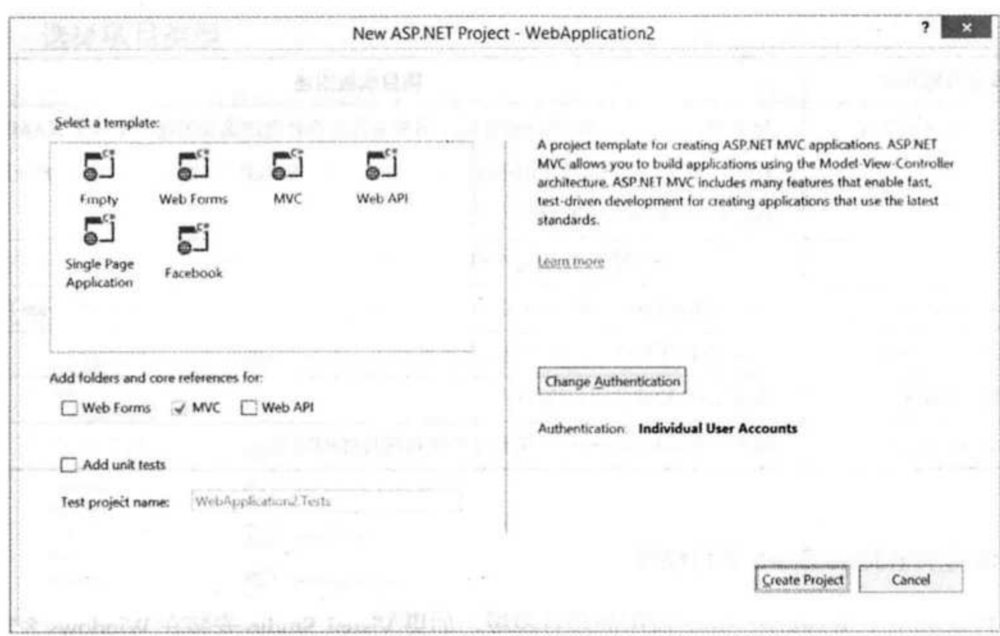


图 17-8

表 17-3 列出了为 Web 应用程序提供的模板，可以为 Web Forms、MVC 和 Web API 选择文件夹和核心引用。

表 17-3

项　　目	项目模板描述
空白应用程序	这个模板没有任何内容，适合于用 HTML 和 CSS 页面创建站点
Web Forms	这个模板默认为 Web Forms 添加文件夹。可以添加 MVC 和 Web API 配置，以混合它们
MVC	这个模板使用“模型-视图-控制器”模式和 Web 应用程序。它可以用于创建现代的 Web 应用程序
Web API	Web API 模板很容易创建 RESTful 服务。这个模板也添加 MVC 文件夹和核心引用，因为服务的文档用 MVC 创建
单页应用程序	这个模板使用 MVC 创建结构，仅使用一个页面，它利用 JavaScript 代码检索服务器中的数据
Facebook	这个模板基于 MVC，创建一个派生自 FacebookRealtimeUpdateController 的控制器，以便与 Facebook 集成

4. 使用 WCF 项目模板

要创建一个 Windows Communication Foundation(WCF)应用程序来实现客户端和服务器之间的通信，可以选择如表 17-4 所示的 WCF 项目模板。

表 17-4

项目模板名称	项目模板描述
WCF 服务库	一个包含示例服务合同和实现以及配置的库。该项目模板被配置为启动一个 WCF 服务宿主，用来托管服务和测试客户端应用程序

(续表)

项目模板名称	项目模板描述
WCF 服务应用程序	一个 Web 项目，它包含一个 WCF 服务合同和实现
WCF 工作流服务应用程序	一个 Web 项目，它托管一个使用工作流运行库的 WCF 服务
联合服务库	一个 WCF 服务库，它包含一个 WCF 服务合同和实现，以托管 RSS 或 ATOM 订阅源

5. 使用 Workflow 项目模板

表 17-5 列出了用于创建 Windows Workflow Foundation(WF)应用程序的项目模板。

表 17-5

项目模板名称	项目模板描述
工作流控制台应用程序	托管工作流的 Windows Workflow Foundation 可执行文件
WCF 工作流服务应用程序	一个 Web 项目，它托管了一个使用工作流运行库的 WCF 服务
活动库	一个可以用于工作流的工作流活动库
活动设计器库	一个库，用于创建活动的 XAML 用户界面来显示活动，在工作流设计器中配置活动

这不是一个完整的 Visual Studio 2013 项目模板列表，但它列出了一些最常用的模板。Visual Studio 2013 主要添加了 Windows Store 项目模板。这些新功能将会在本书其他章节中介绍。特别是，一定要参阅第 31 章和第 38 章。也可以在 New Project 对话框中通过搜索功能找到新的项目模板。

17.3 浏览并编写项目

本节着眼于 Visual Studio 提供用于帮助在项目中添加和浏览代码的功能。学习如何使用 Solution Explorer 浏览文件和代码，使用编辑器的 IntelliSense 和代码片段等功能，浏览其他窗口，如 Properties(属性)窗口和 Document Outline(文档大纲)。

17.3.1 Solution Explorer

创建项目之后，要用到的最重要的工具除了代码编辑器，就是 Solution Explorer。使用这个工具可以浏览项目的所有文件和项，查看所有的类和类成员。Solution Explorer 在 Visual Studio 2013 中得到了极大的改进。

在 Visual Studio 中运行控制台应用程序时，有一个常见的误解，即需要在 Main 方法的最后一行添加一个 Console.ReadLine 方法来保持控制台窗口打开。事实并非如此，通过命令 Debug | Start without Debugging(或按 Ctrl+F5 组合键)可以启动应用程序，而不必通过命令 Debug | Start Debugging(或按 F5 键)来开启。这样可以保持窗口打开直到按下某个键。使用 F5 键来开启应用程序也是有意义的，如果设置了断点，Visual Studio 就会在断点处挂起。

1. 使用项目和解决方案

Solution Explorer 会显示项目和解决方案。理解它们之间的区别是很重要的：

- 项目是一个包含所有源代码文件和资源文件的集合，它们将编译成一个程序集，在某些情况下也可能编译为一个模块。例如，项目可能是一个类库或一个 Windows GUI 应用程序。
- 解决方案是一个包含所有项目的集合，它们组合成一个特定的软件包(应用程序)。

要理解这个区别，可以考虑当发布一个包括多个程序集的项目时会发生什么。例如，可能有用户界面、自定义控件和作为应用程序一部分的库的其他组件。甚至可能为管理员提供不同的用户界面和通过网络调用的服务。应用程序的每一部分可能包含在单独的程序集中，因此 Visual Studio 会认为它们是单独的项目。而且很有可能并行编码这些项目，并将它们彼此结合。因此，在 Visual Studio 中把这些项目当作一个单位来编辑是非常有用的。Visual Studio 允许把所有相关的项目构成一个解决方案，并且当作一个单位来处理，Visual Studio 会读取该单位并允许在该单位上进行工作。

到目前为止，本章已经零散地讨论创建一个控制台项目。在这个例子中，Visual Studio 实际上已经创建一个解决方案，只不过它仅包含一个项目而已。可以在 Solution Explorer 中看到这样的场景(如图 17-9 所示)，它包含一个树型结构，用于定义该解决方案。

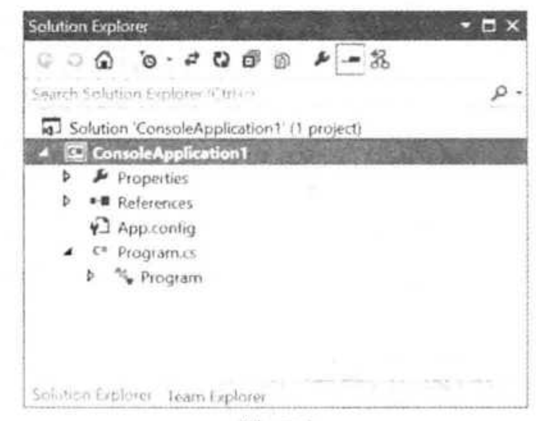


图 17-9

在这个例子中，项目包含了源文件 Program.cs，以及另一个 C#源文件 AssemblyInfo.cs(在 Properties 文件夹中)，该 C#源文件用于描述程序集的信息，指定版本信息(可在第 19 章中查阅该文件的详细信息)。Solution Explorer 也显示了项目引用的程序集。在 Solution Explorer 中展开 References 文件夹就可以看到这些信息。

如果在 Visual Studio 中没有改变任何默认设置，在屏幕右上方就可以找到 Solution Explorer。如果找不到它，则可以进入 View 菜单栏并选择 Solution Explorer。

解决方案是用一个扩展名为.sln 的文件来描述的，在这个示例中，它是 ConsoleApplication1.sln。解决方案文件是一个文本文件，它包含解决方案中所有项目的信息，以及可用于所有包含项目的全局项。

C#项目是用一个扩展名为.csproj 的文件来描述的，在这个示例中，它是 ConsoleApplication1.csproj。这是一个 XML 文件，可以在 Solution Explorer 中直接打开它。但是，为此，需要先卸载这个项目，可以右击项目名称并在上下文菜单中选择 Unload Project 命令来进行卸载。项目卸载之后，在上下文菜单中选择 Edit ConsoleApplication1.csproj，就可以直接访问 XML 文件的内容了。

显示隐藏文件

默认情况下，Solution Explorer 隐藏了一些文件。单击 Solution Explorer 工具栏中的 Show All Files 按钮，可以显示所有隐藏的文件。例如，bin 和 obj 子文件夹存放了编译和中间文件。obj 子文件夹存放各种临时或中间文件；bin 子文件夹存放已编译的程序集。

2. 将项目添加到一个解决方案中

下面各节将介绍 Visual Studio 如何处理 Windows 桌面应用程序和控制台应用程序。最终会创建一个名为 BasicForm 的 Windows 项目，将它添加到当前的解决方案 ConsoleApplication1 中。

创建 BasicForm 项目，得到的解决方案将包含一个 WPF 应用程序和一个控制台应用程序。这种情况并不多见，更有可能的是解决方案包含一个应用程序和许多类库。这么做只是为了展示更多的代码。不过，有时需要创建这样的解决方案，例如，编写一个既可以运行为 WPF 应用程序、又可以运行为命令行实用工具的实用程序。

创建新项目的方式有几种。一种方式是在 File 菜单中选择 New | Project(前面就是这么做的)，或者在 File 菜单中选择 Add | New Project。选择 New Project 命令将打开熟悉的 Add New Project 对话框，如图 17-10 所示。不过，此时 Visual Studio 会在已有 ConsoleApplication1 项目所在的解决方案中创建新项目。

如果选择该选项，就会添加一个新项目，因此 ConsoleApplication1 解决方案现在包含一个控制台应用程序和一个 WPF 应用程序。

Visual Studio 支持语言独立性，所以新项目并不一定是 C#项目。将 C#项目、Visual Basic 项目和 C++项目放在同一个解决方案中是完全可行的。但是，本书的主题是 C#，所以创建 C#项目。

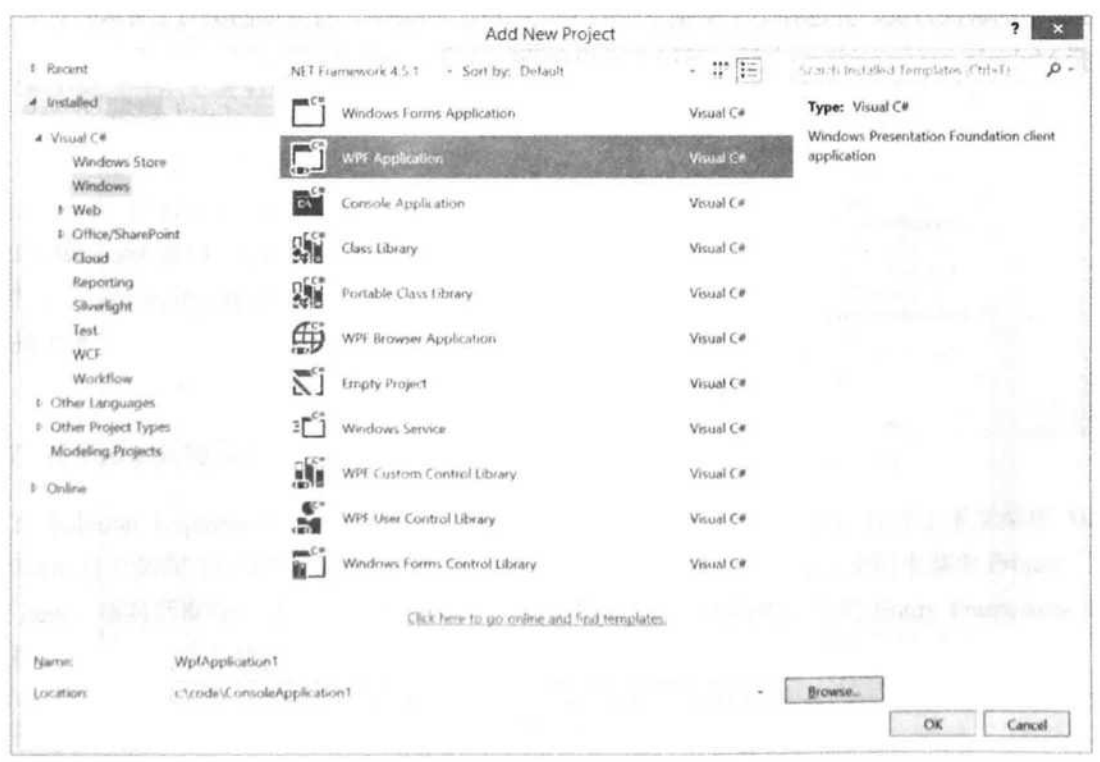


图 17-10

当然，这意味着 ConsoleApplication1 不再适合作为解决方案的名称。要改变名称，可以右击解决方案的名称，并选择上下文菜单中的 Rename 命令。将新的解决方案命名为 DemoSolution。Solution Explorer 现在如图 17-11 所示。

可以看出，Visual Studio 自动为新添加的 WPF 项目引用一些额外的基类，这些基类对于 WPF 功能非常重要。

注意，在 Windows Explorer 中，解决方案文件的名称已经改为 DemoSolution.sln。通常，如果想重命名任何文件，Solution Explorer 窗口是最合适的选择，因为 Visual Studio 会自动更新它在其他项目文件中的引用。如果只使用 Windows Explorer 来重命名文件，可能会破坏解决方案，因为 Visual Studio 不能定位需要读入 IDE 的所有文件。因此需要手动编辑项目和解决方案文件，来更新文件引用。

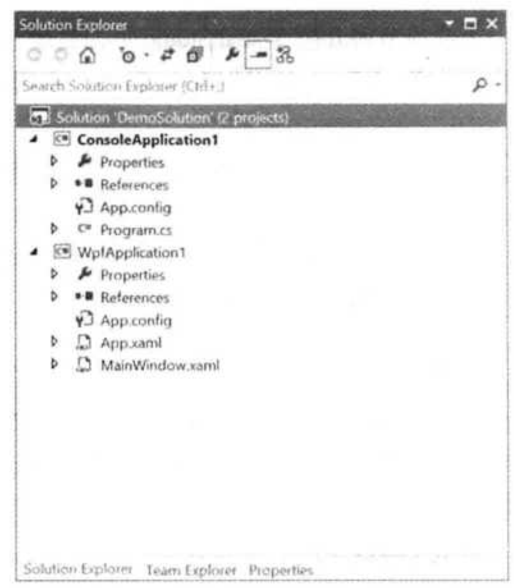


图 17-11

3. 设置启动项目

请记住，如果一个解决方案有多个项目，就需要配置哪个项目作为启动项目来运行。也可以配置多个同时启动的项目。这有多种方式。在 Solution Explorer 中选择一个项目之后，上下文菜单会提供 Set as Startup Project 选项，它允许一次设置一个启动项目。也可以使用上下文菜单中的 Debug | Start new instance 命令，在一个项目后启动另一个项目。要同时启动多个项目，右击 Solution Explorer 中的解决方案，并选择上下文菜单中的 Set Startup Projects，打开如图 17-12 所示的对话框。当选择 Multiple startup projects 之后，可以定义启动哪些项目。

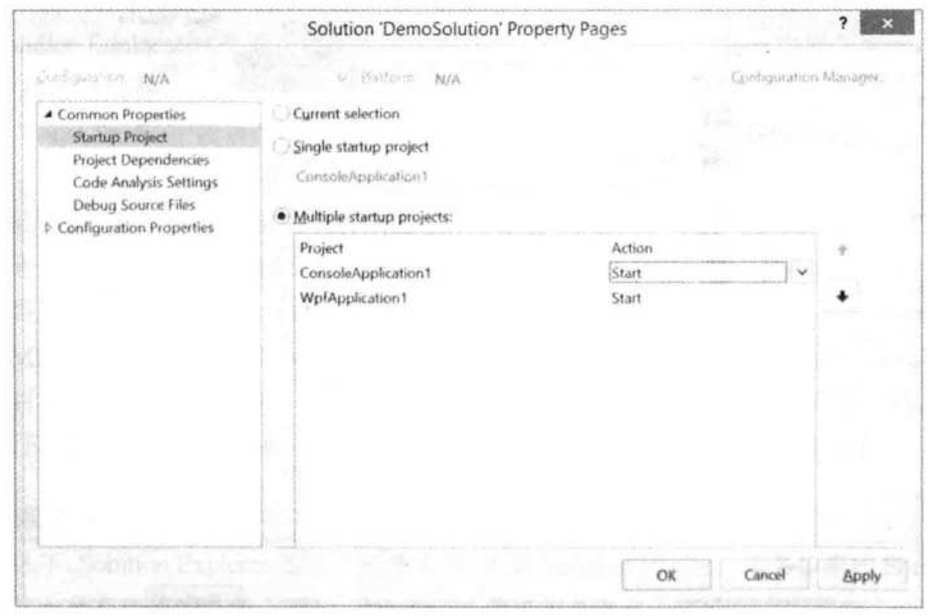


图 17-12

4. 浏览类型和成员

当 Visual Studio 初次创建 WPF 应用程序时，该应用程序比控制台应用程序要多包含一些初始代码。这是因为创建窗口是一个较复杂的过程。第 35 章详细讨论 WPF 应用程序的代码。现在，查看 MainWindow.xaml 中的 XAML 代码，和 MainWindow.xaml.cs 中的 C#代码。这里也有一些隐藏的 C#代码。遍历 Solution Explorer 中的树状结构，在 MainWindow.xaml.cs 的下面可以找到 MainWindow 类。Solution Explorer 在该文件中显示了所有代码文件中的类型。在 MainWindow 类型中，可以看到类的所有成员。_contentLoaded 是一个布尔类型的字段。单击这个字段，会打开 MainWindow.g.i.cs 文件。这个文件是 MainWindow 类的一部分，它由设计器自动生成，包含一些初始化代码。

5. 预览条目

Visual Studio 2013 在 Solution Explorer 中提供的一个新功能是 Preview Selected Items 按钮。启用这个按钮，在 Solution Explorer 中单击一项，就会打开该项的编辑器，这与往常相同。但如果该项以前没有打开过，编辑器的选项卡流就会在最右端显示新打开的项。现在单击另一项，以前打开的项就会关闭。这大大减少了打开的项数。

在预览项的编辑器选项卡中有 Keep Open 按钮，它会使该项目在单击另一项时仍处于打开状态，保持打开的项的选项卡会向左移动。

6. 使用作用域

设置作用域可以让用户专注于解决方案的某一特定部分。Solution Explorer 列表显示的项会越来越多。例如，打开一个类型的上下文菜单，就可以从 Base Types 菜单中选择该类型的基类型。这里可以看到完整的类型继承层次结构，如图 17-13 所示。

因为 Solution Explorer 包含的信息量比在一个屏幕中可以轻松查看的信息量要多，所以可以用 New Solution Explorer View 菜单项一次打开多个 Solution Explorer 窗口，并且可以设置作用域来显示一个特定元素。例如，要显示一个项目或者一个类，可选择上下文菜单中的 Scope to This 命令。要返回到以前的作用域，可单击 Back 按钮。

图 17-13

7. 将项添加到项目中

在 Solution Explorer 中可以直接将不同的项添加到项目中。选择项目，打开上下文菜单 Add | New Item，打开如图 17-14 所示的对话框。打开这个对话框的另一种方式是，使用主菜单 Project | Add New Item。该对话框有很多不同的类别，例如添加类或接口的代码项、使用 Entity Framework 或其他数据访问技术的数据项等。

图 17-14

8. 管理引用

如图 17-15 所示的 Reference Manager 在 Visual Studio 中得到了很大的改进。在 Solution Explorer 中选择 References，并单击上下文菜单 Add Reference，打开这个对话框。在这里可以把引用添加到同一个解决方案中的其他程序集、来自.NET Framework 的程序集、COM 类型库，还可以浏览磁盘上的程序集。

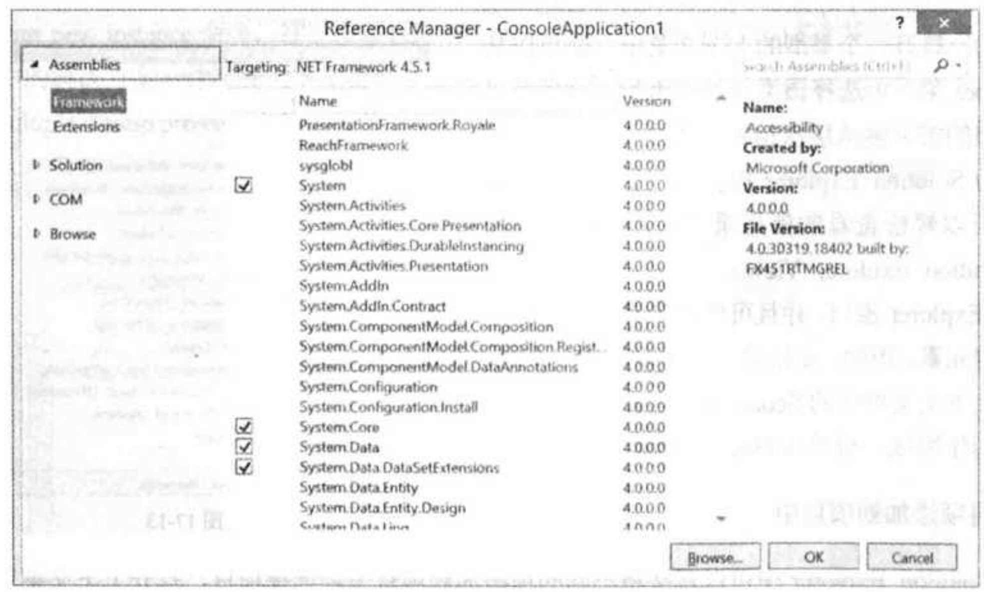


图 17-15

9. 使用 NuGet 包安装、更新 Microsoft 工具与第三方工具

如图 17-16 所示的 NuGet 包管理器是一个重要的工具，用于安装和更新 Microsoft 与第三方库和工具。.NET Framework 的某些部分需要单独安装，例如 Entity Framework 6.0 版本或 TPL DataFlow；此外还有一些 JavaScript 库，如 jQuery 和 Modernizr。如果项目包含了通过 NuGet 包管理器安装的

包，当新版本的包可用时会自动通知。

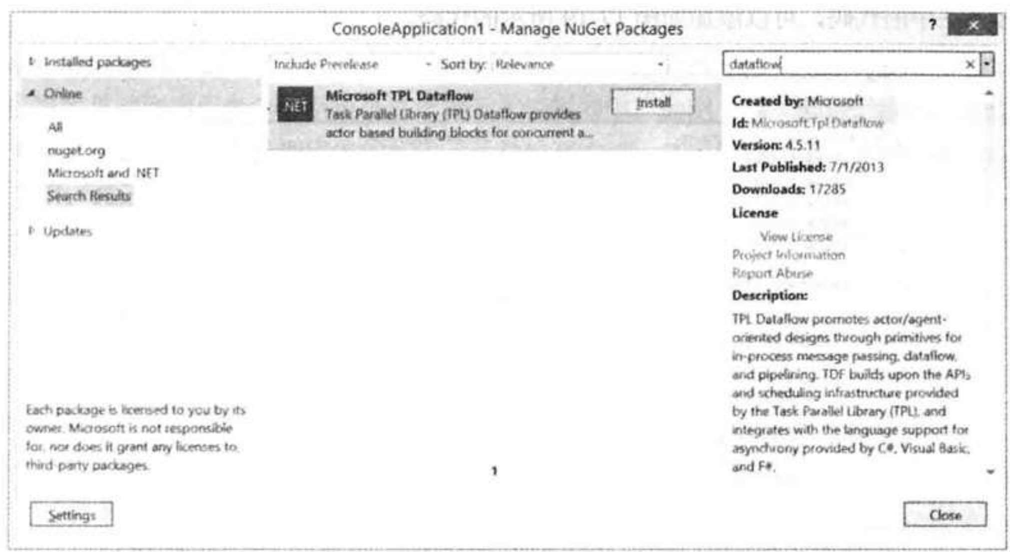


图 17-16

17.3.2 使用代码编辑器

Visual Studio 代码编辑器是进行大部分开发工作的地方。在 Visual Studio 中，从默认配置中移除一些工具栏，并移除了菜单栏、工具栏和选项卡标题的边框，从而增加了代码编辑器的可用空间。下面介绍该编辑器中最有用的功能。

1. 可折叠的编辑器

Visual Studio 中的一个显著功能是使用可折叠的编辑器作为默认的代码编辑器。图 17-17 是前面生成的控制台应用程序代码。注意窗口左侧的小减号，这些符号所标记的点是编辑器认为新代码块(或文档注释)的开始位置。可以单击这些图标来关闭相应代码块的视图，如同关闭树状控件中的节点，如图 17-18 所示。

图 17-17

图 17-18

这意味着在编辑时可以只关注所需的代码区域，隐藏此刻不感兴趣的代码。如果不喜欢编辑器

折叠代码的方式，可以用 C#预处理器指令#region 和#endregion 来指定要折叠的代码块。例如，要折叠 Main 方法中的代码，可以添加如图 17-19 所示的代码。

图 17-19

代码编辑器自动检测#region 块，并通过#region 指令放置一个新的减号标识，这允许关闭该区域。封闭区域中的这段代码允许编辑器关闭它(如图 17-20 所示)，在#region 指令中用指定的注释标记这个区域。然而，编译器会忽略这些指令，跟往常一样编译 Main 方法。

图 17-20

2. IntelliSense

除了可折叠编辑器的功能之外，Visual Studio 的代码编辑器也集成了 Microsoft 流行的 IntelliSense 功能。它不仅减少了输入量，还确保使用正确的参数。IntelliSense 会记住首选项，并从这些选项开始提供列表，而不是使用 IntelliSense 提供的有时相当长的列表。

代码编辑器甚至在编译代码之前就对代码进行语法检查，用短波浪线指示错误。将鼠标指针悬停在带有下划线的文本上，会弹出一个包含了错误描述的小方框。

3. CodeLens

Visual Studio 2013 中的一个新功能是 CodeLens。用户可能修改了一个方法，但忘了调用它的方法。现在很容易找到调用者。引用数会直接显示在编辑器中，如图 17-21 所示。单击引用链接时，会打开 CodeLens ，以便查看调用者的代码，并导航到它们。还可以使用另一个新功能 Code Map 来查看引用，Code Map 参阅 17.8 节。

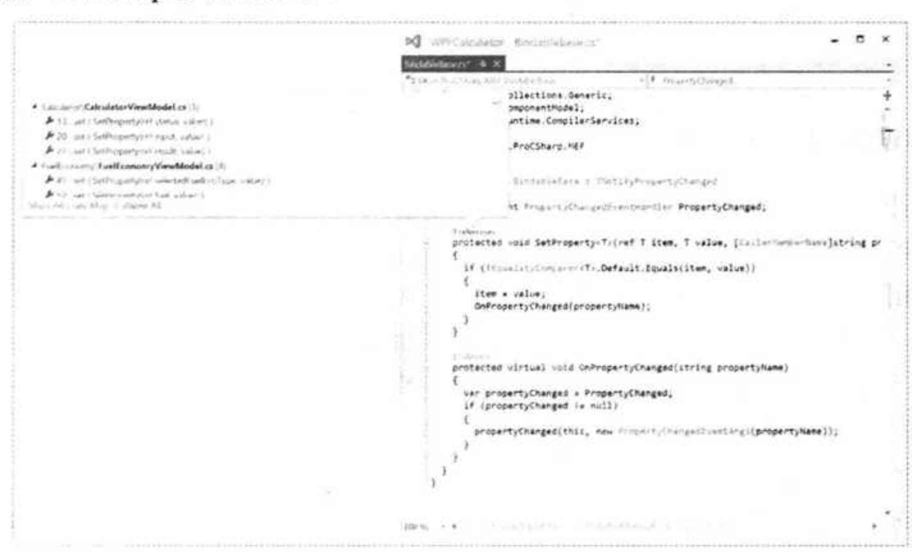


图 17-21

4. 使用代码片段

代码片段提升了代码编辑器的工作效率，仅需要在编辑器中写入 cw<tab><tab>，编辑器就会创建 Console.WriteLine();。Visual Studio 自带很多代码片段，例如，使用快捷方式 do、for、forr、foreach、while 来创建循环，使用 equals 来实现 Equals 方法，使用 attribute 和 exception 来创建 Attribute 和 Exception 派生类型等。选择 Tools | Code Snippets Manager，在打开的 Code Snippets Manager 中可以看到所有可用的代码片段(如图 17-22 所示)。也可以创建自定义的代码片段。

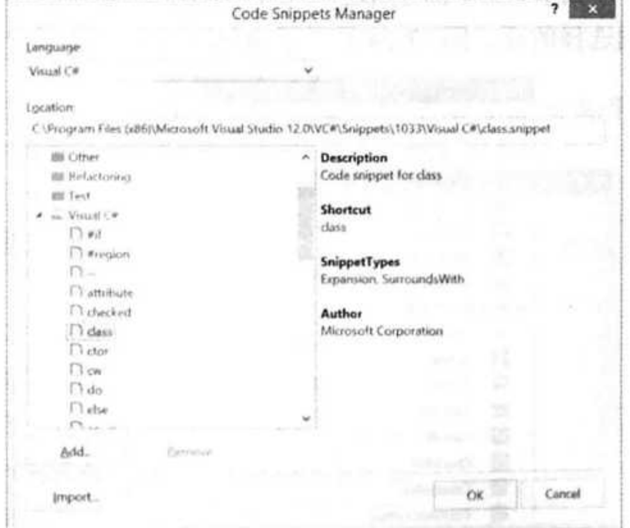


图 17-22

Visual Studio 22013 提供了用于 XAML 的代码片段。代码片段可以从 http://xamlsnippets.codeplex.com 上获得。

17.3.3 学习和理解其他窗口

除了代码编辑器和 Solution Explorer 外，Visual Studio 还提供了许多其他窗口，允许从不同的角

度来查看或管理项目。

本节的其余部分介绍其他几个窗口。如果这些窗口在屏幕上不可见，可以在 View 菜单中选择它们。要显示设计视图或代码编辑器，可以右击 Solution Explorer 中的文件名，并选择上下文菜单中的 View Designer 或 View Code；也可以选择 Solution Explorer 顶部工具栏中的对应项。设计视图和代码编辑器共用同一个选项卡式窗口。

1. 使用设计视图窗口

如果设计一个用户界面应用程序，如 WPF 应用程序、Windows 控件库或 ASP.NET Web Forms 应用程序，可以使用设计视图窗口。这个窗口显示窗体的可视化概览。设计视图窗口经常和工具箱窗口一起使用。工具箱包含许多.NET 组件，可以将它们拖放到程序中。工具箱的组件会根据项目类型而有所不同。图 17-23 显示了 WPF 应用程序的工具箱。

要将自定义的类别添加到工具箱，请执行如下步骤：

(1) 右击任何一个类别。

(2) 选择上下文菜单中的 Add Tab。

也可以选择上下文菜单中的 Choose Items，在工具箱中放置其他工具，这尤其适合于添加自定义的组件或工具箱默认没有显示的.NET Framework 组件。

2. 使用 Properties 窗口

如本书第 I 部分所述，.NET 类可以实现属性。Properties 窗口可用于项目、文件和使用设计视图选择的项。图 17-24 显示了 Windows Service 的 Properties 视图。

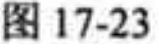


图 17-23

图 17-24

在这个窗口中可以看到一项的所有属性，并对其进行相应的配置。一些属性可以通过在文本框中输入文本来改变，一些属性有预定义的选项，一些属性有自定义的编辑器(例如图 17-25 中 ASP.NET Web Forms 的 More Colors 对话框)。也可以在 Properties 窗口中添加事件处理程序。

在 WPF 应用程序中，Properties 窗口看起来非常不同，如图 17-26 所示。这个窗口提供了很多图形效果，并允许用图形方式来设置属性。它看起来很熟悉，可能是因为它来源于 Expression Blend。如前所述，Visual Studio 和 Blend for Visual Studio 有许多相似之处。

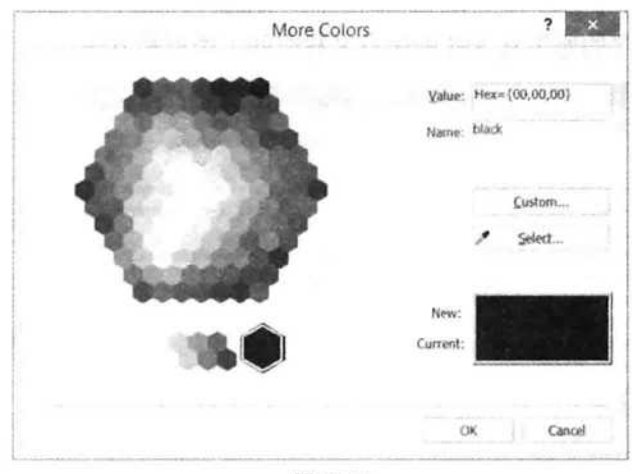


图 17-25

图 17-26

标准的 Properties 窗口是 System.Windows.Forms.PropertyGrid 的一个实例，它在内部使用了第 15 章描述的反射技术来识别属性和要显示的属性值。

3. 使用类视图窗口

Solution Explorer 可以显示类和类的成员，这是类视图的一般功能(如图 17-27 所示)。要调用类视图，可选择 View | Class View。类视图显示代码中的名称空间和类的层次结构。它提供了一个树型结构，可以展开该结构来查看名称空间下包含哪些类，类中包含哪些成员。

图 17-27

类视图的一个杰出功能是，如果右击任何有权访问其源代码的项的名称，然后选择上下文菜单中的 Go To Definition 命令，就会转到代码编辑器中的项定义。另外，在类视图中双击该项(或在代码编辑器中右击想要的项，并从上下文菜单中选择相同的选项)，也可以查看该项的定义。上下文菜单还允许给类添加字段、方法、属性、或索引器。换句话说，在对话框中指定相关成员的详细信息，就会自动添加代码。这个功能对于添加属性和索引器非常有用，因为它可以减少相当多的输入量。

4. 使用 Object Browser 窗口

在.NET 环境中编程的一个重要方面是能够找出基类或从程序集引用的其他库中有哪些可用的方法和其他代码项。这个功能可通过 Object Browser 窗口来获得。在 Visual Studio 2013 中选择 View 菜单中的 Object Browser，可以访问这个窗口。使用这个工具，可以浏览并选择现有的组件集，如.NET 4.5.1、.NET 4.5、.NET 4.0、.NET 3.5、适用于 Windows Store 应用程序的.NET，并查看这个子集中可用的类和类成员。在 Browse 下拉框中选择 Windows(如图 17-28 所示)，来选择 Windows 运行库，也可以找到这个用于 Windows Store 应用程序的原生新 API 的所有名称空间、类型和方法。

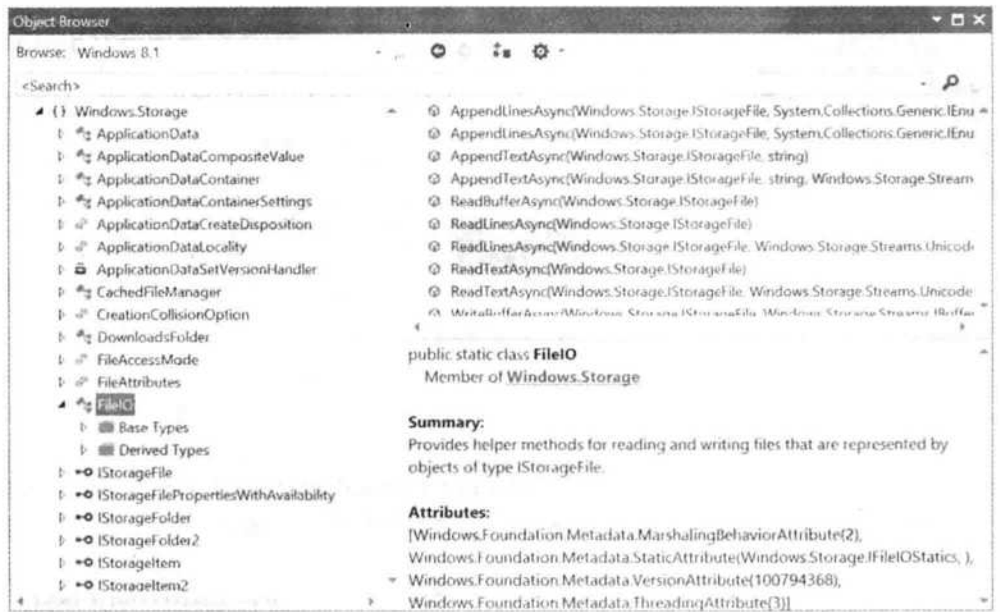


图 17-28

5. 使用 Server Explorer 窗口

使用 Server Explorer 窗口，如图 17-29 所示，可以在编码时找出计算机在网络中的相关信息。在该窗口的 Servers 部分中，可以找到服务运行情况的信息(这对于开发 Windows 服务是非常有用的)，创建新的性能计数，访问事件日志。在 Data Connection 部分中不仅能够连接现有数据库，查询数据，还可以创建新的数据库。Visual Studio 2013 也有一些内置于 Server Explorer 的 Windows Azure 信息，包括 Windows Azure Compute、Mobile Services、Storage、Service Bus 和 Virtual Machines 选项。

6. 使用 Document Outline 窗口

可用于 WPF 应用程序的一个窗口是 Document Outline。如图 17-30 所示，在这个窗口中打开了第 36 章的一个应用程序，从中可以查看 XAML 元素的逻辑结构和层次结构，锁定元素以防止其无意中被改变，在层次结构中轻松地移动元素，在新的元素容器中分组元素和改变布局类型。

使用这个工具还可以创建 XAML 模板，图形化地编辑数据绑定。

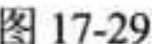


图 17-29

图 17-30

17.3.4 排列窗口

学习 Visual Studio 时会发现，许多窗口有一个有趣的功能会让人联想到工具栏。尤其是，它们都可以浮动(也可以显示在第二个显示器上)，也可以停靠。当它们停靠时，在每个窗口右上角的最小化按钮旁边会显示一个类似图钉的额外图标。这个图标的作用确实像图钉，它可以用来固定打开的窗口。固定窗口(图钉是垂直显示的)的行为与平时使用的窗口一样。但当它们取消固定时(图钉是水平显示的)，则窗口只有获得焦点，才会打开。当失去焦点时(因为单击或者移动鼠标到其他地方)，它们会快速退出到 Visual Studio 应用程序的主边框内。固定和取消固定窗口提供了另一种方式来更好地利用屏幕上有限的空间。

17.4 构建项目

Visual Studio 不仅可以编写项目，它实际上是一个 IDE，管理着项目的整个生命周期，包括生成或编译解决方案。本节讨论如何用 Visual Studio 生成项目。

17.4.1 构建、编译和生成

讨论各种构建选项之前，先要弄清楚一些术语。从源代码转换为可执行代码的过程中，经常看到 3 个不同的术语：构建、编译和生成。这 3 个术语的起源反映了一个事实：直到最近，从源代码到可执行代码的过程涉及多个步骤(在 C++中仍然如此)。这主要是因为一个程序包含了大量的源文件。

例如，在 C++中，每个源文件都需要单独编译。这就产生了所谓的对象文件，每个对象文件包含类似于可执行代码的内容，但每个对象文件只与一个源文件相关。要生成一个可执行文件，这些对象文件需要连接在一起，这个过程官方称为链接。这个合并过程通常称为构建代码(至少在 Windows 平台上是如此)。然而，在 C#术语中，编译器比较复杂，能够将所有的源文件当作一个块来读取和处理。因此，没有真正独立的链接阶段，所以在 C#上下文中，术语“构建”和“编译”可以交替使用。

术语“生成”的含义与“构建”基本相同，虽然它在 C#上下文中没有真正使用。术语“生成”

起源于旧的大型机系统，在该系统中，当一个项目由许多源文件组成时，就在一个单独的文件中写入指令，告诉编译器如何构建项目：包含哪些文件和链接什么库等。这个文件通常称为生成文件，在UNIX系统上它仍然是非常标准的文件。事实上，项目文件和旧的生成文件非常类似，它只是一个新的高级XML变体。可以使用MSBuild命令，将项目文件当作输入，来编译所有的源文件。使用构建文件非常适合于在一个单独的构建服务器上进行构建，其中所有的开发人员仅需要签入他们的代码，构建过程会在深夜自动完成。

17.4.2 调试版本和发布版本

C++开发人员非常熟悉生成两个版本的这种思想，有Visual Basic开发背景的开发人员也不会十分陌生。其关键在于：可执行文件在调试时的目标和行为应与正式发布时不同。准备发布软件时，可执行文件应尽可能小而快。但是，这两个目标与调试代码时的需求不兼容，在接下来的小节中将看到这一点。

1. 优化

在高性能方面，编译器对代码进行的多次优化起到了一定的作用。这意味着编译器在编译代码时，会在代码实现细节中积极找出可以修改的地方。编译器所作的修改并不会改变整体效果，但是会使程序更加高效。例如，假设编译器遇到了下面的源代码：

```
double InchesToCm(double ins)
{
   return ins*2.54;
}

// later on in the code
Y = InchesToCm(X);
```

就可能把它们替换为下面的代码：

```
Y = X * 2.54;
```

类似地，编译器可能把下面的代码：

```
{
   string message = "Hi";
   Console.WriteLine(message);
}
```

替换为：

```
Console.WriteLine("Hi");
```

这样，编译器就不需要在此过程中声明任何非必要的对象引用。

C#编译器会进行怎样的优化无从判断，我们也不知道前两个例子中的优化在特定情况中是否会实际发生，因为编译器的文档没有提供这类细节。不过，对于C#这样的托管语言，上述优化很可能在JIT编译时发生，而不是在C#编译器把源代码编译为程序集时发生。显然，由于专利原因，编写编译器的公司通常不愿意过多地说明他们使用了什么技巧。注意，优化不会影响源代码，而只影响可执行代码的内容。通过前面的示例，可以基本了解优化产生的效果。

问题在于，虽然示例代码中的优化可以加快代码的运行速度，但是它们也增加了调试的难度。在第一个例子中，假设想要在 InchesToCm()方法中设置一个断点，了解该方法的工作机制。如果在编译器做了优化后，可执行代码中不再包含 InchesToCm()方法，怎么可能进行这种操作呢？同样，如果编译后的代码中不再包含 Message 变量，又如何监视该变量的值？

2. 调试器符号

在调试过程中，经常需要查看变量的值，这时使用的是它们在源代码中的名称。问题是可执行代码一般不包含这些名称——编译器用内存地址代替了变量名称。.NET 在一定程度上改变了这种情况，使程序集中的某些项以名称的形式存储，但是这只适用于少量的项(例如公有类和方法)，而且这些名称在 JIT 编译程序集后也仍然会被移除。如果让调试器显示变量 HeightInInches 的值，但是编译器在查看可执行代码时只看到了地址，而没有看到任何对名称 HeightInInches 的引用，自然就得不到期望的结果。

因此，为了正确地调试，需要在可执行文件中提供一些额外的调试信息。这些信息包含变量名和代码行信息，允许调试器确定可执行机器汇编语言指令与源代码中的哪些指令对应。但是，不应该在发布版本中包含这些信息，这既是出于专利考虑(提供调试信息会方便其他人反汇编代码)，也是因为包含调试信息会增加可执行文件的大小。

3. 其他源代码调试指令

一个相关问题是，在调试时，程序经常包含一些额外的代码行，用于显示关键的调试信息。显然，在发布软件前，需要从可执行代码中彻底删除这些相关指令。手动删除是可以的，但是如果能以某种方式标记这些语句，让编译器在编译发布代码时自动忽略它们，不是更方便吗？本书的第 I 部分提到，在 C#中，定义合适的预处理器指令，再结合使用 Conditional 特性(所谓的条件编译)，就可以实现这种操作。

所有这些因素综合到一起，决定了几乎所有商业软件的编译调试方式与最终交付产品的编译方式是稍有区别的。Visual Studio 能够处理这种区别，因为 Visual Studio 在编译代码时，会存储应传递给编译器的所有编译选项信息。为了支持不同类型的构建版本，Visual Studio 需要存储多组编译选项。这些不同的版本信息集合称为配置。在创建项目时，Visual Studio 会自动提供两种配置：调试和发布。

- **调试**：这种配置通常指定编译器不优化编译过程，可执行文件应该包含额外的调试信息，编译器假定调试预处理器指令 Debug 是存在的，除非源代码中显式使用了#undefined Debug 指令。
- **发布**：这种配置指定编译器应优化编译过程，可执行文件不应包含额外的调试信息，编译器不应假定源代码包含特定的预处理器符号。

还可以定义自己的配置，例如设置软件的专业级版本和企业级版本。过去，由于 Windows NT 支持 Unicode 字符编码，但 Windows 95 不支持，因此 C++项目经常使用 Unicode 配置和 MBCS(Multi-Byte Character Set，多字节字符集)配置。

17.4.3 选择配置

Visual Studio 存储了多个配置的细节，那么在准备生成一个项目时，如何决定使用哪个配置？

答案是，项目总是有一个活动的配置，当要求 Visual Studio 生成项目时，就使用这个配置。注意，活动配置是针对每个项目、而不是每个解决方案设置的。

在创建项目时，默认情况下 Debug 配置是活动配置。如果想修改活动配置，可以单击 Build 菜单，选择 Configuration Manager 菜单项。在 Visual Studio 主工具栏的下拉菜单中也可以找到此选项。

17.4.4 编辑配置

除了选择活动配置外，还可以查看及编辑配置。为此，在 Solution Explorer 中选择相关的项目，然后选择 Project 菜单中的 Properties 菜单项，这会打开一个复杂的对话框。打开该对话框的另一个方法是，在 Solution Explorer 中右击项目名称，然后从上下文菜单中选择 Properties。

这个对话框包含多个选项卡，用于选择要查看或编辑的常规属性类别。由于篇幅原因，本节不展示所有的属性类别，只介绍其中两个最重要的选项卡。

图 17-31 显示了这个选项卡式对话框，它列出了应用程序的可用属性。这个屏幕截图显示了本章前面创建的 ConsoleApplication1 项目的常规应用程序设置。

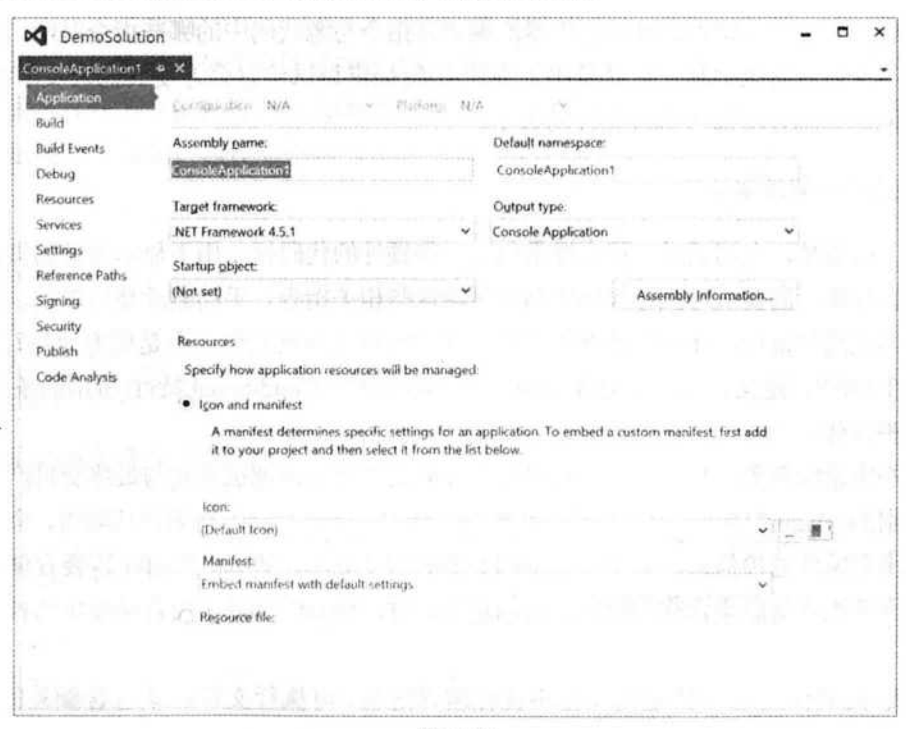


图 17-31

在 Application 选项卡中，可以选择要生成的程序集的名称及类型。可选的类型包括 Console Application、Windows Application 和 Class Library。当然，如果愿意，还可以修改程序集的类型，但是，为什么不在一开始就让 Visual Studio 生成正确类型的项目呢？

图 17-32 显示了生成配置属性。注意，在对话框顶部的下拉列表中可以指定要查看的配置类型。对于 Debug 配置，编译器假定已经定义了 DEBUG 和 TRACE 预处理器符号。此外，编译器不会优化代码，而且会生成额外的调试信息。

一般来说，并不需要调整配置设置。另一方面，如果确实需要修改，现在也已经熟悉可用的不同配置属性。

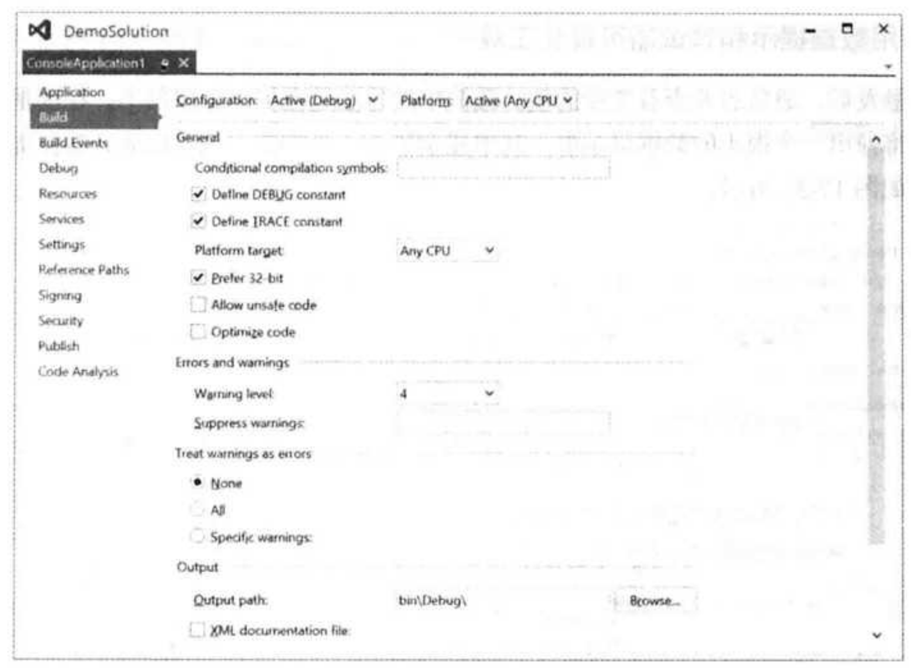


图 17-32

17.5　调试代码

现在，已经准备好运行和调试应用程序了。在 C#中调试应用程序与早于.NET 的语言一样，涉及的主要技巧是设置断点，使用断点检查在代码执行到特定位置时发生了什么情况。

17.5.1　设置断点

在 Visual Studio 中，可在实际执行的任何代码行上设置断点。最简单的方法是在代码编辑器中单击文档窗口最左边的灰色区域，或者在选中合适的行后按 F9 键。这会在该代码行上设置一个断点，当程序执行到该行时将暂停执行，并把控制权转交给调试器。与 Visual Studio 以前的版本一样，断点由代码编辑器中代码行左边的红色圆圈表示。Visual Studio 还使用不同的颜色高亮显示该行代码的文本和背景。再次单击红色圆圈将删除断点。

如果对于特定的问题，每次在特定的代码行暂停执行不足以解决该问题，则还可以设置条件断点。为此，选择 Debug | Windows | Breakpoints。在打开的对话框中，输入想要设置的断点的细节。例如，在该对话框中可以执行以下操作：

- 指定只有遇到断点超过一定次数时，执行才应该中断。
- 指定只有遇到代码行达到一定次数时，才激活断点。例如，执行代码行每达到 20 次时就激活断点。这对于调试大型循环很有帮助。
- 指定只有遇到变量满足一定条件时，执行才应该中断。此时，变量的值将被监视，一旦该值发生变化，断点就会触发。但是，这个选项可能会显著减慢代码的执行。在每行代码执行后就检查某个变量的值是否发生变化，会增加大量的处理器时间。

使用该对话框还可以导出和导入断点设置，如果根据不同的调试场景想要使用不同的断点设置，那么这些选项十分有用。另外，在该对话框中还可以存储调试设置。

17.5.2 使用数据提示和调试器可视化工具

在断点触发后，通常想要查看变量的值。最简单的方法是在代码编辑器中，在变量名的上方悬停光标。这将弹出一个很小的数据提示框，其中显示了该变量的值。也可以展开数据提示框来查看更多细节，如图 17-33 所示。

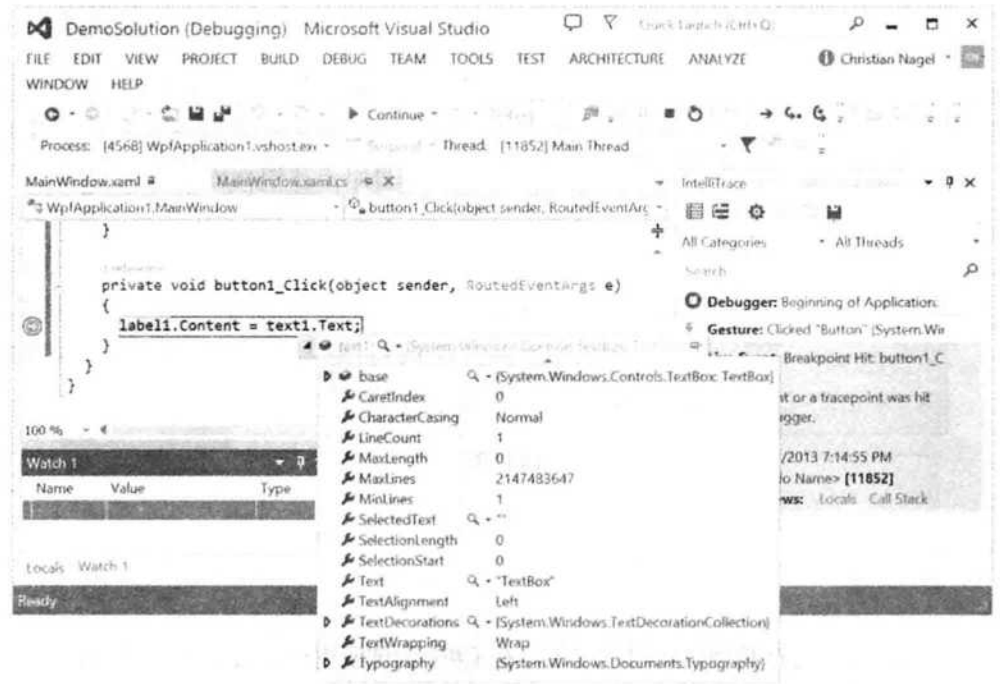


图 17-33

数据提示中的某些值会带有一个放大镜图标。单击这个放大镜图标时，根据数据的类型，会显示一个或多个使用调试器可视化工具(debugger visualizer)的选项。对于 WPF 控件，使用 WPF Visualizer 可以更详细地查看控件，如图 17-34 所示。在此可视化工具中，可以使用可视化树查看运行期间的变量，包括所有的实际属性设置。通过该可视化树可以预览在该树中选择的元素。

图 17-34

如图 17-35 所示的 XML Visualizer 可显示 XML 内容。还有其他许多可视化工具，如 HTML 和 Text 可视化工具，以及显示 DataTable 或 DataSet 内容的可视化工具。

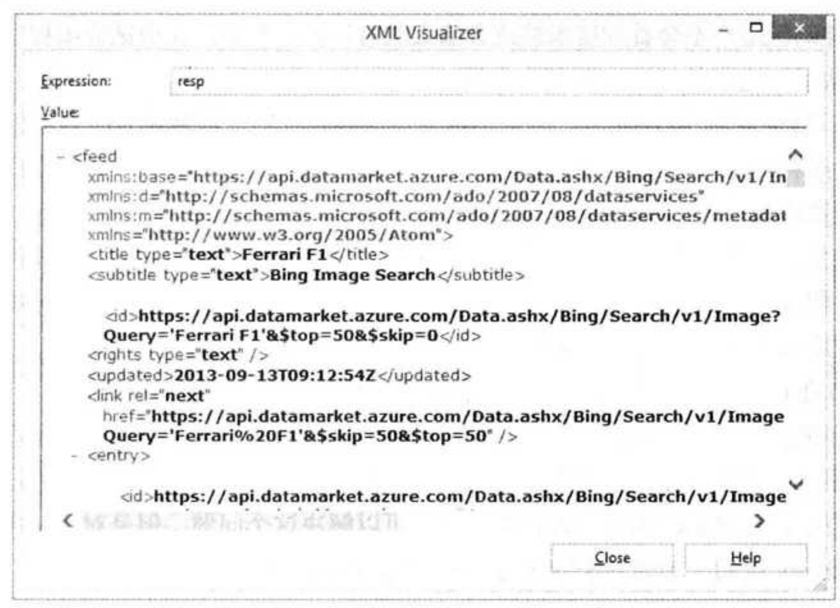


图 17-35

17.5.3　监视和修改变量

有时候，需要连续查看变量的值。此时，可以使用 Autos、Locals 和 Watch 窗口检查变量的内容。这 3 个窗口监视不同的变量：

- Autos——监视在程序执行过程中离断点最近的几个变量。
- Locals——监视在程序执行过程中当前断点所在方法内可访问的变量。
- Watch——监视在程序执行过程中显式指定名称的任何变量。可以把变量拖放到 Watch 窗口中。

只有当使用调试器运行程序时，这些窗口才是可见的。如果看不到它们，则选择 Debug | Windows，然后根据需要选择菜单项。考虑到可能要监视的内容过多，需要进行分组，Watch 窗口提供了 4 个不同的窗口。在这些窗口中都可以查看和修改变量的值，所以不必离开调试器就可以尝试改变程序的不同路径。图 17-36 显示了 Locals 窗口。

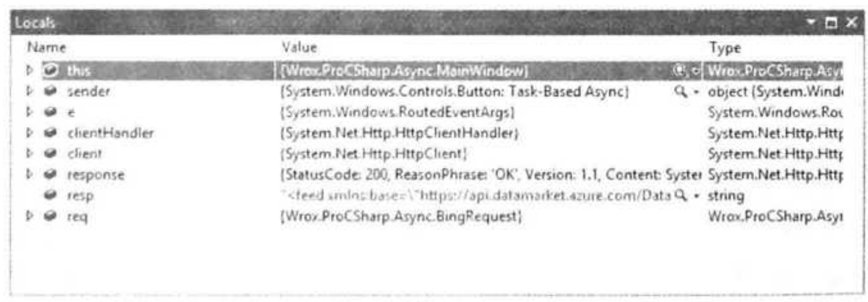


图 17-36

另外，还有一个 Immediate 窗口，虽然它与刚才讨论的其他几个窗口没有直接关系，但仍然是一个可以监视和修改变量的重要窗口。在该窗口中可以查看变量的值。还可以在此窗口中输入并运行代码，这样在调试过程中进行测试时就能够关注细节，试用方法，以及动态修改调试运行。

17.5.4 异常

在准备交付应用程序时，异常是很好的帮手，它们确保错误条件得到了恰当的处理。如果正确使用，异常能够确保用户不会看到技术性或者恼人的对话框。但是，在调试应用程序时，异常就没有那么令人愉快了。它们的问题在于两个方面：

- 如果在调试过程中发生异常，通常不希望自动处理它们，特别是有时自动处理异常意味着应用程序将终止。调试器应能帮助确定为什么会发生异常。当然，如果编写了优良、健壮的防御性代码，程序将能够自动处理几乎任何事情，包括想要检测的 bug！
- 如果发生了某个异常，.NET 运行库就会尝试搜索该异常的处理程序，即使没有为该异常编写处理程序。没有找到异常时，它会终止程序。这时没有了调用栈，意味着所有的变量将超出作用域，所以将无法查看任何变量的值。

当然，可以在 catch 块中设置断点，但是这常常没有多大帮助，因为按照定义，当遇到 catch 块时，执行流已经退出了对应的 try 块。这意味着想要通过检查变量值来确定问题所在时，那些变量已经超出了作用域。甚至不能通过查看栈跟踪来找出在遇到 throw 语句时执行的方法，因为控制流已经离开了该方法。在 throw 语句处设置断点显然可以解决这个问题，但是对于防御性编码，代码中将存在许多 throw 语句。如何判断哪条 throw 语句抛出了该异常？

Visual Studio 为这种问题提供了一个很好的解决方法。Debug 菜单项中包含一个 Exceptions 项。单击它将打开 Exceptions 对话框，如图 17-37 所示。在该对话框中可以指定抛出异常后执行什么操作。例如，可以选择继续执行，或者停止执行并启动调试——此时程序将停止执行，调试器将在 throw 语句位置启动。

图 17-37

这个对话框的强大之处在于允许根据所抛出异常的类型选择相应的操作。例如，可以配置为在遇到.NET 基类抛出的任何异常时进入调试器，但是对于其他异常类型则不进入调试器。

Visual Studio 知道.NET 基类中的所有异常类，以及许多可在.NET 环境外抛出的异常。Visual Studio 不会自动知道用户编写的任何自定义异常类，但是可以把自定义异常类手动添加到 Visual Studio 的异常列表中，并指定哪些自定义异常类应该导致执行立即停止。为此，只需要单击 Add 按钮(在列表中选择一个顶层节点时，将启用该按钮)，并输入自定义异常类的名称即可。

17.5.5　多线程

Visual Studio 为调试多线程程序提供了出色的支持。在调试多线程程序时，必须理解在调试器中运行与不在调试器中运行时，程序的行为会发生变化。遇到断点时，Visual Studio 会停止程序的所有线程，所以此时有机会查看所有线程的当前状态。为了在不同的线程间切换，可以启用 Debug Location 工具栏。这个工具栏有一个针对所有进程的组合框，还有另外一个组合框，用于当前运行的应用程序的所有线程。选择一个不同的线程时，可以看到该线程在哪一行代码暂停，以及当前可在其他线程中访问的变量。Parallel Tasks 窗口(如图 17-38 所示)显示了所有正在运行的任务，包括这些任务的状态、位置、任务名、任务使用的当前线程、所在的应用程序域以及进程标识符。该窗口还显示了不同的线程在什么时候彼此阻塞，导致死锁。

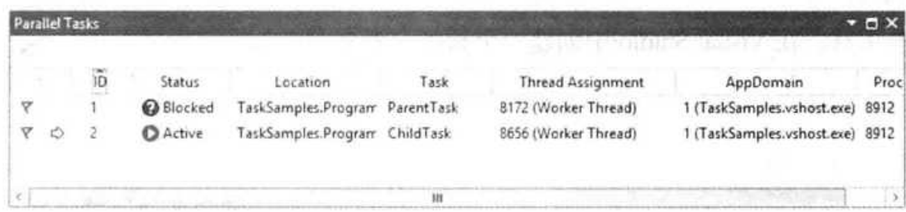


图 17-38

图 17-39 显示了 Parallel Stacks 窗口，该窗口以分层视图的形式显示了不同的线程或任务(取决于所选项)。单击任务或线程可跳转到对应的源代码。

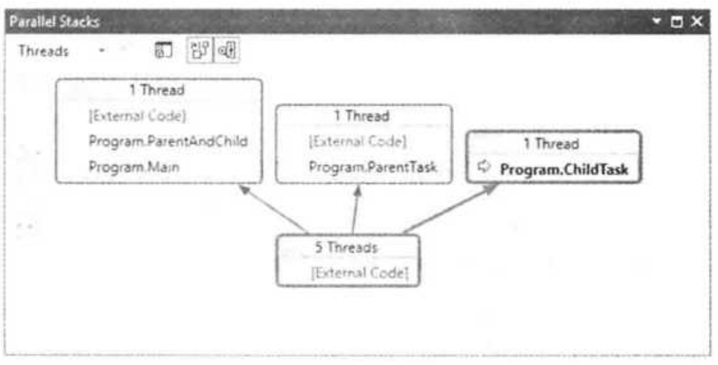


图 17-39

17.5.6　IntelliTrace

IntelliTrace 是另外一个很有帮助的调试功能，但是只有 Visual Studio 2013 Ultimate Edition 提供了该功能。IntelliTrace(也叫做历史调试)提供了历史信息。遇到一个断点后，能够查看以前的信息(如图 17-40 所示)，例如以前的断点、抛出的异常、数据库访问、ASP.NET 事件、跟踪或者用户操作(如单击按钮)。单击以前的事件时，可以查看局部变量、调用栈以及函数调用。使用这种功能时，不需要重启调试器并为发现问题前调用的方法设置断点，就可以轻松地找到问题所在。

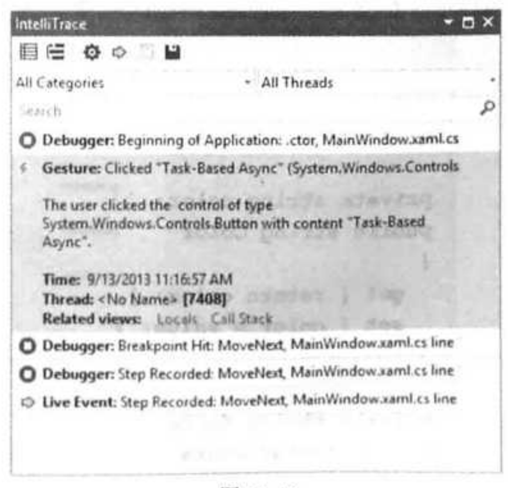


图 17-40

如果调试速度很慢，关闭 IntelliTrace 可能有帮助。

17.6 重构工具

许多开发人员在开发应用程序时首先完成功能，然后修改应用程序，使它们更易于管理和阅读。这个过程称为重构。重构过程包括修改代码来实现更好的性能和可读性，提供类型安全，以及确保应用程序符合标准的面向对象编程实践。更新应用程序时，也需要重构。

Visual Studio 2013 的 C#环境包含一组重构工具，位于 Visual Studio 菜单的 Refactoring 选项中。为了演示这些工具，在 Visual Studio 中创建一个新类，命名为 Car：

```
namespace ConsoleApplication1
{
  public class Car
  {
    public string color;
    public string doors;

    public int Go()
    {
      int speedMph = 100;
      return speedMph;
    }
  }
}
```

现在，假设为了进行重构，需要对代码稍作修改，将变量 color 和 door 封装到公有的.NET 属性中。Visual Studio 2013 的重构功能允许在文档窗口中简单地右击这两个属性，然后选择 Refactor | Encapsulate Field，打开如图 17-41 所示的 Encapsulate Field 对话框。

在该对话框中可以提供属性的名称，然后单击 OK 按钮，将选定的公有字段改为私有字段，并将该字段封装在一个公有的.NET 属性中。在单击 OK 按钮后，代码将修改为如下所示(在修改两个字段后)：

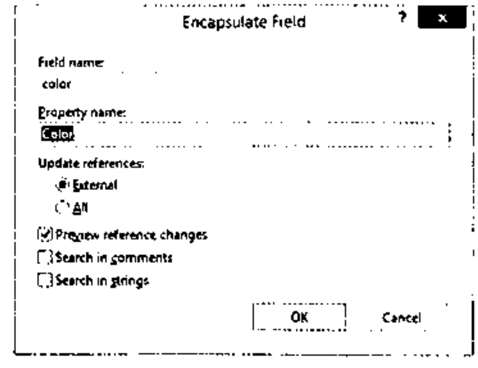


图 17-41

```
namespace ConsoleApplication1
{
  public class Car
  {
    private string color;
    public string Color
    {
      get { return color; }
      set { color = value; }
    }

    private string doors;
    public string Doors
    {
      get { return doors; }
```

```
         set { doors = value; }
       }

       public int Go()
       {
         int speedMph = 100;
         return speedMph;
       }
     }
   }
```

可以看到，使用这些向导重构代码很简单，不仅是一页的代码，重构整个应用程序的代码都一样简单。Visual Studio 的重构工具还提供了以下功能：

- 重命名方法、局部变量、字段等
- 从选定代码中提取方法
- 基于一组已有的类型成员提取接口
- 将局部变量提升为参数
- 重命名参数或修改参数的顺序

Visual Studio 2013 的重构工具是得到更整洁、可读性更好、结构更合理的代码的一种优秀方法。

17.7 体系结构工具

在开始编写程序前，应该从体系结构的角度考虑解决方案，分析需求，然后定义解决方案的体系结构。Visual Studio Ultimate 2013 提供了一些体系结构工具。在 Visual Studio Premium 2013 中也能够查看关系图。

图 17-42 显示了在创建一个建模项目后的 Add New Item 对话框。使用该对话框中的选项可创建 UML 用例图、类图、序列图和活动图。有不少图书专门介绍标准的 UML 关系图，所以这里不讨论它们，而是重点介绍 Microsoft 提供的两种关系图：依赖项关系图(或定向关系图文档)和层关系图。

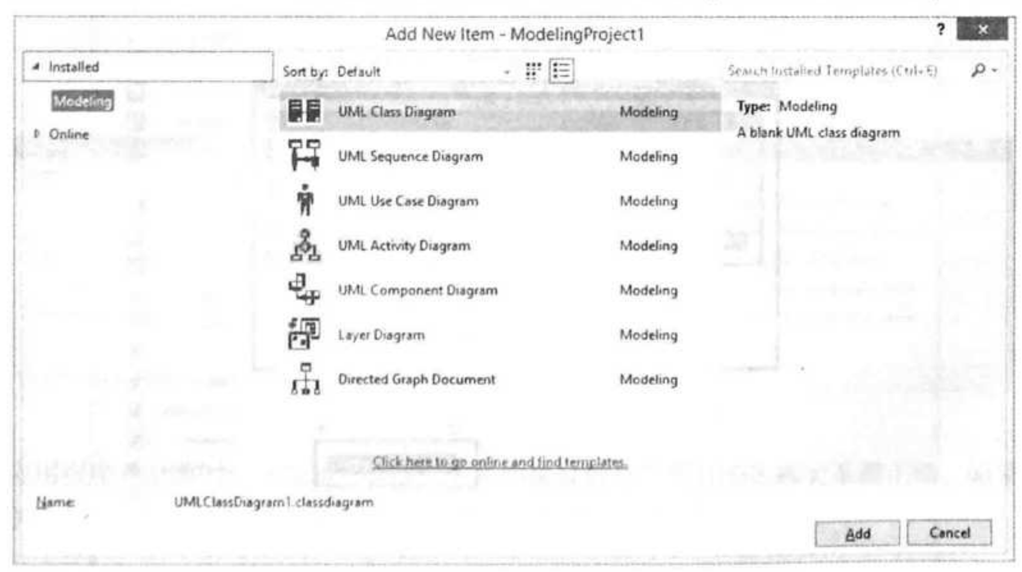


图 17-42

17.7.1 依赖项关系图

在依赖项关系图中，可以查看程序集、类甚至类成员之间的依赖关系。图 17-43 显示了第 30 章中 Calculator 示例的依赖项关系图，该示例包含一个计算器宿主应用程序和几个类库，例如协定程序集以及插件程序集 SimpleCalculator、FuelEconomy 和 TemperatureConversion。通过选择 Architecture | Generate Dependency Graph | For Solution，可以创建依赖项关系图。这会分析解决方案中的所有项目，在一个关系图中显示所有程序集，并在程序集之间绘制连线以显示依赖关系。图 17-43 中移除了外部依赖项，只显示了解决方案内的程序集之间的依赖关系。程序集之间的连线的粗细程度反映了依赖程度。程序集中包含了几个类型和类型的成员，许多类型及其成员在其他程序集中使用。

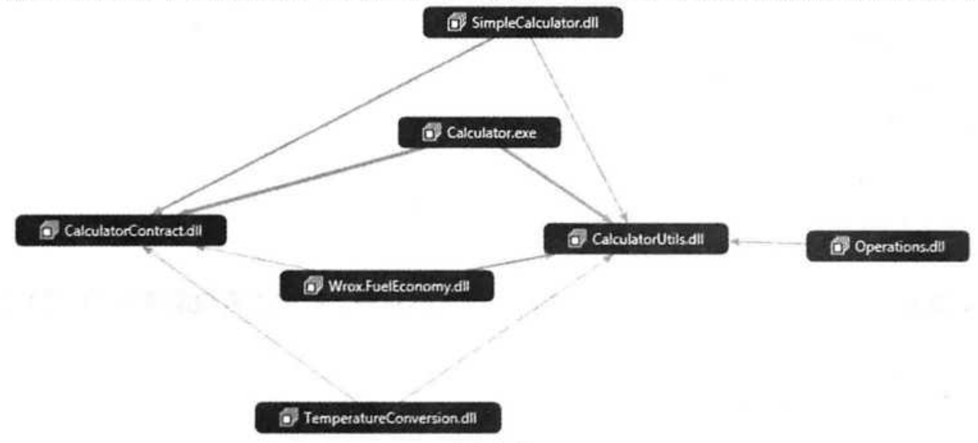


图 17-43

还可以更深入地查看依赖关系。图 17-44 显示了一个更详细的关系图，包括 Calculator 程序集的类及它们的依赖关系。图中还显示了对 CalculatorContract 程序集的依赖。为简单起见，在关系图中移除了其他程序集。在更大的关系图中，还可以缩放关系图的不同部分。

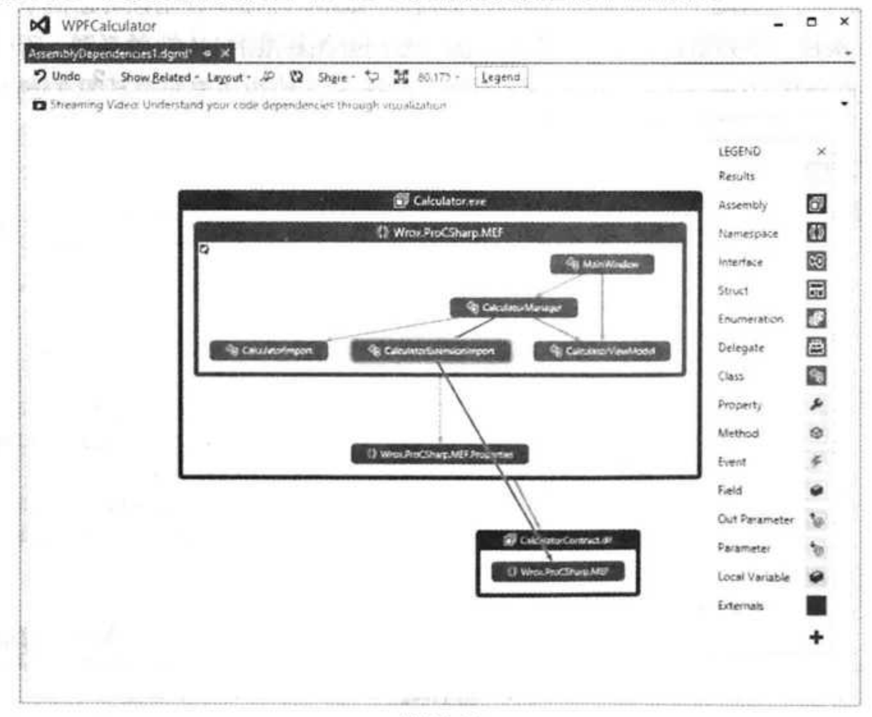


图 17-44

还可以更进一步，显示字段、属性、方法、事件以及它们之间的依赖关系。

17.7.2　层关系图

层关系图与依赖项关系图密切相关。在创建层关系图时，既可以使用依赖项关系图(或者在 Solution Explorer 中选择程序集或类)，也可以在进行任何开发前从头创建。

在分布式解决方案中可以使用不同的层定义客户端和服务器端，例如，将一个层用于 Windows 应用程序，一个层用于服务，一个层用于数据访问库，或者让层基于程序集。层中还可以包含其他层。

在如图 17-45 所示的层关系图中，主层包括 Calculator UI、CalculatorUtils、Contracts 和 AddIns。AddIns 层中又包含 FuelEconomy、TemperatureConversion 和 Calculator 层。层旁边显示的数字反映了链接到该层的项数。

为了创建层关系图，选择 Architecture | New Diagram | Layer Diagram，这会创建一个空关系图。然后，使用工具箱或者 Architecture Explorer，在这个空关系图中添加层。Architecture Explorer 包含一个 Solution View 和一个 Class View，从中可以选择解决方案中的所有项，并把它们添加到层关系图中。构建层关系图只需要选择项并添加到层中。选择一个层，然后单击上下文菜单中的 View Links 可打开如图 17-46 所示的 Layer Explorer，其中显示了选定层包含的所有项。

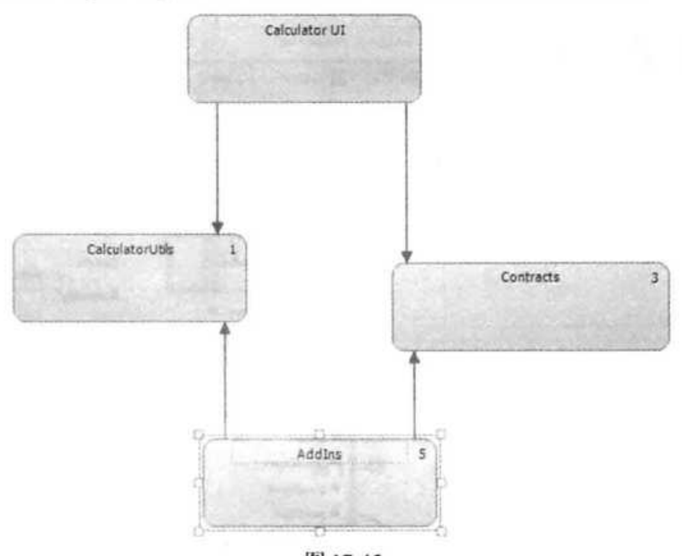


图 17-45

Layer Explorer

Name	Categories	Layer	Supports Validation	Identifier
Wrox.FuelEconomy.dll	Assembly, FileSystem.Category.FileOfType.dll	AddIns	True	(Assembly=Wrox.FuelEconomy)
TemperatureConversion.dll	Assembly, FileSystem.Category.FileOfType.dll	AddIns	True	(Assembly=TemperatureConversion)
SimpleCalculator.dll	Assembly, FileSystem.Category.FileOfType.dll	AddIns	True	(Assembly=SimpleCalculator)
Calculator	Class	AddIns	True	(Assembly=SimpleCalculator Nam...
TemperatureCalculatorEx...	Class	AddIns	True	(Assembly=TemperatureConversio...

Layer Explorer　Properties　Solution Explorer　Team Explorer　Notifications

图 17-46

在应用程序开发期间，可以通过验证层关系图来分析是否所有的依赖关系都正确。如果某个层的依赖关系的方向相反，或者依赖一个错误的层，验证就会返回错误。

17.8　分析应用程序

前一节讨论的体系结构关系图(依赖项关系图和层关系图)并不只是在开始编码前考虑的问题，实际上它们还可以帮助分析应用程序，保持应用程序的开发走在正确的轨道上，确保不会生成错误的依赖关系。Visual Studio 2013 提供了许多有用的工具来帮助分析应用程序，提前解决应用程序中可能发生的问题。本节将讨论其中的一些分析工具。

17.8.1　代码地图

代码地图是一个有助于理解代码的出色的新工具。这个工具会显示方法、属性、字段、变量和事件，以及它们的相互关系，如图 17-47 所示。

为代码中感兴趣的各个部分建立代码地图的简单方法是，运行调试器，单击感兴趣的项，打开上下文菜单 Show on Code Map。这样，代码地图不会填充过多的信息，只显示当前感兴趣的方面。

创建代码地图有不同的方式。除了在运行调试器时创建它之外，还可以在不运行调试器的情况下右击源代码中的元素，来创建代码地图。也可以使用 Solution Explorer，把项添加到代码地图中。

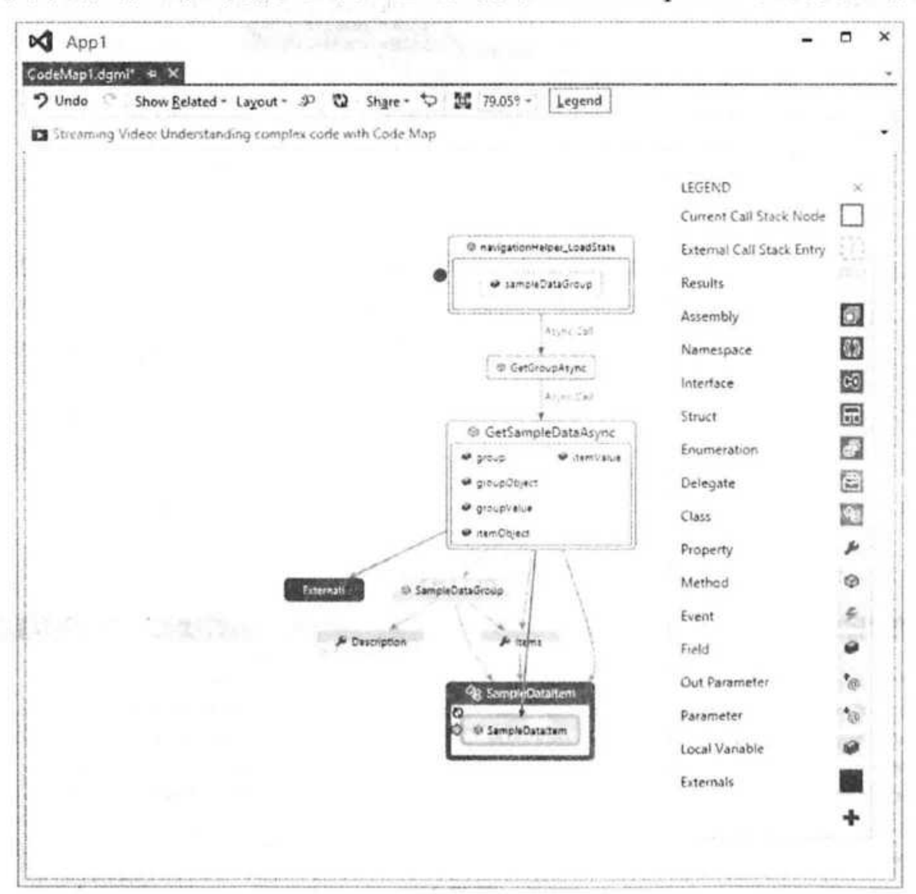


图 17-47

17.8.2　序列图

为了更好地理解一个方法，可以为该方法创建一个序列图。在编辑器内右击方法名，然后从上下文菜单中选择 Generate Sequence Diagram，即可创建该方法的序列图。在创建序列图的对话框中，

可以指定要分析的调用深度；是否要包括来自当前项目、解决方案或解决方案及外部引用的调用；是否排除对属性和 System 对象的调用。

图17-48是第30章的Managed Extensibility Framework示例中创建的WPFCalculator项目的一个序列图。图中显示了OnCalculate方法的序列图。从图中可以看到，OnCalculate()是MainWindow的一个实例方法。首先，检查一个条件，确定currentOperands的长度。只有当这个值为2时，才继续下一步操作，调用CalculatorManager类的InvokeCalculatorAsync()方法。CalculatorManager类调用Task类型的Run()方法，Run()方法启动的延迟调用将调用某个实现了ICalculator接口的对象的Operate()方法。

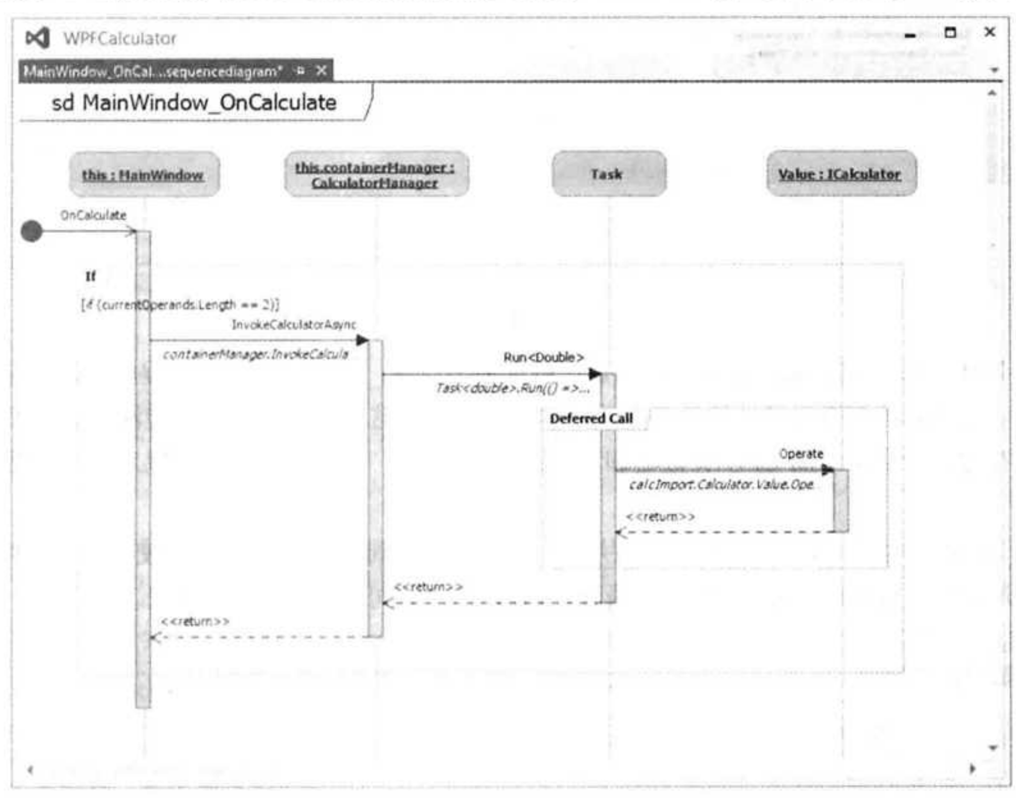


图 17-48

17.8.3　探查器

为了分析应用程序的完整运行，可以使用探查器。探查器是一个性能工具，用于确定方法的调用频率、方法调用所需的时间、使用的内存量等。为了开始使用探查器，一种简单的方法是通过选择 Analyze | Launch Performance Wizard 来打开 Performance Wizard。图 17-49 显示了各种探查方法。第一个选项是 CPU 采样，它的性能开销最小。使用此选项时，每经过固定的时间间隔就对性能信息采样。如果方法调用运行的时间很短，就可能看不到这些方法调用。但是再提一次，这个选项的优势在于性能开销很低。进行探查时总是应该记住，并不只是在监视应用程序的性能，也是在监视数据获取操作的性能。所以不应该同时探查全部数据，因为采样全部数据会影响得到的结果。收集关于.NET 内存分配的信息有助于找出发生内存泄露的地方，以及哪种类型的对象需要多少内存。资源争用数据对分析线程有帮助，能够很容易地看出不同的线程是否会彼此阻塞。

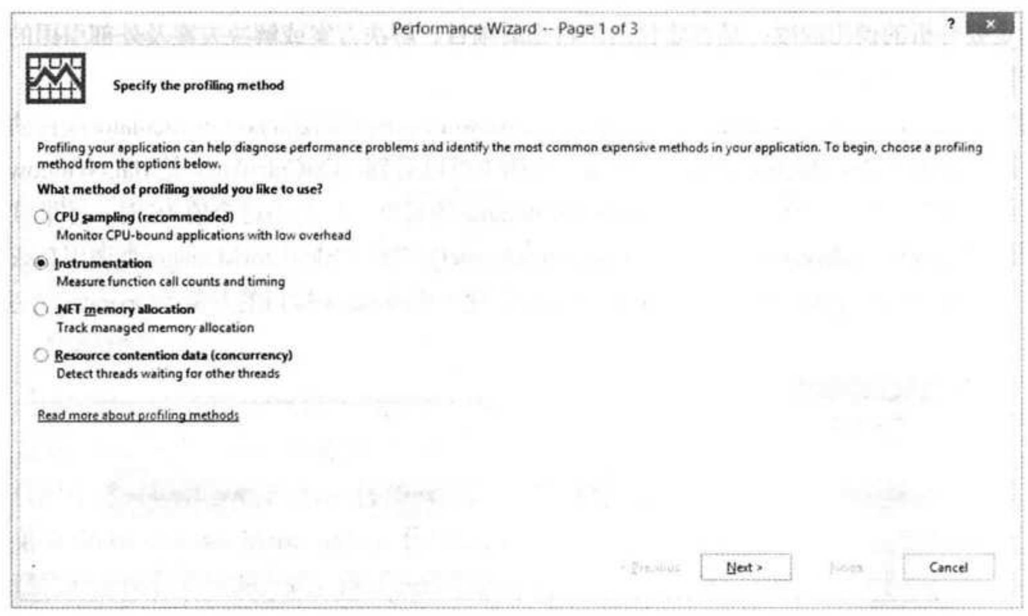


图 17-49

在 Performance Explorer 中配置选项后，可以退出向导，并立即启动应用程序，开始探查。以后还可以通过修改探查设置的属性来更改一些选项。通过这些设置，可以在检测会话中添加内存探查，在探查会话中添加 CPU 计数器和 Windows 计数器，以查看这些信息以及其他一些探查的数据。

图 17-50 显示了一个探查会话的摘要屏幕。从中可以看到应用程序的 CPU 使用率，说明哪些函数占用最长时间的热路径(hot path)，以及使用最多 CPU 时间的函数的排序列表。

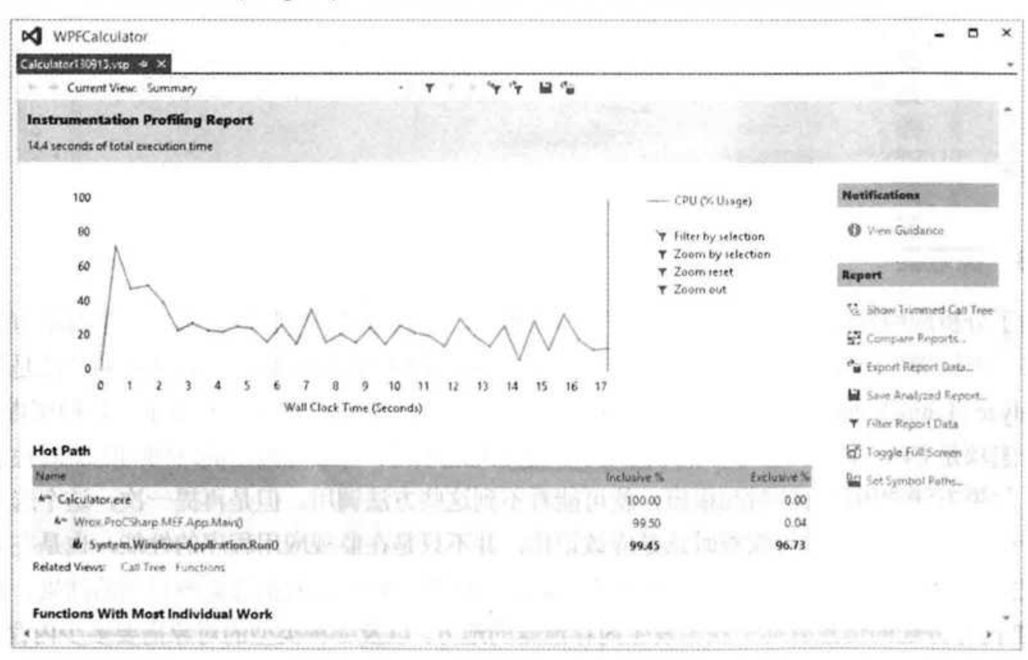


图 17-50

探查器还有许多屏幕，这里无法一一展示。其中有一个函数视图，允许根据函数调用次数进行排序，或者根据函数占用的时间(包括或者不包含函数调用本身)进行排序。这些信息有助于确定哪

些方法的性能值得关注，而其他的方法则可能因为调用得不是很频繁或者不会占用过多时间，所以不必考虑。

在函数内单击，就会显示该函数的详细信息，如图 17-51 所示。这样就可以看到调用了哪些函数，并立即开始单步调试源代码。Caller/Callee 视图也会显示函数的调用关系。

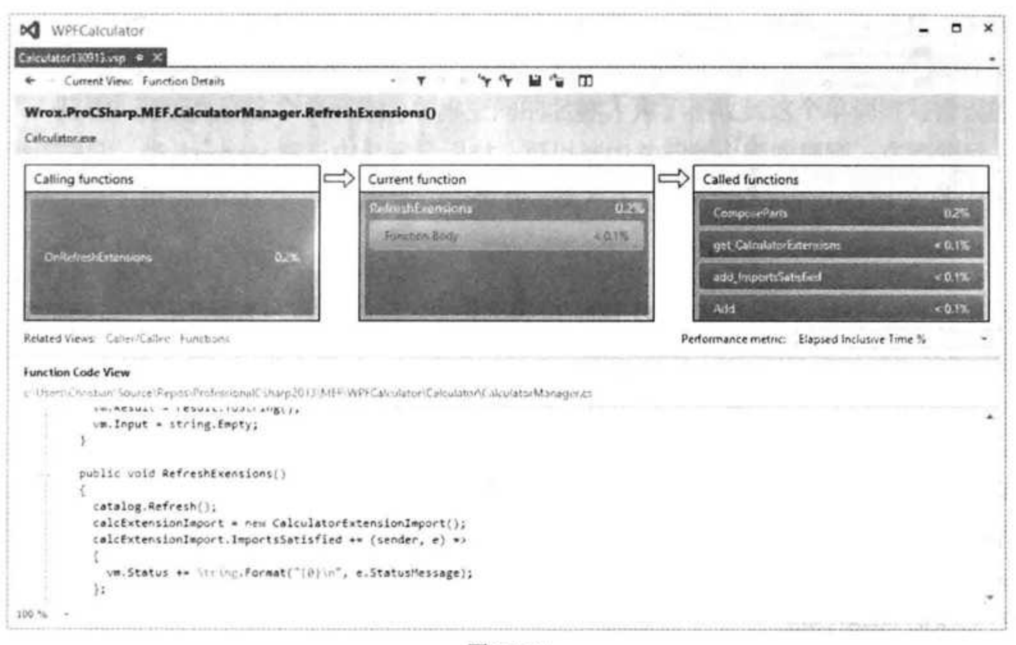


图 17-51

Visual Studio Professional Edition 提供了探查功能。在 Premium Edition 中，可以配置层交互探查，查看生成的 SQL 语句和 ADO.NET 查询花费的时间，以及关于 ASP.NET 页面的信息。

17.8.4 Concurrency Visualizer

Concurrency Visualizer 用于分析应用程序的线程问题。运行此分析工具可得到如图 17-52 所示的摘要屏幕。在该屏幕中，可以比较应用程序需要的 CPU 资源与系统的整体性能。还可以切换到 Threads 视图，查看所有正在运行的应用程序线程及其在各个时间段所处状态的信息。切换到 Cores 视图会显示使用了多少 CPU 核心的信息。如果应用程序只使用了一个 CPU 核心，并且一直处于繁忙状态，那么通过添加一些并行功能，使用更多的 CPU 核心，可能会改进性能。在不同的时间，可能看到不同的线程处于活动状态，但是在给定的时间点，只能有一个线程处于活动状态。在这种情况中，可能需要修改锁定行为。还可以查看线程是否使用了 I/O。如果多个线程的 I/O 使用率都很高，那么磁盘可能是瓶颈，导致线程都在等待彼此完成 I/O。此时，可能需要减少执行 I/O 的线程数，或者使用一个 SSD 磁盘。显然，这些分析工具提供了大量很有帮助的信息。

在 Visual Studio 2013 中，需要通过 Tools | Extensions and Updates 下载并安装 Concurrency Visualizer。

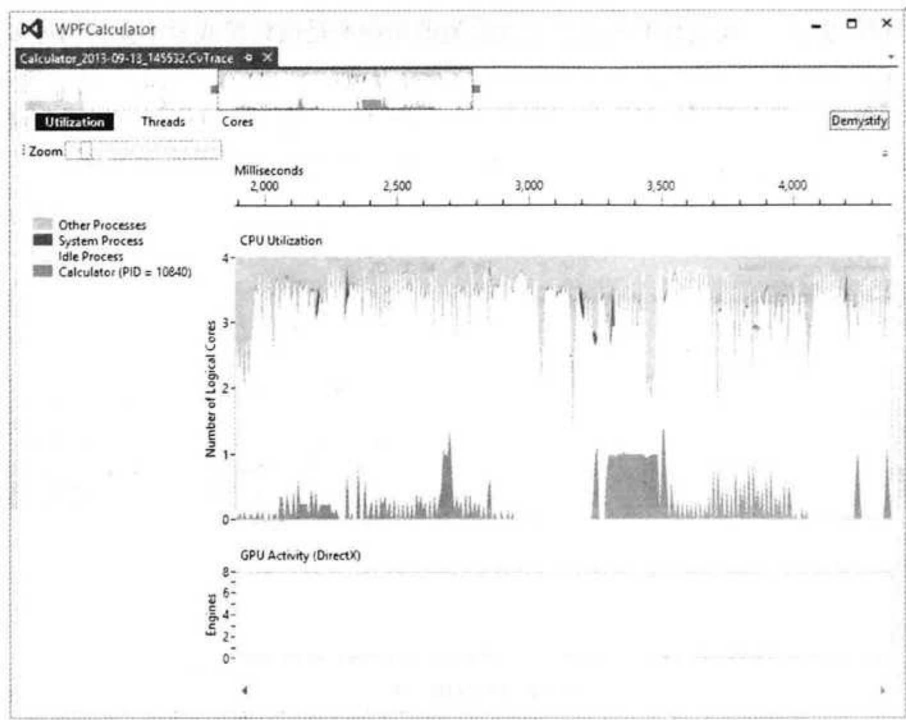


图 17-52

17.8.5 Code Analysis

可以用代码分析规则验证代码。使用 Visual Studio 2013 Professional Edition 可进行静态代码分析。单击项目的属性后，可以看到 Code Analysis 选项卡，在此选项卡中可以选择和编辑一组代码分析规则，这些规则将在生成项目时运行，或者在单独启动 Run Code Analysis 时运行。图 17-53 显示了一个配置的规则集。在规则集中还可以指定该规则将导致警告还是错误。

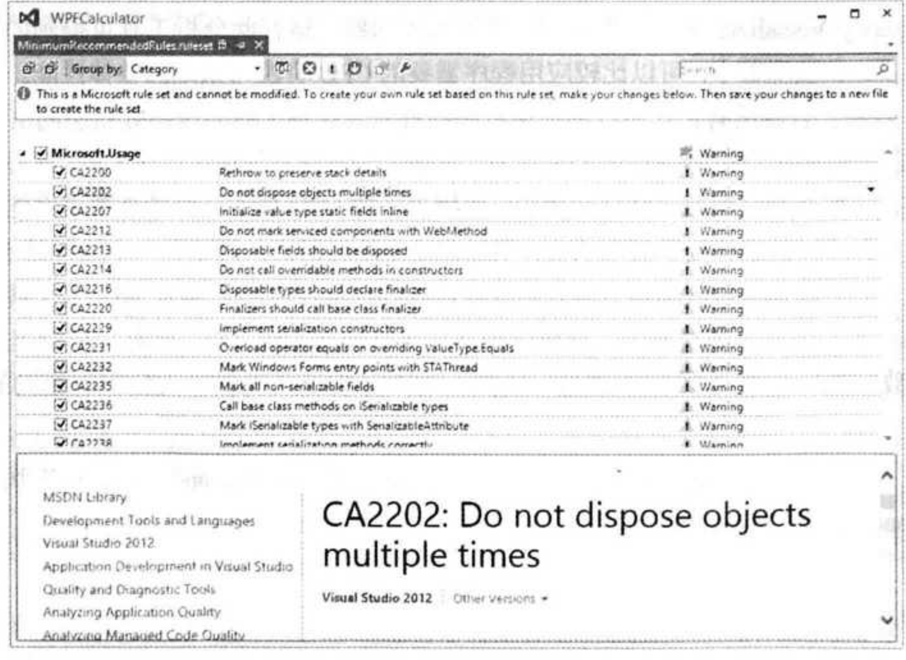


图 17-53

在运行代码分析之前，应该定义要应用的规则。Microsoft 为预定义规则定义了各种规则集，例如 Microsoft Managed Recommended Rules 或 Microsoft Extended Design Guideline Rules。可以创建自己的规则集，或者定义要使用的规则集。在应用规则集时，可能并不同意其中的一些规则，这很正常。可以配置该规则集，排除相应的规则，并且/或者添加适合自己需要的自定义规则。另外，还可以逐个项目抑制规则，或者对适用规则的所有类和方法抑制该规则。例如，假设有一条规则指定 Wrox 的拼写应该与名称空间中使用的形式相同。Visual Studio 使用的拼写检查规则并不包含“Wrox”。但是，应该允许这个单词作为名称空间的名称。为了不收到这个单词拼写错误的消息，可以忽略该规则。当 Analysis 窗口中显示错误时，可以选中并抑制出错的规则。在抑制后，可能发生两种情况：发生错误的标识符会被添加一个特性，或者在 GlobalSuppressions.cs 文件中对应用程序全局抑制该规则：

```
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming",
  "CA1704:IdentifiersShouldBeSpelledCorrectly", MessageId = "Wrox",
  Scope = "namespace", Target = "Wrox.ProCSharp.MEF")]
```

17.8.6　Code Metrics

通过检查代码度量，可以知道代码的可维护程度。图 17-54 中的代码度量显示完整 Calculator 库的可维护程度指数为 82，还包含每个类和方法的细节。这些评级采用颜色编码：红色(0~9)表示可维护程度低；黄色(10~19)表示可维护程度中等；绿色(20~100)表示可维护程度高。Cyclomatic Complexity 列提供了关于不同代码路径的反馈。更多的代码路径意味着需要对每个选项进行更多的单元测试。Depth of Inheritance 列反映了类型的层次。基类数越多，就越难找出某个字段属于哪个基类。Class Coupling 列表明了类型的耦合程度，例如用于参数或局部变量。耦合程度越高，意味着越难维护代码。

图 17-54

17.9　单元测试

编写单元测试有助于代码维护。例如，在更新代码时，想要确信更新不会破坏其他代码。创建自动单元测试可以帮助确保修改代码后，所有功能得以保留。Visual Studio 2013 提供了一个健壮的单元测试框架。

17.9.1　创建单元测试

下面的示例测试了一个非常简单的方法。类 DeepThought 包含 TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything()方法，该方法返回数字 42。为了确保没有人修改该方法，使其返回一个错误的结果，创建如下所示的单元测试：

```
public class DeepThought
{
  public int TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything()
  {
    return 42;
  }
}
```

为了创建单元测试，在 Visual C#项目组中选择 Unit Test Project 模板。单元测试类将带有一个 TestClass 特性，并包含一个标记了 TestMethod 特性的测试方法。单元测试类的实现创建了 DeepThought 实例，并调用了要测试的方法 TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything()。然后，使用 Assert.AreEqual()将该方法的返回值与 42 做比较。如果 Assert.AreEqual()失败，测试就会失败：

```
[TestClass]
public class TestProgram
{
  [TestMethod]
  public void
    TestTheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything()
  {
    int expected = 42;
    DeepThought f1 = new DeepThought();
    int actual =
      f1.TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything();
    Assert.AreEqual(expected, actual);
  }
```

17.9.2 运行单元测试

使用 Test Explorer(通过 Test | Windows | Test Explorer 打开)，可以在解决方案中运行测试(见图 17-55)。

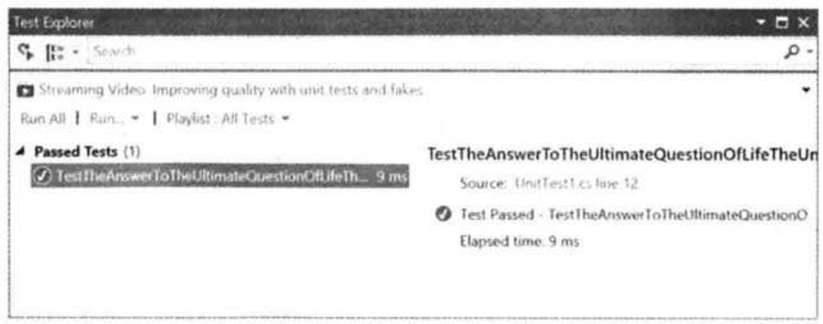


图 17-55

图 17-56 显示了一个失败的测试，列出了失败的所有细节。

当然，这只是一个很简单的场景，测试通常是没有这么简单的。例如，方法可以抛出异常，用其他的路径返回其他值，或者使用了不应该在单个单元中测试的代码(例如数据库访问代码或者调用的服务)。接下来就看一个比较复杂的单元测试场景。

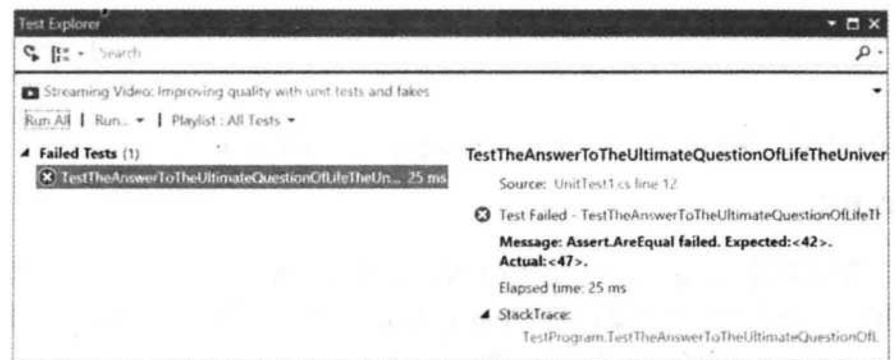


图 17-56

下面代码中类 StringSample 的构造函数有一个字符串参数，并包含方法 GetStringDemo。GetStringDemo 方法根据 first 和 second 参数使用不同的路径，返回不同的结果(可能是类的一个字段成员)：

```
public class StringSample
{
  public StringSample(string init)
  {
    if (init == null)
      throw new ArgumentNullException("init");
    this.init = init;
  }
  private string init;

  public string GetStringDemo(string first, string second)
  {
    if (first == null)
      throw new ArgumentNullException("first");
    if (string.IsNullOrEmpty(first))
      throw new ArgumentException("empty string is not allowed", first);
    if (second == null)
      throw new ArgumentNullException("second");
    if (second.Length > first.Length)
      throw new ArgumentOutOfRangeException("second",
        "must be shorter than first");

    int startIndex = first.IndexOf(second);
    if (startIndex < 0)
    {
      return string.Format("{0} not found in {1}", second, first);
    }
    else if (startIndex < 5)
    {
     return string.Format("removed {0} from {1}: {2}", second, first,
       first.Remove(startIndex, second.Length));
    }
    else
    {
      return init.ToUpperInvariant();
```

```
      }
    }
}
```

单元测试应该测试每个可能的执行路径，并检查异常，如下所述。

17.9.3 预期异常

以 null 为参数调用 StringSample 类的构造函数和 GetStringDemo 方法时，可以预计会发生 ArgumentNullException 异常。在测试代码中很容易测试这一点，只需要像下面的示例那样对测试方法应用 ExpectedException 特性。这样一来，测试方法将成功地捕捉到异常：

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void TestStringSampleNull()
{
   StringSample sample = new StringSample(null);
}
```

对于 GetStringDemo 方法抛出的异常，可以采取类似的处理。

17.9.4 测试全部代码路径

为了测试全部代码路径，可以创建多个测试，每个测试针对一条代码路径。下面的测试示例将字符串 a 和 b 传递给 GetStringDemo 方法。因为第二个字符串没有包含在第一个字符串内，所以 if 语句的第一个路径生效。结果将被相应地检查：

```
[TestMethod]
public void GetStringDemoAB()
{
  string expected = "b not found in a";
  StringSample sample = new StringSample(String.Empty);
  string actual = sample.GetStringDemo("a", "b");
  Assert.AreEqual(expected, actual);
}
```

下一个测试方法测试 GetStringDemo 方法的另一个路径。在这个示例中，第二个字符串包含在第一个字符串内，并且索引小于 5，所以将执行 if 语句的第二个代码块：

```
[TestMethod]
public void GetStringDemoABCDBC()
{
  string expected = "removed bc from abcd: ad";
  StringSample sample = new StringSample(String.Empty);
  string actual = sample.GetStringDemo("abcd", "bc");
  Assert.AreEqual(expected, actual);
}
```

其他所有代码路径都可以以类似的方式测试。为了查看单元测试覆盖了哪些代码，以及还缺少什么代码，可以打开 Code Coverage Results 窗口，如图 17-57 所示。

Code Coverage Results

Christian_SUMMITDAY 2013-09-13 15_34_48

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Christian_SUMMITDAY 2013-09-...	74	61.67 %	46	38.33 %
testdeepthought.dll	12	38.71 %	19	61.29 %
unittestingsample.dll	62	69.66 %	27	30.34 %
UnitTestingSample	62	69.66 %	27	30.34 %
ChampionsLoader	3	100.00 %	0	0.00 %
DeepThought	0	0.00 %	2	100.00 %
Formula1	44	93.62 %	3	6.38 %
Formula1.<>c_Displ...	5	100.00 %	0	0.00 %
StringSample	10	31.25 %	22	68.75 %

图 17-57

17.9.5 外部依赖

许多方法都依赖于不受应用程序本身控制的某些功能，例如调用 Web 服务或者访问数据库。在测试外部资源的可用性时，可能服务或数据库并不可用。更糟的是，数据库和服务可能在不同的时间返回不同的数据，这就很难与预期的数据进行比较。在单元测试中，必须排除这种情况。

下面的示例依赖于外部的某些功能。方法 ChampionsByCountry()访问一个 Web 服务器上的 XML 文件，该文件以 Firstname、Lastname、Wins 和 Country 元素的形式列出了 F1 世界冠军。这个列表按国家筛选，并使用 Wins 元素的值按数字顺序排序。返回的数据是一个 XElement，其中包含了转换后的 XML 代码：

```
public XElement ChampionsByCountry(string country)
{
  XElement champions = XElement.Load(
    "http://www.cninnovation.com/downloads/Racers.xml");
  var q = from r in champions.Elements("Racer")
          where r.Element("Country").Value == country
          orderby int.Parse(r.Element("Wins").Value) descending
          select new XElement("Racer",
            new XAttribute("Name", r.Element("Firstname").Value + " " +
              r.Element("Lastname").Value),
            new XAttribute("Country", r.Element("Country").Value),
            new XAttribute("Wins", r.Element("Wins").Value));
  return new XElement("Racers", q.ToArray());
}
```

关于 LINQ to XML 的更多信息，请参考第 34 章。

应该为这个方法创建一个单元测试。测试不应依赖于服务器上的数据源。一方面，服务器可能不可用。另一方面，服务器上的数据可能随时间发生改变，返回新的冠军和其他值。正确的测试应该确保按预期方式完成筛选，并以正确的顺序返回正确筛选后的列表。

为了创建独立于数据源的单元测试，一种方法是重构 ChampionsByCountry()方法的实现，使用一个返回 XElement 的工厂将 XElement.Load()方法替换为可以独立于数据源的方法。接口 IChampionsLoader 定义了一个可以替换前述方法的 LoadChampions()方法：

```
public interface IChampionsLoader
{
   XElement LoadChampions();
}
```

类 ChampionsLoader 使用 XElement.Load()方法实现了接口 IChampionsLoader：

```
public class ChampionsLoader : IChampionsLoader
{
  public XElement LoadChampions()
  {
    return XElement.Load("http://www.cninnovation.com/downloads/Racers.xml");
  }
}
```

现在就能够修改 ChampionsByCountry()方法的实现，使用接口来加载冠军，而不是直接使用 XElement.Load 方法()。新的方法命名为 ChampionsByCountry2，以便有两个版本可用于单元测试。IChampionsLoader 传递给类 Formula1 的构造函数，然后 ChampionsByCountry2()将使用这个加载器：

```
public class Formula1
{
  private IChampionsLoader loader;
  public Formula1(IChampionsLoader loader)
  {
    this.loader = loader;
  }

  public XElement ChampionsByCountry2(string country)
  {
    var q = from r in loader.LoadChampions().Elements("Racer")
            where r.Element("Country").Value == country
            orderby int.Parse(r.Element("Wins").Value) descending
            select new XElement("Racer",
              new XAttribute("Name", r.Element("Firstname").Value + " " +
                r.Element("Lastname").Value),
              new XAttribute("Country", r.Element("Country").Value),
              new XAttribute("Wins", r.Element("Wins").Value));
    return new XElement("Racers", q.ToArray());
  }
}
```

在典型实现中，会把一个 ChampionsLoader 传递给 Formula1 构造函数，以从服务器检索赛车手。创建单元测试时，可以实现一个自定义方法来返回 F1 冠军，如方法 Formula1SampleData()所示：

```
    internal static string Formula1SampleData()
    {
      return @"
<Racers>
  <Racer>
    <Firstname>Nelson</Firstname>
    <Lastname>Piquet</Lastname>
    <Country>Brazil</Country>
    <Starts>204</Starts>
```

```
    <Wins>23</Wins>
  </Racer>
  <Racer>
    <Firstname>Ayrton</Firstname>
    <Lastname>Senna</Lastname>
    <Country>Brazil</Country>
    <Starts>161</Starts>
    <Wins>41</Wins>
  </Racer>
  <Racer>
    <Firstname>Nigel</Firstname>
    <Lastname>Mansell</Lastname>
    <Country>England</Country>
    <Starts>187</Starts>
    <Wins>31</Wins>
  </Racer>
  //... more sample data
```

为了验证应该返回的结果，使用 Formula1VerificationData()方法创建验证数据，用样本数据匹配请求：

```
    internal static XElement Formula1VerificationData()
    {
      return XElement.Parse(@"
<Racers>
  <Racer Name=""Mika Hakkinen"" Country=""Finland"" Wins=""20"" />
  <Racer Name=""Kimi Raikkonen"" Country=""Finland"" Wins=""18"" />
</Racers>");
    }
```

测试数据的加载器实现了与 ChampionsLoader 类相同的接口：IChampionsLoader。这个加载器仅使用样本数据，而不访问 Web 服务器：

```
public class F1TestLoader : IChampionsLoader
{
  public XElement LoadChampions()
  {
    return XElement.Parse(Formula1SampleData());
  }
}
```

现在，很容易创建一个使用样本数据的单元测试：

```
[TestMethod]
public void TestChampionsByCountry2()
{

  Formula1 f1 = new Formula1(new F1TestLoader());
  XElement actual = f1.ChampionsByCountry2("Finland");

  Assert.AreEqual(Formula1VerificationData().ToString(),
    actual.ToString());
}
```

当然，真正的测试不应该只覆盖传递 Finland 作为一个字符串并在测试数据中返回两个冠军这样一种情况。还应该针对其他情况编写测试，例如传递没有匹配结果的字符串，返回两个以上的冠军的情况，可能还包括数字排序顺序与字母数字排序顺序不同的情况。

17.9.6 Fakes Framework

有时无法重构要测试的方法，使其独立于数据源。这时，Fakes Framework 能够提供很大的帮助。Fakes Framework 是 Visual Studio Ultimate Edition 提供的一个框架。

ChampionsByCountry()方法像前面一样进行测试。其实现使用 XElement.Load()直接访问 Web 服务器上的一个文件。Fakes Framework 允许将 XElement.Load()方法改为其他方法，从而只针对测试用例修改 ChampionsByCountry()的实现：

```
public XElement ChampionsByCountry(string country)
{
  XElement champions = XElement.Load(
    "http://www.cninnovation.com/downloads/Racers.xml");
  var q = from r in champions.Elements("Racer")
          where r.Element("Country").Value == country
          orderby int.Parse(r.Element("Wins").Value) descending
          select new XElement("Racer",
            new XAttribute("Name", r.Element("Firstname").Value + " " +
              r.Element("Lastname").Value),
            new XAttribute("Country", r.Element("Country").Value),
            new XAttribute("Wins", r.Element("Wins").Value));
  return new XElement("Racers", q.ToArray());
}
```

为了在单元测试项目的引用中使用 Fakes Framework，选择包括 XElement 类的程序集。XElement 类在 System.Xml.Linq 程序集中。选择 System.Xml.Linq 程序集，打开上下文菜单，可看到 Add Fakes Assembly 菜单项。选择该菜单项将创建 System.Xml.Linq.4.0.0.0.Fakes 程序集，它在 System.Xml.Linq.Fakes 名称空间中包含一些填充码类。System.Xml.Linq 程序集中的所有类型在这个名称空间中都有填充码版本，例如，XAttribute 有 ShimXAttribute，XDocument 有 ShimXDocument。本例中只需要使用 ShimXElement。对于 XElement 类中的每个公有的重载成员，ShimXElement 中都包含对应的成员。ShimXElement 重载了 XElement 的 Load()方法，使其可以接收 String、Stream、TextReader 和 XMLReader 类型作为参数，另外还提供了可接受第二个参数 LoadOptions 的重载版本。具体来说，ShimXElement 定义了 LoadString()、LoadStream()、LoadTextReader()、LoadXmlReader()成员方法，以及还可以接受 LoadOptions 的成员方法，如 LoadStringLoadOptions()、LoadStreamLoadOptions()等。所有这些成员都是委托类型，该委托类型允许指定一个自定义方法，在测试方法时，将用自定义方法代替原方法调用。单元测试方法 TestChampionsByCountry()将 Formula1.ChampionsByCountry 方法中带有一个参数的 XElement.Load()方法替换为 XElement.Parse()，以访问样本数据。ShimXElement.LoadString 指定了新方法的实现。使用填充码版本时，需要使用 ShimsContext.Create 方法创建一个上下文。该上下文一直处于活动状态，直到在 using 代码块的末尾调用 Dispose()方法：

```
[TestMethod]
public void TestChampionsByCountry()
```

```
{
  using (ShimsContext.Create())
  {
    ShimXElement.LoadString = s => XElement.Parse(Formula1SampleData());

    Formula1 f1 = new Formula1();
    XElement actual = f1.ChampionsByCountry("Finland");

    Assert.AreEqual(Formula1VerificationData().ToString(),
      actual.ToString());
  }
}
```

虽然最好的方法是让要测试的代码具有灵活的实现，但在测试代码时，可以将 Fakes Framework 作为修改代码实现的一种有用方式，使其不依赖于外部资源。

17.10 Windows Store 应用程序、WCF、WF 等

本节将讨论一些具体的应用程序类型。前面介绍了控制台应用程序和 WPF 应用程序，现在就看看 WCF、WF 和 Windows Store 应用程序。Windows Store 应用程序是 Visual Studio 2012 中新增的类型，要在 Windows 8.1 上创建 Windows Store 应用程序，需要 Visual Studio 2013。使用 Visual Studio 2013 还可以维护已有的 Windows 8 应用程序，但不能创建新的 Windows 8 应用程序。要创建 Windows 8 应用程序，需要 Visual Studio 2012。

17.10.1 使用 Visual Studio 生成 WCF 应用程序

WCF 服务库是一个项目模板，用于创建在客户端应用程序中调用的服务。在使用这样的服务时，客户端请求需要在 HTTP、TCP 或其他网络协议上使用 SOAP 协议，或者使用 REST 风格的通信。

WCF 服务应用程序模板会自动创建服务协定、操作协定、数据协定和一个服务实现文件，用户只需要提供一个很小的样本实现。

运行 WCF 服务应用程序将启动一个服务器和一个客户端应用程序来测试该服务。服务器应用程序的对话框如图 17-58 所示。如果主机由于某种原因未能启动，可在 Windows 通知区域打开此对话框，了解失败的原因。如果不应该启动主机，则可在项目属性的 WCF 选项中禁用主机。

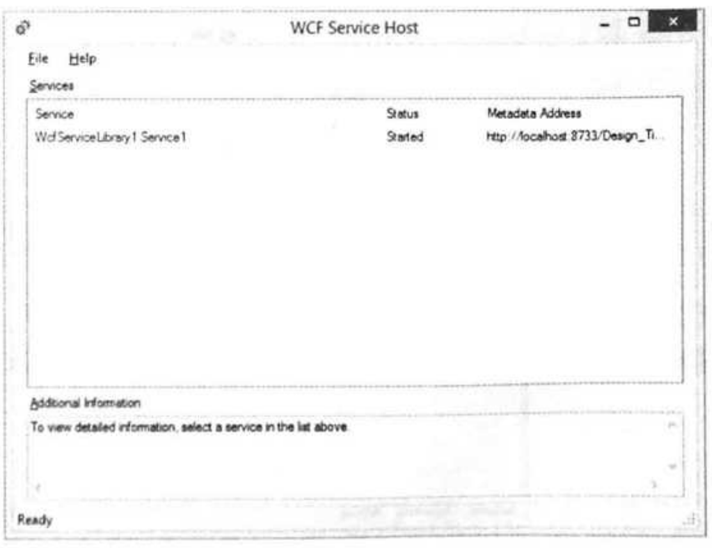


图 17-58

由于使用了调试命令行参数设置/client:"WcfTestClient.exe"，WCF 测试客户端也会启动，如图 17-59 所示。使用此对话框，可以进行多种不同的服务调用(并不是所有的调用都支持)。该对话框既使得测试变得简单，又提供了关于所发送的

SOAP 消息的信息。

图 17-59

第 43 章将详细讨论 WCF 应用程序。

17.10.2　使用 Visual Studio 生成 WF 应用程序

在 Visual Studio 中生成应用程序时，Windows Workflow 应用程序类型也与其他应用程序类型有显著区别。为了进行演示，在 New Project 对话框的 Workflow 部分选择 Workflow Console Application 项目类型。这将创建一个控制台应用程序，其中包括一个 Workflow1.xaml 文件。

当生成使用 Windows Workflow Foundation 的应用程序时，可以注意到它们严重依赖于设计视图。使用设计器时，可以创建变量，以及从工具箱中拖动许多不同的活动到设计视图上。仔细查看工作流(见图 17-60)，可以发现它包含一个 while 循环、一个序列以及基于条件的动作(例如 if-else 语句)。

图 17-60

第 45 章将详细讨论 Windows Workflow Foundation。

17.10.3 使用 Visual Studio 2013 生成 Windows Store 应用程序

如果 Visual Studio 2013 安装在 Windows 8.1 上，就可以创建 Windows Store 应用。新的 Hub 应用程序(XAML)模板已经包含填充了示例数据的 3 个页面。使用这个模板时，在 Solution Explorer 中会看到几个文件。Assets 文件夹包含一些预定义的图标。Common 文件夹包含一些辅助类，如与命令一起使用的 RelayCommand、转换器、挂起管理器，管理生命周期的 NavigationHelper。DataModel 文件夹包含一些可生成样本数据的类，以及包含代码隐藏的一些 XAML 页面。单击 Package.appxmanifest 文件可打开该包的 Manifest Editor，如图 17-61 所示。这个编辑器专门用于 Windows Store 应用程序，可用来配置 UI 以定义应用程序的名称和磁贴，配置应用程序的功能和声明，以及配置应用程序的打包方式。

图 17-61

如图 17-62 所示，运行应用程序后，可看到模板已经按照 Windows Store 应用程序的要求定义了格式和样式。显然，与从头创建所有样式相比，使用模板要简单得多。读者可能知道一些使用此项目模板创建的 Windows Store 应用程序。

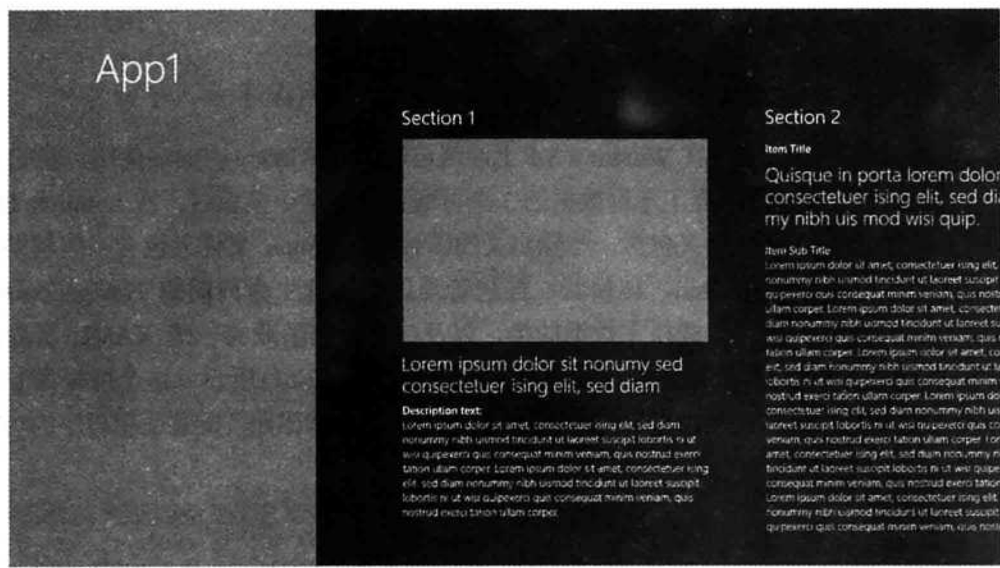


图 17-62

第 31、38 和 39 章将详细讨论 Windows Store 应用程序。

17.11 小结

本章探讨了.NET 环境中最重要的编程工具之一：Visual Studio 2013。大部分内容都用在讲解这个工具如何简化 C#编程。

Visual Studio 2013 是最便于编程的开发环境之一。它不只方便了开发人员实现快速应用程序开发(RAD)，还使得开发人员能够深入探索应用程序的创建机制。本章主要关注如何使用 Visual Studio 进行重构、生成多个版本、分析现有代码、创建单元测试以及使用 Fakes Framework。

本章还了解了.NET Framework 4.5.1 提供的最新项目类型，包括 Windows Presentation Foundation、Windows Communication Foundation、Windows Workflow Foundation 和 Windows Store 应用程序。

第 18 章将介绍应用程序的部署。

第18章

部　　署

本章要点

- 部署要求
- 部署场景
- 使用 ClickOnce 进行部署
- Web 应用程序的部署
- Windows Store 应用程序的部署

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- WPFSampleApp
- WebSampleApp
- WinStoreSplitApp
- Win81PackageSample

18.1　部署是应用程序生命周期的一部分

编译源代码并完成测试后，开发过程并没有结束。在这个阶段，需要把应用程序提供给用户。无论是 ASP.NET 应用程序、WPF 客户端应用程序，还是 Windows Store 应用程序，软件都必须部署到目标环境中。

应该在应用程序设计的早期阶段考虑部署，因为它会影响到应用程序本身使用的技术。

.NET Framework 使部署工作比以前容易得多，因为不再需要注册 COM 组件，也不需要编写新的注册表配置单元。

本章将介绍可用于应用程序部署的选项，包括 ASP.NET 应用程序和富客户端应用程序(包括 Windows Store 应用程序)的部署选项。

18.2 部署的规划

部署常常是开发过程之后的工作，如果不精心规划，就可能会导致严重的问题。为了避免在部署过程中出错，应在最初的设计阶段就对部署过程进行规划。任何部署问题，如服务器的容量、桌面的安全性或从哪里加载程序集等，都应从一开始就纳入设计，这样部署过程才会比较顺利。

另一个必须在开发过程早期解决的问题是，在什么环境下测试部署。应用程序代码的单元测试和部署选项的测试可以在开发人员的系统中进行，而部署必须在类似于目标系统的环境中测试。这一点非常重要，可以消除目标计算机上不存在的依赖项。例如，第三方的库很早就安装在项目开发人员的计算机上，但目标计算机可能没有安装这个库。在部署软件包中很容易忘记包含这个库。在开发人员的系统上进行的测试不可能发现这个错误，因为库已经存在。归档依赖关系可以帮助消除这种潜在的错误。

部署过程对于大型应用程序可能非常复杂。提前规划部署，在实现部署过程时可以节省时间和精力。

选择合适的部署选项，必须像开发系统的其他方面那样给予特别关注和认真规划。选择错误的选项会使把软件交付给用户的过程充满艰难险阻。

18.2.1 部署选项

本节概述.NET 开发人员可以使用的部署选项。其中大多数选项将在本章后面详细论述。

- xcopy——xcopy 实用工具允许把一个程序集或一组程序集复制到应用程序文件夹中，从而减少了开发时间。由于程序集是自我包含的(即描述程序集的元数据包含在程序集中)，因此不需要在注册表中注册。

  每个程序集都跟踪它需要执行的其他程序集。默认情况下，程序集会在当前的应用程序文件夹中查找依赖项。把程序集移动到其他文件夹的过程将在本章后面讨论。
- ClickOnce——ClickOnce 技术可以构建自动更新的、基于 Windows 的应用程序。ClickOnce 允许把应用程序发布到网站、文件共享甚或 CD 上。在对应用程序进行更新并生成新版本后，开发小组可以把它们发布到相同的位置或站点上。最终用户在使用应用程序时，程序会检查是否有更新版本。如果有，就进行更新。
- Windows Installer——ClickOnce 有一些限制，在某些场合中不能使用。如果安装要求管理员权限(例如部署 Windows 服务)，Windows Installer 就是最佳选项。
- 部署 Web 应用程序——当部署一个网站时，会用 IIS 创建一个虚拟站点，并把运行应用程序所需的文件复制到该服务器中。使用 Visual Studio 时，有不同的文件复制选项：使用 FTP 协议，访问网络共享，或者使用前些年常用的选项，即 FrontPage Server Extensions(FPSE)。本章后面将讨论一种更新的技术：创建 Web 部署包。
- Windows Store 应用程序——这些应用程序可以从 Windows Store 部署，也可以使用 PowerShell 脚本从企业环境部署。本章后面将介绍如何创建 Windows Store 应用程序的包。

18.2.2 部署要求

最好看一下基于.NET 的应用程序的运行要求。在执行任何托管的应用程序之前，CLR 对目标

平台都有一定的要求。

首先必须满足的要求是操作系统。目前，下面的操作系统可以运行基于.NET 4.5 的应用程序：

- Windows Vista SP2
- Windows 7
- Windows 8 (已包含.NET 4.5)
- Windows 8.1 (已包含.NET 4.5.1)

下面的服务器平台也支持运行基于.NET 4.5 的应用程序：

- Windows Server 2008 SP2
- Windows Server 2008 R2
- Windows Server 2012(已包含.NET 4.5)
- Windows Server 2012 R2(已包含.NET 4.5.1)

用 Visual Studio 2012 创建的 Windows Store 应用程序运行在 Windows 8 和 8.1 上。用 Visual Studio 2013 创建的 Windows Store 应用程序运行在 Windows 8.1 上。

在部署.NET 应用程序时，还必须考虑硬件要求。硬件的最低要求是：客户端和服务器都有 1GHz 的 CPU，以及 512MB 的 RAM。

要获得最佳性能，应增加 RAM：RAM 越大，.NET 应用程序运行得就越好。对于服务器应用程序更是如此。可以使用性能监视器来分析应用程序的 RAM 使用情况。

18.2.3 部署.NET 运行库

使用.NET 开发应用程序时，需要依赖.NET 运行库。这似乎很明显，但有时可以忽略这一点。表 18-1 列出了必须发布的版本号和文件名。Windows 8.1 和 Windows Server 2012 R2 已经包含.NET 4.5.1。

表 18-1

.NET 版本	文件名
2.0.50727.42	dotnetfx.exe
3.0.4506.30	dotnetfx3.exe (包括 x86 和 x64)
3.5.21022.8	dotnetfx35.exe (包括 x86、x64 和 ia64)
4.0.0.0	dotnetfx40.exe (包括 x86、x64 和 ia64)
4.5.50501	dotnetFx45.exe (包括 x86 和 x64)
4.5.51641	dotnetFx451. exe(包括 x86 和 x64)

18.3 传统的部署选项

如果在应用程序的初始设计阶段考虑了部署，部署就只是简单地把一组文件复制到目标计算机上。本节就讨论这种简单的部署情况和不同的部署选项。

为了了解如何设置部署选项，必须有一个要部署的应用程序。我们使用了 ClientWPF 解决方案，它需要 AppSupport 库。

ClientWPF 是使用 WPF 的富客户端应用程序。AppSupport 项目是一个类库，它包含一个简单的

类，该类返回一个包含当前日期和时间的字符串。

示例应用程序使用AppSupport项目，用一个包含当前日期的字符串填写一个标签。为了使用这些示例，首先加载并构建AppSupport项目。然后在ClientWPF项目中设置对新构建的AppSupport.dll的引用。

下面是AppSupport程序集的代码：

```
using System;

namespace AppSupport
{
  public class DateService
  {
    public string GetLongDateInfoString()
    {
      return string.Format("Today's date is {0:D}", DateTime.Today);
    }

    public string GetShortDateInfoString()
    {
      return string.Format("Today's date is {0:d}", DateTime.Today);
    }
  }
}
```

这个简单的程序集足以说明可用的部署选项。

18.3.1 xcopy部署

xcopy部署过程就是把一组文件复制到目标计算机上的一个文件夹中，然后在客户端上执行应用程序。这个术语来自于DOS命令xcopy.exe。无论程序集的数目是多少，如果把文件复制到同一个文件夹中，应用程序就会执行，不需要编辑配置设置或注册表。

为了理解xcopy部署的工作原理，执行下面的步骤：

(1) 打开示例下载文件中的ClientWPF解决方案(ClientWPF.sln)。

(2) 把目标改为Release，进行完整的编译。

(3) 使用File Explorer导航到项目文件夹\ClientWPF\bin\Release，双击ClientWPF.exe，运行应用程序。

(4) 单击对应的按钮，会看到当前日期显示在两个文本框中。这将验证应用程序是否能正常运行。当然，这个文件夹是Visual Studio放置输出的地方，所以应用程序能正常工作。

(5) 新建一个文件夹，命名为ClientWPFTest。把这两个程序集(AppSupport.dll和ClientWPFTest.exe)从Release文件夹复制到这个新文件夹中，然后删除Release文件夹。再次双击ClientWPF.exe文件，验证它是否正常工作。

这就是需要完成的所有工作：xcopy部署只需要把程序集复制到目标计算机上，就可以部署功能完善的应用程序。这里使用的示例非常简单，但这并不意味着这个过程对较复杂的应用程序无效。实际上，使用这种方法对可以部署的程序集的大小或数目没有限制。

不想使用xcopy部署的原因是它不能把程序集放在全局程序集缓存(GAC)中，或者不能在“开始”菜单中添加图标。如果应用程序仍依赖于某种类型的COM库，就不能很容易地注册COM组件。

18.3.2 xcopy 和 Web 应用程序

xcopy 部署也可以用于 Web 应用程序，但文件夹结构有点不同。我们必须建立 Web 应用程序的虚拟目录，并配置合适的用户权限。这个过程通常需要使用 IIS 管理工具来完成。

在建立虚拟目录后，Web 应用程序文件就可以复制到虚拟目录中。复制 Web 应用程序的文件有点困难，需要考虑几个配置文件和页面使用的图像。

18.3.3 Windows Installer

Microsoft 倾向于使用 ClickOnce 技术来安装 Windows 应用程序，稍后将详细讨论这种技术。但是，ClickOnce 有一些局限：ClickOnce 安装不需要管理员权限，应用程序会被安装到用户有权限的目录中。如果系统由多个用户使用，则需要针对所有用户安装应用程序。而且，使用 ClickOnce 技术不能安装共享 COM 组件并在注册表中配置它们，不能在 GAC 中安装程序集，也不能注册 Windows 服务。所有这些任务都需要管理员权限。

关于在 GAC 中安装程序集的详细信息，请阅读第 19 章。

为了执行这些管理员任务，需要创建一个 Windows 安装程序包。安装程序包就是使用了 Windows Installer 技术的 MSI 文件(可以从 setup.exe 启动)。

创建 Windows 安装程序包的功能不再包含在 Visual Studio 中(它曾是 Visual Studio 2010 的一部分)。不过，可以在 Visual Studio 2013 中使用免费的 InstallShield Limited Edition。它提供了一个项目文件，其中包含了下载及向 Flexera Software 注册 InstallShield 的信息。

InstallShield Limited Edition 提供了一个简单的向导，可以根据应用程序信息(名称、网站、版本号)创建安装程序包；设置安装需求(支持哪些操作系统，以及安装过程需要计算机上已经安装哪些软件)；创建应用程序文件及“开始”菜单和桌面上的快捷方式；设置注册表项。还可以选择提示用户同意许可协议。

如果只需要这些功能，不需要在安装过程中显示自定义对话框，那么 InstallShield Limited Edition 就可以作为一个不错的部署解决方案。否则，就需要安装另外一个产品，例如 InstallShield 的完整版本(www.flexerasoftware.com/products/installshield.htm)或者免费的 WiX 工具包(http://wix.codeplex.com)。

本章将详细讨论 ClickOnce、Web Deploy 包和 Windows Store 应用程序的部署。

18.4 ClickOnce

ClickOnce 是一种允许应用程序自动更新的部署技术。应用程序发布到共享文件、网站或 CD 这样的媒介上。之后，ClickOnce 应用程序就可以自动更新，而无需用户的干涉。

ClickOnce 还解决了安全权限问题。一般情况下，要安装应用程序，用户需要有管理员权限。而利用 ClickOnce，没有管理员权限的用户也可以安装和运行应用程序。但是，应用程序将安装到特定用户的目录中。如果有多个用户使用同一个系统，则每个用户都需要安装该应用程序。

18.4.1 ClickOnce 操作

ClickOnce 应用程序有两个基于 XML 的清单文件，其中一个是应用程序的清单，另一个是部署清单。这两个文件描述了部署应用程序所需的所有信息。

应用程序清单包含应用程序的相关信息，例如需要的权限、要包括的程序集和其他依赖项。部署清单包含了应用程序的部署信息，例如应用程序清单的设置和位置。这些清单的完整模式在.NET SDK 文档中给出。

如前所述，ClickOnce 有一些限制。例如，程序集不能添加到 GAC 文件夹中，以及不能在注册表中配置 Windows 服务。在这些情况下，使用 Windows Installer 比较好，但 ClickOnce 也适用于许多应用程序。

18.4.2 发布 ClickOnce 应用程序

ClickOnce 需要知道的全部信息都包含在两个清单文件中。为 ClickOnce 部署发布应用程序的过程就是生成清单，并把文件放在正确的位置。清单文件可以在 Visual Studio 中生成。还可以使用一个命令行工具 mage.exe，它还有一个带 GUI 的版本 mageUI.exe。

在 Visual Studio 2013 中创建清单文件有两种方式。在 Project Properties 对话框的 Publish 选项卡底部有两个按钮，一个是 Publish Wizard 按钮，另一个是 Publish Now 按钮。Publish Wizard 按钮要求回答几个关于应用程序的部署问题，然后生成清单文件，把所有需要的文件复制到部署位置。Publish Now 按钮使用在 Publish 选项卡中设置的值创建清单文件，并把文件复制到部署位置。

为了使用命令行工具 mage.exe，必须传递各个 ClickOnce 属性的值。使用 mage.exe 可以创建和更新清单文件。在命令提示符中输入 **mage.exe-help**，就会显示传递所需值的语法。

mage.exe 的 GUI 版本(mageUI.exe)类似于 Visual Studio 2012 中的 Publish 选项卡。使用 GUI 工具可以创建和更新应用程序清单及部署清单文件。

ClickOnce 应用程序会显示在控制面板中“添加/删除程序”对话框对应的小程序内，这与其他安装的应用程序一样。一个主要区别是用户可以选择卸载应用程序或回滚到以前的版本。ClickOnce 在 ClickOnce 应用程序缓存中保存以前的版本。

现在开始创建一个 ClickOnce 应用程序。在此之前，系统中必须安装了 IIS，并且必须以提升的权限启动 Visual Studio。ClickOnce 安装程序会直接发布到本地 IIS，而本地 IIS 是需要管理员权限的。

在 Visual Studio 中打开 ClientWPF 项目，选择 Project Properties 对话框的 Publish 选项卡，然后单击 Publish Wizard 按钮。第一个屏幕如图 18-1 所示，它要求指定发布位置。这里使用本地 IIS：http://localhost/ProCSharpSample。

下一个屏幕包括一个在“开始”菜单中添加快捷方式的选项，以便使应用程序在在线和离线时都可用。这里保留默认选项。现在就准备好发布了。一个浏览器窗口会打开，用于安装应用程序(如图 18-2 所示)。

在单击 Install 按钮之前，我们先看看向导对 ClickOnce 做了哪些设置。

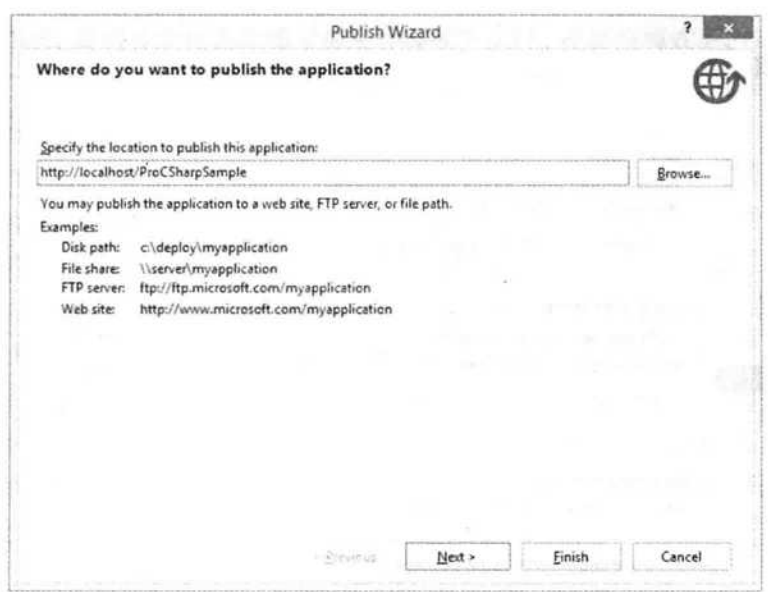


图 18-1

图 18-2

18.4.3　ClickOnce 设置

两个清单文件都有几个属性，可以在 Visual Studio 项目设置内的 Publish 选项卡(如图 18-3 所示)中配置它们的许多属性。最重要的属性是应用程序应从什么地方部署。这里使用了 IIS，但是也可以使用网络共享或者 CD。

Publish 选项卡中有一个 Application Files 按钮，单击它会打开一个对话框，其中列出了应用程序需要的所有程序集和配置文件。单击 Prerequisite 按钮会显示与应用程序一起安装的常见必备程序列表。这个必备程序列表由 Microsoft Installer 包定义，必须在安装 ClickOnce 应用程序前安装。回过头查看图 18-2，会发现.NET Framework 4.5 被列为一个必备条件，只有安装了.NET Framework 4.5 以后才能通过该网页安装 ClientWPF。可以选择从发布应用程序的位置上安装必备程序，也可以从供应商的网站上安装必备程序。

图 18-3

单击 Updates 按钮会显示一个对话框(如图 18-4 所示)，其中包含了如何更新应用程序的信息。当有应用程序的新版本时，可以使用 ClickOnce 更新应用程序。其选项包括：每次启动应用程序时检查是否有更新版本，或在后台检查更新版本。如果选择后台选项，就可以输入两次检查的指定间隔时间。此时可以使用允许用户拒绝或接收更新版本的选项。它可用于在后台进行强制更新，这样用户就不知道进行了更新。下次运行应用程序时，会使用新版本替代旧版本。还可以使用另一个位置存储更新文件。这样，原始安装软件包在一个位置，用于给新用户安装应用程序，而所有的更新版本放在另一个位置上。

图 18-4

安装应用程序时，可以让它以在线模式或离线模式下运行。在离线模式下，应用程序可以从“开始”菜单中运行，就好像它是用 Windows Installer 安装的。在线模式表示应用程序只能在有安装文件夹的情况下运行。

使用 Publish Wizard 时，对项目设置所作的改动要比在 Publish 选项卡中看到的更多。在 Signing 选项卡中，可以看到 ClickOnce 清单已被签名。对于当前部署，创建了一个测试证书。测试证书只能用于测试。在进入生产环境前，需要从某个证书颁发机构获得一个应用程序签名证书，并用该证书签署清单。在 Security 选项卡中，可以看到 ClickOnce 安全设置已被启用，并且默认情况下应用程序被配置为一个完全信任的应用程序。这种配置让应用程序具有与用户相同的权限，可以做用户能够做的所有操作。在安装过程中，会询问用户是否信任应用程序。可以把此配置修改为部分信任的应用程序，以应用较低的 ClickOnce 安全权限。例如，从 Internet 上下载的应用程序只能在一个隔离的存储区内读写，而不能访问整个文件系统。第 22 章将详细介绍.NET 代码访问安全性。

18.4.4 ClickOnce 文件的应用程序缓存

用 ClickOnce 发布的应用程序不能安装在 Program Files 文件夹中，它们会放在应用程序缓存中，应用程序缓存驻留在当前用户的 Documents and Settings 文件夹的 Local Settings 子文件夹下。控制部署的这个方面意味着，可以把应用程序的多个版本同时放在客户端 PC 上。如果应用程序设置为在线运行，就会保留用户访问过的每个版本。对于设置为本地运行的应用程序，会保留当前版本和以前的版本。

所以，把 ClickOnce 应用程序回滚到以前的版本是一个非常简单的过程。如果用户进入控制面板中“添加/删除程序”对话框对应的小程序，则所显示的对话框将允许删除 ClickOnce 应用程序或回滚到以前的版本(如图 18-5 所示)。管理员可以修改清单文件，使之指向以前的版本。之后，下次用户运行应用程序时，会检查是否更新版本。应用程序不查找要部署的新程序集，而是还原以前的版本，但不需要用户的交互。

图 18-5

18.4.5 应用程序的安装

现在，我们从前面看过的浏览器屏幕(见图 18-2)启动应用程序的安装。单击 Install 按钮，会显示如图 18-6 所示的对话框。因为系统不信任测试证书的颁发机构，所以会显示红色标记。单击 More Information 链接，可以获得证书的更多信息，并知道应用程序需要完全信任的访问权限。如果信任

应用程序，就单击 Install 按钮，安装应用程序。

图 18-6

安装完成后，在“开始”菜单和控制面板的“程序和功能”中都可以找到该应用程序。

18.4.6 ClickOnce 部署 API

使用 ClickOnce 设置，可以将应用程序配置为自动检查更新，但是通常这种方法并不可行。可能一些超级用户应该更早得到应用程序的新版本。如果他们对新版本感到满意，那么也应该向其他用户授予接收更新的权限。在这种场景中，可以使用自己的用户管理信息数据库，通过编程更新应用程序。

对于使用编程方式进行的更新，可以使用 System.Deployment 程序集和 System.Deployment 名称空间中的类来检查应用程序版本信息及进行更新。下面的代码片段(代码文件 MainWindow.xaml.cs)包含了应用程序中的 Update 按钮的单击处理程序。它首先检查 ApplicationDeployment 类的 IsNetwork-Deployed 属性，判断应用程序是否是 ClickOnce 部署的应用程序。然后，使用 CheckForUpdateAsync() 方法检查服务器上由 ClickOnce 设置指定的更新目录中是否有新版本可用。收到关于更新的信息后，CheckForUpdateCompleted 事件就会触发。这个事件处理程序的第二个参数(类型为 CheckForUpdate-CompletedEventArgs)包含了关于更新、版本号、是否是强制更新等信息。如果有更新可用，则通过调用 UpdateAsync()方法自动安装更新：

```
private void OnUpdate(object sender, RoutedEventArgs e)
{
  if (ApplicationDeployment.IsNetworkDeployed)
  {
    ApplicationDeployment.CurrentDeployment.CheckForUpdateCompleted +=
      (sender1, e1) =>
      {
        if (e1.UpdateAvailable)
        {
          ApplicationDeployment.CurrentDeployment.UpdateCompleted +=
          (sender2, e2) =>
            {
              MessageBox.Show("Update completed");
            };
```

```
            ApplicationDeployment.CurrentDeployment.UpdateAsync();
          }
          else
          {
            MessageBox.Show("No update available");
          }

        };
      ApplicationDeployment.CurrentDeployment.CheckForUpdateAsync();
    }
  }
```

使用部署 API 代码时，可以在应用程序中直接手动测试更新。

18.5 Web 部署

对于 Web 应用程序，需要部署控制器(MVC)的二进制文件、代码隐藏(Web Forms)以及 HTML、JavaScript 文件、样式表和配置文件。

部署 Web 应用程序最简单的方法是使用 Web Deploy。On-premises IIS 和 Windows Azure 网站都可以使用这种功能。使用 Web Deploy 时，会创建一个可以直接上传到 IIS 的包。这个包是一个压缩文件，其中包含了 Web 应用程序所需的全部信息，包括数据库文件。

18.5.1 Web 应用程序

为了演示 Web Deploy，我们使用 Internet Application 模板创建了一个新的 ASP.NET MVC 5 项目。这会自动创建一个带 Home 和 About 页面的应用程序，包括登录和注册部分，如图 18-7 所示。

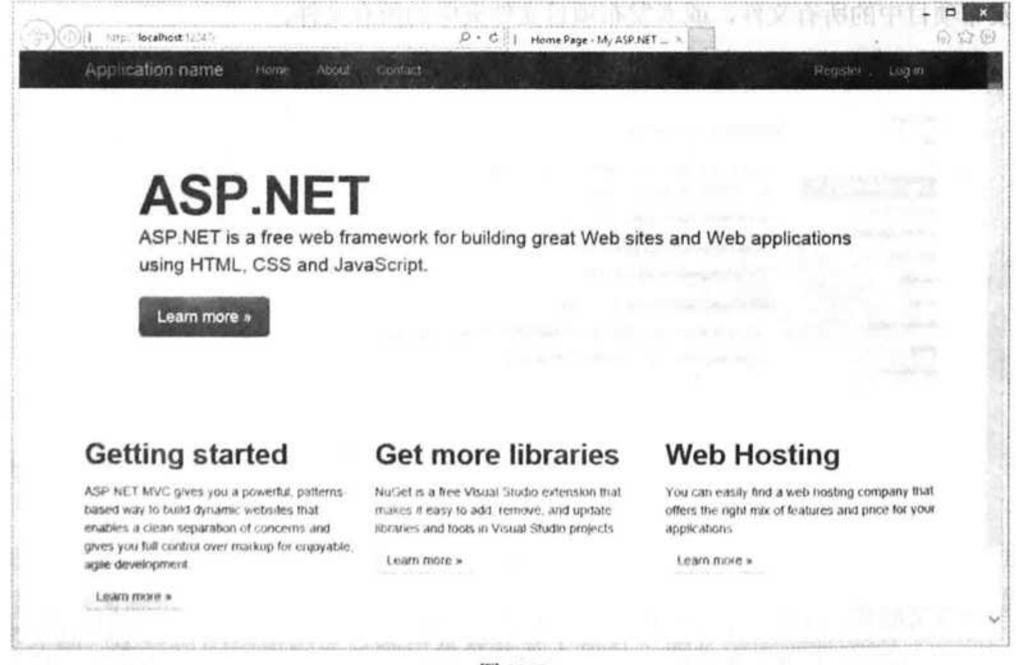


图 18-7

18.5.2 配置文件

配置文件是 Web 应用程序的一个重要部分。在部署时，需要考虑这个文件的不同版本。例如，如果为运行在本地系统上的 Web 应用程序使用一个不同的数据库，那么针对开发用服务器有一个特殊的测试数据库，当然对于生产服务器还有一个实时数据库。这些服务器的连接字符串不同，调试配置也不同。如果为它们创建了各自的 Web.config 文件，然后在本地的 Web.config 文件中添加了一个新的配置值，就可能会忘记修改其他的配置文件。

Visual Studio 提供了一个特殊的功能来处理这种情况。可以创建一个配置文件，然后定义如何针对开发用服务器和部署服务器转换该文件。默认情况下，对于 ASP.NET Web 项目，在 Solution Explorer 中除了 Web.config 文件以外，还会看到 Web.debug.config 和 Web.release.config 文件。这两个文件只包含转换后的版本。还可以添加其他配置文件，例如针对开发用服务器的配置文件。具体方法是，在 Solution Explorer 中选择解决方案，打开 Configuration Manager，然后添加一个新的配置(例如 Staging 配置)即可。一旦有新配置可用，就可以选择 Web.config 文件，然后从上下文菜单中选择 Add Config Transform 选项，这会添加一个以配置命名的配置转换文件，如 Web.Staging.config。

转换配置文件的内容只是定义了从原配置文件的转换，例如修改 system.web 下的编译元素，从中删除 debug 特性，如下所示：

```
<system.web>
<compilation xdt:Transform="RemoveAttributes(debug)" />
```

18.5.3 创建 Web Deploy 包

为了定义 Web 应用程序的部署方式，项目属性对话框提供了 Package/Publish Web 设置(如图 18-8 所示)。在配置中，可以选择仅发布运行应用程序所需的文件，这将排除所有的 C#源文件。其他选项包括发布项目中的所有文件，或者发布项目文件夹中的所有文件。

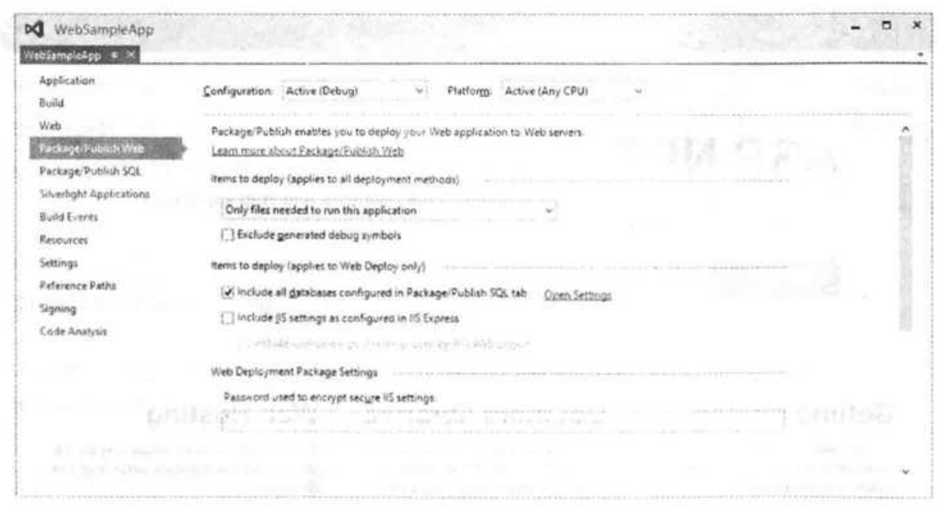


图 18-8

对于要部署的项，可以指定在包中包含在 Package/Publish SQL 选项卡中定义的数据库。在该选项卡中，可以从配置文件导入数据库，以及通过创建 SQL 脚本来创建模式及加载数据。这些脚本可以包含在包中，在目标系统上创建一个数据库。

Package/Publish Web 选项卡中的其他配置选项包括压缩文件的名称和 IIS 应用程序的名称。把包部署到 IIS 时，为包定义的名称就是 IIS 应用程序默认的名称。部署 Web 应用程序的管理员会用一个不同的名称覆盖该名称。

配置完包以后，可以从 Solution Explorer 的上下文菜单中选择 Publish 菜单项来创建一个包。显示的第一个对话框允许创建或选择一个配置文件。配置文件用来把包部署到不同的服务器上，例如，可以定义一个配置文件来部署到开发用服务器，定义另一个配置文件来部署到生产服务器。如果在 Windows Azure 网站上运行自己的站点，那么可以从 Windows Azure 下载一个配置文件并用 Publish Web 工具导入。该配置文件包含服务器的 URL 以及用户名和密码。向导的第二个对话框用于指定发布方式。有效的方式包括创建一个 Web Deploy Package(就是现在正在做的工作)，直接执行 Web Deploy 部署到服务器，以及使用 FTP、文件系统或者 FrontPage Server Extensions。图 18-9 中已经选中了 Web Deploy Package，所以允许定义包的位置和网站的名称。第三个对话框用于指定应该部署到包的配置。如果前面创建了 Staging 配置，那么现在 Debug、Release 和 Staging 配置都将可用。

在完成向导并单击 Publish 按钮后，就创建了 Web Deploy 包。可以打开这个包，查看其中包含的文件。如果 IIS 正在运行，则可以打开 IIS Manager 来部署压缩文件，并创建一个新的 Web 应用程序。

图 18-9

18.6 Windows Store 应用程序

安装 Windows Store 应用程序则完全不同。对于正常的.NET 应用程序，按照前面的介绍使用 xcopy 部署来复制带 DLL 的可执行文件是一种可行的方法。但是，对于 Windows Store 应用程序则不能这么做。未打包的应用程序只能用在有开发人员许可的系统上。

Windows Store 应用程序是需要打包的，这使得应用程序能够放到 Windows Store 中被多数人使用。在部署 Windows Store 应用程序时，还有另外一种方法，不需要把该应用程序添加到 Windows

Store 中。这种方法称为旁加载(Sideleading)。在这些选项中，有必要创建一个应用程序包，所以我们就从创建一个应用程序包开始。

18.6.1 创建应用程序包

Windows Store 应用程序包是一个带有.appx 文件扩展名的文件，实际上是一个压缩文件。该文件包含了所有的 XAML 文件、二进制文件、图片和配置。使用 Visual Studio 或命令行工具 MakeAppx.exe 都可以创建包。

Visual Studio 在 Windows Store 类别下有一个应用程序模板 Split App(XAML)，使用该模板可以创建包含了一些核心功能的简单 Windows Store 应用程序。该模板包含了两个可以导航的页面。所创建的应用程序的名称是 WinStoreSplitApp。

对于打包过程来说，重要的是 Assets 文件夹中的图像。Logo、SmallLogo 和 StoreLogo 文件代表应该用自定义应用程序图标代替的应用程序图标。文件 Package.appxmanifest 是一个 XML 文件，包含了应用程序包所需的所有定义。打开这个文件会调出 Package Editor，它包含 4 个选项卡：Application UI、Capabilities、Declarations 和 Packaging。Packaging 选项卡如图 18-10 所示。在这里可以配置包名、在 Windows Store 中显示的图标、版本号和证书。默认情况下，只创建一个用于测试目的的证书。在部署应用程序前，必须使用由 Windows 信任的某个证书颁发机构提供的证书替换此证书。

图 18-10

Application UI 选项卡允许配置应用程序名称、应用程序描述以及小图标和大图标。可配置的能力随系统功能和应用程序使用的设备而异，例如音乐库、摄像头等。用户会得到应用程序使用了哪些功能的通知。如果应用程序没有指定需要某个功能，在运行期间就不能使用该功能。在 Declarations 选项卡中，应用程序可以注册更多的功能，例如用作共享目标，或者指定某些功能是否应该在后台运行。

在 Visual Studio 中，单击 Solution Explorer 中的项目，然后从上下文菜单中选择 Store | Create App Package，可以创建一个包。在 Create App Package 向导中，首先需要指定是否要把应用程序上传到 Windows Store。如果不必上传，那么可以使用旁加载来部署包，如后面所述。如果还没有注册一个 Windows Store 账户，就选择旁加载选项。在该向导的第二个对话框中，为包选择 Release 而不是 Debug Code。在这里还可以选择为哪个平台生成包：x86、x64 和 ARM CPU。这就是构建包所需的全部工作。要想查看包中的内容，可以把文件的扩展名从.appx 改为.zip，然后就可以看到其中包含的所有图像、元数据和二进制文件。

18.6.2 Windows App Certification Kit

在创建应用程序包时，向导的最后一个对话框启用了 Windows App Certification Kit。该工具的命令行版本是 appcertui.exe。可以使用此命令行并传递包以进行测试。

把应用程序部署到 Windows Store 时，应用程序必须满足一些需求。多数需求是可以提前检查的。

运行此工具时，需要有些耐心。它需要几分钟的时间来测试应用程序并给出结果。在此期间，不应与该工具或正在运行的应用程序交互。表 18-2 显示了该工具对应用程序执行的测试。

表 18-2

测 试	描 述
崩溃和挂起测试	应用程序不能崩溃或停止响应。长时间运行的任务应该异步运行，以免阻塞应用程序
应用程序清单兼容测试	确认应用程序清单的内容正确。另外，应用程序在安装后可能只有一个磁贴。用户可以在配置应用程序时添加额外的磁贴，但最开始只允许有一个磁贴
Windows 安全特性测试	确认应用程序不会在未经用户同意的情况下删除用户的数据，且不会引入病毒或恶意软件
支持的 API 测试	应用程序只能使用 Windows 8.1 API(Windows 运行库和.NET 的一个子集)，而且不能依赖于没有此限制的库。应用程序只能依赖于 Windows Store 中的应用程序
性能测试	应用程序必须在 5 秒钟内启动，挂起时间不能超过 2 秒
应用程序清单资源测试	应用程序必须针对支持的所有语言包含经过本地化的资源，必须替换 Visual Studio 模板为磁贴和商店徽标创建的图像文件

图 18-11 显示了测试运行失败得到的部分结果。没有提供图像文件。

图 18-11

18.6.3 旁加载

为了让应用程序得到最多的用户，应该把应用程序发布到 Windows Store 中。Windows Store 的许可很灵活，可以选择销售给个人的许可，或者批量许可。对于后者，可以根据唯一 ID 和设备确定运行应用程序的用户。

对于企业而言，如果应用程序不适合放到 Windows Store 中，就可以使用旁加载。

旁加载对涉及的系统有一些要求：PC 机需要与活动目录相连，还需要设置一个组策略，允许安装所有可信应用程序。这个组策略添加了一个注册表键 HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\Appx\AllowAllTrustedApps，其键值设为 1。最后一个要求是必须用可信任的证书签署应用程序。这个证书可以是自定义证书，它将证书服务器列为受信任的根证书颁发机构。

Windows 8.1 企业版默认启用了旁加载。对于其他平台，需要购买旁加载的密钥。

在分发给所有客户端系统的原始 Windows 8.1 镜像中，可以为所有的用户预安装定制的应用程序，也可以使用下面的 PowerShell cmdlet 安装这些应用程序：

```
add-appxpackage Package.appx
```

18.6.4 Windows 部署 API

新的 Windows 运行库定义了名称空间 Windows.Management.Deployment。该名称空间中包含的 PackageManager 类可以用来以编程方式部署 Windows 8 包。该类中的 AddPackageAsync 方法在系统中添加一个包，RemovePackageAsync 则删除包。

下面的代码片段(代码文件 Win81PackageSample/Program.cs)演示了 PackageManager 类的用法。PackageManager 类只能在桌面应用程序中使用，所以这里创建了一个.NET 控制台应用程序：

```
using System;
using System.Collections.Generic;
using System.IO;
using Windows.ApplicationModel;
using Windows.Management.Deployment;

namespace Win8PackageSample
{
  class Program
  {
    static void Main()
    {
      var pm = new PackageManager();
      IEnumerable<Package> packages = pm.FindPackages();
      foreach (var package in packages)
      {
        try
        {
          Console.WriteLine("Architecture: {0}",
             package.Id.Architecture.ToString());
          Console.WriteLine("Family: {0}", package.Id.FamilyName);
```

```
                Console.WriteLine("Full name: {0}", package.Id.FullName);
                Console.WriteLine("Name: {0}", package.Id.Name);
                Console.WriteLine("Publisher: {0}", package.Id.Publisher);
                Console.WriteLine("Publisher Id: {0}", package.Id.PublisherId);
                if (package.InstalledLocation != null)
                  Console.WriteLine(package.InstalledLocation.Path);
                Console.WriteLine();
              }
              catch (FileNotFoundException ex)
              {
                Console.WriteLine("{0}, file: {1}", ex.Message, ex.FileName);
              }
            }
            Console.ReadLine();
          }
        }
      }
```

为了在.NET 应用程序中引用 Windows 8.1 的 Windows 运行库，必须使用 Reference Manager 中的 Windows 选项卡来添加对 Windows 的引用。在项目文件中添加<TargetPlatformVersion>8.0</TargetPlatformVersion>，可以启用此选项卡。另外还需要在项目文件中手动添加对 System.Runtime 程序集的引用：<Reference Include="System.Runtime" />。

因为 PackageManager 类需要管理员权限，所以在项目中添加了一个包含 requestedExecutionLevel requireAdministrator 的应用程序清单。这会自动在提升权限的模式下启动应用程序。Visual Studio 创建的 app.manifest 文件还定义了一组 ID，用于指定支持特定的平台版本。下面的示例代码取消了对 Windows 8.1 版本的注释：

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
  xmlns="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="requireAdministrator"
          uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>
    </application>
  </compatibility>
```

```
</asmv1:assembly>
```

运行此应用程序将提供系统中安装的所有包的信息。以下内容节选自该应用程序的输出：

```
Architecture: X64
Family: Microsoft.BingFinance_8wekyb3d8bbwe
Full name: Microsoft.BingFinance_3.0.1.174_x64__8wekyb3d8bbwe
Name: Microsoft.BingFinance
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
Publisher Id: 8wekyb3d8bbwe
C:\Program Files\WindowsApps\Microsoft.BingFinance_
3.0.1.174_x64__8wekyb3d8bbwe

Architecture: X64
Family: Microsoft.BingFoodAndDrink_8wekyb3d8bbwe
Full name: Microsoft.BingFoodAndDrink_3.0.1.177_x64__8wekyb3d8bbwe
Name: Microsoft.BingFoodAndDrink
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation,
L=Redmond, S=Washington, C=US
Publisher Id: 8wekyb3d8bbwe
C:\Program Files\WindowsApps\Microsoft.BingFoodAndDrink_
3.0.1.177_x64__8wekyb3d8bbwe

Architecture: X64
Family: Microsoft.BingHealthAndFitness_8wekyb3d8bbwe
Full name: Microsoft.BingHealthAndFitness_3.0.1.176_x64__8wekyb3d8bbwe
Name: Microsoft.BingHealthAndFitness
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation,
L=Redmond, S=Washington, C=US
Publisher Id: 8wekyb3d8bbwe
C:\Program Files\WindowsApps\Microsoft.BingHealthAndFitness_
3.0.1.176_x64__8weky
```

18.7 小结

部署是应用程序生命周期的一个重要部分，会影响到应用程序中使用的技术，所以应该从项目的一开始就进行考虑。本章介绍了不同应用程序类型的部署。

本章介绍了如何使用 ClickOnce 部署 Windows 应用程序。ClickOnce 提供了一种方便的自动更新能力，也可以在应用程序中直接触发，在 System.Deployment API 中可以看到这一点。在介绍 Web 应用程序部署的小节中，我们查看了 Web Deploy 包，使用自定义的托管 IIS 和 Windows Azure 网站很容易部署它。

本章还介绍了 Windows Store 应用程序的部署。可以把 Windows Store 应用程序发布到 Windows Store 中，也可以不使用 Windows Store，而是使用 PowerShell 在企业环境中部署它们。

从第 19 章开始，我们将介绍.NET Framework 的基础，首先将介绍程序集。

第Ⅲ部分

基　　础

- 第 19 章　程序集
- 第 20 章　诊断
- 第 21 章　任务、线程和同步
- 第 22 章　安全性
- 第 23 章　互操作
- 第 24 章　文件和注册表操作
- 第 25 章　事务处理
- 第 26 章　网络
- 第 27 章　Windows 服务
- 第 28 章　本地化
- 第 29 章　核心 XAML
- 第 30 章　Managed Extensibility Framework
- 第 31 章　Windows 运行库

第19章

程 序 集

本章要点

- 程序集概述
- 创建程序集
- 使用应用程序域
- 共享程序集
- 版本问题
- 在不同技术之间共享程序集

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Application Domains
- Dynamic Assembly
- Shared Demo

19.1 程序集的含义

程序集是.NET 用于部署和配置单元的术语。本章主要讨论什么是程序集，如何使用它们，它们的功能为什么这么强大。

本章将讲述如何动态地创建程序集，如何把程序集加载到应用程序域中，如何在不同的应用程序之间共享程序集。本章还会介绍版本问题，这是共享程序集的一个重要方面。

程序集是.NET 应用程序的部署单元。.NET 应用程序包含一个或多个程序集。通常扩展名是 EXE 或 DLL 的.NET 可执行程序称为程序集。程序集和本地 DLL 或 EXE 有什么区别？它们的文件扩展名虽然相同，但.NET 程序集包含元数据，这些元数据描述了程序集中定义的所有类型及其成员的信息，即方法、属性、事件和字段。

.NET 程序集的元数据还提供了程序集中文件的相关信息、版本信息和所使用的程序集的信息。.NET 程序集为以前使用本地 DLL 时面临的众多问题提供了一个解决方案。

程序集是自我描述的安装单元，由一个或多个文件组成。程序集可以是包括元数据的 DLL 或 EXE，它也可以由多个文件组成，例如，资源文件、模块和 EXE。

程序集可以是私有或共享的。在简单的.NET 应用程序中，最好只使用私有程序集工作。私有程序集没有特殊的管理、注册和版本设置等问题，只有用户自己的应用程序在使用私有程序集时才有版本问题。其他应用程序不受影响，因为它们有自己的程序集副本。在这种应用程序中使用的私有组件应与应用程序一起安装。因为私有程序集位于应用程序所在的目录或子目录下，所以应用程序不会有版本冲突问题。其他应用程序都不会重写私有的程序集。当然，仍可以使用私有程序集的版本号。这非常有助于代码的修改(因为自己就可以检测到：这些程序集有不同的版本，某个地方一定发生了变化)，但它不是.NET 所必需的。

在使用共享程序集时，几个应用程序都使用同一个程序集，且与它有一定的依赖关系。共享程序集减少了磁盘和内存空间的需求。使用共享程序集时，要遵循许多规则。共享程序集必须有一个版本号和一个唯一的名称，通常它安装在全局程序集缓存(global assembly cache，GAC)中。GAC 允许共享系统上同一个程序集的不同版本。

19.1.1 程序集的功能

程序集的功能可以总结如下：

- 程序集是自描述的。不再需要考虑注册表键、从其他地方获得类型库等问题。程序集包含描述程序集的元数据。元数据包括从程序集中导出的类型和一个清单。下一节将介绍清单。
- 版本的相互依赖性在程序集的清单中进行了记录。任何被引用的程序集的版本都存储在程序集的清单中，这样就很容易确定因错误的版本号而引起的部署失败了。以后使用的引用程序集版本可以由开发人员和系统管理员配置。本章后面将介绍可用的版本策略及其工作方式。
- 程序集可以并行加载。从 Windows 2000 开始，就可以获得并行功能，其中同一个 DLL 的不同版本可以在系统上同时使用。你检查过目录<windows>\winsxs 吗？.NET 允许同一个程序集的不同版本在一个进程中使用！那么这有什么用呢？如果程序集 A 引用共享程序集 Shared 的版本 1，程序集 B 引用共享程序集 Shared 的版本 2，而用户同时使用程序集 A 和程序集 B，则应用程序需要使用共享程序集 Shared 的这两个版本，在.NET 中，应加载和使用两个版本。.NET 4 运行库允许一个进程中有多个 CLR 版本(2 和 4)。例如，这样就可以加载有不同 CLR 要求的插件。在同一个进程的不同 CLR 版本中，对象之间没有直接通信的.NET 方式，但可以使用其他技术，如 COM。
- 应用程序使用应用程序域来确保其独立性。使用应用程序域，许多应用程序就可以独立地运行在一个进程中。运行在一个应用程序域内的一个应用程序中的错误不会直接影响同一个进程中运行在另一个应用程序域内的其他应用程序。
- 安装非常简单，只需要复制一个程序集中的所有文件，一条 xcopy 命令就足够了。这个特性称为 ClickOnce 部署。但是在一些情况下不能进行 ClickOnce 部署，而需要正常的 Windows 安装。第 18 章讨论过应用程序的部署。

19.1.2 程序集的结构

程序集由描述它的程序集元数据、描述导出类型和方法的类型元数据、MSIL 代码和资源组成。所有这些部分都在一个文件中，或者分布在几个文件中。

在第一个例子中，程序集元数据、类型元数据、MSIL 代码和资源都在一个文件 Component.dll 中，如图 19-1 所示，这个程序集由一个文件组成。

第二个例子介绍的是分布在 3 个文件中的一个程序集，如图 19-2 所示。Component.dll 包含程序集元数据、类型元数据和 MSIL 代码，但不包含资源。这个程序集使用了一张图片 picture.jpeg，该图片没有嵌在 Component.dll 中，而是在程序集元数据中引用。程序集元数据还引用了一个模块 util.netmodule，该模块自身只包含一个类的类型元数据和 MSIL 代码，不包含程序集元数据，所以这个模块自身没有版本信息，也不能单独安装。第二个例子中的这 3 个文件构成了一个程序集，这个程序集是一个安装单元，还可以在另一个文件中放置程序集清单。

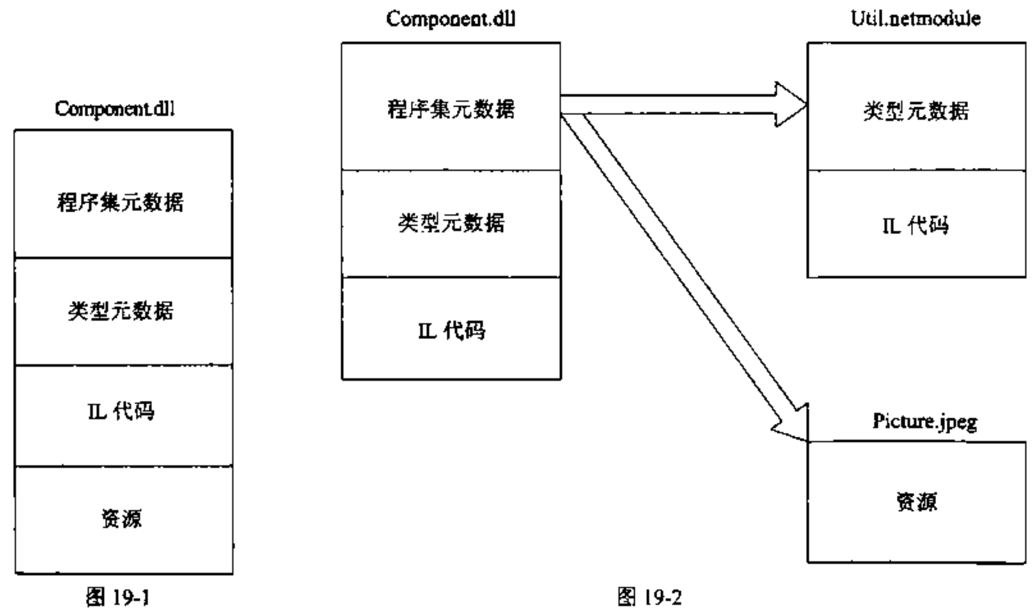


图 19-1　　　　图 19-2

19.1.3 程序集清单

程序集的一个重要部分是程序集清单，它是元数据的一部分，描述了程序集和引用它所需要的所有信息，并列出了它所有的依赖关系。清单由以下部分组成：

- 标识(名称、版本、文化和公钥)。
- 属于该程序集的一个文件列表。一个程序集至少要有一个文件，也可以包含多个文件。
- 被引用程序集的列表。在程序集清单中说明了在程序集中使用的所有程序集，这些引用信息包括版本号和公钥。公钥用于唯一地标识程序集。本章后面将讨论公钥。
- 一组许可请求——运行这个程序集需要的许可。许可详见第 22 章。
- 导出的类型，假定它们在一个模块中定义，该模块从程序集中引用，程序集就包含它们；否则它们就不是程序集清单的一部分。模块是可重用的单元。类型描述在程序集中存储为元数据，使用属性和方法可以从这些元数据中获得结构和类，它替代了以前用 COM 描述类

型的类型库。使用COM客户端很容易在程序集清单的外部生成一个类型库。反射机制使用已导出类型的信息，便于后面绑定到类。有关反射的内容，请参见第15章。

19.1.4 名称空间、程序集和组件

也许你目前会混淆名称空间、类型、程序集和组件。名称空间如何与程序集的概念相匹配？名称空间完全独立于程序集。在一个程序集中可以有不同的名称空间，同一个名称空间也可以分布在多个程序集上。名称空间只是类型名的一种扩展，它属于类型名的范畴。

例如，除了许多其他名称空间外，程序集mscorlib和system都包含名称空间System.Threading。尽管程序集包含相同的名称空间，但没有相同的类名。

19.1.5 私有程序集和共享程序集

程序集可以是共享的，也可以是私有的。私有程序集或者位于应用程序所在的同一个目录下，或者位于其子目录中。使用私有程序集时，不需要考虑与其他类的命名冲突或版本问题。在构建过程中引用的程序集会复制到应用程序的目录下。私有程序集是构建程序集的一般方式，特别是在同一个公司中构建应用程序和组件时，就更是如此。

尽管私有程序集可能仍有命名冲突(应用程序可能包含多个私有程序集，这些程序集本来有冲突，或者一个私有程序集中的名称与应用程序使用的一个共享程序集中的名称冲突)，但命名冲突会大大减少。如果使用了多个私有程序集或使用了其他应用程序中的共享程序集，最好利用恰当命名的名称空间和类型，使命名冲突降到最少。

在使用共享程序集时，必须遵循一些规则。程序集必须是唯一的，因此，必须有一个唯一的名称(称为强名)。强名的一部分是一个强制的版本号。当组件由另一个开发商构建，而不是应用程序的开发商构建时，以及一个大型应用程序分布在几个子项目中时，常常需要使用共享程序集。另外，一些技术(如.NET Enterprise Services)需要在特定的情形下使用共享程序集。

19.1.6 附属程序集

附属程序集是只包含资源的程序集，它尤其适用于本地化。因为程序集有一种相关的文化，所以资源管理器会查找包含特定文化资源的附属程序集。

附属程序集的更多信息可参见第28章。

19.1.7 查看程序集

程序集可以使用命令行实用工具ildasm来查看，这是一个MSIL反汇编程序。从命令行中启动ildasm，把程序集作为其参数，或者选择File | Open命令，就可以打开程序集。

图19-3是ildasm打开的一个示例程序SharedDemo.dll，后面会构建这个程序。ildasm显示了程序集清单，以及Wrox.ProCSharp.Assemblies名称空间中的SharedDemo类型。打开该程序集清单时，就可以看到版本号、程序集特性、被引用的程序集及其版本。打开类的方法，就可以查看MSIL代码。

19.2 构建程序集

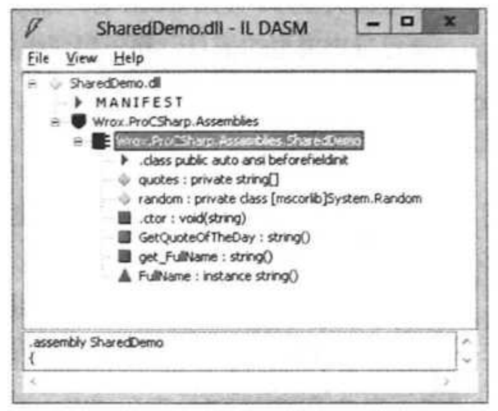


图 19-3

前面学习了程序集的含义，下面要构建一些程序集。当然，本书前面已经构建了一些程序集，因为.NET可执行程序也是一个程序集。下面要介绍程序集的特定选项。

19.2.1 创建模块和程序集

在 Visual Studio 中，所有的 C#项目类型都会创建一个程序集。无论是选择 DLL 项目类型，还是 EXE 项目类型，都会创建一个程序集。使用命令行 C#编译器 csc，也可以创建模块。模块是一个没有程序集特性的 DLL(所以它不是程序集，但可以在以后把它添加到程序集中)。命令

```
csc /target:module hello.cs
```

创建模块 hello.netmodule，可以使用 ildasm 查看这个模块。

模块也有一个清单，但在该清单中没有.assembly 条目(除了引用的外部程序集之外)，因为模块没有程序集特性。不能用模块来配置版本或许可，而只能在程序集范围内进行。在模块的清单中可以找到程序集的引用。使用 csc 的/addmodule 选项，可以把模块添加到已有的程序集中。

为了比较模块和程序集，下面创建一个简单的 A 类，并用下面的命令编译它：

```
csc /target:module A.cs
```

编译器生成 A.netmodule 文件，它不包括程序集的信息(使用 ildasm 可以查看清单信息)。模块的清单显示了被引用的程序集 mscorlib 和.module 条目，如图 19-4 所示。

图 19-4

下面创建一个程序集 B，它包括模块 A.netmodule。不需要用一个源文件来生成这个程序集，生成该程序集的命令如下：

```
csc /target:library /addmodule:A.
netmodule /out:B.dll
```

在使用 ildasm 查看程序集时，只能找到一个清单。在清单中，引用了程序集 mscorlib。接着看

看带有散列算法和版本的程序集部分。算法的数量决定了用于创建程序集的散列代码的算法类型。在通过编程创建程序集时，可以选择该算法。清单包含属于该程序集的所有模块的一个列表。在图19-5中，可以看出.file A.netmodule是该程序集的一部分，从模块中导出的类是程序集清单的一部分，从程序集本身导出的类则不是。

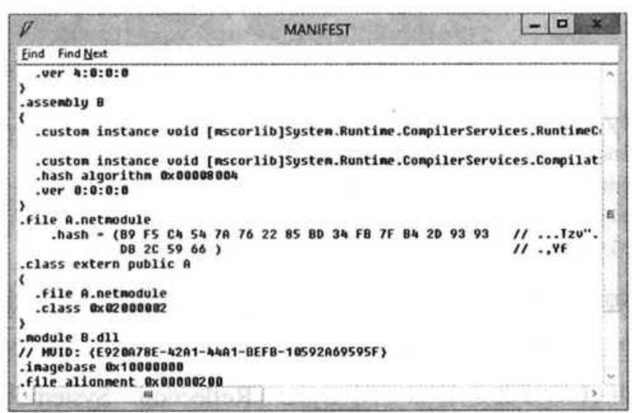


图19-5

模块可以更快地启动程序集，因为并不是所有的类型都在一个文件中。模块只在需要时加载。使用模块的另一个原因是，要用多种编程语言来创建一个程序集。一个模块用Visual Basic编写，另一个模块用C#编写，这两个模块可以包括在一个程序集中。

19.2.2 程序集的特性

在创建Visual Studio项目时，会自动生成源文件AssemblyInfo.cs，这个文件在Solution Explorer窗口的Properties对话框中。在该文件中，可以使用一般的源代码编辑器配置程序集的特性。下面是从项目模板中生成的一个文件：

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("ClassLibrary1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("CN innovation")]
[assembly: AssemblyProduct("ClassLibrary1")]
[assembly: AssemblyCopyright("Copyright @ CN innovation 2013]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]
// The following GUID is for the ID of the typelib if this project is exposed
// to COM
[assembly: Guid("21649c19-6609-4607-8fc0-d75f1f27a8ff")]
```

```
//
// Version information for an assembly consists of the following four
// values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision
// Numbers by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

这个文件用于配置程序集清单。编译器读取程序集特性，把特定的信息插入到程序集清单中。

assembly:前缀把特性标记为程序集级别特性。与其他特性相反，程序集级别特性与特定的语言元素无关，用于程序集特性的参数是名称空间 System.Reflection、System.Runtime.CompilerServices 和 System.Runtime.InteropServices 中的类。

第 15 章介绍了特性、如何创建和使用自定义特性的内容。

表 19-1 是 System.Reflection 名称空间中定义的程序集特性列表。

表 19-1

程序集的特性	说　明
AssemblyCompany	指定公司名
AssemblyConfiguration	指定构建信息，例如零售或调试信息
AssemblyCopyright and AssemblyTrademark	包含版权和商标信息
AssemblyDefaultAlias	如果程序集名不容易理解(例如，动态地创建程序集名时的 GUID)，就可以使用该特性。使用这个特性可以指定别名
AssemblyDescription	描述程序集或产品。如果查看可执行文件的属性，这个值就会显示为 Comments
AssemblyProduct	指定了程序集所属产品的名称
AssemblyTitle	给程序集提供一个友好的名称。该名称可以包含空格。使用文件属性时，这个值就显示为 Description
AssemblyCulture	定义程序集的文化。这个特性对附属程序集很重要
AssemblyInformationalVersion	在引用程序集时，这个特性不用于版本检查，它仅用于版本信息。该特性非常适合于指定使用多个程序集的应用程序的版本。打开可执行程序的属性，这个值就显示为 Product Version
AssemblyVersion	这个特性给出了程序集的版本号。本章后面讨论版本问题
AssemblyFileVersion	这个特性定义了文件的版本。这个值显示在 Windows 文件属性窗口中，但它对.NET 的行为没有影响

下面是配置这些特性的一个示例：

```
[assembly: AssemblyTitle("Professional C#")]
[assembly: AssemblyDescription("Sample Application")]
[assembly: AssemblyConfiguration("Retail version")]
[assembly: AssemblyCompany("Wrox Press")]
[assembly: AssemblyProduct("Wrox Professional Series")]
[assembly: AssemblyCopyright("Copyright (C) Wrox Press
2013]
[assembly: AssemblyTrademark("Wrox is a registered
trademark of " +
      "John Wiley & Sons, Inc.")]
[assembly: AssemblyCulture("")]

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

在 Visual Studio 2013 中，可以在项目属性对话框中选择 Application 选项卡，然后单击 Assembly Information 按钮来配置这些特性，如图 19-6 所示。

图 19-6

19.2.3 创建和动态加载程序集

在开发期间，添加对程序集的引用，使之包含在程序集引用中，该类型的程序集就可用于编译器。在运行期间，只要实例化了一种类型的程序集，或者使用了该类型的一个方法，就会加载所引用的程序集。除了使用这种自动操作之外，还可以通过编程加载程序集。为此，可以使用 Assembly 类的静态方法 Load()。这个方法是重载的，其中可以使用 AssemblyName 给它传递程序集的名称或字节数组。

还可以快速创建程序集，如下面的例子所示。这个例子把 C#代码输入文本框后，启动 C#编译器，就会动态地创建一个新程序集，并调用编译的代码。

要动态地编译 C#代码，可以使用 Microsoft.CSharp 名称空间中的 CSharpCodeProvider 类。使用这个类可以编译代码，从 DOM 树、文件和源代码中生成程序集。

该应用程序的 UI 是使用 WPF 创建的，图 19-7 显示了这个 UI 的设计视图。窗口由一个要输入 C#代码的文本框、一个按钮和一个 TextBlock WPF 控件组成，TextBlock 横跨最后一行的所有列，以显示对应的结果。

图 19-7

为了动态地编译并运行 C#代码，CodeDriver 类定义了 CompileAndRun()方法。这个方法编译文本框中的代码，并启动所生成的方法(代码文件 DynamicAssembly/CodeDriver.cs)。

```
using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriver
  {
    private string prefix =
        "using System;" +
        "public static class Driver" +
        "{" +
        " public static void Run()" +
        " {";

    private string postfix =
        " }" +
        "}";


    public string CompileAndRun(string input, out bool hasError)
    {
      hasError = false;
      string returnData = null;

      CompilerResults results = null;
      using (var provider = new CSharpCodeProvider())
      {
        var options = new CompilerParameters();
        options.GenerateInMemory = true;

        var sb = new StringBuilder();
        sb.Append(prefix);
        sb.Append(input);
        sb.Append(postfix);

        results = provider.CompileAssemblyFromSource(options, sb.ToString());
      }

      if (results.Errors.HasErrors)
      {
        hasError = true;
        var errorMessage = new StringBuilder();
        foreach (CompilerError error in results.Errors)
        {
          errorMessage.AppendFormat("{0} {1}", error.Line,
              error.ErrorText);
        }
        returnData = errorMessage.ToString();
      }
      else
      {
```

```
          TextWriter temp = Console.Out;
          var writer = new StringWriter();
          Console.SetOut(writer);
          Type driverType = results.CompiledAssembly.GetType("Driver");

          driverType.InvokeMember("Run", BindingFlags.InvokeMethod |
             BindingFlags.Static | BindingFlags.Public, null, null, null);
          Console.SetOut(temp);

          returnData = writer.ToString();
        }

        return returnData;
      }
    }
  }
```

CompileAndRun()方法需要一个字符串参数input，在其中可以传递一行或多行C#代码。因为调用的每个方法都必须包含在方法和类中，所以变量prefix和postfix定义了动态创建的Driver类的结构和包含参数中代码的Run()方法。使用StringBuilder，把prefix、postfix和input变量中的代码合并起来，创建一个完整的、可编译的类。再使用这个得到的字符串，通过CSharpCodeProvider类编译代码。CompileAssemblyFromSource()方法动态地创建一个程序集。因为这个程序集仅需要在内存中使用，所以设置了编译器参数选项GenerateInMemory。

如果所传递的源代码包含错误，它们就会显示在CompilerResults的Errors集合中。错误和返回数据一起返回，变量hasError设置为true。

如果源代码编译成功，就调用新的Driver类的Run()方法。这个方法的调用通过反射来实现。新编译的程序集可以使用CompilerResults.CompiledType来访问，在这个程序集中，新的Driver类由变量driverType引用。接着使用Type类的InvokeMember()方法调用Run()方法。因为这个方法定义为公共静态方法，所以必须相应地设置BindingFlags。要查看程序写到控制台上的结果，需要把控制台重定向到StringWriter中，最终使用returnData变量返回程序的全部结果。

用InvokeMember()方法运行代码需要使用.NET反射功能，该功能详见第15章。

WPF按钮的Click事件连接到Compile_Click()方法上，在该方法中，实例化CodeDriver类，并调用CompileAndRun()方法。从文本框textCode中提取输入，把结果写到TextBlock控件textOutput中(代码文件DynamicAssembly/DynamicAssemblyWindow.xaml.cs)。

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
  textOutput.Background = Brushes.White;
  var driver = new CodeDriver();
  bool isError;
  textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
  if (isError)
  {
    textOutput.Background = Brushes.Red;
```

```
    }
}
```

现在可以启动应用程序，在 TextBox 中输入 C#代码，如图 19-8 所示，编译并运行代码。

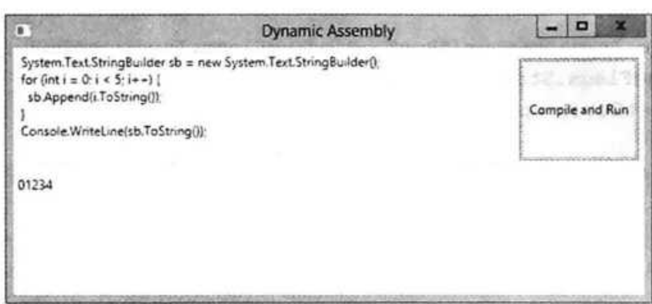


图 19-8

前面编写的程序有一个缺点：每次单击 Compile And Run 按钮时，都会创建并加载一个新程序集，于是程序需要越来越多的内存。不能从应用程序中卸载程序集。要卸载程序集，需要使用应用程序域。

19.3 应用程序域

在.NET 之前的技术中，进程作为独立的边界来使用，每个进程都有其私有的虚拟内存；运行在一个进程中的应用程序不能写入另一个应用程序的内存，也不会因为这种方式破坏其他应用程序。该进程用作应用程序之间的一个独立而安全的边界。在.NET 体系结构中，应用程序有一个新的边界：应用程序域。使用托管 IL 代码，运行库可以确保在同一个进程中不能访问另一个应用程序的内存。多个应用程序可以运行在一个进程的多个应用程序域中，如图 19-9 所示。

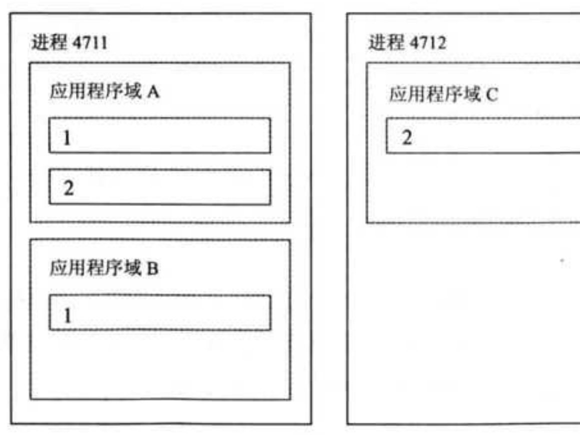


图 19-9

把程序集加载到应用程序域中。在图 19-9 中，进程 4711 有两个应用程序域。在应用程序域 A 中，实例化对象 1 和 2，对象 1 在程序集 1 中，对象 2 在程序集 2 中。在进程 4711 中，第二个应用程序域有对象 1 的一个实例。要最小化占用的内存，在应用程序域中，程序集的代码应只加载一次。实例和静态成员不能在应用程序域之间共享，也不能直接访问另一个应用程序域中的对象。此时需要一个代理(proxy)。所以在图 19-9 中，如果没有代理，应用程序域 B 中的对象 1 就不能直接访问应

用程序域 A 中的对象 1 或 2。

AppDomain 类用于创建和终止应用程序域，加载、卸载程序集和类型，以及枚举应用程序域中的程序集和线程。下面用一个小例子来说明应用程序域。

首先，创建一个 C#控制台应用程序 AssemblyA，在 Main()方法中添加一个 Console.WriteLine()方法，这样，我们就知道这个方法什么时候调用。另外，添加 Demo 类，其构造函数的参数是两个 int 值，用这两个参数和 AppDomain 类创建实例。从将要创建的第二个应用程序中加载程序集 AssemblyA.exe(代码文件 AssemblyA/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Assemblies
{
  public class Demo
  {
    public Demo(int val1, int val2)
    {
      Console.WriteLine("Constructor with the values {0}, {1} in domain " +
          "{2} called", val1, val2, AppDomain.CurrentDomain.FriendlyName);
    }
  }

  class Program
  {
    static void Main()
    {
      Console.WriteLine("Main in domain {0} called",
      AppDomain.CurrentDomain.FriendlyName);
    }
  }
}
```

运行应用程序，结果如下所示：

```
Main in domain AssemblyA.exe called.
```

创建的第二个项目也是一个 C#控制台应用程序：DomainTest。首先，使用 AppDomain 类的 FriendlyName 属性显示当前域的名称。调用 CreateDomain()方法，新建应用程序域 NewAppDomain，然后把程序集 AssemblyA 加载到新域中，调用 ExecuteAssembly()方法来调用 Main()方法(代码文件 DomainTest/Program.cs)：

```
using System;
using System.Reflection;

namespace Wrox.ProCSharp.Assemblies
{
  class Program
  {
    static void Main()
    {
      AppDomain currentDomain = AppDomain.CurrentDomain;
      Console.WriteLine(currentDomain.FriendlyName);
```

```
            AppDomain secondDomain = AppDomain.CreateDomain("New AppDomain");
            secondDomain.ExecuteAssembly("AssemblyA.exe");
        }
    }
}
```

在启动程序 DomainTest.exe 前，通过 DomainTest 项目引用 AssemblyA.exe 程序集。通过 Visual Studio 2013 引用程序集，就是把程序集复制到项目的目录中，这样项目才能找到这个程序集。如果找不到这个程序集，就会抛出 System.IO.FileNotFoundException 异常。

运行 DomainTest.exe 程序后，会得到如下所示的控制台输出。DomainTest.exe 是第一个应用程序域的友好名称。第二行是 New AppDomain 中新加载的程序集的输出结果。在进程查看器中看不到进程 AssemblyA.exe 的执行，因为没有新建进程，AssemblyA 加载到 DomainTest.exe 进程中。

```
DomainTest.exe
Main in domain New AppDomain called
```

在新加载的程序集中，还可以新建一个实例，以替代调用 Main()方法。在下面的例子中，用 CreateInstance()方法替代 ExecuteAssembly()方法，它的第一个参数是程序集名 AssemblyA，第二个参数定义了应实例化的类型 Wrox.ProCSharp.Assemblies.AppDomains.Demo，第三个参数 true 表示不区分大小写。System.Reflection.BindingFlags.CreateInstance 是一个绑定的标志枚举值，用来指定应调用的构造函数：

```
AppDomain secondDomain = AppDomain.CreateDomain("New AppDomain");
// secondDomain.ExecuteAssembly("AssemblyA.exe");
secondDomain.CreateInstance("AssemblyA",
    "Wrox.ProCSharp.Assemblies.Demo", true,
    BindingFlags.CreateInstance, null, new object[] {7, 3},
    null, null);
```

在应用程序成功运行后，会得到如下所示的控制台输出：

```
DomainTest.exe
Constructor with the values 7, 3 in domain New AppDomain called
```

前面介绍了如何创建和调用应用程序域。在运行库宿主上，会自动创建应用程序域。大多数应用程序类型仅有默认的应用程序域。ASP.NET 为运行在 Web 服务器上的每个 Web 应用程序创建一个应用程序域。Internet Explorer 创建运行托管控件的应用程序域。对于应用程序，如果要卸载一个程序集，创建应用程序域就非常有效。卸载程序集只能通过终止应用程序域来进行。

如果程序集是动态加载的，且需要在使用完后卸载程序集，应用程序域就非常有用。在主应用程序域中，不能删除已加载的程序集，但可以终止应用程序域，在该应用程序域中加载的所有程序集都会从内存中清除。

了解了应用程序域后，就可以修改前面创建的 WPF 程序了。新类 CodeDriverInAppDomain 使用 AppDomain.CreateDomain 新建了一个应用程序域。在这个新应用程序域中，使用 CreateInstanceAndUnwrap()方法实例化 CodeDriver 类。使用 CodeDriver 实例，调用 CompileAndRun()

方法，之后再次卸载新应用程序域。

```
using System;
using System.Runtime.Remoting;

namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriverInAppDomain
  {
    public string CompileAndRun(string code, out bool hasError)
    {
      AppDomain codeDomain = AppDomain.CreateDomain("CodeDriver");

      CodeDriver codeDriver = (CodeDriver)
      codeDomain.CreateInstanceAndUnwrap("DynamicAssembly",
          "Wrox.ProCSharp.Assemblies.CodeDriver");
      string result = codeDriver.CompileAndRun(code, out hasError);

      AppDomain.Unload(codeDomain);

      return result;
    }
  }
}
```

CodeDriver类本身现在同时在主应用程序域和新应用程序域中使用，因此不能删除这个类使用的代码。如果要删除这些代码，就可以定义一个由CodeDriver类实现的接口，再在主应用程序域中使用这个接口。但是在这个例子中，这并不是个问题，因为只需要删除用Driver类动态创建的程序集即可。

要在另一个应用程序域中访问CodeDriver类，CodeDriver类就必须派生于基类MarshalByRefObject。只有派生自这个基类的类才能通过另一个应用程序域来访问。在主应用程序域中，实例化一个代理，以通过应用程序域之间的信道调用这个类的方法(代码文件DynamicAssembly/CodeDriver.cs)。

```
using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriver: MarshalByRefObject
  {
```

Compile_Click()事件处理程序现在可以修改为使用CodeDriverInAppDomain类，而不是使用CodeDriver类(代码文件DynamicAssembly/DynamicAssemblyWindow.xaml.cs)：

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
  var driver = new CodeDriverInAppDomain();
  bool isError;
  textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
  if (isError)
  {
    textOutput.Background = Brushes.Red;
  }
}
```

现在可以单击应用程序的 Compile And Run 按钮任意多次，生成的程序集总是会被卸载。

使用 AppDomain 类的 GetAssemblies()方法，就可以查看应用程序域中加载的程序集。

19.4 共享程序集

程序集可以由一个应用程序单独使用，在默认情况下不共享程序集。在使用共享程序集时，需要考虑一些特定的要求。本节将探讨共享程序集所需的信息。必须使用强名唯一地标识共享程序集，强名通过给程序集的签名来创建。本节还将解释延迟签名的过程。共享程序集一般安装在全局程序集缓存(GAC)中，我们还会讨论如何使用 GAC。

19.4.1 强名

共享程序集名必须是全局唯一的，并且必须可以保护该名称。任何其他人不能使用同一个名称创建程序集。

COM 使用全局唯一标识符(GUID)解决了第一个问题。但第二个问题仍没有解决，因为每个人都可以盗用这个 GUID，用相同的标识符创建不同的对象。通过.NET 程序集的强名可以解决这两个问题。

强名由下述项组成：

- 程序集本身的名称
- 版本号。有了版本号，可以同时使用同一个程序集的不同版本。不同的版本可以同时存在，并可以同时加载到同一个进程上。
- 公钥保证强名是独一无二的。它也保证被引用的程序集不能从另一个源中替代。
- 文化。文化详见第 28 章。

共享程序集必须有一个强名，来唯一地标识该程序集。

强名是一个简单的文本名称，附带版本号、公钥和文化。不能使用每个程序集新建公钥，但可以在公司中有一个这样的公钥，这样该密钥就唯一地标识了公司的程序集。

但是，这个密钥不能用作信任密钥。程序集可以利用 Authenticode 签名来建立信任关系。Authenticode 签名的密钥可以与针对强名使用的密钥不同。

对于开发目的来说，可以使用不同的公钥，以后换为真正的密钥。该特性将在“程序集的延迟签名”一节中介绍。

为了唯一地标识公司中的程序集，应使用名称空间层次结构来给类命名。下面是介绍如何组织名称空间的一个简单例子：Wrox 出版社使用主名称空间 Wrox 来标识其类和名称空间。在名称空间 Wrox 下面的层次结构中，必须对名称空间进行组织，使所有的类都是唯一的。本书中的每一章都使用 Wrox.ProCSharp.<Chapter>形式的不同名称空间。本章使用 Wrox.ProCSharp. Assemblies 名称空间。这样，如果在某两章中都有 Hello 类，就不会有名称冲突，因为它们在不同的名称空间中。可以在不同的书中使用的实用工具类则放在 Wrox.Utilities 名称空间中。

公司名一般用作名称空间的第一部分，但它不一定是唯一的，因此必须使用某种机制来建立强名。此时可以使用公钥。因为在强名中使用公钥/私钥规则，所以不能访问私钥的人，就不能破坏性地创建一个程序集，让客户无意中调用该程序集。

19.4.2 使用强名获得完整性

在创建共享组件时，必须使用公钥/私钥对。编译器把公钥写入程序集清单，创建属于该程序集的所有文件的散列表，用私钥对这个散列表签名，该私钥不存储在程序集中。这样就可以确保没有人可以修改该程序集。签名可以使用公钥来验证。

在开发过程中，客户端程序集必须引用共享程序集。编译器把被引用程序集的公钥写入客户端程序集的清单中。要减少存储量，就不应把公钥写入客户端程序集的清单，而应写入公钥标记。公钥标记是公钥散列表中的最后 8 个字节，且是唯一的。

在运行期间加载共享程序集时(如果客户程序集是使用本机映像生成器安装的，则应在安装期间加载)，共享程序集的散列表可以使用存储在客户端程序集中的公钥来验证。除了私钥的主人外，其他人都不能修改共享的组件程序集。供应商 A 创建了一个组件 Math，在客户端上引用该组件，黑客的组件就无法替代它。只有私钥的主人才能用新版本替换原来的共享组件。只要共享程序集来自期望的发布商，就保证了完整性。

图 19-10 显示了一个共享组件，它的公钥由客户端程序集引用，该程序集在其清单中包含共享程序集的公钥标记。

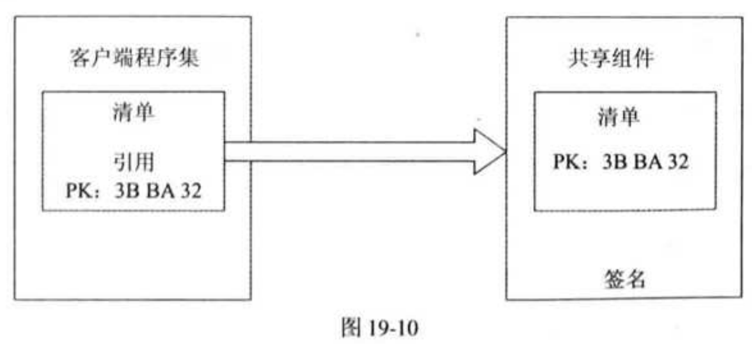


图 19-10

19.4.3 全局程序集缓存

顾名思义，全局程序集缓存(Global Assembly Cache，GAC)就是可全局使用的程序集的缓存。大多数共享程序集都安装在这个缓存中；另外，也可以使用共享目录(也在服务器上)。

GAC位于<windows>\Microsoft.NET\assembly目录下。在这个目录中，有多个GACxxx目录。GACxxx目录包含共享程序集，GAC_MSIL目录包含带纯.NET代码的程序集；GAC_32包含专用于32位平台的程序集。在64位系统上，GAC-64目录包含专用于64位平台的程序集。

在<windows>\assembly\NativeImages_<runtime version>目录下，有编译为本机代码的程序集。如果再深入该目录结构，就会看到与程序集名类似的一些目录名，其下是一个版本目录和程序集本身。这样就可以安装同一个程序集的不同版本。

gacutil.exe实用工具可以使用命令行安装、卸载和显示程序集。gacutil的一些选项如下所示：

- gacutil /l——显示程序集缓存中的所有程序集。
- gacutil /i mydll——把共享程序集mydll安装到程序集缓存上。即使程序集已安装，也可以使用选项/f强制安装到GAC中。如果修改了程序集，但没有改变版本号，使用选项/f就很有用。
- gacutil /u mydll——卸载程序集mydll。

在产品系统中，应使用安装程序，把共享程序集安装在GAC中。部署参见第18章。

在.NET 4之前，共享程序集的目录是<windows>\assembly。这个目录包含一个Windows shell扩展名，从而更方便地显示程序集和版本号。.NET 4程序集不能使用这个shell扩展。

19.4.4 创建共享程序集

在本例中，创建一个共享程序集，再创建一个使用该共享程序集的客户。创建共享程序集与创建私有程序集的区别不大。首先创建一个简单的Visual C#类库项目SharedDemo。把名称空间改为Wrox.ProCSharp.Assemblies，把类名改为SharedDemo。输入下面的代码。类的构造函数把文件的所有行都读取到一个数组中。文件名作为参数传递给构造函数。GetQuoteOfTheDay()方法只返回这个数组的一个随机字符串(代码文件SharedDemo/SharedDemo.cs)。

```
using System;
using System.IO;

namespace Wrox.ProCSharp.Assemblies
{
  public class SharedDemo
  {
    private string[] quotes;
    private Random random;

    public SharedDemo(string filename)
```

```
        {
          quotes = File.ReadAllLines(filename);
          random = new Random();
        }

        public string GetQuoteOfTheDay()
        {
          int index = random.Next(1, quotes.Length);
          return quotes[index];
        }
      }
    }
```

19.4.5 创建强名

要共享这个程序集，需要一个强名。要创建这个名称，可以使用强名工具(sn)：

```
sn -k mykey.snk
```

强名实用工具生成和编写一个公钥/私钥对，并把该密钥对写到文件中，此处的文件是mykey.snk。

在 Visual Studio 2012 中，可以选择 Signing 选项卡，用项目属性标记程序集，如图 19-11 所示。还可以使用这个工具创建密钥。但不需要为每个项目创建密钥文件。整个公司可以只使用几个密钥。最好根据安全要求创建不同的密钥，详见第 22 章。

用 Visual Studio 设置 signing 选项，会给编译器设置添加/keyfile 选项。Visual Studio 还允许创建用密码保护的密钥文件。这种文件的扩展名是.pfx，如图 19-11 所示。

AssemblySamples - SharedDemo

SharedDemo

Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Code Analysis

Configuration: N/A Platform: N/A

Sign the ClickOnce manifests

Certificate

Timestamp server URL

Sign the assembly

Choose a strong name key file:

mykey.pfx

Change Password...

Delay sign only

When delay signed, the project will not run or be debuggable.

图 19-11

在重新生成该文件后，公钥就在该程序集清单中。可以使用 ildasm 验证这一点，如图 19-12 所示。

```
MANIFEST
Find  Find Next
  // --- The following custom attribute is added automatically, do not uncor
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute

  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilati
  .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCo

  .publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00   // .$....
                00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00   // .$..RS
                57 F2 67 E3 0C E3 EF 08 7F DA C7 82 00 9B D3 12   // W.g...
                EB 46 D7 56 BD 92 C5 44 30 8A 7E BC B7 36 BD 66   // .F.V..
                8A E2 6A 28 3E 85 AF 2C 9E C3 C0 EC 52 27 1D 5B   // ..j(>.
                33 B2 9D 44 BC 48 98 09 77 31 A4 80 4C A2 CE BB   // 3..D.H
                C3 EE 6D 95 BE E4 F9 13 30 D5 5A BB F4 56 8E 9C   // ..m...
                8C 94 EA D8 66 86 77 6F 15 B1 6C F8 76 59 51 67   // ....f.
                B8 7C FE 34 11 49 CE 81 99 C8 42 27 63 45 EC B7   // .|.4.I
                1D 4C 15 40 6E 70 B3 8E 00 D1 A3 BA 85 8D E5 B9 ) // .L.@np
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module SharedDemo.dll
// MVID: {036C1A39-5414-4077-91BE-63E3406E8876}
```

图 19-12

19.4.6 安装共享程序集

程序集中有了公钥后，就可以使用全局程序集缓存工具 gacutil 及其/i 选项把它安装到全局程序集缓存中。即使程序集已安装，也可以使用选项/f 强制把它写到 GAC 中。

```
gacutil /i SharedDemo.dll /f
```

然后，可以使用全局程序集缓存查看器或 gacutil /l SharedDemo 检查共享程序集的版本，看看它是否安装成功。

19.4.7 使用共享程序集

要使用共享程序集，应创建一个 C#控制台应用程序 Client。把名称空间的名称改为 Wrox.ProCSharp.Assemblies。以引用私有程序集的方式引用共享程序集：使用菜单 Project | Add Reference 命令。

有了共享程序集，引用属性 Copy Local 就可以设置为 false，这样，共享程序集就不会复制到输出文件的目录中，而会从 GAC 中加载。

在项目条目中添加 quotes.txt 文件，并把 Copy to Output Directory 属性设置为 Copy if newer。下面是 Client 应用程序的代码(代码文件 Client/Program.cs)：

```
using System;
namespace Wrox.ProCSharp.Assemblies
{
  class Program
  {
    static void Main()
    {
      var quotes = new SharedDemo("Quotes.txt");
      for (int i=0; i < 3; i++)
      {
        Console.WriteLine(quotes.GetQuoteOfTheDay());
        Console.WriteLine();
      }
```

```
      }
    }
  }
```

在使用ildasm查看客户端程序集的清单时(如图19-13所示)，可以看到对共享程序集SharedDemo的引用：.assembly extern SharedDemo。这些引用信息的一部分是版本号(详见后面的内容)和公钥的标记。

```
MANIFEST
Find  Find Next
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .
  .ver 4:0:0:0
}
.assembly extern SharedDemo
{
  .publickeytoken = (F9 46 43 3F DA E2 51 2D )                         // .
  .ver 1:0:0:0
}
.assembly Client
{
  .custom instance void [mscorlib]System.Runtime.Versioning.TargetFramework
```

图19-13

公钥的标记也可以使用强名实用工具sn在共享程序集中查看：sn –T会显示程序集中的公钥标记，sn –Tp显示标记和公钥。注意使用大写字母T！

该程序引用了示例文件，其结果如下所示。

```
"We don't like their sound. And guitar music is on the way out."
- Decca Recording, Co., in rejecting the Beatles, 1962

"The ordinary 'horseless carriage' is at present a luxury for the wealthy; and
although its price will probably fall in the future, it will never come into as
common use as the bicycle." - The Literary Digest, 1889

"Landing and moving around the moon offers so many serious problems for human
beings that it may take science another 200 years to lick them", Lord Kelvin
(1824-1907)
```

19.4.8 程序集的延迟签名

公司的私钥应安全地存储。大多数公司都不允许所有的开发人员都能访问私钥。只有几个有安全权限的人才能访问它。这就是为什么程序集的签名可以以后(如在发布之前)添加的原因。在全局程序集特性AssemblyDelaySign设置为true时，签名就不会存储在程序集中，但保留了足够的空间，以便以后添加它。不使用密钥，就不能测试程序集，在全局程序集缓存中安装它。但是，可以使用临时密钥进行测试，以后再用真正的公司密钥代替这个临时密钥。

延迟程序集的签名需要执行下面的步骤：

(1) 首先必须用强名实用工具sn创建一个公钥/私钥对，生成的文件mykey.snk包含公钥和私钥。

```
sn -k mykey.snk
```

(2) 接着提取公钥，使之可以用于开发人员。选项-p提取密钥文件的公钥。文件mypub.snk仅包含公钥。

```
sn -p mykey.snk mykeypub.snk
```

公司中的所有开发人员都可以使用这个密钥文件mykeypub.snk，用/delaysign+选项编译程序集。

这样，签名就没有添加到程序集中，但可以以后添加它。在 Visual Studio 2012 中，延迟签名选项可以在 Signing 设置的复选框中设置。

(3) 关闭签名的验证功能，因为程序集没有包含签名。

```
sn -Vr SharedDemo.dll
```

(4) 在发布之前，程序集可以用 sn 实用工具重新签名。–R 选项用于对以前已签名或延迟签名的程序集进行重新签名。程序集的重新签名可以由部署应用程序的软件包且有权访问用于发布的私钥的人员完成。

```
sn -R MyAssembly.dll mykey.snk
```

签名的验证功能只能在开发过程中关闭。不经过验证是不能发布程序集的，因为这个程序集可能被恶意的程序集替代。

程序集的重新签名可以通过在 MSBuild 文件中定义任务来自动完成，相关内容参见第 17 章。

19.4.9 引用

GAC 中的程序集可以包含与它们相关联的引用。这些引用负责：如果应用程序仍需要引用程序集，缓存的程序集就不能删除。例如，如果 Microsoft 安装程序包(.msi 文件)安装了一个共享程序集，就只能通过卸载应用程序删除它，而不能从全局程序集缓存中直接删除它。从全局程序集缓存中删除程序集会得到一条错误消息："程序集<name>不能卸载，因为其他应用程序还需要它。"

```
"Assembly <name> could not be uninstalled because it is required by other
applications."
```

使用 gacutil 实用工具和选项/r 可以设置程序集的引用。/r 选项需要一个引用类型、一个引用 ID 和一个描述。引用的类型可以是下面 3 个选项中的一个：UNINSTALL_KEY、FILEPATH 或 OPAQUE。UNINSTALL_KEY 由 MSI 使用，定义一个注册表键之后卸载它需要使用该选项。用 FILEPATH 可以指定一个目录，应用程序的根目录是有用的目录。OPAQUE 引用类型允许设置任意类型的引用。

命令行：

```
gacutil /i shareddemo.dll /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

通过引用客户端应用程序的目录，在全局程序集缓存中安装程序集 shareddemo。这个程序集的另一种安装可以使用另一条路径安装，或使用 OPAQUE ID 安装，如下面的命令行所示：

```
gacutil /i shareddemo.dll /r OPAQUE 4711 "Opaque installation"
```

现在全局程序集缓存中只有一个程序集，但它有两个引用。为了从全局程序集缓存中删除程序集，必须删除这两个引用：

```
gacutil /u shareddemo /r OPAQUE 4711 "Opaque installation"
gacutil /u shareddemo /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

要删除共享程序集，选项/u 需要不带文件扩展名.DLL 的程序集。而安装共享程序集的选项/i 需要包含文件扩展名的完整文件名。

第 18 章介绍了程序集的部署，其中在 MSI 软件包中处理引用总数。

19.4.10　本机映像生成器

使用本机映像生成器 Ngen.exe，可以在安装期间把 IL 代码编译为本机代码。这样程序启动就比较快，因为不再需要在运行时进行编译。比较预编译的程序集和在其中需要运行 JIT 编译器的程序集，其性能在编译 IL 代码后没有太大区别。使用本机映像生成器获得的最大改进是应用程序的启动比较快，因为不需要运行 JIT。而且，因为 IL 代码已被编译，所以在运行期间也不需要 JIT。如果应用程序没有使用大量的 CPU 时间，那么可能看不到很显著的改进。减少应用程序的启动时间是使用本机映像生成器的主要原因。如果从可执行文件中创建本机映像，也应从可执行文件加载的所有 DLL 中创建本机映像。否则，就仍需要运行 JIT 编译器。

ngen 实用工具在本机映像缓存中安装本机映像，本机映像缓存的物理目录是<windows>\assembly\NativeImages<RuntimeVersion>。

使用 ngen install myassembly 可以把 MSIL 代码编译为本机代码，并把它安装到本机映像缓存中。如果要把程序集安装到本机映像缓存中，就应在安装程序中完成上述操作。

使用 ngen 和选项 display 还可以显示本机映像缓存中的所有程序集。如果把程序集名添加到选项 display 中，就可以得到依赖于这个程序集的所有程序集的信息。长长的列表之后，会显示已安装程序集的所有版本。

```
C:\>ngen display System.Core
Microsoft (R) CLR Native Image Generator - Version 4.0.30319.17626
Copyright (c) Microsoft Corporation. All rights reserved.

NGEN Roots that depend on "System.Core":

C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
AddInViews\Microsoft.VisualStudio.Tools.Applications.Runtime.v10.0.dll
C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
HostSideAdapters\Microsoft.VisualStudio.Tools.Office.Excel.HostAdapter.v10.0.
dll
C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
HostSideAdapters\Microsoft.VisualStudio.Tools.Office.HostAdapter.v10.0.dll
c:\Program Files (x86)\Microsoft Expression\Blend 4\
Microsoft.Windows.Design.Extensibility\
Microsoft.Windows.Design.Extensibility.dll
...

Native Images:

System.AddIn, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.AddIn, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

如果系统的安全设置有变化，就不能保证预编译的本机映像达到运行应用程序的安全要求。这就是本机映像在系统配置变化时就无效的原因。使用命令 ngen update，会重新生成所有的本机映像，以包含新配置。

安装.NET 4.5 时，会安装 Native Runtime Optimization Service。这个服务可以用于延迟本机映像的编译，重新生成已失效的本机映像。

安装程序可以使用 ngen install myassembly /queue 命令，通过本机映像服务延迟将 myassembly 编译为本机映像的过程。ngen update /queue 重新生成已失效的本机映像。使用 ngen queue 的选项 pause、continue 和 status，可以控制服务，并获取状态信息。

为什么不能在开发人员的系统中创建本机映像，而只能在产品系统中发布本机映像？因为本机映像生成器会处理与目标系统一起安装的 CPU，编译为 CPU 类型优化的代码。而在安装应用程序的过程中，CPU 是已知的。

19.5 配置.NET 应用程序

在 COM 之前，应用程序配置通常使用 INI 文件完成。而有了 COM 以后，则主要用注册表进行配置。所有的 COM 组件都是在注册表中进行配置的。IIS 的第一个版本也完全在注册表中进行配置。注册表的优势是可以在一个集中的位置进行各种配置。其缺点在于开放 API 可能使应用程序在不合适的位置添加配置值。而且，使用注册表配置，就不能使用 xcopy 部署了。IIS 的后续版本改为使用自定义二进制配置格式，只能通过 IIS 管理 API 访问。到了今天，IIS 则使用 XML 文件进行配置。XML 配置文件也是存储.NET 应用程序的配置值的首选位置。配置文件可以简单地复制。配置文件使用 XML 语法来指定应用程序的启动和运行库配置。

本节将介绍：

- 可以使用 XML 基本配置文件进行哪些配置
- 用强名引用的程序集如何重新定向到另一个版本中
- 如何指定程序集的目录，以便在子目录中查找私有程序集，在公共目录或服务器上查找共享程序集

19.5.1 配置类别

可以把配置分为如下几类：

- **启动设置**——用于指定需要的运行库版本。同一个系统上可能安装了不同版本的运行库，使用<startup>元素来指定运行库版本。
- **运行库设置**——用于指定运行库如何进行垃圾回收，如何进行程序集绑定。也可以使用这些设置指定版本策略和代码库(code base)。本章的后面将详细介绍运行库设置。
- **WCF 设置**——用于利用 WCF 配置应用程序。这些配置参见第 43 章。
- **安全设置**——详见第 22 章。第 22 章将介绍加密配置和许可。

这些设置可以在 3 种配置文件中给出：

- **应用程序配置文件**——包含应用程序的特定设置，如程序集的绑定信息，远程对象的配置等。这个配置文件放在可执行文件所在的目录下，它与可执行文件同名，但最后添加了.config扩展名。ASP.NET配置文件命名为web.config。
- **计算机配置文件**——可以用于系统范围的配置。也可以在这里指定程序集绑定和远程配置。在绑定过程中，应在考虑应用程序配置文件之前考虑计算机配置文件。应用程序配置可以重写计算机配置中的设置。应用程序配置文件应优先用于应用程序特定的设置，这样计算机配置文件会比较小，也容易管理。计算机配置文件位于%runtime_install_path%\config\Machine.config中。
- **发行者策略文件**——由组件的创建者用于指定共享程序集可以与旧版本兼容。如果新程序集版本仅修改了共享组件的一个错误，就不必把应用程序配置文件放在使用该组件的每个应用程序目录中，发行者可以添加一个发行者策略文件，从而把它标记为“可兼容的”。当组件不能用于所有的应用程序时，就可以在应用程序配置文件中重写发行者策略设置。与其他配置文件不同，发行者策略文件存储在全局程序集缓存中。

为了理解如何使用这些配置文件，回忆一下客户如何根据程序集是共享的还是私有的来查找程序集(也称为绑定)。私有程序集必须位于应用程序所在的目录或子目录下。进程probing可用于查找这样的程序集。如果程序集没有强名，probing就不使用版本号。

共享程序集可以安装在全局程序集缓存中，或放在一个目录、网络共享或Web站点上。我们用codeBase的配置来指定这样一个目录。在绑定共享程序集时，公钥、版本和文化都很重要。所需程序集的引用记录在客户端程序集的清单中，包括名称、版本和公钥标记。所有的配置文件都要检查，以应用正确的版本策略。全局程序集缓存和在配置文件中指定的代码库也要检查，然后检查应用程序的目录，之后应用探测规则。

19.5.2 绑定程序集

前面已经介绍了如何把共享程序集安装到全局程序集缓存中。除了把共享程序集安装到全局程序集缓存中，还可以使用配置文件配置特定的共享目录。如果要在服务器上使用共享组件，就可以使用这个功能。如果要在应用程序之间共享一个程序集，但不希望它在全局程序集缓存中共享它，就可以把该程序集放在一个共享目录中。

查找程序集的正确目录有两种方式：使用XML配置文件中的codeBase元素，或者使用probing元素。codeBase元素配置只可用于共享程序集，而probing元素可用于私有和共享程序集。

1. <codeBase>

使用应用程序配置文件可以配置<codeBase>元素。下面的应用程序配置文件把对程序集SharedDemo的搜索重定向为从网络上加载它：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
             publicKeyToken="f946433fdae2512d" />
        <codeBase version="1.0.0.0"
```

```
            href="http://www.christiannagel.com/WroxUtils/SharedDemo.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

<codeBase>元素有特性 version 和 href，使用 version 特性必须指定程序集的原始引用版本，使用 href 特性可以定义应从中加载程序集的目录。在本例中，使用 HTTP 协议所在的路径。使用 href="file://C:/WroxUtils/SharedDemo.dll"可以指定本地系统上的一个目录或一个共享。

2. <probing>

如果没有配置<codeBase>元素，程序集也没有存储在全局程序集缓存中，运行库就会利用 probing 元素来查找程序集。.NET 运行库会在应用程序目录或与所搜索程序集同名的子目录中查找文件扩展名为.dll 或.exe 的程序集。如果没有找到程序集，会继续搜索。可以在应用程序配置文件的<runtime>部分中，用<probing>元素配置搜索目录。使用.NET Framework 配置工具选择应用程序的属性，也可以很容易地完成这个 XML 配置。使用.NET Framework 配置工具中的搜索路径可以配置该探测所在的目录。

得到的 XML 文件包含如下条目：

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

<probing>元素只有一个必需的 privatePath 特性。这个应用程序配置文件告诉运行库，应在应用程序的根目录下搜索程序集，再在 bin 和 util 目录中搜索。这两个目录是应用程序根目录的子目录。不可能在应用程序根目录或其子目录的外部引用私有程序集。在应用程序根目录外部的程序集必须有一个共享名，并可以使用<codeBase>元素来引用。

19.6 版本问题

对于私有程序集，版本问题并不重要，因为被引用的程序集会与客户端一起复制。客户端使用其私有目录下的程序集。但是，共享程序集就不是这样。下面先看看共享时发生的一些传统问题。

使用共享组件，则表示多个客户端应用程序可以使用同一个组件。在用新版本更新共享组件时，新版本会中断已有客户端。我们不可能不发布新版本，因为我们需要按照客户的需求在已有组件的新版本中提供新功能。我们可以仔细编程，让程序保持向后兼容。但事情并不总是很顺利。

这个问题的解决方案是使用一种体系结构，它允许安装共享组件的不同版本，客户端使用它们在构建过程中引用的版本。这解决了许多问题，但并没有解决全部问题。如果从客户端引用的组件中检测到了一个错误，该怎么办？我们可以更新这个组件，并确保客户端使用新版本，而不是在构建过程中引用的那个版本。

因此，根据新版本中更改的类型，有时要使用更新的版本，有时则要使用旧一点的引用版本。.NET 体系结构允许这两种情况。在.NET 中，默认情况下使用原来引用的程序集。使用配置文件可以把引用重新定向到一个不同的版本。版本问题在绑定的体系结构中有非常关键的作用——客户端获得存储了其组件的正确程序集。

19.6.1 版本号

程序集的版本号由四部分组成，例如 1.1.400.3300，各部分分别是：

```
<Major>.<Minor>.<Build>.<Revision>.
```

这些号码根据应用程序配置来使用。

如果进行的改动与以前的版本不兼容，最好改变主版本号或次版本号；如果兼容，最好只改变内部版本号或修订版本号。这样就可以假定把程序集重定向到只修改了内部版本号和修订版本号的新版本中是安全的。

在 Visual Studio 2013 中，使用项目设置中的程序集信息可以指定程序集的版本号。项目设置将程序集特性[AssemblyVersion]写入 AssemblyInfo.cs 文件：

```
[assembly: AssemblyVersion("1.0.0.0")]
```

除了全部定义版本号四部分之外，还可以在第 3 或第 4 个位置放置星号：

```
[assembly: AssemblyVersion("1.0.*")]
```

在这个设置中，前两个数字指定主版本号和次版本号，星号表示内部版本号和修订版本号是自动生成的。内部版本号是自从 2000 年 1 月 1 日以来的天数，修订版本号表示自从当地时间的午夜开始的秒数除以 2。自动设置的版本号在开发期间很有帮助，但在发布之前，最好定义特定的版本号。

这个版本存储在程序集清单的.assembly 部分中。

在客户端应用程序中引用程序集，会在客户端应用程序的程序集清单中存储引用的程序集版本。

19.6.2 通过编程方式获取版本

要查看在客户端应用程序中使用的程序集的版本，可以在前面创建的 SharedDemo 类中添加只读属性 FullName，以返回程序集的强名。要简化 Assembly 类的使用，必须导入 System.Reflection 名称空间(代码文件 SharedDemo/SharedDemo.cs)。

```
public string FullName
{
  get
  {
    return Assembly.GetExecutingAssembly().FullName;
  }
}
```

Assembly 类的 FullName 属性包含了类名、版本、位置和公钥标记，在客户端应用程序中调用 FullName 时，可以在其结果中看到这些内容。

在客户端应用程序中，创建共享组件后，在 Main()方法中添加对 FullName 的一个调用(代码文件 Client/Program.cs)：

```
static void Main()
{
   var quotes = new SharedDemo("Quotes.txt");
   Console.WriteLine(quotes.FullName);
```

一定要使用 gacutil 在全局程序集缓存中再次注册共享程序集 SharedDemo 的新版本。如果没有找到引用的版本，就会抛出一个 System.IO.FileLoadException 异常，因为没有绑定到正确的程序集。

成功运行后，可以看到引用的程序集的完整名称，如下所示。

```
SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken= f946433fdae2512d
```

这个客户端程序可以用于测试这个共享组件的不同配置。

19.6.3 绑定到程序集版本

使用配置文件可以指定应绑定到共享程序集的另一个版本。假定创建共享程序集 SharedDemo 的一个新版本，其主版本号和次版本号是 1.1。我们不想重新建立客户端，只想在已有的客户端中使用程序集的新版本。这适用于下述场合：共享程序集有一个错误需要修改，或者因为新版本是兼容的，所以要删除旧版本。

运行 gacutil.exe，可以看到 SharedDemo 程序集安装了 1.0.0.0 和 1.0.3300.0 版本。

```
> gacutil -l SharedDemo

Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.17626
Copyright (c) Microsoft Corporation. All rights reserved.

The Global Assembly Cache contains the following assemblies:
  SharedDemo, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=f946433fdae2512d, processorArchitecture=x86
  SharedDemo, Version=1.0.3300.0, Culture=neutral,
  PublicKeyToken=f946433fdae2512d, processorArchitecture=x86

Number of items = 2
```

图 19-14 显示了客户端应用程序的清单，其中客户引用了程序集 SharedDemo 的 1.0.0.0 版本。

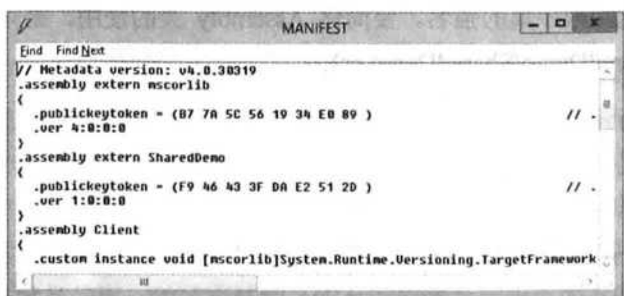


图 19-14

现在同样需要使用应用程序配置文件。与前面一样，重定向的程序集需要用<assemblyIdentity>元素指定。这个元素使用名称、文化和公钥标记标识程序集。为了重定向到另一个版本上，要使用

<bindingRedirect>元素。oldVersion 特性指定应把程序集的哪个版本重定向到新版本上。使用 oldVersion 特性可以指定一个范围，例如，应重定向 1.0.0.0~1.0.3300.0 之间所有程序集的版本。新版本用 newVersion 特性指定(配置文件 Client/App.config)。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
                          publicKeyToken="f946433fdae2512d" />
        <bindingRedirect oldVersion="1.0.0.0-1.0.3300.0"
                         newVersion="1.0.3300.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

19.6.4 发行者策略文件

使用全局程序集缓存器中的共享程序集，可以使用发行者策略避免版本冲突问题。假定有一个共享程序集由一些应用程序使用。如果在共享程序集中有一个关键的错误，会出现什么情况？前面看到，不需要重新建立所有使用该共享程序集的应用程序，因为可以使用配置文件重定向到这个共享程序集的新版本上。我们也许不了解所有使用该共享程序集的应用程序，但要为所有这些应用程序修改错误。此时可以创建发行者策略文件，把所有这些应用程序重新定向到该共享程序集的新版本上。

发行者策略文件只能应用于安装在全局程序集缓存中的共享程序集。

要建立发行者策略，必须：

- 创建发行者策略文件
- 创建发行者策略程序集
- 把发行者策略程序集添加到全局程序集缓存器中

1. 创建发行者策略文件

发行者策略文件是一个把已有版本或某个版本范围重定向到新版本的 XML 文件。这里使用的语法与应用程序配置文件相同，所以可以使用前面创建的文件，把旧版本 1.0.0.0~1.0.3300.00 重定向到新版本 1.0.3300.00 上。把前面创建的文件重命名为 mypolicy.config，把它用作发行者策略文件。

2. 创建发行者策略程序集

要把发行者策略文件与共享程序集关联起来，必须创建一个发行者策略程序集，并把它放到全局程序集缓存器中。可以创建这种文件的工具是程序集链接器 al。/linkresource 选项把发行者策略文件添加到生成的程序集中。生成的程序集的名称必须以 policy 开头，其后是应重定向的程序集的主次

版本号，以及共享程序集的文件名。在本例中，发行者策略程序集必须命名为policy.1.0.SharedDemo.dll，才能重定向主版本号为1、次版本号为0的SharedDemo程序集。必须用/keyfile选项把一个密钥添加到这个发行者密钥中，新添加的这个密钥与用于签名共享程序集 SharedDemo 的密钥相同，这样才能保证该版本从同一个发行者重定向。

```
al /linkresource:mypolicy.config /out:policy.1.0.SharedDemo.dll
/keyfile:.\.\mykey.snk
```

3. 将发行者的策略程序集添加到全局程序集缓存中

现在，可以使用实用工具gacutil把发行者策略程序集添加到全局程序集缓存中：

```
gacutil -i policy.1.0.SharedDemo.dll
```

如果已发布了同一个策略文件，就可以使用选项-f。现在，可以删除位于客户端应用程序目录中的应用程序配置文件，并启动该客户端应用程序。尽管客户端程序集引用的是1.0.0.0版本，但因为有了发行者策略，所以我们使用共享程序集的新版本1.0.3300.0。

4. 重写发行者策略

有了发行者策略，共享程序集的发行者就可以保证程序集的新版本与旧版本兼容。从传统DLL的变化来看，这种保证并不总可靠。也许只有一个应用程序在使用新的共享程序集。为了修改使用新版本的应用程序中的错误，可以使用应用程序配置文件，重写发行者策略。

添加 XML<publisherPolicy>元素和 apply="no"特性，就可以禁用发行者策略(配置文件 Client/App.config)。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
                          publicKeyToken="f946433fdae2512d" />
        <publisherPolicy apply="no" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

禁用发行者策略后，可以在应用程序配置文件中配置不同版本的重定向。

19.6.5 运行库的版本

不仅可以安装和使用程序集的多个版本，还可以安装和使用.NET运行库(CLR)的多个版本。CLR的1.0、1.1、2.0和4.0(及未来)版本可以同时安装在一个操作系统上。Visual Studio 2012面向通过.NET 2.0、3.0、3.5在CLR 2.0上运行的应用程序，以及通过.NET 4、4.5在CLR 4.0上运行的应用程序。

如果应用程序是用CLR 2.0构建的，可能不做修改就可以在只安装了CLR 4.0版本的系统上运行它。但是，如果应用程序是用CLR 4.0构建的，就不能在只安装了CLR 2.0版本的系统上运行它。

在应用程序配置文件中，不仅可以重定向被引用的程序集的版本，还可以定义运行库所需的版

本。在应用程序配置文件中可以指定应用程序需要的版本。<supportedRuntime>元素标记应用程序支持的运行库版本。<supportedRuntime>的顺序定义系统上可用的多个运行库版本的优先级。这里的配置首选.NET 4 运行库，也支持 2.0。为此，构建的应用程序必须面向.NET Framework 2.0、3.0 或 3.5。

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" />
    <supportedRuntime version="v2.0.50727" />
  </startup>
</configuration>
```

作为可选项，可以使用 sku 特性定义 SKU。SKU 定义了.NET Framework 的版本，例如 4.0 with SP1，或者客户端配置文件。下面的代码段需要.NET 4.5 版本：

```
<supportedRuntime version="v4.0" sku=".NET Framework,Version=4.5" />
```

为指定.NET 4.0 with SP2 的客户端配置文件，需要指定此字符串：

```
.NET Framework,Version=4.02,Profile=Client
```

在注册表键 HKLM\SOFTWARE\Microsoft\.NETFramework\v4.0.30319\SKUs 中可以找到全部可用的 SKU。

19.7 在不同的技术之间共享程序集

并不是只能在.NET 应用程序之间共享程序集，在不同的技术之间(例如.NET 和 Windows Store 应用程序之间)，也是可以共享代码或程序集的。本节将介绍各种可用的选项，以及各自的优缺点。具体的需求将决定在具体的环境中哪种选项最合适。

19.7.1 共享源代码

第一种选项实际上并不是共享程序集的某种形式，而是共享源代码。在不同的技术之间共享源代码时，可以使用 C#预处理器指令定义条件编译符号，如下面的代码段所示。在这里，方法 PlatformString 返回一个字符串，这个字符串根据是否定义了 SILVERLIGHT、NETFX_CORE 或者二者均未定义而发生变化：

```
    public string PlatformString()
    {
#if SILVERLIGHT
      return "Silverlight";
#elif NETFX_CORE
      return Windows Store App
#else
      return "Default";
#endif
    }
```

可以在一个普通的.NET 库中定义具有平台依赖性的代码。对于其他库，例如 Windows Store App

的类库或者 Silverlight 5 类库，符号的定义方式如图 19-15 所示，这里使用了一个 Windows Store App 的类库。

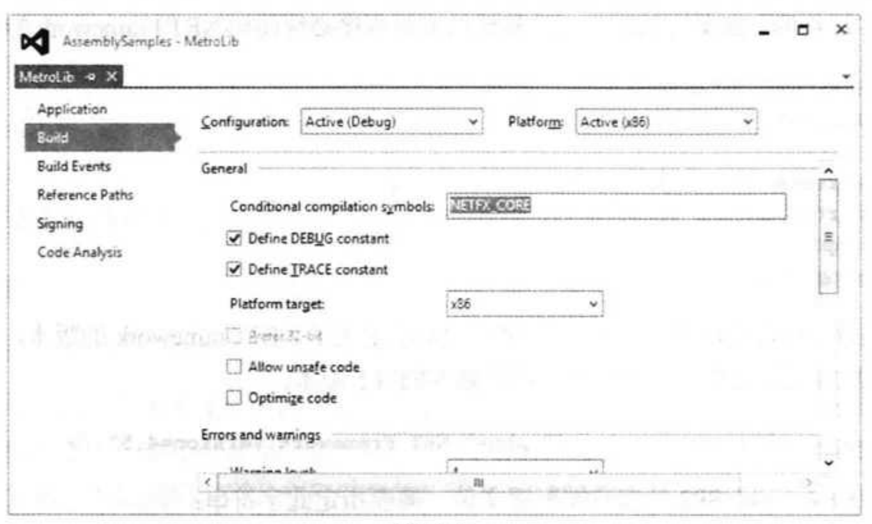


图 19-15

对于其他项目，可以在 Solution Explorer 中使用 Add as Link 选项添加已有项。这样一来，源代码只存在一次，而添加了该链接的所有项目都可以编辑这些代码。Visual Studio 编辑器根据打开并编辑文件的项目突出显示了当前活动的#if 块中的代码。在图 19-16 中，3 个不同的项目都链接了同一个文件 Demo.cs。链接在 Solution Explorer 中以不同的图标显示。

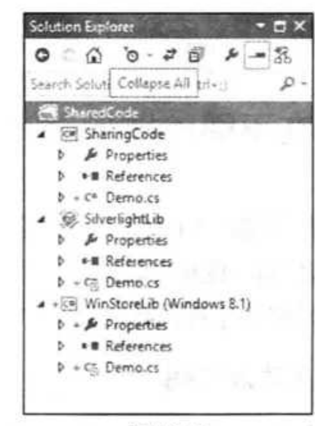


图 19-16

在共享源代码时，每种项目类型都可以充分利用源代码的功能。但是，必须定义不同的代码片段来处理不同项目类型的差别。为此，可以使用预处理器指令来处理方法的不同实现、不同的方法甚至整个类型的不同实现。

19.7.2 可移植类库

使用可移植类库可以共享二进制程序集而不是源代码。Visual Studio 2013 为创建可移植类库提供了一个新的模板。在这个库中，可以配置多个目标框架，如图 19-17 所示。这里选择的目标框架是.NET Framework 4.5 和.NET for Windows Store apps。选定框架可以使用所有的引用、类和方法。

如果选择了所有的框架，那么可以使用的类就非常有限。可用的类和类成员会显示在 Object Browser 中，如图 19-18 所示。

图 19-17

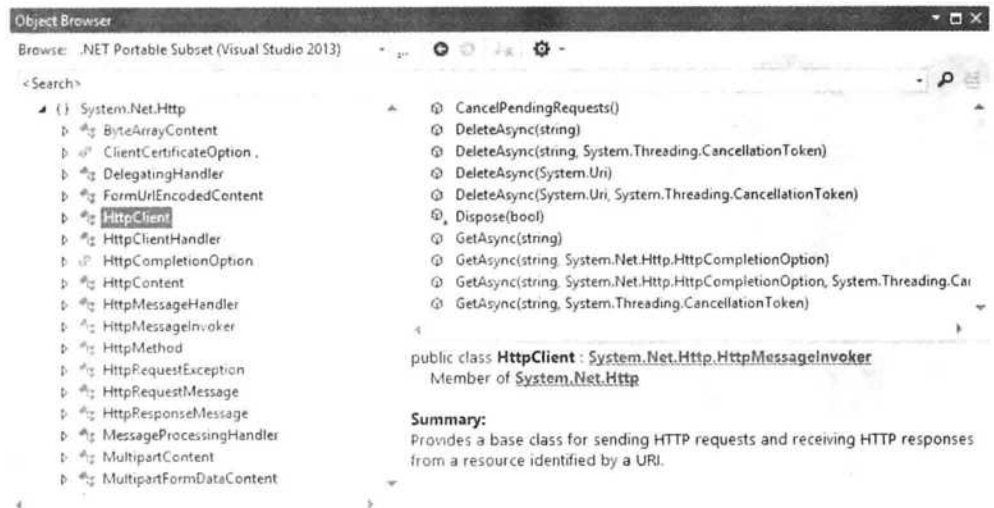


图 19-18

例如，使用.NET Framework 4.5 和.NET for Metro style apps 时，可以使用 MEF 和 WCF 的一个子集。WPF、Windows Forms、ASP.NET 和 ADO.NET 中的类都是不可用的。可以在可移植库中创建一个视图模型，用于 MVVM 模式。使用可移植库时，视图模型类不能使用引用了 ADO.NET 的库。当然，在 Windows 应用程序中使用数据库是很常见的场景。对于这种情况，可以使用一些访问数据库的服务器端代码，并使用一种通信协议来访问服务。

MVVM 模式(Model-View-ViewModel)使用一个中间层(view-model)将用户界面(view)与数据(model)分隔开。这种模式常用在 WPF 应用程序中。

19.8　小结

程序集是.NET 平台的安装单元。Microsoft 从以前的体系结构(如 COM)中吸取教训，进行了全新的设计，以避免出现旧问题。本章讨论了程序集的特性：程序集是自描述的，不需要类型库和注册信息。

因为版本的依赖性会准确地记录下来，这样，使用程序集时，旧 DLL 的问题就不复存在了。由于具备这些特性，因此开发、部署和管理就容易多了。

本章讨论了私有和共享程序集之间的差异，并介绍了如何创建共享程序集。对于私有程序集，不必关心唯一性和版本冲突问题，因为这些程序集仅由一个应用程序复制和使用。共享程序集需要使用一个密钥来保持其唯一性、确定其版本。我们介绍了全局程序集缓存，它可以用作共享程序集的智能存储器。

使用本机映像生成器可以更快地启动应用程序。有了本机映像生成器，就不需要运行 JIT 编译器，因为本机代码是在安装期间创建的。

本章还阐述了避免版本冲突问题，以使用与在开发过程中使用的版本不同的程序集。这通过发行者策略和应用程序配置文件实现。最后还讨论了 probing 元素如何使用私有程序集。

我们还讨论了如何动态地加载程序集，在运行期间创建程序集。如果要了解更多的信息，就可以参阅第 30 章介绍的.NET 4 中的插件模型。

第 20 章将介绍应用程序的诊断，即如何在开发环境和生产环境中找出应用程序失败的原因。

第20章

诊　　断

本章要点

- 代码协定
- 跟踪
- 事件日志
- 性能监视

本章源代码下载地址(wrox.com):

打开网页 http://www.wrox.com/go/prosharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Code Contracts
- Tracing Demo
- Tracing Demo with EventLog
- EventSource Sample
- Event Log
- Event Log Reader
- Performance Counter

20.1　诊断概述

本章介绍如何获得关于正在运行的应用程序的实时信息，找出应用程序在生产过程中某些问题的原因，或者监视需要的资源，以尽早适应较高的用户负载。这就是名称空间 System.Diagnostics 的作用。这个名称空间提供了用于跟踪、事件日志、性能测量以及代码协定的类。

当然，在应用程序中标记错误的一种方式是抛出异常。然而，有可能应用程序不抛出异常，但仍不像期望的那样运行。应用程序可能在大多数系统上都运行良好，只在几个系统上出问题。在实时系统上，改变配置值可以改变日志行为，获得应用程序运行状况的详细实时信息。这可以用跟踪

功能来实现。

如果应用程序出了问题，就需要通知系统管理员。事件查看器是一个常用的工具，并不是只有系统管理员才需要使用它。使用事件查看器可以交互地监视应用程序的问题，通过添加订阅功能来了解发生的特定事件。事件日志机制允许写入应用程序的相关信息。

为了分析应用程序需要的资源，在指定的时间间隔内(例如每 5 分钟)监视应用程序，这样就可以获得 24 小时或者一周的数据，而不必填写数以 TB 计的数据，并且还可以规划另一个应用程序的分布或系统资源的扩展。性能监视器可以用来获得这些数据。使用性能计数器可以写入来自应用程序的实时数据。

按协定设计是.NET Framework 提供的另一项功能。按协定设计的特征是使用方法的签名定义参数的类型，但是不提供可以把哪些值传递给方法的信息。使用 System.Diagnostics.Contracts 名称空间中的类可以定义前提条件、后置条件和常量，它们不仅可以在运行期间检查，还可以使用静态的协定分析器检查。本章介绍这些功能，并说明如何在应用程序中使用它们。

20.2 代码协定

按协定设计是 Eiffel 编程语言的一个理念，定义了前提条件、后置条件和常量。现在，.NET 在 System.Diagnostics.Contracts 名称空间中包含的类可用于静态检查代码和在运行期间检查代码，这些类可由所有的.NET 语言使用。利用这个功能可以定义方法中的前提条件、后置条件和常量。前提条件列出了参数必须满足的要求，后置条件定义了返回数据必须满足的要求，常量定义了方法中变量必须满足的要求。

协定信息可以编译到调试代码和发布代码中。还可以定义一个单独的合同程序集，也可以进行许多静态检查，而无须运行应用程序。也可以在接口上定义协定，使接口的实现代码满足协定的要求。协定工具可以重写程序集，在运行库检查的代码中插入协定检查，在编译期间检查协定，在生成的 XML 文档中添加协定信息。

图 20-1 显示了 Visual Studio 2013 中代码协定的项目属性。这里可以定义应进行什么级别的运行库检查，指定在协定失败时是否打开断言对话框，配置静态检查。把 Perform Runtime Contract Checking 设置为 Full 来定义 CONTRACTS_FULL 符号。因为许多协定方法都用[Confitional("CONTRACTS_FULL")]特性注解，所以所有的运行库检查都只使用这个设置。

要使用代码协定，可以使用.NET 4 在 System.Diagnostics.Contracts 名称空间中可用的类。但 Visual Studio 2013 没有包含这方面的工具。需要从 Microsoft Research 上下载 NuGet 包 Code Contracts Editor Extensions。

代码协定用 Contract 类定义。在方法中定义的所有协定要求，无论是前提条件还是后置条件，都必须放在方法的开头。也可以给 ContractFailed 事件赋予一个全局的事件处理程序，运行期间失败的每个协定都会调用这个事件处理程序。调用 SetHandled()方法时包含 ContractFailedEventArgs 类型的参数 e，会禁止失败时的标准行为：抛出异常(代码文件 CodeContractSamples/Program.cs)。

图 20-1

```
Contract.ContractFailed += (sender, e) =>
{
  Console.fdhWriteLine(e.Message);
  e.SetHandled();
};
```

20.2.1 前提条件

前提条件检查传递给方法的参数。使用 Contract 类的 Requires()方法可以定义前提条件。使用 Requires()方法时，必须传送一个布尔值，第二个参数是一个可选的消息字符串，当条件不满足时显示该消息。下面的示例要求，参数 min 应小于等于参数 max。

```
static void MinMax(int min, int max)
{
  Contract.Requires(min <= max);
  //...
}
```

使用 Requires 方法的泛型变体可以指定当条件不满足时调用的异常类型。如果参数 o 是空，下面的协定就抛出一个 ArgumentNullException 异常。如果事件处理程序把 ContractFailed 事件设置为 handled，就不抛出该异常。另外，如果配置了 Assert on Contract Failure，就执行 Trace.Assert()方法停止程序，而不是抛出所定义的异常。

```
static void Preconditions(object o)
{
  Contract.Requires<ArgumentNullException>(o != null,
     "Preconditions, o may not be null");
  //...
```

因为 Requires<TException>()方法没有用[Conditional("CONTRACTS_FULL")]特性注解，并且在 DEBUG 符号上它也没有条件，所以这个运行库检查可以在任意情况下进行。如果条件不满足，Requires<TException>()方法就抛出所定义的异常。

为了检查用作参数的集合，Contract 类提供了 Exists()和 ForAll()方法。ForAll()方法检查集合中的每一项，看看它们是否满足条件。在示例中，检查集合中的每一项的值是否小于 12。使用 Exists()方法可以检查集合中的任意一项是否满足条件。

```
static void ArrayTest(int[] data)
{
  Contract.Requires(Contract.ForAll(data, i => i < 12));
```

Exists()和 ForAll()方法都有一个重载版本，其中可以给该重载版本传递两个整数 fromInclusive 和 toExclusive，而不是传递 IEnumerable<T>。把一个数值范围(除去 toExclusive)传递给第 3 个参数定义的谓词。Exists()和 ForAll()方法可以用于前提条件、后置条件和常量。

20.2.2 后置条件

后置条件定义了方法执行完后共享数据和返回值的保证。尽管后置条件定义了关于返回值的一些保证，但它们必须放在方法的开头；所有的协定要求都必须放在方法的开头。

Ensures()和 EnsuresOnThrow<TException>()方法是后置条件。下面的协定确保变量 sharedState 在方法执行完后小于 6。该值在方法执行期间可以改变。

```
private static int sharedState = 5;
static void Postcondition()
{
  Contract.Ensures(sharedState < 6);

  sharedState = 9;
  Console.WriteLine("change sharedState invariant {0}", sharedState);
  sharedState = 3;
  Console.WriteLine("before returning change it to a valid value {0}",
     sharedState);
}
```

使用 EnsuresOnThrow<TException>()方法，可以保证如果抛出了指定的异常，共享状态就满足某条件。

为了保证返回某个值，可以对 Ensure()方法的协定使用特定的值 Result<T>。这里的结果是 int 类型，因为它是用泛型类型 T 给 Result()方法定义的。Ensure()方法的协定保证返回值小于 6。

```
static int ReturnValue()
{
  Contract.Ensures(Contract.Result<int>() < 6);
  return 3;
}
```

还可以比较新旧值。为此应使用 OldValue<T>()方法，它返回在方法入口给变量传递的初始值。下面的协定确保，返回的结果大于参数 x 中的旧值。

```
static int ReturnLargerThanInput(int x)
{
  Contract.Ensures(Contract.Result<int>() > Contract.OldValue<int>(x));
  return x + 3;
}
```

如果方法的返回值用 out 修饰符来修饰，而不仅仅使用 return 语句，就可以用 ValueAtResult()方法定义条件。下面的协定指定，变量 x 在返回时必须大于 5 且小于 20，变量 y 与 5 的取模结果在返回时必须等于 0。

```
static void OutParameters(out int x, out int y)
{
  Contract.Ensures(Contract.ValueAtReturn<int>(out x) > 5 &&
    Contract.ValueAtReturn<int>(out x) < 20);
  Contract.Ensures(Contract.ValueAtReturn<int>(out y) % 5 == 0);
  x = 8;
  y = 10;
}
```

20.2.3 不变量

不变量为对象生命周期中的变量定义了协定。Contract.Requires()方法定义了输入要求，Contract.Ensures()方法定义了方法结束时的要求。Contract.Invariant()方法定义了在对象的整个生命周期中都必须满足的条件。

下面的代码段对成员变量 x 进行不变量检查，要求 x 必须大于 5。x 被初始化为 10，这是符合协定的。对 Contract.Invariant 的调用只能放在应用了 ContractInvariantMethod 特性的方法内。这个方法可以是公有或私有的，可以有任何名称(建议名称为 ObjectInvariant)，并且只能包含对协定不变量进行检查的代码。

```
private int x = 10;

[ContractInvariantMethod]
private void ObjectInvariant()
{
  Contract.Invariant(x > 5);
}
```

不变量检查总是在公有方法的末尾进行的。在接下来的例子中，方法 Invariant 将 3 赋值给变量 x，这导致在方法的末尾不符合协定要求：

```
public void Invariant()
{
  x = 3;
  Console.WriteLine("invariant value: {0}", x);

  // contract failure at the end of the method
}
```

20.2.4 纯粹性

在协定方法中可以使用自定义方法，但是这些方法必须是纯粹的方法。纯粹指的是自定义方法不会修改对象的任何可见状态。

通过应用Pure特性，可以把方法和类型标记为纯粹的方法。默认情况下，属性的get访问器被认为是纯粹的方法。代码协定工具的当前版本对纯粹性没有强制性要求。

20.2.5 接口的协定

对于接口，可以定义方法、属性和事件，派生自该接口的类必须实现这些方法、属性和事件。在接口的声明中，不能定义接口的实现方式，现在可以使用代码协定来实现。

看看下面的例子。IPerson接口定义了FirstName、LastName和Age属性以及ChangeName()方法。这个接口的特殊之处是ContractClass特性。这个特性应用于IPerson接口，并指定PersonContract类用作这个接口的代码协定(代码文件CodeContractsSamples/IPerson.cs)。

```
[ContractClass(typeof(PersonContract))]
public interface IPerson
{
  string FirstName { get; set; }
  string LastName { get; set; }
  int Age { get; set; }
  void ChangeName(string firstName, string lastName);
}
```

PersonContract类实现了IPerson接口，并为所有的成员定义了代码协定。这用属性的get访问器定义，还可以用不允许改变状态的所有方法来定义。FirstName和LastName属性的get访问器也用Contract.Result()方法指定，结果必须是一个字符串。Age属性的get访问器定义了一个后置条件，确保返回值在0～120之间。FirstName和LastName属性的set访问器要求，传递的值非空。Age属性的set访问器定义了一个前提条件，要求传递的值在0～120之间(代码文件CodeContractsSamples/PersonContract.cs)。

```
[ContractClassFor(typeof(IPerson))]
public abstract class PersonContract : IPerson
{
  string IPerson.FirstName
  {
    get { return Contract.Result<String>(); }
    set { Contract.Requires(value != null); }
  }
  string IPerson.LastName
  {
    get { return Contract.Result<String>(); }
    set { Contract.Requires(value != null); }
  }
  int IPerson.Age
  {
    get
    {
      Contract.Ensures(Contract.Result<int>() >= 0 &&
                       Contract.Result<int>() < 121);
```

```
      return Contract.Result<int>();
    }
    set
    {
      Contract.Requires(value >= 0 && value < 121);
    }
  }
  void IPerson.ChangeName(string firstName, string lastName)
  {
    Contract.Requires(firstName != null);
    Contract.Requires(lastName != null);
  }
}
```

现在实现IPerson接口的类必须满足所有的协定要求。Person类是满足协定的接口的一个简单实现(代码文件CodeContractsSamples/Person.cs)。

```
public class Person : IPerson
{
  public Person(string firstName, string lastName)
  {
    this.FirstName = firstName;
    this.LastName = lastName;
  }

  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public int Age { get; set; }

  public void ChangeName(string firstName, string lastName)
  {
    this.FirstName = firstName;
    this.LastName = lastName;
  }
}
```

使用Person类时，也必须满足协定。例如，不允许给属性赋予空值：

```
var p = new Person { FirstName = "Tom", LastName = null };
// contract error
```

也不允许给Age属性赋予无效值：

```
var p = new Person { FirstName = "Tom", LastName = "Turbo" };
p.Age = 133; // contract error
```

20.2.6 简写

.NET 4.5为代码协定提供了一项新功能，即简写。如果某个协定被重复使用，就可以利用一个重用机制。向包含多个协定的方法应用ContractAbbreviator特性，然后就可以在需要此协定的其他方法内使用这个方法：

```
[ContractAbbreviator]
private static void CheckCollectionContract(int[] data)
```

```
{
  Contract.Requires<ArgumentNullException>(data != null);
  Contract.Requires(Contract.ForAll(data, x => x < 12));
}
```

现在就可以在一个方法内使用 CheckCollectionContract，检查参数的值是否为空，集合中的每个值是否小于 12：

```
private static void Abbrevations(int[] data)
{
  CheckCollectionContract(data);
}
```

20.2.7 协定和遗留代码

有大量遗留代码时，经常会使用 if 语句检查参数，如果某个条件没有满足，就抛出一个异常。使用代码协定时，不必重写验证，只要添加一行代码就可以了：

```
static void PrecondtionsWithLegacyCode(object o)
{
  if (o == null) throw new ArgumentNullException("o");
Contract.EndContractBlock();
```

EndContractBlock 指定前面的代码作为协定来处理。如果还使用了其他协定语句，就没有必要使用 EndContractBlock 了。

在遗留代码中使用程序集时，如果使用了代码协定配置，必须把程序集模式设为 Custom Parameter Validation。

20.3 跟踪

利用跟踪功能可以从正在运行的应用程序中查看消息。为了获得关于正在运行的应用程序的信息，可以在调试器中启动应用程序。在调试过程中，可以单步执行应用程序，在特定的代码行上设置断点，并在满足某些条件时设置断点。调试的问题是包含发布代码的程序与包含调试代码的程序以不同的方式运行。例如，程序在断点处停止运行时，应用程序的其他线程也会挂起。另外，在发布版本中，编译器生成的输出进行了优化，因此会产生不同的效果。在经过优化的发布代码中，垃圾回收要比在调试代码中更加积极。方法内的调用次序可能发生变化，甚至一些方法会被彻底删除，改为就地调用。此时也需要从程序的发布版本中获得信息。跟踪消息要写入调试代码和发布代码中。

下面的场景描述了跟踪功能的作用。在部署应用程序后，它运行在一个系统中时没有问题，而在另一个系统上很快出现了问题。在出问题的系统上打开详细的跟踪功能，就会获得应用程序中所出现问题的详细信息。在运行没有问题的系统上，将跟踪功能配置为把错误消息重定向到 Windows 事件日志系统中。系统管理员会查看重要的错误，跟踪功能的系统开销非常小，因为仅在需要时配

置跟踪级别。

跟踪体系结构有 4 个主要部分：

- 源是跟踪信息的源头，使用源可以发送跟踪消息。
- 开关定义了要记录的信息级别。例如，可以只请求错误信息或详细的信息。
- 跟踪侦听器定义了写入跟踪消息的位置。
- 侦听器可以有关联的筛选器。筛选器定义了侦听器应写入哪些跟踪消息。这样，就可以给同一个源头使用不同的侦听器，写入不同级别的信息。

图 20-2 显示了主要的跟踪类和它们在 Visual Studio 类图中的连接方式。TraceSource 类使用一个开关来定义要记录的信息。TraceSource 类有一个相关联的 TraceListenerCollection 类，它指定写入跟踪消息的位置。该集合由 TraceListener 对象组成，每个侦听器都连接到一个 TraceFilter 类。

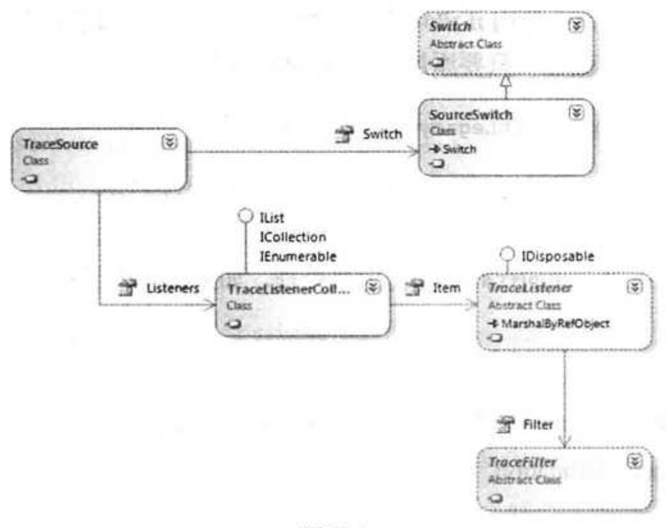


图 20-2

有几种.NET 技术使用跟踪源，只需要打开跟踪源，就能看到发生了什么情况。例如，WPF 定义的源有 System.Windows.Data、System.Windows.RoutedEvent、System.Windows.Markup、System.Windows.Media.Animation。而在 WPF 中，打开跟踪功能时，不仅需要配置侦听器，还要在 HKEY_CURRENT_USER\Software\MicrosoftTracing\WPF 注册键中把一个新的 DWORD 设置为 ManagedTracing，其值设置为 1。另一种方法就是通过编程打开跟踪功能。

System.Net 名称空间中的类使用跟踪源 System.Net，WCF 使用跟踪源 System.ServiceModel 和 System.ServiceModel.MessageLogging。关于 WCF 跟踪的内容参见第 43 章。

20.3.1 跟踪源

用 TraceSource 类可以写入跟踪消息。跟踪需要编译器设置的 Trace 标志。在 Visual Studio 项目中，Trace 标志在调试版本和发布版本中已进行了默认设置，但可以通过项目的 Build 属性修改它。

与写入跟踪消息的 Trace 相比，TraceSource 类较难使用，但它提供的选项较多。

要写入跟踪消息，需要创建一个新的 TraceSource 实例。在构造函数中，定义了跟踪源的名称。TraceInformation()方法在跟踪输出中写入一条信息型消息。TraceEvent()方法不只写入信息型消息，还需要一个 TraceEventType 类型的枚举值，来定义跟踪消息的类型。TraceEventType.Error 把消息指定为一条错误消息。可以用跟踪开关将它定义为只查看错误消息。

TraceEvent()方法的第二个参数需要一个标识符。ID 可以用于应用程序自身。例如，可以使用 id 1 进入某个方法，用 id 2 退出某个方法。因为 TraceEvent()方法是重载的，所以只需要 TraceEventType 和 ID 这两个参数。使用重载方法的第 3 个参数，可以传递写入跟踪输出的消息。TraceEvent()方法还可以传递格式字符串和任意数量的参数，其方式与 Console.WriteLine()方法相同。TraceInformation()方法只是调用标识符为 0 的 TraceEvent()方法。TraceInformation()方法是 TraceEvent()方法的一个简化版本。在 TraceData()方法中，可以传递任意对象，例如，异常实例，来替代消息。

要确保数据由侦听器写入，且不存储在内存中，就需要执行 Flush()方法。如果不再需要跟踪源，就可以调用 Close()方法，它关闭与跟踪源相关的所有侦听器。Close()方法也会执行 Flush()方法(代码文件 TracingDemo/Program.cs)。

```
public class Program
{
  internal static TraceSource trace =
     new TraceSource("Wrox.ProCSharp.Instrumentation");

  static void TraceSourceDemo1()
  {
    trace.TraceInformation("Info message");

    trace.TraceEvent(TraceEventType.Error, 3, "Error message");
    trace.TraceData(TraceEventType.Information, 2, "data1", 4, 5);
    trace.Close();
  }
```

可以在应用程序中使用不同的跟踪源。为不同的库定义不同的跟踪源，这样就可以为应用程序的不同部分打开不同的跟踪级别。要使用跟踪源，必须知道它的名称。跟踪源的常用名称与程序集的名称相同。

TraceEventType 枚举作为一个参数传递给 TraceEvent()方法，该枚举定义了下面的级别，来指定问题的严重程度：Verbose、Information、Warning、Error 和 Critical。Critical 定义了致命错误或使应用程序崩溃的错误；Error 表示可恢复的错误。Verbose 级别的跟踪消息可给出详细的调试信息。TraceEventType 还定义了操作级别：Start、Stop、Suspend 和 Resume。这些级别在逻辑操作中定义了及时事件。现在编写的代码还没有显示任何跟踪消息，因为与跟踪源相关的开关是关闭的。

20.3.2 跟踪开关

要启用或禁用跟踪消息，可以配置一个跟踪开关。跟踪开关是派生自抽象基类 Switch 的类。派生类是 BooleanSwitch、TraceSwitch 和 SourceSwitch。BooleanSwitch 类可以打开和关闭，其他两个类提供了由 SourceLevels 枚举定义的范围级别。要配置跟踪开关，必须知道与 SourceLevels 枚举相关的值。SourceLevels 定义的值有 Off、Error、Warning、Info 和 Verbose。

设置 TraceSource 类的 Switch 属性，可以用编程方式关联跟踪开关。这里关联的开关是 SourceSwitch 类型，其名称是 Wrox.ProCSharp.Diagnostics，级别为 Verbose。

```
internal static SourceSwitch traceSwitch =
    new SourceSwitch("Wrox.ProCSharp.Diagnostics")
    { Level = SourceLevels.Verbose };
internal static TraceSource trace =
    new TraceSource("Wrox.ProCSharp.Diagnostics")
    { Switch = traceSwitch };
```

把级别设置为 Verbose，表示应写入所有的跟踪消息。如果把值设置为 Error，就应只显示错误消息。把值设置为 Information，表示显示错误、警告和信息型消息。在 Output 窗口中运行调试器，再次写入跟踪消息，就可以看到这些消息。

通常，要改变开关级别，而不希望重新编译应用程序，则最好只改变配置。跟踪源可以在应用程序配置文件中配置。跟踪的配置在<system.diagnostics>元素中完成。跟踪源在<source>元素中定义为<sources>的一个子元素。配置文件中跟踪源的名称必须精确匹配程序代码中跟踪源的名称。这里跟踪源的开关类型是 System.Diagnostics.SourceSwitch，名称是 MySourceSwitch。该开关在<switches>部分中定义，开关的级别设置为 Verbose(代码文件 TracingDemo/App.config)。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Diagnostics" switchName="MySourceSwitch"
          switchType="System.Diagnostics.SourceSwitch" />
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

现在，要修改跟踪级别，只需要修改配置文件，而无须重新编译代码。修改配置文件后，必须重新启动应用程序。

20.3.3 跟踪侦听器

默认情况下，把跟踪信息写入 Visual Studio 调试器的 Output 窗口中。只要改变应用程序的配置，就可以将跟踪输出重定向到不同的位置。

跟踪信息应写入什么位置由跟踪侦听器确定。跟踪侦听器派生自抽象基类 TraceListener。.NET 包含的几个跟踪侦听器会把跟踪事件写入不同的目标位置。对于基于文件的跟踪侦听器，使用基类 TextWriterTraceListener，写入 XML 文件的派生类 XmlWriterTraceListener，以及写入有分隔符的文

件的派生类 DelimitedListTraceListener。写入事件日志可使用 EventLogTraceListener 类或 EventProviderTraceListener 类来完成。EventProviderTraceListener 类使用自 Windows Vista 以来新增的事件文件格式。还可以合并 Web 跟踪和 System.Diagnostics 跟踪功能，使用 WebPageTraceListener 把 System.Diagnostics 跟踪信息也写入 Web 跟踪文件 trace.axd 中。

.NET Framework 发布了许多可写入跟踪信息的侦听器。如果侦听器不满足用户的需要，就可以从基类 TraceListener 中派生一个类，从而创建自定义侦听器。使用自定义侦听器，可以将跟踪信息写入 Web 服务，将消息写入手机等。其实手机在闲暇时间接收到数百条消息并没有那么有趣，而且使用 Verbose 跟踪级别，这会变得相当昂贵。

创建一个侦听器对象，并将它赋予 TraceSource 类的 Listeners 属性，就可以用编程方式配置跟踪侦听器。但是，只改变配置，就定义另一个侦听器会比较有趣。

可以把侦听器配置为<source>元素的子元素。在侦听器中，可以定义侦听器类的类型，并使用 initializeData 指定侦听器的输出应写入什么位置。下面的配置将 XmlWriterTraceListener 类定义为写入 demotrace.xml 文件中，并将 DelimitedListTraceListener 类定义为写入 demotrace.txt 文件中(代码文件 TracingDemo/App.config)。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Diagnostics" switchName="MySourceSwitch"
          switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
              type="System.Diagnostics.XmlWriterTraceListener"
              traceOutputOptions="None"
              initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" delimiter=":"
              type="System.Diagnostics.DelimitedListTraceListener"
              traceOutputOptions="DateTime, ProcessId"
              initializeData="c:/logs/mytrace.txt" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

在侦听器中，还可以指定把哪些额外信息写入跟踪日志中。这些信息和 XML 特性 traceOutputOptions 一起由 TraceOptions 枚举定义。该枚举定义了 Callstack、DateTime、LogicalOperationStack、ProcessId、ThreadId 和 None。需要的信息可以用逗号分隔符添加到 XML 特性 traceOutputOptions 中，如带分隔符的跟踪侦听器所示。

下面是 DelimitedListTraceListener 类中带分隔符的文件输出，包括进程 ID 和日期/时间：

```
"Wrox.ProCSharp.Diagnostics":Start:0:"Main started"::7724:""::
"2012-05-11T14:31:50.8677211Z"::
"Wrox.ProCSharp.Diagnostics":Information:0:"Info message"::7724:"Main"::
```

```
"2012-05-11T14:31:50.8797132Z"::
"Wrox.ProCSharp.Diagnostics":Error:3:"Error message"::7724:"Main"::
"2012-05-11T14:31:50.8817119Z"::
"Wrox.ProCSharp.Diagnostics":Information:2::"data1","4","5":7724:"Main"::
"2012-05-11T14:31:50.8817119Z"::
```

XmlWriterTraceListener 类中的 XML 输出总是包含计算机名、进程 ID、线程 ID、消息、创建时间、源和活动 ID。其他字段，如调用栈、逻辑操作栈、时间戳等，都依赖于跟踪输出选项。

可以使用 XmlDocument 类、XPathNavigator 类和 XElement 类分析 XML 文件中的内容。这些类详见第 34 章。

如果侦听器应由多个跟踪源使用，就可以将侦听器配置添加到<sharedListeners>元素中，它独立于跟踪源。配置为共享侦听器的侦听器名称必须从跟踪源的侦听器中引用：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Diagnostics" switchName="MySourceSwitch"
          switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
              type="System.Diagnostics.XmlWriterTraceListener"
              traceOutputOptions="None"
              initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="delimitedListener" delimiter=":"
          type="System.Diagnostics.DelimitedListTraceListener"
          traceOutputOptions="DateTime, ProcessId"
          initializeData="c:/logs/mytrace.txt" />
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

20.3.4 筛选器

每个侦听器都有一个 Filter 属性，它定义了侦听器是否应写入跟踪消息。例如，多个侦听器可以与同一个跟踪源一起使用。其中一个侦听器将详细消息写入日志文件中，另一个侦听器将错误消息写入事件日志中。在侦听器写入跟踪信息之前，调用相关筛选器对象的 ShouldTrace()方法，确定是否应写入跟踪信息。

筛选器是派生自抽象基类 TraceFilter 的类。.NET 提供了两个已实现的筛选器：SourceFilter 和 EventTypeFilter。使用 SourceFilter 筛选器，可以指定只从特定的源中写入跟踪信息。EventTypeFilter

筛选器是对开关功能的扩展。使用开关，可以根据跟踪的严重程度，确定事件源是否应将跟踪消息写入侦听器中。如果写入跟踪消息，侦听器就可以使用筛选器确定是否应写入该消息。

改变的配置现在确定，只有严重程度为警告或更高，带分隔符的侦听器才应写入跟踪消息，因为定义了 EventTypeFilter 筛选器。XML 侦听器指定了 SourceFilter 筛选器，只接收源 Wrox.ProCSharp.Tracing 中的跟踪消息。如果定义了大量跟踪源，以便将跟踪消息写入同一个侦听器中，就可以修改该侦听器的配置，只处理指定源中的跟踪消息。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
          switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
          <add name="delimitedListener" delimiter=":"
              type="System.Diagnostics.DelimitedListTraceListener"
              traceOutputOptions="DateTime, ProcessId"
              initializeData="c:/logs/mytrace.txt">
            <filter type="System.Diagnostics.EventTypeFilter"
                initializeData="Warning" />
          </add>
          <add name="xmlListener"
              type="System.Diagnostics.XmlWriterTraceListener"
              traceOutputOptions="None"
              initializeData="c:/logs/mytrace.xml">
            <filter type="System.Diagnostics.SourceFilter"
                initializeData="Wrox.ProCSharp.Diagnostics" />
          </add>
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

跟踪体系结构可以扩展。可以编写一个派生自 TraceListener 基类的自定义侦听器，同样，也可以创建一个派生自 TraceFilter 的自定义筛选器。因此，可以创建一个筛选器，根据时间、最近发生的异常或天气，指定写入跟踪消息。

20.3.5 相关性

在跟踪日志中，可以用几种方式查看不同方法的关系。要查看跟踪事件的调用栈，只需要进行一个配置，就可以用 XML 侦听器跟踪调用栈。还可以定义一个能在日志消息中显示的逻辑调用栈。也可以定义活动，以映射跟踪消息。

为了显示调用栈和带跟踪消息的逻辑调用栈，可以把 XmlWriterTraceListener 类配置为对应的 traceOuputOptions。MSDN 文档(http://msdn.microsoft.com/en-us/library/System.Diagnostics.XmlWriter-TraceListener(v=vs.110).aspx)给出了可以配置为通过这个侦听器进行跟踪的所有其他选项的详细内容。

```
<sharedListeners>
  <add name="xmlListener" type="System.Diagnostics.XmlWriterTraceListener"
      traceOutputOptions="LogicalOperationStack, Callstack"
      initializeData="c:/logs/mytrace.xml">
  </add>
</sharedListeners>
```

为了在跟踪日志中查看相关性，可在 Main()方法中设置 ActivityId 属性，把一个新的活动 ID 赋予 CorrelationManager。TraceEventType.Start 和 TraceEventType.Stop 类型的事件在 Main()方法的开始和结束时触发。另外，使用 StartLogicalOperation()和 StopLogicalOperation()方法分别启动和停止一个逻辑操作“Main”。

```
static void Main()
{
  // start a new activity
  if (Trace.CorrelationManager.ActivityId == Guid.Empty)
  {
    Guid newGuid = Guid.NewGuid();
    Trace.CorrelationManager.ActivityId = newGuid;
  }
  trace.TraceEvent(TraceEventType.Start, 0, "Main started");

  // start a logical operation
  Trace.CorrelationManager.StartLogicalOperation("Main");

  TraceSourceDemo1();
  StartActivityA();
  Trace.CorrelationManager.StopLogicalOperation();
  Thread.Sleep(3000);
  trace.TraceEvent(TraceEventType.Stop, 0, "Main stopped");
}
```

在 Main()方法中调用的 StartActivityA()方法会把 CorrelationManager 的 ActivityId 设置为一个新的 GUID，以新建一个活动。在该活动停止之前，把 CorrelationManager 的 ActivityId 重置为以前的值。这个方法调用 Foo()方法，并用 Task.Factory.StartNew()方法新建一个任务。创建了这个任务后，就可以查看不同的线程如何显示在跟踪查看器中。

任务可参见第 21 章。

```
private static void StartActivityA()
{
  Guid oldGuid = Trace.CorrelationManager.ActivityId;
  Guid newActivityId = Guid.NewGuid();
  Trace.CorrelationManager.ActivityId = newActivityId;
```

```
    Trace.CorrelationManager.StartLogicalOperation("StartActivityA");

    trace.TraceEvent(TraceEventType.Verbose, 0,
        "starting Foo in StartNewActivity");
    Foo();

    trace.TraceEvent(TraceEventType.Verbose, 0,
        "starting a new task");
    Task.Run(() => WorkForATask());

    Trace.CorrelationManager.StopLogicalOperation();
    Trace.CorrelationManager.ActivityId = oldGuid;
}
```

从StartActivityA()方法内部启动的Foo()方法启动了一个新的逻辑操作。逻辑操作Foo在逻辑操作StartActivityA内部启动。

```
private static void Foo()
{
    Trace.CorrelationManager.StartLogicalOperation("Foo operation");

    trace.TraceEvent(TraceEventType.Verbose, 0, "running Foo");

    Trace.CorrelationManager.StopLogicalOperation();
}
```

在StartActivityA()方法内部创建的任务运行WorkForATask()方法。这里仅把包含启动、停止信息和详细信息的跟踪事件写入跟踪日志中。

```
private static void WorkForATask()
{
    trace.TraceEvent(TraceEventType.Start, 0, "WorkForATask started");

    trace.TraceEvent(TraceEventType.Verbose, 0, "running WorkForATask");

    trace.TraceEvent(TraceEventType.Stop, 0, "WorkForATask completed");
}
```

为了分析跟踪信息，可以启动服务跟踪查看器工具svctraceviewer.exe。这个工具主要用于分析WCF跟踪，但它也可以用于分析用XmlWriterTraceListener类写入的任何跟踪信息。图20-3显示了服务跟踪查看器的Activity选项卡，每个活动都显示在左边，事件显示在右边。选择一个事件时，可以指定在XML或格式化的视图中显示完整的消息。在格式化的视图中，基本信息、应用程序数据、逻辑操作栈和调用栈都进行了很好的格式化。

图20-4显示了该对话框的Graph选项卡，在这个视图中，不同的进程或线程可以选择性地显示在各自的泳道中。用Task类新建线程时，会显示第二个区域，以选择线程视图。

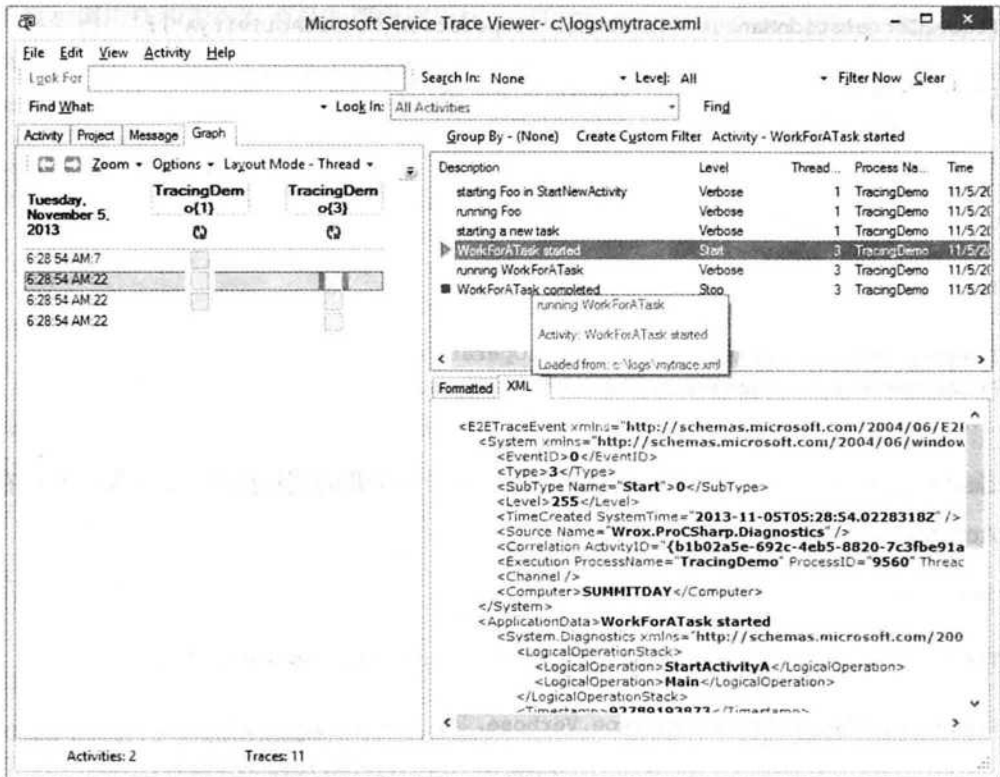


图 20-3

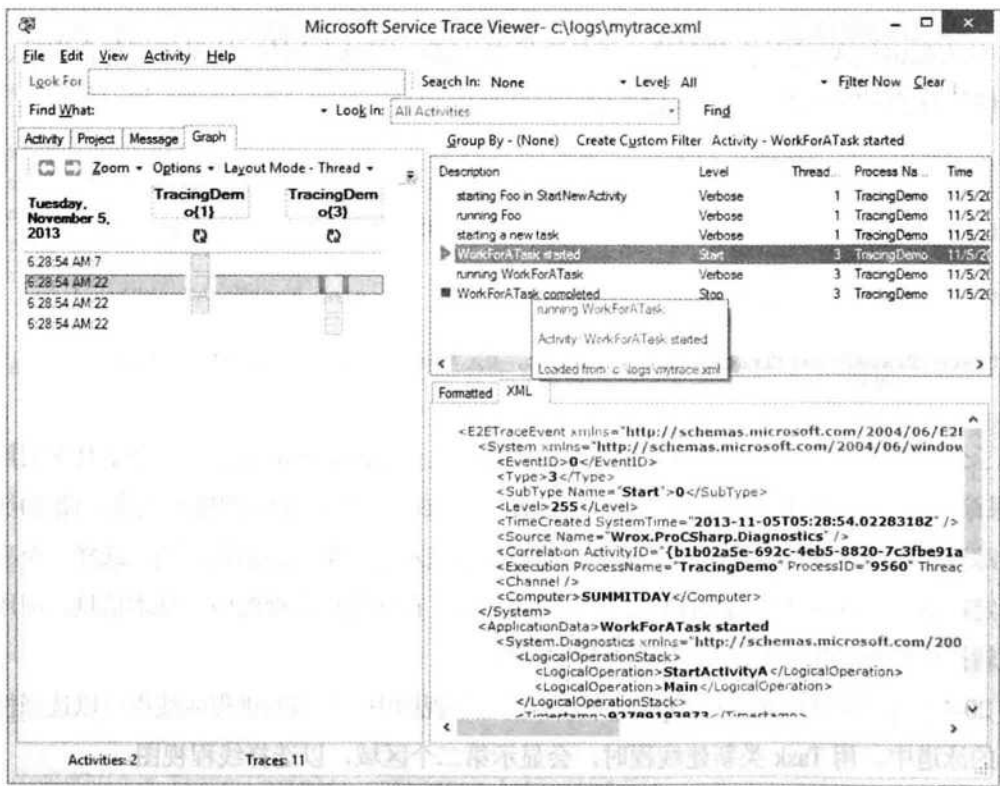


图 20-4

20.3.6 使用ETW进行跟踪

使用Event Tracing for Windows(ETW)可以快速地进行跟踪。Windows将ETW用于跟踪、事件日志和性能计数。使用ETW写入跟踪时，可以把EventProviderTraceListener配置为侦听器，如下面的代码段所示。type特性用于动态地找出类。类名通过程序集的强名和类名同时指定。对于initializeData特性，必须指定一个GUID，以唯一地标识侦听器。可以使用命令行工具uuidgen或图形工具guidgen创建一个GUID。

```
<sharedListeners>
  <add name="etwListener"
      type="System.Diagnostics.Eventing.EventProviderTraceListener,
      System.Core, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089"
      initializeData="{8ADA630A-F1CD-48BD-89F7-02CE2E7B9625}"/>
```

在修改配置之后，再次运行程序以使用ETW写入跟踪之前，需要使用logman命令启动一个跟踪会话。在这个命令中使用的start选项会启动一个新会话来进行记录。-p选项定义了提供程序的名称，这里使用了GUID来标识提供程序。-o选项定义了输出文件，-ets选项则把命令直接发送给事件跟踪系统，并不进行调度：

```
logman start mysession -p {8ADA630A-F1CD-48BD-89F7-02CE2E7B9625}
  -o mytrace.etl -ets
```

在运行应用程序后，可以使用stop命令停止跟踪会话：

```
logman stop mysession -ets
```

日志文件是二进制格式的。为了得到一个容易阅读的格式，可以使用tracerpt工具。使用此工具时，可以使用-of选项指定要提取的格式，包括CSV、XML和EVTX。

```
tracerpt mytrace.etl -o mytrace.xml -of XML
```

Windows操作系统包含了命令行工具logman和tracerpt。

20.3.7 使用EventSource

EventSource是一个用于跟踪的新类，自从.NET 4.5以来就可以使用了。这个类提供了一种新的跟踪方式，且完全基于ETW。这个类和可与EventSource一起使用的类型在System.Diagnostics.Tracing名称空间中定义。

使用EventSource的一种简单方式是创建一个派生自EventSource类型的类，再调用基类的WriteEvent方法，定义写入跟踪信息的方法。MyProjectEventSource类定义了强类型化的成员，例如Startup和CallService，它们调用基类的WriteEvent方法(代码文件EventSourceSample/MyProjectEventSource.cs)：

```
public class MyProjectEventSource : EventSource
{
```

```
  private MyProjectEventSource()
  {
  }

  public static MyProjectEventSource Log = new MyProjectEventSource();

  public void Startup()
  {
    base.WriteEvent(1);
  }

  public void CallService(string url)
  {
    base.WriteEvent(2, url);
  }

  public void ServiceError(string message, int error)
  {
    base.WriteEvent(3, message, error);
  }
}
```

在简单的情形中，应只写入信息型消息，不需要其他内容。除了给跟踪日志传递事件ID之外，WriteEvent方法还有14个重载版本，可以传送string、int和long值的消息，以及任意多个对象。

在这个实现代码中，MyProjectEventSource类型的成员可以用于写入跟踪消息，如Program类中所示(代码文件EventSourceSample/Program.cs)。Main方法建立一个调用Startup方法的跟踪日志，再调用NetworkRequestSample方法，通过CallService方法创建一个跟踪日志，并在出错时建立一个跟踪日志：

```
class Program
{
  static void Main()
  {
    MyProjectEventSource.Log.Startup();
    NetworkRequestSample();
    Console.ReadLine();
  }

  private static async void NetworkRequestSample()
  {
    try
    {
      var client = new HttpClient();
      string url = "http://www.cninnovaton.com";
      MyProjectEventSource.Log.CallService(url);
      string result = await client.GetStringAsync(url);
      Console.WriteLine("Complete................");
    }
    catch (Exception ex)
    {
      MyProjectEventSource.Log.ServiceError(ex.Message, ex.HResult);
    }
  }
}
```

跟踪消息可以使用 logman 实用工具在进程外部访问(参见上一节)。也可以使用 PerfView 实用工具读取跟踪消息。PerfView 可以从 Microsoft 下载中心 http://www.microsoft.com/downloads 下载。

要在进程内部访问跟踪消息，可以使用 EventListener 基类。只需要创建一个派生自 EventListener 的类，再重写 OnEventWritten 方法。对于这个方法，把跟踪消息传递为 EventWrittenEventArgs 类型的参数。示例实现代码发送了事件信息，包括有效载荷(这些额外数据传递给 EventSource 的 WriteEvent 方法)(代码文件 EventSourceSample/MyListener.cs)：

```
class MyListener : EventListener
{

  protected override void OnEventSourceCreated(EventSource eventSource)
  {
    Console.WriteLine("created {0} {1}", eventSource.Name, eventSource.Guid);
  }

  protected override void OnEventWritten(EventWrittenEventArgs eventData)
  {
    Console.WriteLine("event id: {0} source: {1}", eventData.EventId,
    eventData.EventSource.Name);
    foreach (var payload in eventData.Payload)
    {
      Console.WriteLine("\t{0}", payload);
    }
  }
}
```

侦听器在 Program 类的 Main 方法中激活。事件源可以调用 EventSource 类的静态方法 GetSources 来访问：

```
IEnumerable<EventSource> eventSources = EventSource.GetSources();
InitListener(eventSources);
```

InitListener 方法调用自定义侦听器的 EnableEvents 方法，并传递每个事件源。示例代码注册为监听并记录每个消息设置 EventLevel.LogAlways。也可以指定只写入信息型消息，这也包括错误，或只包含错误。指定这个级别非常类似于 20.3.2 节讨论的跟踪源开关。

```
private static void InitListener(IEnumerable<EventSource> sources)
{
  listener = new MyListener();
  foreach (var source in sources)
  {
    listener.EnableEvents(source, EventLevel.LogAlways);
  }
}
```

20.3.8 使用 EventSource 进行高级跟踪

对于许多应用程序，按上一节所述使用 EventSource 就足够了。也可以更多地控制跟踪，如代码文件 EventSourceSample/MyAdvancedProjectEventSource.cs 所示。

默认情况下，事件源的名称与类名相同，但可以使用 EventSource 特性，来改变该名称和唯一标识符。每个事件跟踪方法都可以带有 Event 特性。在这里可以定义事件的 ID、opcode、跟踪级别、自定义关键字和任务。侦听器可以根据关键字来过滤跟踪消息。要指定关键字，可以在标志样式的枚举中设置单个位：

```
[EventSource(Name="EventSourceSample", Guid="45FFF0E2-7198-4E4F-9FC3-DF6934680096")]
class MyAdvancedProjectEventSource : EventSource
{
  public class Keywords
  {
    public const EventKeywords Network = (EventKeywords)1;
    public const EventKeywords Database = (EventKeywords)2;
    public const EventKeywords Diagnostics = (EventKeywords)4;
    public const EventKeywords Performance = (EventKeywords)8;
  }

  public class Tasks
  {
    public const EventTask CreateMenus = (EventTask)1;
    public const EventTask QueryMenus = (EventTask)2;
  }
  private MyAdvancedProjectEventSource()
  {
  }

  public static MyAdvancedProjectEventSource Log = new MyAdvancedProjectEventSource();

  [Event(1, Opcode=EventOpcode.Start, Level=EventLevel.Verbose)]
  public void Startup()
  {
    base.WriteEvent(1);
  }
  [Event(2, Opcode=EventOpcode.Info, Keywords=Keywords.Network,
    Level=EventLevel.Verbose, Message="{0}")]
  public void CallService(string url)
  {
    base.WriteEvent(2);
  }

  [Event(3, Opcode=EventOpcode.Info, Keywords=Keywords.Network,
    Level=EventLevel.Error, Message="{0} error: {1}")]
  public void ServiceError(string message, int error)
  {
    base.WriteEvent(3);
  }

  [Event(4, Opcode=EventOpcode.Info, Task=Tasks.CreateMenus,
    Level=EventLevel.Verbose, Keywords=Keywords.Network)]
  public void SomeTask()
  {
    base.WriteEvent(4);
  }
}
```

有了事件定义的新用法，除了日志级别之外，侦听器还需要指定应记录的关键字：

```
listener.EnableEvents(source, EventLevel.Verbose, (EventKeywords)15L);
```

20.4 事件日志

系统管理员使用事件查看器获得关于系统和应用程序正常运行的重要消息和信息型消息。应将应用程序的错误消息写入事件日志，从而可以用事件查看器读取这些信息。

如果配置了 EventLogTraceListener 类，跟踪消息就可以写入事件日志中。EventLogTraceListener 类有一个与之相关联的 EventLog 对象，可以方便写入事件日志项。也可以直接使用 EventLog 类读写事件日志。

本节介绍如下内容：

- 事件日志体系结构
- System.Diagnostics 名称空间中用于事件日志的类
- 在服务和其他应用程序类型中添加事件日志
- 用 EventLog 类的 EnableRaisingEvents 属性创建事件日志侦听器
- 使用资源文件定义消息

图 20-5 显示了使用分布式 COM 访问失败的一个日志项。

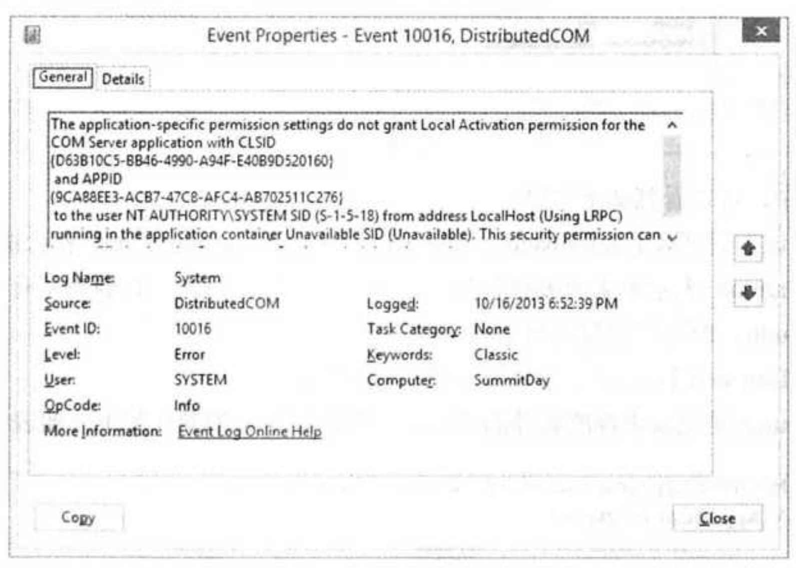


图 20-5

对于自定义事件日志，可以使用 System.Diagnostics 名称空间中的类。

20.4.1 事件日志体系结构

事件日志信息存储在几个日志文件中。最重要的日志文件是应用程序、安全性和系统日志文件。查看事件日志服务的注册表配置，会注意到 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog 下的几项带有指向特定文件的配置。系统日志文件用于系统驱动程序和设备驱动程序。应用程序和服务则将事件写入应用程序日志中。安全性日志是应用程序的只读日志。操作

系统的审计功能使用安全性日志。每个应用程序还可以创建自定义类别和日志文件，在其中写入事件日志项。例如，Media Center 就可以这么做。

使用管理工具“事件查看器”就可以读取这些事件。要启动事件查看器，可以直接在 Visual Studio 的 Server Explorer 窗口中右击 Event Logs 项，从弹出的上下文菜单中选择 Launch Event Viewer 命令。事件查看器如图 20-6 所示。

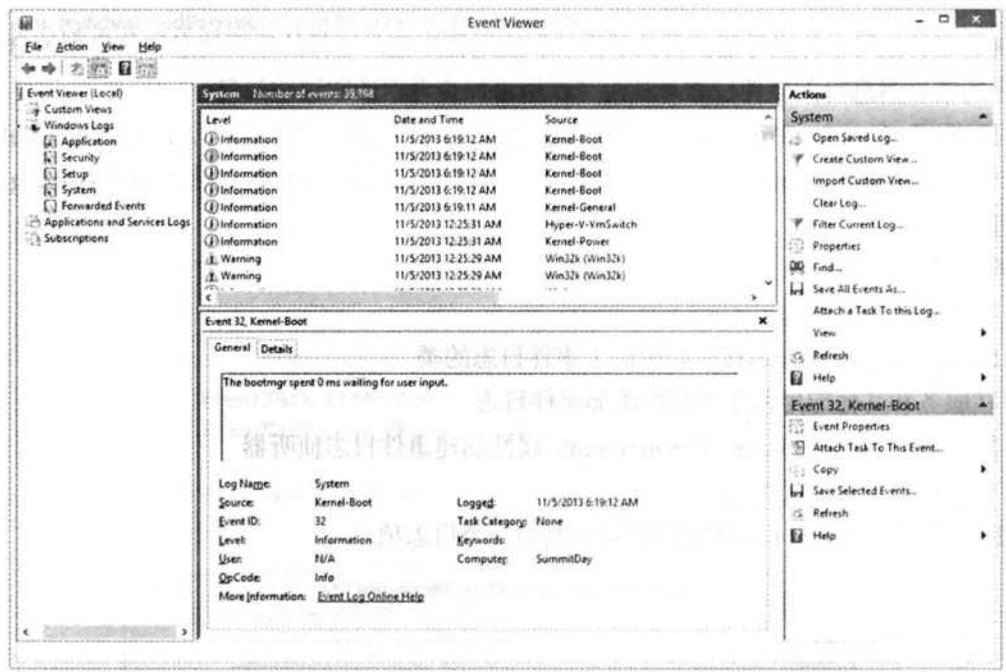


图 20-6

在事件日志中，可以看到如下信息：

- **类型**——该类型可以是 Information、Warning 或 Error。Information 是表示成功操作的不常用类型；Warning 表示不太重要的问题；Error 表示重要问题。其他类型有 FailureAudit 和 SuccessAudit，但这些类型仅用于安全性日志。
- **日期**——Date and Time 显示事件发生的日期和时间。
- **源**——Source 是记录事件的软件的名称。应用程序日志的源在如下注册表键中配置：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\
Application\[ApplicationName]
```

在这个键下，EventMessageFile 的值配置为指向一个资源 DLL，该 DLL 保存了错误消息。

- **事件 ID**——Event 标识符指定某条事件消息。
- **类别**——Category 可以定义为允许在使用 Event Viewer 时筛选事件日志。类别可以通过事件源来定义。

20.4.2 事件日志类

写入事件日志时，可以使用两个不同的Windows API。在Windows Vista发布后可用的一个API由System.Diagnostics.Eventing名称空间中的类封装，另一个封装类在System.Diagnostics名称空间中。

本书介绍了 System.Diagnostics 名称空间中的事件日志类。System.Diagnostics.Eventing 名称空间中的其他事件日志没有对.NET 提供特别好的支持，需要使用一些命令行工具和不安全的 C#代码。

System.Diagnostics 名称空间中的一些类用于事件日志，如表 20-1 所示。

表 20-1

类	说 明
EventLog	使用 EventLog 类可以读写事件日志中的项，把应用程序建立为事件源
EventLogEntry	EventLogEntry 类表示事件日志中的一项。使用 EventLogEntryCollection 类可以遍历 EventLogEntry 项
EventLogInstaller	EventLogInstaller 类是 EventLog 组件的安装程序。EventLogInstaller 类调用 EventLog.CreateEventSource()方法创建事件源
EventLogTraceListener	使用 EventLogTraceListener 类有助于把跟踪信息写入事件日志中。这个类实现了抽象类 TraceListener

事件日志的核心是 EventLog 类。这个类的成员如表 20-2 所示。

表 20-2

EventLog 类的成员	说 明
Entries	使用 Entries 属性可以读取事件日志。Entries 属性返回一个 EventLogEntryCollection，它包含的 EventLogEntry 对象存储了事件的相关信息。不需要调用 Read()方法。只要访问这个属性，就会填充该集合
Log	使用 Log 属性可指定用于读写事件日志的记录
LogDisplayName	LogDisplayName 是一个只读属性，返回日志的显示名称
MachineName	使用 MachineName 可以指定在哪个系统上读写日志项
Source	Source 属性指定要写入日志项的源
CreateEventSource()	CreateEventSource()新建一个事件源和一个日志文件
DeleteEventSource()	要删除一个事件源，可以调用 DeleteEventSource()方法
SourceExists()	在创建事件源之前，可以使用这个元素验证该源是否已存在
WriteEntry() WriteEvent()	用 WriteEntry()或 WriteEvent()方法可以写入事件日志项。WriteEntry()方法比较简单，只需要传递一个字符串即可。WriteEvent()方法比较灵活，因为用户可以使用独立于应用程序的消息文件，还可以支持本地化
Clear()	Clear()方法删除事件日志中的所有项
Delete()	Delete()方法删除一个完整的事件日志

20.4.3 创建事件源

在写入事件之前，必须创建一个事件源。为此，可以使用 EventLog 类或 EventLogInstaller 类的 CreateEventSource 方法。在创建事件源时需要管理员权限，所以最好用一个安装程序定义新事件源。

下面的例子验证了事件日志源 EventLogDemoApp 已存在。如果它不存在，就实例化一个 EventSourceCreationData 类型的对象，该对象定义源名 EventLogDemoApp 和日志名 ProCSharpLog。这里，该源的所有事件都写入 ProCSharpLog 事件日志。默认为应用程序日志。

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";

if (!EventLog.SourceExists(sourceName))
{
  var eventSourceData = new EventSourceCreationData(sourceName, logName);

  EventLog.CreateEventSource(eventSourceData);
}
```

事件源的名称是写入事件的应用程序的标识符。系统管理员在读取日志时，该信息有助于识别事件日志项，将它们映射到应用程序类别上。事件日志源的名称包括：如用于性能监视器的 LoadPerf，用于 Microsoft SQL Server 的 MSSQLSERVER，用于 Windows 安装程序的 MsiInstaller，以及 Winlogon、Tcpip、Time-Service 等。

为事件日志设置名称 Application，会把事件日志项写入应用程序日志中。也可以创建自己的日志，方法是指定不同的应用程序日志名。日志文件位于<windows>\System32\WinEvt \Logs 目录中。

使用 EventSourceCreationData 类，还可以为事件日志指定更多的特性，如表 20-3 所示。

表 20-3

EventSourceCreationData	说 明
Source	Source 属性可获取或设置事件源的名称
LogName	LogName 属性定义了事件日志项写到什么日志中。默认为应用程序日志
MachineName	使用 MachineName 属性可以定义读写日志项的系统
CategoryResourceFile	使用 CategoryResourceFile 属性可以定义类别的资源文件。类别有助于筛选单一源中的事件日志项
CategoryCount	CategoryCount 属性定义了类别资源文件中的类别个数
MessageResourceFile	除了指定程序(该程序用来写入事件)中应写入事件日志中的消息之外，消息还可以在一个资源文件中定义，该文件由 MessageResourceFile 属性指定。该资源文件中的信息是可以本地化的
ParameterResourceFile	资源文件中的消息可以带参数。参数可以用一个资源文件中定义的字符串替代，该资源文件由 ParameterResourceFile 属性指定

20.4.4 写入事件日志

要写入事件日志项，可以使用 EventLog 类的 WriteEntry()或 WriteEvent()方法。EventLog 类有一

个静态方法 WriteEntry()和一个实例方法 WriteEntry()。静态方法 WriteEntry()的参数是事件源。该源也可以用 EventLog 类的构造函数设置。在下面的构造函数中，定义了日志名、本地计算机和事件源名。接着将消息作为 WriteEntry()方法的第一个参数，写入 3 个事件日志项。WriteEntry()方法是重载版本。可以指定的第二个参数是 EventLogEntryType 类型的枚举。使用 EventLogEntryType 类型，可以指定事件日志项的严重程度。其值可以是 Information、Warning 和 Error，以及用于审计的 FailureAudit 和 SuccessAudit。根据该类型，在 Event Viewer 窗口中会显示不同的图标。第 3 个参数指定与应用程序相关的事件 ID，它可以由应用程序使用。另外，还可以传递与应用程序相关的二进制数据和类别。

```
using (var log = new EventLog(logName, ".", sourceName))
{
  log.WriteEntry("Message 1");
  log.WriteEntry("Message 2", EventLogEntryType.Warning);
  log.WriteEntry("Message 3", EventLogEntryType.Information, 33);
}
```

20.4.5 资源文件

除了在 C#代码中为事件日志定义消息，把它传递给 WriteEntry()方法之外，还可以创建消息资源文件，在该资源文件中定义消息，将消息的标识符传递给 WriteEvent()方法。资源文件还支持本地化。

消息资源文件是本地资源文件，它与.NET 资源文件没有共同之处。.NET 资源文件详见第 28 章。

资源文件是一个文本文件，扩展名是.mc。这个文件用于定义消息的语法非常严格。示例文件 EventLogMessages.mc 包含 4 个类别，之后是事件消息。每个消息都有一个 ID，它可以由写入日志项的应用程序使用。可以从应用程序中传递的参数在消息文本中用%语法定义(资源文件 EventLogDemo/EventLogDemoMessages.mc)。

```
; // EventLogDemoMessages.mc
; // ********************************************************

; // - Event categories -
; // Categories must be numbered consecutively starting at 1.
; // ********************************************************

MessageId=0x1
Severity=Success
SymbolicName=INSTALL_CATEGORY
Language=English
Installation
.

MessageId=0x2
Severity=Success
SymbolicName=DATA_CATEGORY
Language=English
Database Query
.
```

```
MessageId=0x3
Severity=Success
SymbolicName=UPDATE_CATEGORY
Language=English
Data Update
.

MessageId=0x4
Severity=Success
SymbolicName=NETWORK_CATEGORY
Language=English
Network Communication
.

; // - Event messages -
; // *********************************

MessageId = 1000
Severity = Success
Facility = Application
SymbolicName = MSG_CONNECT_1000
Language=English
Connection successful.
.

MessageId = 1001
Severity = Error
Facility = Application
SymbolicName = MSG_CONNECT_FAILED_1001
Language=English
Could not connect to server %1.
.

MessageId = 1002
Severity = Error
Facility = Application
SymbolicName = MSG_DB_UPDATE_1002
Language=English
Database update failed.
.

MessageId = 1003
Severity = Success
Facility = Application
SymbolicName = APP_UPDATE
Language=English
Application %%5002 updated.
.

; // - Event log display name -
; // ********************************************************
```

```
MessageId = 5001
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_DISPLAY_NAME_MSGID
Language=English
Professional C# Sample Event Log
.

; // - Event message parameters -
; // Language independent insertion strings
; // ***********************************************************


MessageId = 5002
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_SERVICE_NAME_MSGID
Language=English
EventLogDemo.EXE
.
```

消息文件的语法可参见MSDN文档中的“消息文本文件”。(http://msdn.microsoft.com/en-us/library/windows/desktop/dd996906.aspx)。

使用消息编译器mc.exe，创建一个二进制的消息文件。下面的命令会把包含消息的源文件编译为一个扩展名为.bin的消息文件和Messages.rc文件，后者包含对二进制消息文件的引用。

```
mc -s EventLogDemoMessages.mc
```

接着，需要使用资源编译器rc.exe。下面的命令会创建资源文件EventLogDemoMessages.RES：

```
rc EventLogDemoMessages.rc
```

使用链接器，可以把二进制消息文件EventLogDemoMessages.RES绑定到一个本地DLL上。

```
link /DLL /SUBSYSTEM:WINDOWS /NOENTRY /MACHINE:x86 EventLogDemoMessages.RES
```

现在，就可以用下面的代码注册一个事件源，来定义资源文件了。首先检查事件源EventLogDemoApp是否存在。如果它不存在，就必须创建事件日志。接着验证资源文件是否可用。MSDN文档中的一些示例说明了如何把消息文件写入<windows>\system32目录中，但这里不应这么做。把消息DLL复制到与程序相关的目录中，该目录可以使用SpecialFolder枚举的值ProgramFiles指定。如果需要在多个应用程序中共享消息文件，就可以把它放在Enviornment.SpecialFolder.CommonProgramFiles中。

如果该文件存在，就实例化EventSourceCreationData类型的一个新对象。在构造函数中，定义了源的名称和日志的名称。使用CatagoryResourceFile、MessageResourceFile和ParameterResourceFile属性定义资源文件的引用。在创建了事件源后，就可以使用事件源在注册表中找到资源文件的信息。CreateEventSource()方法注册新的事件源和日志文件。最后，EventLog类的RegisterDisplayName()方法指定日志显示在事件查看器中的名称。从消息文件中提取ID 5001(代码文件EventLogDemo/Program.cs)。

如果要删除以前创建的事件源，就可以使用 EventLog.DeleteEventSource(sourceName)。要删除日志，可以调用 EventLog.Delete(logName)。

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";
string resourceFile = Environment.GetFolderPath(
     Environment.SpecialFolder.ProgramFiles) +
     @"\procsharp\EventLogDemoMessages.dll";

if (!EventLog.SourceExists(sourceName))
{
   if (!File.Exists(resourceFile))
   {
     Console.WriteLine("Message resource file does not exist");
     return;
   }

   var eventSource = new EventSourceCreationData(sourceName, logName);

   eventSource.CategoryResourceFile = resourceFile;
   eventSource.CategoryCount = 4;
   eventSource.MessageResourceFile = resourceFile;
   eventSource.ParameterResourceFile = resourceFile;

   EventLog.CreateEventSource(eventSource);
}
else
{
   logName = EventLog.LogNameFromSourceName(sourceName, ".");
}

var evLog = new EventLog(logName, ".", sourceName);
evLog.RegisterDisplayName(resourceFile, 5001);
```

现在，可以使用 WriteEvent()方法而不是 WriteEntry()写入事件日志项。WriteEvent()方法需要把一个 EventInstance 类型的对象作为参数。使用 EventInstance 可以指定消息 ID、类别和 EventLogEntryType 类型的严重程度。除 EventInstance 参数之外，WriteEvent()方法的参数还允许接收消息的参数，这些消息包含字节数组格式的参数和二进制数据。

```
using (var log = new EventLog(logName, ".", sourceName))
{
   var info1 = new EventInstance(1000, 4,
      EventLogEntryType.Information);
   log.WriteEvent(info1);
   var info2 = new EventInstance(1001, 4,
      EventLogEntryType.Error);
   log.WriteEvent(info2, "avalon");

   var info3 = new EventInstance(1002, 3,
      EventLogEntryType.Error);
```

```
        byte[] additionalInfo = { 1, 2, 3 };
        log.WriteEvent(info3, additionalInfo);
    }
```

消息标识符可以定义带常量值的类，该值在应用程序中为标识符提供更有意义的名称。

使用事件查看器可以读取事件日志项。

20.5 性能监视

性能监视可以用于获取关于应用程序的正常行为的信息，将系统的行为与预先建立好的标准进行比较，以及观察变化和趋势(特别是运行在服务器上的应用程序的变化和趋势)。当越来越多的用户访问应用程序时，不等到第一个用户开始抱怨性能问题，系统管理员就已经采取了措施，在需要的地方增加了资源。性能监视器(PerfMon)是一个强大的工具，有助于查看所有性能计数，以便未雨绸缪。开发人员可以使用这个工具来理解正在运行的应用程序以及其基础技术。

Microsoft Windows 有许多性能对象，如 System、Memory、Objects、Process、Processor、Thread、Cache 等。这些对象有许多要监视的计数。例如，使用 Process 对象可以监视所有进程实例或特定进程实例的用户时间、句柄数、页面错误、线程数等。.NET Framework 和一些应用程序，如 SQL Server，还添加了与应用程序相关的对象。

20.5.1 性能监视类

System.Diagnostics 名称空间为性能监视提供了如下类：

- PerformanceCounter 类可以用于监视计数和写入计数。还可以使用这个类创建新的性能类别。
- PerformanceCounterCategory 类可以查看所有已有的类别，以及创建新类别。可以以编程方式获得一个类别中的所有计数器。
- PerformanceCounterInstaller 类用于安装性能计数器，它的用法类似于前面讨论的 EventLog-Installer 类。

20.5.2 性能计数器生成器

示例应用程序 PerformanceCounterDemo 是一个简单的 Windows 应用程序，它只有两个按钮，用于说明如何编写性能计数。使用一个按钮的处理程序注册性能计数器的类别，使用另一个按钮的处理程序写入性能计数器的值。采用同样的方式，可以在 Windows 服务(参见第 27 章)、网络应用程序(参见第 26 章)和任何其他要接收实时计数的应用程序中添加性能计数器。

使用 Visual Studio，可以创建新的性能计数器类别，具体方法是在 Server Explorer 中选择性能计数器，并在弹出的上下文菜单中选择菜单项 Create New Category 命令。这会启动 Performance Counter Builder(性能计数器生成器)，如图 20-7 所示。将性能计数器类别的名称设置为 Wrox Performance Counters。表 20-4 列出了示例应用程序的所有性能计数器。

要使用 Visual Studio 创建性能计数器类别，Visual Studio 必须以提升权限的模式启动。

表 20-4

性能计数器	说 明	类 型
# of button clicks	按钮单击的总次数	NumberOfItems32
# of button clicks/sec	一秒内按钮单击次数	RateOfCountsPerSecond32
# of mouse move events	鼠标移动事件的总数	NumberOfItems32
# of mouse move events/sec	一秒内鼠标移动事件的总数	RateOfCountsPerSecond32

Performance Counter Builder 将配置写入性能数据库中。这也可以用 System.Diagnostics 名称空间中的 PerformanceCounterCategory 类的 Create()方法动态完成。使用 Visual Studio 很容易在以后添加其他系统的安装程序。

Performance Counter Builder
Define a new category and the list of counters you want it to contain, or edit the values for an existing category.
Category name:
Wrox Performance Counters
Category description:
Sample Performance Counters
Counter list builder
To add a new counter, click on the New button then specify the details.
Counters:
# of Button Clicks
# of Button Clicks / sec
Counter
Name:
# of Button Clicks / sec
Type:
RateOfCountsPerSecond32
Counter description:
# of button clicks in one second
New
Delete
OK
Cancel

图 20-7

下面的代码段说明了如何以编程方式添加性能类别。使用 Visual Studio 中的工具，只能创建全局的性能类别，该类别不能为运行应用程序的不同进程指定不同的值。以编程方式创建性能类别，可以监视不同应用程序的性能计数，这里就采用这种方式。

首先，给类别名定义一个 const，再定义一个包含性能计数类别名的 SortedList<TKey, TValue>(代码文件 PerformanceCounterDemo/MainWindow.xaml.cs):

```
private const string perfomanceCounterCategoryName =
    "Wrox Performance Counters";
private SortedList<string, Tuple<string, string>> perfCountNames;
```

perfCountNames 变量的列表在 InitializePerformanceCountNames()方法中填充。有序列表的值定义为 Tuple<string, string>，以确定性能计数器的名称和描述。

```
private void InitializePerfomanceCountNames()
{
  perfCountNames = new SortedList<string, Tuple<string, string>>();
  perfCountNames.Add("clickCount", Tuple.Create("# of button Clicks",
      "Total # of button clicks"));
  perfCountNames.Add("clickSec", Tuple.Create("# of button clicks/sec",
      "# of mouse button clicks in one second"));
  perfCountNames.Add("mouseCount", Tuple.Create("# of mouse move events",
      "Total # of mouse move events"));
  perfCountNames.Add("mouseSec", Tuple.Create("# of mouse move events/sec",
      "# of mouse move events in one second"));
}
```

接着在OnRegisterCounts()方法中创建性能计数器类别。验证该类别不存在后，就创建一个 CounterCreationData 数组，用性能计数的类型和名称填充该数组。接着，使用 PerformanceCounterCategory.Create()方法创建新类别。PerformanceCounterCategoryType.MultiInstance 指定这些计数不是全局的，不同的实例可以有不同的值。

```
private void OnRegisterCounts(object sender, RoutedEventArgs e)
{
  if (!PerformanceCounterCategory.Exists(
      perfomanceCounterCategoryName))
  {
    var counterCreationData = new CounterCreationData[4];
    counterCreationData[0] = new CounterCreationData
    {
      CounterName = perfCountNames["clickCount"].Item1,
      CounterType = PerformanceCounterType.NumberOfItems32,
      CounterHelp = perfCountNames["clickCount"].Item2
    };
    counterCreationData[1] = new CounterCreationData
    {
      CounterName = perfCountNames["clickSec"].Item1,
      CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
      CounterHelp = perfCountNames["clickSec"].Item2,
    };
    counterCreationData[2] = new CounterCreationData
    {
      CounterName = perfCountNames["mouseCount"].Item1,
      CounterType = PerformanceCounterType.NumberOfItems32,
      CounterHelp = perfCountNames["mouseCount"].Item2,
    };
    counterCreationData[3] = new CounterCreationData
    {
      CounterName = perfCountNames["mouseSec"].Item1,
      CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
      CounterHelp = perfCountNames["mouseSec"].Item2,
    };
    var counters = new CounterCreationDataCollection(counterCreationData);
```

```
    var category = PerformanceCounterCategory.Create(
        perfomanceCounterCategoryName,
        "Sample Counters for Professional C#",
        PerformanceCounterCategoryType.MultiInstance,

        counters);
    MessageBox.Show(String.Format("category {0} successfully created",
        category.CategoryName));
}
```

20.5.3 添加 PerformanceCounter 组件

对于 Windows Forms 或 Windows Service 应用程序，可以从工具箱中添加 PerformanceCounter 组件，或者从 Server Explorer 中把该组件拖放到设计界面上。

对于 WPF 应用程序，就不能实现上述操作。但定义性能计数器所需的手工劳动并不多，因为它使用了 InitializePerformanceCounters()方法。在下例中，所有性能计数的 CategoryName 都在 const 字符串 performanceCounterCategoryName 中设置，CounterName 在有序列表中设置。由于应用程序要写入性能计数，因此 ReadOnly 属性必须设置为 false。如果所编写的应用程序出于显示目的仅读取性能计数，就可以使用 ReadOnly 属性的默认值 true。PerformanceCounter 对象的 InstanceName 属性设置为应用程序名。如果计数器配置为全局计数，就不能设置 InstanceName 属性。

```
private PerformanceCounter performanceCounterButtonClicks;
private PerformanceCounter performanceCounterButtonClicksPerSec;
private PerformanceCounter performanceCounterMouseMoveEvents;
private PerformanceCounter performanceCounterMouseMoveEventsPerSec;

private void InitializePerformanceCounters()
{
  performanceCounterButtonClicks = new PerformanceCounter
  {
    CategoryName = perfomanceCounterCategoryName,
    CounterName = perfCountNames["clickCount"].Item1,
    ReadOnly = false,
    MachineName = ".",
    InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
    InstanceName = this.instanceName
  };
  performanceCounterButtonClicksPerSec = new PerformanceCounter
  {
    CategoryName = perfomanceCounterCategoryName,
    CounterName = perfCountNames["clickSec"].Item1,
    ReadOnly = false,
    MachineName = ".",
    InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
    InstanceName = this.instanceName
  };
  performanceCounterMouseMoveEvents = new PerformanceCounter
  {
    CategoryName = perfomanceCounterCategoryName,
    CounterName = perfCountNames["mouseCount"].Item1,
    ReadOnly = false,
    MachineName = ".",
    InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
```

```
        InstanceName = this.instanceName
      };
      performanceCounterMouseMoveEventsPerSec = new PerformanceCounter
      {
        CategoryName = perfomanceCounterCategoryName,
        CounterName = perfCountNames["mouseSec"].Item1,
        ReadOnly = false,
        MachineName = ".",
        InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
        InstanceName = this.instanceName
      };
    }
```

为了计算性能值，需要添加 clickCountPerSec 和 mouseMoveCountPerSec 字段：

```
public partial class MainWindow : Window
{
  // Performance monitoring counter values
  private int clickCountPerSec = 0;
  private int mouseMoveCountPerSec = 0;
```

给按钮的 Click 事件添加事件处理程序，给按钮的 MouseMove 事件添加事件处理程序，在处理程序中添加如下代码：

```
private void OnButtonClick(object sender, RoutedEventArgs e)
{
  this.performanceCounterButtonClicks.Increment();
  this.clickCountPerSec++;
}

private void OnMouseMove(object sender, MouseEventArgs e)
{
  this.performanceCounterMouseMoveEvents.Increment();
  this.mouseMoveCountPerSec++;
}
```

PerformanceCounter 对象的 Increment()方法给计数器递增 1。如果需要给计数器添加其他信息，例如，添加一个字节的收发计数，就可以使用 IncrementBy()方法。对于显示以秒为单位的值的性能计数，只需要递增两个变量 clickCountPerSec 和 mouseMovePerSec。

为了显示每秒更新后的值，可给 MainWindow 的成员添加一个 DispatcherTimer 类。

```
private DispatcherTimer timer;
```

这个计时器在构造函数中配置和启动。DispatcherTimer 类是 System.Windows.Threading 名称空间中的一个计时器。对于非 WPF 应用程序，可以使用第 21 章要讨论的其他计时器。计时器调用的代码在一个匿名方法中指定：

```
public MainWindow()
{
  InitializeComponent();
  InitializePerfomanceCountNames();
  InitializePerformanceCounts();
  if (PerformanceCounterCategory.Exists(perfomanceCounterCategoryName))
```

```
    {
      buttonCount.IsEnabled = true;
      timer = new DispatcherTimer(TimeSpan.FromSeconds(1),
          DispatcherPriority.Background,
          delegate
          {
            this.performanceCounterButtonClicksPerSec.RawValue =
                this.clickCountPerSec;
            this.clickCountPerSec = 0;
            this.performanceCounterMouseMoveEventsPerSec.RawValue =
                this.mouseMoveCountPerSec;
                this.mouseMoveCountPerSec = 0;
          },
          Dispatcher.CurrentDispatcher);
      timer.Start();
    }
  }
```

20.5.4 perfmon.exe

现在就可以监视应用程序了。从控制面板的“管理工具”中可以启动“性能监视器”，在“性能监视器”中，单击工具栏上的“+”按钮，可以添加性能计数。Wrox PerformanceCounters 服务显示为一个性能对象。所有已配置的计数器都显示在可用计数器列表中，如图 20-8 所示。

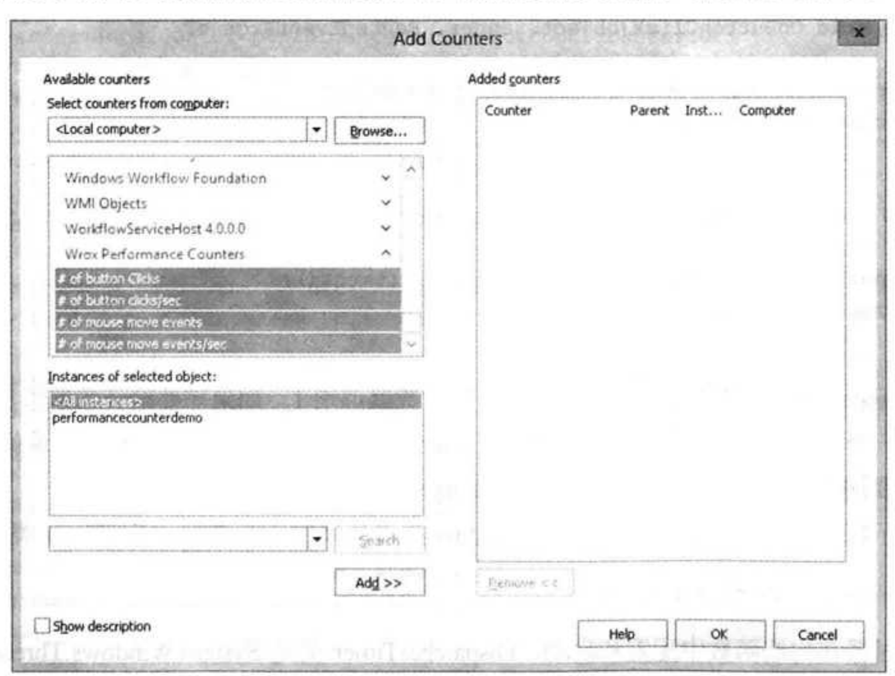


图 20-8

在性能监视器中添加了计数器后，就可以查看服务随时间变化的实际值，如图 20-9 所示。使用这个性能工具，还可以创建日志文件，在以后分析性能数据。

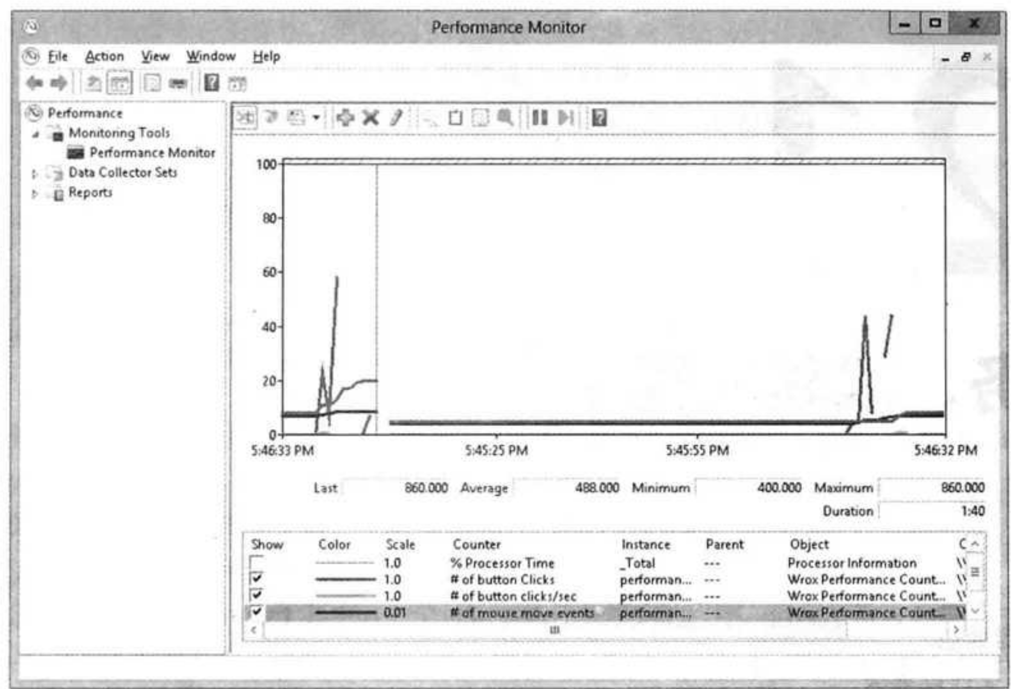


图 20-9

20.6 小结

本章介绍了跟踪和日志功能，它们有助于找出应用程序中的问题。应尽早规划，把这些功能内置于应用程序中。这可以避免以后的许多故障排除问题。

使用跟踪功能，可以把调试消息写入应用程序，也可以用于最终发布的产品。如果出了问题，就可以修改配置值，从而打开跟踪功能，并找出问题。

事件日志为系统管理员提供信息，帮助找出应用程序的某些严重问题。性能监视有助于分析应用程序的负载，提前规划将来可能需要的资源。

第21章

任务、线程和同步

本章要点

- 多线程概述
- 使用 Parallel 类
- 任务
- 取消架构
- 线程类和线程池
- 线程问题
- 同步技术
- 计时器

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Parallel
- Task
- Cancellation
- ThreadClass
- Synchronization
- DataFlow

21.1 概述

使用线程有几个原因。假设从应用程序中进行网络调用需要一定的时间。我们不希望用户界面停止响应，让用户一直等待直到从服务器返回一个响应。用户可以同时执行其他一些操作，或者甚至取消发送给服务器的请求。这些都可以使用线程来实现。

对于所有需要等待的操作，例如，因为文件、数据库或网络访问都需要一定的时间，此时就可以启动一个新线程，同时完成其他任务。即使是处理密集型的任务，线程也是有帮助的。一个进程的多个线程可以同时运行在不同的 CPU 上，或多核 CPU 的不同内核上。

还必须注意运行多个线程时的一些问题。它们可以同时运行，但如果线程访问相同的数据，就很容易出问题。必须实现同步机制。

第 13 章介绍了如何通过新关键字 async 和 wait 来使用异步方法。

本章介绍编写多线程应用程序所需的基础知识，用到的名称空间主要是 System.Threading 和 System.Threading.Tasks。

线程是程序中独立的指令流。使用 C#编写任何程序时，都有一个入口点：Main()方法。程序从 Main()方法的第一条语句开始执行，直到这个方法返回为止。

这种程序结构非常适合于其中有一个可识别的任务序列的程序，但程序常常需要同时完成多个任务。线程对客户端和服务器端应用程序都非常重要。在 Visual Studio 编辑器中输入 C#代码时，系统会分析代码，用下划线标出遗漏的分号或其他语法错误，这用一个后台线程完成。Microsoft Word 的拼写检查器也会做相同的事。一个线程等待用户输入，而另一个线程进行后台搜索。第 3 个线程将写入的数据存储在临时文件中，第 4 个线程从 Internet 上下载其他数据。

运行在服务器上的应用程序中，等待客户请求的线程，称为侦听器线程。只要接收到请求，就把它传递给另一个工作线程，之后继续与客户通信。侦听器线程会立即返回，接收下一个客户发送的下一个请求。

进程包含资源，如 Window 句柄、文件系统句柄或其他内核对象。每个进程都分配了虚拟内存。一个进程至少包含一个线程。操作系统会调度线程。线程有一个优先级、实际上正在处理的程序的位置计数器、一个存储其局部变量的栈。每个线程都有自己的栈，但程序代码的内存和堆由一个进程的所有线程共享。这使一个进程的所有线程之间的通信非常快——该进程的所有线程都寻址相同的虚拟内存。但是，这也使处理比较困难，因为多个线程可以修改同一个内存位置。

进程管理的资源包括虚拟内存和 Window 句柄，其中至少包含一个线程。线程是运行程序所必需的。在.NET 4 之前，必须直接使用 Thread 类和 ThreadPool 类编写线程。现在，.NET 对这两个类做了抽象，允许使用 Parallel 类和 Task 类。在一些特殊的场景中，仍然需要 Thread 类和 ThreadPool 类。作为一种好的习惯，应该使用最易用的类，而只在确实需要高级功能的时候使用更复杂的类。大多数程序都没有使用手写的 IL 代码。但是，在有些情况下甚至也需要手写的 IL 代码。

为编写能够利用并行性的代码，必须区分两种主要的场景：任务并行性和数据并行性。对于任务并行性，使用 CPU 的代码被并行化。CPU 的多个核心会被利用起来，更快速地完成包含多个任务的活动，而不是在一个核心中按顺序一个一个地执行任务。对于数据并行性，则使用了数据集合。在集合上执行的工作被划分为多个任务。当然，任务并行性和数据并行性可以混合起来。

Parallel LINQ 提供了任务并行性的一种变体，详见第 11 章。

21.2 Parallel 类

Parallel 类是对线程的一个很好的抽象。该类位于 System.Threading.Tasks 名称空间中，提供了数据和任务并行性。

Parallel 类定义了并行的 for 和 foreach 的静态方法。对于 C#的 for 和 foreach 语句而言，循环从一个线程中运行。Parallel 类使用多个任务，因此使用多个线程来完成这个作业。

Parallel.For()和 Parallel.ForEach()方法在每次迭代中调用相同的代码，而 Parallel.Invoke()方法允许同时调用不同的方法。Parallel.Invoke 用于任务并行性，而 Parallel.ForEach 用于数据并行性。

21.2.1 用 Parallel.For()方法循环

Parallel.For()方法类似于 C#的 for 循环语句，也是多次执行一个任务。使用 Parallel.For()方法，可以并行运行迭代。迭代的顺序没有定义。

在 For()方法中，前两个参数定义了循环的开头和结束。示例从 0 迭代到 9。第 3 个参数是一个 Action<int>委托。整数参数是循环的迭代次数，该参数被传递给委托引用的方法。Parallel.For()方法的返回类型是 ParallelLoopResult 结构，它提供了循环是否结束的信息(代码文件 ParallelSamples/Program.cs)。

```
ParallelLoopResult result =
  Parallel.For(0, 10, i =>
  {
    Console.WriteLine("{0}, task: {1}, thread: {2}", i,
      Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(10);
  });
Console.WriteLine("Is completed: {0}", result.IsCompleted);
```

在 Parallel.For()的方法体中，把索引、任务标识符和线程标识符写入控制台中。从输出可以看出，顺序是不能保证的。如果再次运行这个程序，可以看到不同的结果。程序这次的运行顺序是 0-2-4-6-8……，有 5 个任务和 5 个线程。任务不一定映射到一个线程上。线程也可以被不同的任务重用。

```
0, task: 1, thread: 1
2, task: 2, thread: 3
4, task: 3, thread: 4
6, task: 4, thread: 5
8, task: 5, thread: 6
5, task: 3, thread: 4
7, task: 4, thread: 5
9, task: 5, thread: 6
3, task: 2, thread: 3
1, task: 1, thread: 1
Is completed: True
```

在前面的例子中，使用了.NET 4.5 中新增的 Thread.Sleep 方法，而不是 Task.Delay 方法。Task.Delay 是一个异步方法，用于释放线程供其他任务使用。下面的代码使用 await 关键字，所以一旦完成延迟，就立即开始调用这些代码。延迟后执行的代码和延迟前执行的代码可以运行在不同的线程中。

下面用 Task.Delay 方法修改前面的例子，在延迟一定时间后将任务线程和循环迭代的信息写入控制台：

```
ParallelLoopResult result =
    Parallel.For(0, 10, async i =>
        {
          Console.WriteLine("{0}, task: {1}, thread: {2}", i,
            Task.CurrentId, Thread.CurrentThread.ManagedThreadId);

          await Task.Delay(10);
          Console.WriteLine("{0}, task: {1}, thread: {2}", i,
            Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
        });
Console.WriteLine("is completed: {0}", result.IsCompleted);
```

其结果如下所示。在输出中可以看到，调用 Thread.Delay 方法后，线程发生了变化。例如，在循环迭代 2 在延迟前的线程 ID 为 3，在延迟后的线程 ID 为 4。在输出中还可以看到，任务不再存在，只有线程留下了，而且这里重用了前面的线程。另外一个重要的方面是，Parallel 类的 For 方法并没有等待延迟，而是直接完成。Parallel 类只等待它创建的任务，而不等待其他后台活动。在延迟后，也有可能完全看不到方法的输出，出现这种情况的原因是主线程(是一个前台线程)结束，所有的后台线程被终止。本章后面将讨论前台线程和后台线程。

```
2, task: 2, thread: 3
0, task: 1, thread: 1
4, task: 3, thread: 5
6, task: 4, thread: 6
8, task: 5, thread: 4
3, task: 2, thread: 3
7, task: 2, thread: 3
9, task: 5, thread: 4
5, task: 3, thread: 5
1, task: 1, thread: 1
is completed: True
5, task: , thread: 6
6, task: , thread: 6
7, task: , thread: 6
3, task: , thread: 6
8, task: , thread: 6
4, task: , thread: 6
0, task: , thread: 6
9, task: , thread: 5
2, task: , thread: 4
1, task: , thread: 3
```

从这里可以看到，虽然使用.NET 4.5 和 C# 5.0 的异步功能十分方便，但是知道后台发生了什么仍然很重要，而且必须留意一些问题。

提前停止 Parallel.For

也可以提前中断 Parallel.For()方法，而不是完成所有迭代。For()方法的一个重载版本接受第 3 个 Action<int, ParallelLoopState>类型的参数。使用这些参数定义一个方法，就可以调用 ParallelLoopState 的 Break()或 Stop()方法，以影响循环的结果。

注意，迭代的顺序没有定义(代码文件 ParallelSamples/Program.cs)。

```
ParallelLoopResult result =
  Parallel.For(10, 40, async (int i, ParallelLoopState pls) =>
  {
    Console.WriteLine("i: {0} task {1}", i, Task.CurrentId);
    await Task.Delay(10);
    if (i > 15)
      pls.Break();
  });
Console.WriteLine("Is completed: {0}", tresult.IsCompleted);
Console.WriteLine("lowest break iteration: {0}",
  result.LowestBreakIteration);
```

应用程序这次的运行说明，迭代在值大于 15 时中断，但其他任务可以同时运行，有其他值的任务也可以运行。利用 LowestBreakIteration 属性，可以忽略其他任务的结果。

```
10 task 1
24 task 3
31 task 4
38 task 5
17 task 2
11 task 1
12 task 1
13 task 1
14 task 1
15 task 1
16 task 1
Is completed: False
lowest break iteration: 16
```

Parallel.For()方法可能使用几个线程来执行循环。如果需要对每个线程进行初始化，就可以使用 Parallel.For<TLocal>()方法。除了 from 和 to 对应的值之外，For()方法的泛型版本还接受 3 个委托参数。第一个参数的类型是 Func<TLocal>，因为这里的例子对于 TLocal 使用字符串，所以该方法需要定义为 Func<string>，即返回 string 的方法。这个方法仅对用于执行迭代的每个线程调用一次。

第二个委托参数为循环体定义了委托。在示例中，该参数的类型是 Func<int, ParallelLoopState, string , string>。其中第一个参数是循环迭代，第二个参数 ParallelLoopState 允许停止循环，如前所述。循环体方法通过第 3 个参数接收从 init 方法返回的值，循环体方法还需要返回一个值，其类型是用泛型 for 参数定义的。

For()方法的最后一个参数指定一个委托 Action<TLocal>；在该示例中，接收一个字符串。这个方法仅对于每个线程调用一次，这是一个线程退出方法。

```
Parallel.For<string>(0, 20, () =>
  {
    // invoked once for each thread
```

```
    Console.WriteLine("init thread {0}, task {1}",
      Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
      return String.Format("t{0}",
        Thread.CurrentThread.ManagedThreadId);
  },
  (i, pls, str1) =>
  {
    // invoked for each member
    Console.WriteLine("body i {0} str1 {1} thread {2} task {3}", i, str1,
      Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
    Thread.Sleep(10);
    return String.Format("i {0}", i);
  },
  (str1) =>
  {
    // final action on each thread
    Console.WriteLine("finally {0}", str1);
  });
```

运行一次这个程序的结果如下:

```
init thread 1, task 1
init thread 5, task 4
init thread 3, task 2
init thread 4, task 3
init thread 6, task 5
body i 10 str1 t4 thread 4 task 3
body i 1 str1 i 0 thread 1 task 1
body i 1 str1 t6 thread 6 task 5
body i 15 str1 t5 thread 5 task 4
body i 5 str1 t3 thread 3 task 2
body i 11 str1 i 10 thread 4 task 3
body i 16 str1 i 15 thread 5 task 4
body i 2 str1 i 1 thread 6 task 5
body i 4 str1 i 0 thread 1 task 1
body i 17 str1 i 16 thread 5 task 4
body i 3 str1 i 2 thread 6 task 5
body i 6 str1 i 4 thread 1 task 1
body i 13 str1 i 5 thread 3 task 2
body i 12 str1 i 11 thread 4 task 3
body i 7 str1 i 6 thread 1 task 1
finally i 3
body i 14 str1 i 13 thread 3 task 2
finally i 17
body i 18 str1 i 12 thread 4 task 3
finally i 14
body i 8 str1 i 7 thread 1 task 1
body i 19 str1 i 18 thread 4 task 3
body i 9 str1 i 8 thread 1 task 1
finally i 19
finally i 9
```

21.2.2 使用 Parallel.ForEach()方法循环

Parallel.ForEach()方法遍历实现了 IEnumerable 的集合，其方式类似于 foreach 语句，但以异步方

式遍历。这里也没有确定遍历顺序。

```
string[] data = {"zero", "one", "two", "three", "four", "five",
  "six", "seven", "eight", "nine", "ten", "eleven", "twelve"};
ParallelLoopResult result =
  Parallel.ForEach<string>(data, s =>
{
  Console.WriteLine(s);
});
```

如果需要中断循环，就可以使用ForEach()方法的重载版本和ParallelLoopState参数。其方式与前面的For()方法相同。ForEach()方法的一个重载版本也可以用于访问索引器，从而获得迭代次数，如下所示：

```
Parallel.ForEach<string>(data, (s, pls, l) =>
  {
    Console.WriteLine("{0} {1}", s, l);
  });
```

21.2.3 通过Parallel.Invoke()方法调用多个方法

如果多个任务应并行运行，就可以使用 Parallel.Invoke()方法，它提供了任务并行性模式。Parallel.Invoke()方法允许传递一个Action委托数组，在其中可以指定应运行的方法。示例代码传递了要并行调用的Foo()和Bar()方法(代码文件ParallelSamples/Program.cs)：

```
static void ParallelInvoke()
{
  Parallel.Invoke(Foo, Bar);
}

static void Foo()
{
  Console.WriteLine("foo");
}

static void Bar()
{
  Console.WriteLine("bar");
}
```

Parallel类使用起来十分方便，而且既可以用于任务，又可以用于数据并行性。如果需要更细致的控制，并且不想等到Parallel类结束后再开始动作，就可以使用Task类。当然，结合使用Task类和Parallel类也是可以的。

21.3 任务

为了更好地控制并行动作，可以使用System.Threading.Tasks名称空间中的Task类。任务表示应完成的某个工作单元。这个工作单元可以在单独的线程中运行，也可以以同步方式启动一个任务，这需要等待主调线程。使用任务不仅可以获得一个抽象层，还可以对底层线程进行很多控制。

在安排需要完成的工作时，任务提供了非常大的灵活性。例如，可以定义连续的工作——在一个任务完成后该执行什么工作。这可以根据任务成功与否来区分。另外，还可以在层次结构中安排任务。例如，父任务可以创建新的子任务。这可以创建一种依赖关系，这样，取消父任务，也会取消其子任务。

21.3.1　启动任务

要启动任务，可以使用 TaskFactory 类或 Task 类的构造函数和 Start()方法。Task 类的构造函数在创建任务上提供的灵活性较大。

在启动任务时，会创建 Task 类的一个实例，利用 Action 或 Action<object>委托(不带参数或带一个 object 参数)，可以指定应运行的代码。下面定义的方法带一个参数。在实现代码中，把任务的 ID 和线程的 ID 写入控制台中，并且如果线程来自一个线程池，或者线程是一个后台线程，也要写入相关信息。把多条消息写入控制台的操作是使用 lock 关键字和 taskMethodLock 同步对象进行同步的。这样，就可以并行调用 TaskMethod，而且多次写入控制台的操作也不会彼此交叉。否则，title 可能由一个任务写入，而线程信息由另一个任务写入(代码文件 TaskSamples/Program.cs)：

```
static object taskMethodLock = new object();
static void TaskMethod(object title)
{
  lock (taskMethodLock)
  {
    Console.WriteLine(title);
    Console.WriteLine("Task id: {0}, thread: {1}",
      Task.CurrentId == null ? "no task" : Task.CurrentId.ToString(),
      Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine("is pooled thread: {0}",
      Thread.CurrentThread.IsThreadPoolThread);
    Console.WriteLine("is background thread: {0}",
      Thread.CurrentThread.IsBackground);
    Console.WriteLine();
  }
}
```

接下来的几小节描述了启动新任务的不同方法。

1. 使用线程池的任务

在本节中，可以看到启动使用了线程池中线程的任务的不同方式。线程池提供了一个后台线程的池，“线程池”一节中将详细介绍。现在，只需要知道线程池独自管理线程，根据需要增加或减少线程池中的线程数。线程池中的线程用于实现一些动作，之后仍然返回线程池中。

创建任务的第一种方式是使用实例化的 TaskFactory 类，在其中把 TaskMethod 方法传递给 StartNew 方法，就会立即启动任务。第二种方式是使用 Task 类的静态属性 Factory 来访问 TaskFactory，以及调用 StartNew()方法。它与第一种方式很类似，也使用了工厂，但是对工厂创建的控制则没有那么全面。第三种方式是使用 Task 类的构造函数。实例化 Task 对象时，任务不会立即运行，而是指定 Created 状态。接着调用 Task 类的 Start()方法，来启动任务。第四种方式是.NET 4.5 新增的，即调用 Task 类的 Run 方法，立即启动任务。Run 方法没有可以传递 Action<object>委托的重载版本，但是通过传递 Action 类型的 lambda 表达式并在其实现中使用参数，可以模拟这种行为。

```
static void TasksUsingThreadPool()
{
  var tf = new TaskFactory();
  Task t1 = tf.StartNew(TaskMethod, "using a task factory");

  Task t2 = Task.Factory.StartNew(TaskMethod, "factory via a task");

  var t3 = new Task(TaskMethod, "using a task constructor and Start");
  t3.Start();

  Task t4 = Task.Run(() => TaskMethod("using the Run method"));
}
```

这些版本返回的输出如下所示。它们都创建一个新任务，并使用线程池中的一个线程：

```
using a task factory
Task id: 1, thread: 6
is pooled thread: True
is background thread: True

factory via a task
Task id: 2, thread: 4
is pooled thread: True
is background thread: True

using the Run method
Task id: 3, thread: 5
is pooled thread: True
is background thread: True

using a task constructor and Start
Task id: 4, thread: 3
is pooled thread: True
is background thread: True
```

使用 Task 构造函数和 TaskFactory 的 StartNew 方法时，可以传递 TaskCreationOptions 枚举中的值。利用这个创建选项，可以改变任务的行为，如接下来的小节所示。

2. 同步任务

任务不一定要使用线程池中的线程，也可以使用其他线程。任务也可以同步运行，以相同的线程作为主调线程。下面的代码段使用了 Task 类的 RunSynchronously 方法：

```
private static void RunSynchronousTask()
{
  TaskMethod("just the main thread");
  var t1 = new Task(TaskMethod, "run sync");
  t1.RunSynchronously();
}
```

这里 TaskMethod 方法首先在主线程上直接调用，然后在新创建的 Task 上调用。从如下所示的控制台输出可以看到，主线程是一个前台线程，没有任务 ID，也不是线程池中的线程。调用 RunSynchronously 方法时，会使用相同的线程作为主调线程，但是如果以前没有创建任务，就会创

建一个任务：

```
just the main thread
Task id: no task, thread: 1
is pooled thread: False
is background thread: False

run sync
Task id: 1, thread: 1
is pooled thread: False
is background thread: False
```

3. 使用单独线程的任务

如果任务的代码应该长时间运行，就应该使用 TaskCreationOptions.LongRunning 告诉任务调度器创建一个新线程，而不是使用线程池中的线程。此时，线程可以不由线程池管理。当线程来自线程池时，任务调度器可以决定等待已经运行的任务完成，然后使用这个线程，而不是在线程池中创建一个新线程。对于长时间运行的线程，任务调度器会立即知道等待它们完成不是明智的做法。下面的代码片段创建了一个长时间运行的任务：

```
private static void LongRunningTask()
{
  var t1 = new Task(TaskMethod, "long running",
    TaskCreationOptions.LongRunning);
  t1.Start();
}
```

实际上，使用 TaskCreationOptions.LongRunning 选项时，不会使用线程池中的线程，而是会创建一个新线程：

```
long running
Task id: 1, thread: 3
is pooled thread: False
is background thread: True
```

21.3.2 Future——任务的结果

任务结束时，它可以把一些有用的状态信息写到共享对象中。这个共享对象必须是线程安全的。另一个选项是使用返回某个结果的任务。这种任务也叫做 future，因为它在将来返回一个结果。早期版本的 Task Parallel Library(TPL)的类名也叫做 Future，现在它是 Task 类的一个泛型版本。使用这个类时，可以定义任务返回的结果的类型。

由任务调用来返回结果的方法可以声明为任何返回类型。下面的示例方法 TaskWithResult()利用一个元组返回两个 int 值。该方法的输入可以是 void 或 object 类型，如下所示(代码文件 TaskSamples/Program.cs)：

```
static Tuple<int, int> TaskWithResult(object division)
{
  Tuple<int, int> div = (Tuple<int, int>)division;
  int result = div.Item1 / div.Item2;
  int reminder = div.Item1 % div.Item2;
  Console.WriteLine("task creates a result...");
```

```
    return Tuple.Create<int, int>(result, reminder);
}
```

元组参见第6章。

定义一个调用TaskWithResult()方法的任务时，要使用泛型类Task<TResult>。泛型参数定义了返回类型。通过构造函数，把这个方法传递给Func委托，第二个参数定义了输入值。因为这个任务在object参数中需要两个输入值，所以还创建了一个元组。接着启动该任务。Task实例t1的Result属性被禁用，并一直等到该任务完成。任务完成后，Result属性包含任务的结果。

```
var t1 = new Task<Tuple<int,int>>(TaskWithResult,
    Tuple.Create<int, int>(8, 3));
t1.Start();
Console.WriteLine(t1.Result);
t1.Wait();
Console.WriteLine("result from task: {0} {1}", t1.Result.Item1,
    t1.Result.Item2);
```

21.3.3 连续的任务

通过任务，可以指定在任务完成后，应开始运行另一个特定任务，例如，一个使用前一个任务的结果的新任务，如果前一个任务失败了，这个任务就应执行一些清理工作。

任务处理程序或者不带参数，或者带一个对象参数，而连续处理程序有一个Task类型的参数，这里可以访问起始任务的相关信息(代码文件TaskSamples/Program.cs)：

```
static void DoOnFirst()
{
    Console.WriteLine("doing some task {0}", Task.CurrentId);
    Thread.Sleep(3000);
}

static void DoOnSecond(Task t)
{
    Console.WriteLine("task {0} finished", t.Id);
    Console.WriteLine("this task id {0}", Task.CurrentId);
    Console.WriteLine("do some cleanup");
    Thread.Sleep(3000);
}
```

连续任务通过在任务上调用ContinueWith()方法来定义。也可以使用TaskFactory类来定义。t1.OnContinueWith(DoOnSecond)方法表示，调用DoOnSecond()方法的新任务应在任务t1结束时立即启动。在一个任务结束时，可以启动多个任务，连续任务也可以有另一个连续任务，如下面的例子所示：

```
Task t1 = new Task(DoOnFirst);
Task t2 = t1.ContinueWith(DoOnSecond);
Task t3 = t1.ContinueWith(DoOnSecond);
Task t4 = t2.ContinueWith(DoOnSecond);
```

无论前一个任务是如何结束的，前面的连续任务总是在前一个任务结束时启动。使用 TaskContinuationOptions 枚举中的值，可以指定，连续任务只有在起始任务成功(或失败)结束时启动。一些可能的值是 OnlyOnFaulted、NotOnFaulted、OnlyOnCanceled、NotOnCanceled 和 OnlyOnRanToCompletion。

```
Task t5 = t1.ContinueWith(DoOnError,
  TaskContinuationOptions.OnlyOnFaulted);
```

使用第 13 章介绍过的 await 关键字时，编译器生成的代码会使用连续任务。

21.3.4　任务层次结构

利用任务连续性，可以在一个任务结束后启动另一个任务。任务也可以构成一个层次结构。一个任务启动一个新任务时，就启动了一个父/子层次结构。

下面的代码段在父任务内部新建一个任务对象并启动任务。创建子任务的代码与创建父任务的代码相同，唯一的区别是这个任务从另一个任务内部创建。(代码文件 TaskSamples/Program.cs。)

```
static void ParentAndChild()
{
  var parent = new Task(ParentTask);
  parent.Start();
  Thread.Sleep(2000);
  Console.WriteLine(parent.Status);
  Thread.Sleep(4000);
  Console.WriteLine(parent.Status);
}

static void ParentTask()
{
  Console.WriteLine("task id {0}", Task.CurrentId);
  var child = new Task(ChildTask);
  child.Start();
  Thread.Sleep(1000);
  Console.WriteLine("parent started child");
}

static void ChildTask()
{
  Console.WriteLine("child");
  Thread.Sleep(5000);
  Console.WriteLine("child finished");
}
```

如果父任务在子任务之前结束，父任务的状态就显示为 WaitingForChildrenToComplete。所有的子任务也结束时，父任务的状态就变成 RanToCompletion。当然，如果父任务用 TaskCreationOptions 枚举中的 DetachedFromParent 创建子任务时，这就无效。

取消父任务，也会取消子任务。接下来就讨论取消架构。

21.4 取消架构

.NET 4.5 包含一个取消架构，允许以标准方式取消长时间运行的任务。每个阻塞调用都应支持这种机制。当然目前，并不是所有阻塞调用都实现了这个新技术，但越来越多的阻塞调用都支持它。已经提供了这种机制的技术有任务、并发集合类、并行 LINQ 和几种同步机制。

取消架构基于协作行为，它不是强制的。长时间运行的任务会检查它是否被取消，并返回控制权。

支持取消的方法接受一个 CancellationToken 参数。这个类定义了 IsCancellationRequested 属性，其中长时间运行的操作可以检查它是否应终止。长时间运行的操作检查取消的其他方式有：取消标记时，使用标记的 WaitHandle 属性，或者使用 Register()方法。Register()方法接受 Action 和 ICancelableOperation 类型的参数。Action 委托引用的方法在取消标记时调用。这类似于 ICancelableOperation，其中实现这个接口的对象的 Cancel()方法在执行取消操作时调用。

21.4.1 Parallel.For()方法的取消

本节以一个使用 Parallel.For()方法的简单例子开始。Parallel 类提供了 For()方法的重载版本，在重载版本中，可以传递 ParallelOptions 类型的参数。使用 ParallelOptions 类型，可以传递一个 CancellationToken 参数。CancellationToken 参数通过创建 CancellationTokenSource 来生成。由于 CancellationTokenSource 实现了 ICancelableOperation 接口，因此可以用 CancellationToken 注册，并允许使用 Cancel()方法取消操作。本例没有直接调用 Cancel 方法，而是使用了.NET 4.5 中的一个新方法 CancelAfter，在 500 毫秒后取消标记。

在 For()循环的实现代码内部，Parallel 类验证 CancellationToken 的结果，并取消操作。一旦取消操作，For()方法就抛出一个 OperationCanceledException 类型的异常，这是本例捕获的异常。使用 CancellationToken 可以注册取消操作时的信息。为此，需要调用 Register()方法，并传递一个在取消操作时调用的委托(代码文件 CancellationSamples/Program.cs)。

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("*** token canceled"));

// send a cancel after 500 ms
cts.CancelAfter(500);

try
{
  ParallelLoopResult result =
    Parallel.For(0, 100, new ParallelOptions()
      {
        CancellationToken = cts.Token,
      },
      x =>
      {
        Console.WriteLine("loop {0} started", x);
        int sum = 0;
        for (int i = 0; i < 100; i++)
        {
          Thread.Sleep(2);
          sum += i;
```

```
            }
            Console.WriteLine("loop {0} finished", x);
        });
    }
    catch (OperationCanceledException ex)
    {
        Console.WriteLine(ex.Message);
    }
```

运行应用程序，会得到如下结果，第 0、1、25、75 和 50 次迭代都启动了。这在一个有 4 个内核 CPU 的系统上运行。通过取消操作，所有其他的迭代操作都在启动之前就取消了。启动的迭代操作允许完成，因为取消操作总是以协作方式进行，以避免在取消迭代操作的中间泄露资源。

```
loop 0 started
loop 1 started
loop 25 started
loop 75 started
loop 50 started
** token cancelled
loop 75 finished
loop 0 finished
loop 50 finished
loop 25 finished
loop 1 finished
The operation was canceled.
```

21.4.2 任务的取消

同样的取消模式也可用于任务。首先，新建一个 CancellationTokenSource。如果仅需要一个取消标记，就可以访问 Task.Factory.CancellationToken，以使用默认的取消标记。接着，与前面的代码类似，在 500 毫秒后取消任务。在循环中执行主要工作的任务通过 TaskFactory 对象接受取消标记。在构造函数中，把取消标记赋予 TaskFactory。这个取消标记由任务用于检查 CancellationToken 的 IsCancellationRequested 属性，以确定是否请求了取消。

```
static void CancelTask()
{
  var cts = new CancellationTokenSource();
  cts.Token.Register(() => Console.WriteLine("*** task cancelled"));

  // send a cancel after 500 ms
  cts.CancelAfter(500);

  Task t1 = Task.Run(() =>
    {
      Console.WriteLine("in task");
      for (int i = 0; i < 20; i++)
      {
        Thread.Sleep(100);
        CancellationToken token = cts.Token;
        if (token.IsCancellationRequested)
        {
          Console.WriteLine("cancelling was requested, " +
             "cancelling from within the task");
```

```
          token.ThrowIfCancellationRequested();
          break;
        }
        Console.WriteLine("in loop");
      }
      Console.WriteLine("task finished without cancellation");
    }, cts.Token);

  try
  {
    t1.Wait();
  }
  catch (AggregateException ex)
  {
    Console.WriteLine("exception: {0}, {1}", ex.GetType().Name, ex.Message);
    foreach (var innerException in ex.InnerExceptions)
    {
      Console.WriteLine("inner excepion: {0}, {1}",
        ex.InnerException.GetType().Name, ex.InnerException.Message);
    }
  }
}
```

运行应用程序，可以看到任务启动了，运行了几个循环，并获得了取消请求。之后取消任务，并抛出 TaskCanceledException 异常，它是从方法调用 ThrowIfCancellationRequested()中启动的。调用者等待任务时，会捕获 AggregateException 异常，它包含内部异常 TaskCanceledException。例如，如果在一个也被取消的任务中运行 Parallel.For()方法，这就可以用于取消的层次结构。任务的最终状态是 Canceled。

```
in task
in loop
in loop
in loop
in loop
*** task cancelled
cancelling was requested, cancelling from within the task
exception: AggregateException, One or more errors occurred.
inner excepion: TaskCanceledException, A task was canceled.
```

21.5 线程池

本节介绍任务的后台基础：线程池。创建线程需要时间。如果有不同的短任务要完成，就可以事先创建许多线程，在应完成这些任务时发出请求。这个线程数最好在需要更多的线程时增加，在需要释放资源时减少。

不需要自己创建这样一个列表。该列表由 ThreadPool 类托管。这个类会在需要时增减池中线程的线程数，直到最大的线程数。池中的最大线程数是可配置的。在四核 CPU 中，默认设置为 1023 个工作线程和 1000 个 I/O 线程。也可以指定在创建线程池时应立即启动的最小线程数，以及线程池中可用的最大线程数。如果有更多的作业要处理，线程池中线程的个数也到了极限，最新的作业就

要排队，且必须等待线程完成其任务。

下面的示例应用程序首先要读取工作线程和 I/O 线程的最大线程数，把这些信息写入控制台中。接着在 for 循环中，调用 ThreadPool.QueueUserWorkItem()方法，传递一个 WaitCallback 类型的委托，把 JobForAThread()方法赋予线程池中的线程。线程池收到这个请求后，就会从池中选择一个线程，来调用该方法。如果线程池还没有运行，就会创建一个线程池，并启动第一个线程。如果线程池已经在运行，且有一个空闲线程来完成该任务，就把该任务传递给这个线程(代码文件 ThreadPoolSamples/Program.cs)。

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      int nWorkerThreads;
      int nCompletionPortThreads;
      ThreadPool.GetMaxThreads(out nWorkerThreads, out nCompletionPortThreads);
      Console.WriteLine("Max worker threads: {0}, " +
        "I/O completion threads: {1}", nWorkerThreads, nCompletionPortThreads);

      for (int i = 0; i < 5; i++)
      {
        ThreadPool.QueueUserWorkItem(JobForAThread);
      }
      Thread.Sleep(3000);
    }

    static void JobForAThread(object state)
    {
      for (int i = 0; i < 3; i++)
      {
        Console.WriteLine("loop {0}, running inside pooled thread {1}",
          i, Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(50);
      }
    }
  }
}
```

运行应用程序时，可以看到 1023 个工作线程的当前设置。5 个任务只由 4 个线程池中的线程处理(因为这是一个四核系统)，读者运行该程序的结果可能与此不同。也可以改变任务的睡眠时间和要处理的任务数，得到完全不同的结果。

```
Max worker threads: 1023, I/O completion threads: 1000
loop 0, running inside pooled thread 4
loop 0, running inside pooled thread 6
loop 0, running inside pooled thread 5
loop 0, running inside pooled thread 3
loop 1, running inside pooled thread 3
```

```
loop 1, running inside pooled thread 6
loop 1, running inside pooled thread 5
loop 1, running inside pooled thread 4
loop 2, running inside pooled thread 6
loop 2, running inside pooled thread 4
loop 2, running inside pooled thread 5
loop 2, running inside pooled thread 3
loop 0, running inside pooled thread 4
loop 1, running inside pooled thread 4
loop 2, running inside pooled thread 4
```

线程池使用起来很简单，但它有一些限制：

- 线程池中的所有线程都是后台线程。如果进程的所有前台线程都结束了，所有的后台线程就会停止。不能把入池的线程改为前台线程。
- 不能给入池的线程设置优先级或名称。
- 对于 COM 对象，入池的所有线程都是多线程单元(multithreaded apartment，MTA)线程。许多 COM 对象都需要单线程单元(single-threaded apartment，STA)线程。
- 入池的线程只能用于时间较短的任务。如果线程要一直运行(如 Word 的拼写检查器线程)，就应使用 Thread 类创建一个线程(或者在创建 Task 时使用 LongRunning 选项)。

21.6 Thread 类

如果需要更多控制，可以使用 Thread 类。该类允许创建前台线程，以及设置线程的优先级。

使用 Thread 类可以创建和控制线程。下面的代码是创建和启动一个新线程的简单例子。Thread 类的构造函数重载为接受 ThreadStart 和 ParameterizedThreadStart 类型的委托参数。ThreadStart 委托定义了一个返回类型为 void 的无参数方法。在创建了 Thread 对象后，就可以用 Start()方法启动线程(代码文件 ThreadSamples/Program.cs)：

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      var t1 = new Thread(ThreadMain);
      t1.Start();
      Console.WriteLine("This is the main thread.");
    }

    static void ThreadMain()
    {
      Console.WriteLine("Running in a thread.");
    }
  }
}
```

运行这个程序时，得到两个线程的输出：

```
This is the main thread.
Running in a thread.
```

不能保证哪个结果先输出。线程由操作系统调度，每次哪个线程在前面可以不同。

前面探讨了 lambda 表达式如何与异步委托一起使用。lambda 表达式还可以与 Thread 类一起使用，将线程方法的实现代码传送给 Thread 构造函数的实参：

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      var t1 = new Thread(() => Console.WriteLine("running in a thread, id: {0}",
         Thread.CurrentThread.ManagedThreadId));
      t1.Start();
      Console.WriteLine("This is the main thread, id: {0}",
         Thread.CurrentThread.ManagedThreadId);
    }
  }
}
```

应用程序的输出显示了线程名和 ID：

```
This is the main thread, id: 1
Running in a thread, id: 3.
```

21.6.1　给线程传递数据

给线程传递一些数据可以采用两种方式。一种方式是使用带 ParameterizedThreadStart 委托参数的 Thread 构造函数，另一种方式是创建一个自定义类，把线程的方法定义为实例方法，这样就可以初始化实例的数据，之后启动线程。

要给线程传递数据，需要某个存储数据的类或结构。这里定义了包含字符串的 Data 结构，但可以传递任意对象。

```
public struct Data
{
  public string Message;
}
```

如果使用了 ParameterizedThreadStart 委托，线程的入口点必须有一个 object 类型的参数，且返回类型为 void。对象可以强制转换为任意数据类型，这里是把消息写入控制台。

```
static void ThreadMainWithParameters(object o)
{
  Data d = (Data)o;
  Console.WriteLine("Running in a thread, received {0}", d.Message);
}
```

通过 Thread 类的构造函数，可以将新的入口点赋予 ThreadMainWithParameters，传递变量 d，以调用 Start()方法。

```
static void Main()
{
  var d = new Data { Message = "Info" };
  var t2 = new Thread(ThreadMainWithParameters);
  t2.Start(d);
}
```

给新线程传递数据的另一种方式是定义一个类(参见 MyThread 类)，在其中定义需要的字段，将线程的主方法定义为类的一个实例方法：

```
public class MyThread
{
  private string data;

  public MyThread(string data)
  {
    this.data = data;
  }

  public void ThreadMain()
  {
    Console.WriteLine("Running in a thread, data: {0}", data);
  }
}
```

这样，就可以创建 MyThread 的一个对象，给 Thread 类的构造函数传递对象和 ThreadMain()方法。线程可以访问数据。

```
var obj = new MyThread("info");
var t3 = new Thread(obj.ThreadMain);
t3.Start();
```

21.6.2 后台线程

只要有一个前台线程在运行，应用程序的进程就在运行。如果多个前台线程在运行，而 Main()方法结束了，应用程序的进程就仍然是激活的，直到所有前台线程完成其任务为止。

在默认情况下，用 Thread 类创建的线程是前台线程。线程池中的线程总是后台线程。

在用 Thread 类创建线程时，可以设置 IsBackground 属性，以确定该线程是前台线程还是后台线程。Main()方法将线程 t1 的 IsBackground 属性设置为 false(默认值)。在启动新线程后，主线程就把结束消息写入控制台中。新线程会写入启动消息和结束消息，在这个过程中它要睡眠 3 秒。在新线程会完成其工作前，这 3 秒钟有利于主线程结束。

```
class Program
{
  static void Main()
  {
    var t1 = new Thread(ThreadMain)
      { Name = "MyNewThread", IsBackground = false };
    t1.Start();
```

```
        Console.WriteLine("Main thread ending now.");
    }

    static void ThreadMain()
    {
        Console.WriteLine("Thread {0} started", Thread.CurrentThread.Name);
        Thread.Sleep(3000);
        Console.WriteLine("Thread {0} completed", Thread.CurrentThread.Name);
    }
}
```

尽管主线程会提前完成其工作，但在启动应用程序时，会看到写入控制台的完成消息。原因是新线程也是一个前台线程。

```
Main thread ending now.
Thread MyNewThread1 started
Thread MyNewThread1 completed
```

如果将用来启动新线程的 IsBackground 属性改为 true，显示在控制台上的结果就会不同。在控制台上，可以看到相同的结果——新线程的启动消息，但没有结束消息。如果线程没有正常结束，就也有可能看不到启动消息。

```
Main thread ending now.
Thread MyNewThread1 started
```

后台线程非常适合于完成后台任务。例如，如果关闭 Word 应用程序，拼写检查器继续运行其进程就没有意义了。在关闭应用程序时，拼写检查器线程就可以关闭。但是，组织 Outlook 消息库的线程应一直是激活的，直到关闭 Outlook，它才结束。

21.6.3 线程的优先级

前面提到，线程由操作系统调度。给线程指定优先级，就可以影响调度顺序。在改变优先级之前，必须理解线程调度器。操作系统根据优先级来调度线程。调度优先级最高的线程以在 CPU 上运行。线程如果在等待资源，它就会停止运行，并释放 CPU。

线程必须等待时有几个原因，例如，响应睡眠指令、等待磁盘 I/O 的完成，等待网络包的到达等。如果线程不是主动释放 CPU，线程调度器就会抢占该线程。线程有一个时间量，这意味着它可以持续使用 CPU，直到这个时间到达(这是指没有更高优先级的线程时)。如果优先级相同的多个线程等待使用 CPU，线程调度器就会使用一个循环调度规则，将 CPU 逐个交给线程使用。如果线程被其他线程抢占，它就会排在队列的最后。

只有优先级相同的多个线程在运行，才用得上时间量和循环规则。优先级是动态的。如果线程是 CPU 密集型的(一直需要 CPU，且不等待资源)，其优先级就降低为用该线程定义的基本优先级。如果线程在等待资源，它的优先级会提高。由于优先级的提高，线程很有可能在下次等待结束时获得 CPU。

在 Thread 类中，可以设置 Priority 属性，以影响线程的基本优先级。Priority 属性需要 ThreadPriority 枚举定义的一个值。定义的级别有 Highest、AboveNormal、Normal、BelowNormal 和 Lowest。

在给线程指定较高的优先级时要小心，因为这可能降低其他线程的运行概率。根据需要，可以短暂地改变优先级。

21.6.4 控制线程

调用 Thread 对象的 Start()方法，可以创建线程。但是，在调用 Start()方法后，新线程仍不是处于 Running 状态，而是处于 Unstarted 状态。只要操作系统的线程调度器选择了要运行的线程，线程就会改为 Running 状态。读取 Thread.ThreadState 属性，就可以获得线程的当前状态。

使用 Thread.Sleep()方法，会使线程处于 WaitSleepJoin 状态，在等待 Sleep()方法定义的时间段后，线程就会再次被唤醒。

要停止另一个线程，可以调用 Thread.Abort()方法。调用这个方法时，会在接到终止命令的线程中抛出一个 ThreadAbortException 类型的异常。用一个处理程序捕获这个异常，线程可以在结束前完成一些清理工作。如果调用了 Thread.ResetAbort，线程还有机会在接收到 ThreadAbortException 异常后继续运行。如果线程没有重置终止，接收到终止请求的线程的状态就从 AbortRequested 改为 Aborted。

如果需要等待线程的结束，就可以调用 Thread.Join()方法。Thread.Join()方法会停止当前线程，并把它设置为 WaitSleepJoin 状态，直到加入的线程完成为止。

21.7 线程问题

用多个线程编程并不容易。在启动访问相同数据的多个线程时，会间歇性地遇到难以发现的问题。如果使用任务、并行 LINQ 或 Parallel 类，也会遇到这些问题。为了避免这些问题，必须特别注意同步问题和多个线程可能发生的其他问题。下面探讨与线程相关的问题：争用条件和死锁。

21.7.1 争用条件

如果两个或多个线程访问相同的对象，并且对共享状态的访问没有同步，就会出现争用条件。为了说明争用条件，下面的例子定义一个 StateObject 类，它包含一个 int 字段和一个 ChangeState()方法。在 ChangeState()方法的实现代码中，验证状态变量是否包含 5。如果它包含，就递增其值。下一条语句是 Trace.Assert，它立刻验证 state 现在是否包含 6。

在给包含 5 的变量递增了 1 后，可能认为该变量的值就是 6。但事实不一定是这样。例如，如果一个线程刚刚执行完 If (state == 5)语句，它就被其他线程抢占，调度器运行另一个线程。第二个线程现在进入 if 体，因为 state 的值仍是 5，所以将它递增到 6。第一个线程现在再次被调度，在下一条语句中，state 递增到 7。这时就发生了争用条件，并显示断言消息(代码文件 ThreadingIssues/SampleTask.cs)。

```
public class StateObject
{
  private int state = 5;
```

```
    public void ChangeState(int loop)
    {
      if (state == 5)
      {
        state++;
        Trace.Assert(state == 6, "Race condition occurred after " +
          loop + " loops");
      }
      state = 5;
    }
}
```

下面通过给任务定义一个方法来验证这一点。SampleTask 类的 RaceCondition()方法将一个 StateObject 类作为其参数。在一个无限 while 循环中，调用 ChangeState()方法。变量 i 仅用于显示断言消息中的循环次数。

```
public class SampleTask
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
    StateObject state = o as StateObject;

    int i = 0;
    while (true)
    {
      state.ChangeState(i++);
    }
  }
}
```

在程序的 main()方法中，新建了一个 StateObject 对象，它由所有任务共享。通过使用传递给 Task 的 Run 方法的 lambda 表达式调用 RaceCondition 方法来创建 Task 对象。然后，主线程等待用户输入。但是，由于可能出现争用，所以程序很有可能在读取用户输入前就挂起：

```
static void RaceConditions()
{
  var state = new StateObject();
  for (int i = 0; i < 2; i++)
  {
    Task.Run(() => new SampleTask().RaceCondition(state));
  }
}
```

启动程序，就会出现争用条件。多久以后出现第一个争用条件要取决于系统以及将程序构建为发布版本还是调试版本。如果构建为发布版本，该问题的出现次数就会比较多，因为代码被优化了。如果系统中有多个 CPU 或使用双核/四核 CPU，其中多个线程可以同时运行，则该问题也会比单核 CPU 的出现次数多。在单核 CPU 中，因为线程调度是抢占式的，也会出现该问题，只是没有那么频繁。

图 21-1 显示在 64 个循环后，出现争用条件的程序断言。多次启动应用程序，总是会得到不同的结果。

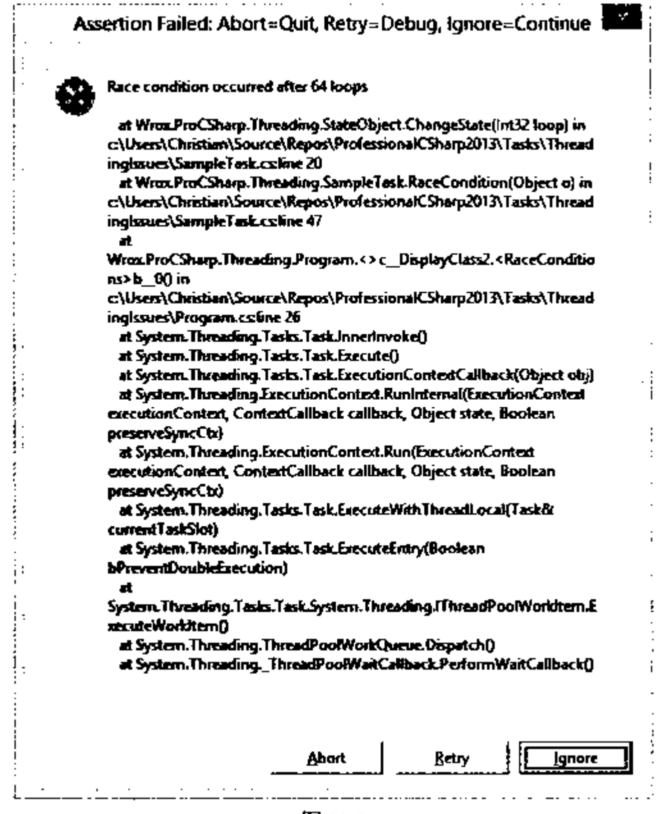


图 21-1

要避免该问题，可以锁定共享的对象。这可以在线程中完成：用下面的 lock 语句锁定在线程中共享的 state 变量。只有一个线程能在锁定块中处理共享的 state 对象。由于这个对象在所有的线程之间共享，因此如果一个线程锁定了 state，另一个线程就必须等待该锁定的解除。一旦接受锁定，线程就拥有该锁定，直到该锁定块的末尾才解除锁定。如果改变 state 变量引用的对象的每个线程都使用一个锁定，就不会出现争用条件。

```
public class SampleTask
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
    StateObject state = o as StateObject;

    int i = 0;
    while (true)
    {
      lock (state) // no race condition with this lock
      {
        state.ChangeState(i++);
      }
    }
  }
}
```

在使用共享对象时，除了进行锁定之外，还可以将共享对象设置为线程安全的对象。在下面的

代码中，ChangeState()方法包含一条 lock 语句。由于不能锁定 state 变量本身(只有引用类型才能用于锁定)，因此定义一个 object 类型的变量 sync，将它用于 lock 语句。如果每次 state 的值更改时，都使用同一个同步对象来锁定，就不会出现争用条件。

```
public class StateObject
{
  private int state = 5;
  private object sync = new object();

  public void ChangeState(int loop)
  {
    lock (sync)
    {
      if (state == 5)
      {
        state++;
        Trace.Assert(state == 6, "Race condition occurred after " +
           loop + " loops");
      }
      state = 5;
    }
  }
}
```

21.7.2　死锁

过多的锁定也会有麻烦。在死锁中，至少有两个线程被挂起，并等待对方解除锁定。由于两个线程都在等待对方，就出现了死锁，线程将无限等待下去。

为了说明死锁，下面实例化 StateObject 类型的两个对象，并把它们传递给 SampleTask 类的构造函数。创建两个任务，其中一个任务运行 Deadlock1()方法，另一个任务运行 Deadlock2()方法(代码文件 ThreadingIssues/Program.cs)：

```
var state1 = new StateObject();
var state2 = new StateObject();
new Task(new SampleTask(state1, state2).Deadlock1).Start();
new Task(new SampleTask(state1, state2).Deadlock2).Start();
```

Deadlock1()和 Deadlock2()方法现在改变两个对象 s1 和 s2 的状态，所以生成了两个锁。Deadlock1 方法先锁定 s1，接着锁定 s2。Deadlock2()方法先锁定 s2，再锁定 s1。现在，有可能 Deadlock1()方法中 s1 的锁定会被解除。接着，出现一次线程切换，Deadlock2()方法开始运行，并锁定 s2。第二个线程现在等待 s1 锁定的解除。因为它需要等待，所以线程调度器再次调度第一个线程，但第一个线程在等待 s2 锁定的解除。这两个线程现在都在等待，只要锁定块没有结束，就不会解除锁定。这是一个典型的死锁(代码文件 ThreadingIssues/SampleTask.cs)。

```
public class SampleTask
{
  public SampleTask(StateObject s1, StateObject s2)
  {
    this.s1 = s1;
    this.s2 = s2;
```

```
    }

    private StateObject s1;
    private StateObject s2;

    public void Deadlock1()
    {
      int i = 0;
      while (true)
      {
        lock (s1)
        {
          lock (s2)
          {
            s1.ChangeState(i);
            s2.ChangeState(i++);
            Console.WriteLine("still running, {0}", i);
          }
        }
      }
    }

    public void Deadlock2()
    {
      int i = 0;
      while (true)
      {
        lock (s2)
        {
          lock (s1)
          {
            s1.ChangeState(i);
            s2.ChangeState(i++);
            Console.WriteLine("still running, {0}", i);
          }
        }
      }
    }
  }
```

结果是，程序运行了许多次循环，不久就没有响应了。“仍在运行”的消息仅写入控制台中几次。同样，死锁问题的发生频率也取决于系统配置，每次运行的结果都不同。

在 Visual Studio 2013 中，可以在调试模式下运行程序，单击 Break All 按钮，打开 Parallel Tasks 窗口，如图 21-2 所示。其中显示，线程处于 Deadlock 状态。

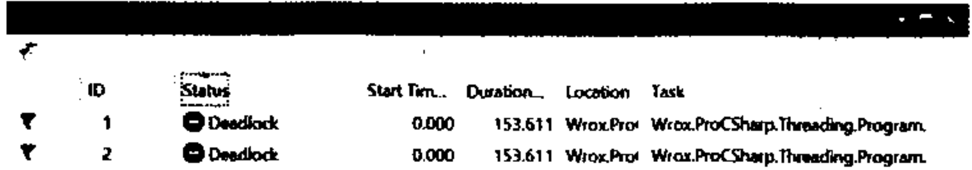


图 21-2

死锁问题并不总是如图 21-2 那样明显。一个线程锁定了 s1，接着锁定 s2；另一个线程锁定了 s2，接着锁定 s1。在本例中只需要改变锁定顺序，这两个线程就会以相同的顺序进行锁定。但是，锁定可能隐藏在方法的深处。为了避免这个问题，可以在应用程序的体系架构中，从一开始就设计好锁定顺序，也可以为锁定定义超时时间。如何定义超时时间详见下一节的内容。

21.8 同步

要避免同步问题，最好不要在线程之间共享数据。当然，这并不总是可行的。如果需要共享数据，就必须使用同步技术，确保一次只有一个线程访问和改变共享状态。注意，同步问题与争用条件和死锁有关。如果不注意这些问题，就很难在应用程序中找到问题的原因，因为线程问题是不定期发生的。

本节讨论可以用于多个线程的同步技术：

- lock 语句
- Interlocked 类
- Monitor 类
- SpinLock 结构
- WaitHandle 类
- Mutex 类
- Semaphore 类
- Event 类
- Barrier 类
- ReaderWriterLockSlim 类

lock 语句、Interlocked 类和 Monitor 类可用于进程内部的同步。Mutex 类、Event 类、SemaphoreSlim 类和 ReaderWriterLockSlim 类提供了多个进程之间的线程同步。

21.8.1 lock 语句和线程安全

C#为多个线程的同步提供了自己的关键字：lock 语句。lock 语句是设置锁定和解除锁定的一种简单方式。在添加 lock 语句之前，先进入另一个争用条件。SharedState 类说明了如何使用线程之间的共享状态，并共享一个整数值(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
   public int State { get; set; }
}
```

Job 类包含 DoTheJob()方法，该方法是新任务的入口点。通过其实现代码，将 SharedState 变量的 State 递增 50 000 次。sharedState 变量在这个类的构造函数中初始化(代码文件 Synchronization-Samples/Job.cs)：

```
public class Job
{
  SharedState sharedState;
```

```
    public Job(SharedState sharedState)
    {
      this.sharedState = sharedState;
    }
    public void DoTheJob()
    {
      for (int i = 0; i < 50000; i++)
      {
        sharedState.State += 1;
      }
    }
  }
```

在Main()方法中，创建一个SharedState对象，并把它传递给20个Task对象的构造函数。在启动所有的任务后，Main()方法进入另一个循环，等待20个任务都执行完毕。任务执行完毕后，把共享状态的合计值写入控制台中。因为执行了50 000次循环，有20个任务，所以写入控制台的值应是1 000 000。但是，事实常常并非如此(代码文件SynchronizationSamples/Program.cs)。

```
  class Program
  {
    static void Main()
    {
      int numTasks = 20;
      var state = new SharedState();
      var tasks = new Task[numTasks];

      for (int i = 0; i < numTasks; i++)
      {
        tasks[i] = Task.Run(() => new Job(state).DoTheJob());
      }

      for (int i = 0; i < numTasks; i++)
      {
        tasks[i].Wait();
      }
      Console.WriteLine("summarized {0}", state.State);
    }
  }
```

多次运行应用程序的结果如下所示：

```
summarized 314430
summarized 310683
summarized 315653
summarized 299973
summarized 326617
```

每次运行的结果都不同，但没有一个结果是正确的。如前所述，调试版本和发布版本的区别很大。根据使用的CPU类型，结果也不一样。如果将循环次数改为比较小的值，就会多次得到正确的值，但不是每次都正确。这个应用程序非常小，很容易看出问题，但该问题的原因在大型应用程序中就很难确定。

必须在这个程序中添加同步功能，这可以用lock关键字实现。用lock语句定义的对象表示，要

等待指定对象的锁定。只能传递引用类型。锁定值类型只是锁定了一个副本，这没有什么意义。如果对值类型使用了 lock 语句，C#编译器就会发出一个错误。进行了锁定后——只锁定了一个线程，就可以运行 lock 语句块。在 lock 语句块的最后，对象的锁定被解除，另一个等待锁定的线程就可以获得该锁定块了。

```
lock (obj)
{
  // synchronized region
}
```

要锁定静态成员，可以把锁放在 object 类型上：

```
lock (typeof(StaticClass))
{
}
```

使用 lock 关键字可以将类的实例成员设置为线程安全的。这样，一次只有一个线程能访问相同实例的 DoThis()和 DoThat()方法。

```
public class Demo
{
  public void DoThis()
  {
    lock (this)
    {
      // only one thread at a time can access the DoThis and DoThat methods
    }
  }
  public void DoThat()
  {
    lock (this)
    {
    }
  }
}
```

但是，因为实例的对象也可以用于外部的同步访问，而且我们不能在类自身中控制这种访问，所以应采用 SyncRoot 模式。通过 SyncRoot 模式，创建一个私有对象 syncRoot，将这个对象用于 lock 语句。

```
public class Demo
{
  private object syncRoot = new object();

  public void DoThis()
  {
    lock (syncRoot)
    {
      // only one thread at a time can access the DoThis and DoThat methods
    }
  }
  public void DoThat()
  {
```

```
        lock (syncRoot)
        {
        }
    }
}
```

使用锁定需要时间，且并不总是必须的。可以创建类的两个版本，一个同步版本，一个异步版本。下一个示例通过修改Demo类来说明。Demo类本身并不是同步的，这可以在DoThis()和DoThat()方法的实现中看出。该类还定义了IsSynchronized属性，客户可以从该属性中获得类的同步选项信息。为了获得该类的同步版本，可以使用静态方法Synchronized()传递一个非同步对象，这个方法会返回SynchronizedDemo类型的对象。SynchronizedDemo实现为派生自基类Demo的一个内部类，并重写基类的虚成员。重写的成员使用了SyncRoot模式。

```
public class Demo
{
  private class SynchronizedDemo: Demo
  {
    private object syncRoot = new object();
    private Demo d;

    public SynchronizedDemo(Demo d)
    {
      this.d = d;
    }
    public override bool IsSynchronized
    {
      get { return true; }
    }

    public override void DoThis()
    {
      lock (syncRoot)
      {
        d.DoThis();
      }
    }

    public override void DoThat()
    {
      lock (syncRoot)
      {
        d.DoThat();
      }
    }
  }

  public virtual bool IsSynchronized
  {
    get { return false; }
  }

  public static Demo Synchronized(Demo d)
  {
```

```
    if (!d.IsSynchronized)
    {
      return new SynchronizedDemo(d);
    }
    return d;
  }

  public virtual void DoThis()
  {
  }

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

必须注意，在使用 SynchronizedDemo 类时，只有方法是同步的。对这个类的两个成员的调用并没有同步。

首先修改异步的 SharedState 类，以使用 SyncRoot 模式。如果试图用 SyncRoot 模式锁定对属性的访问，使 SharedState 类变成线程安全的，就仍会出现前面描述的争用条件(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
  private int state = 0;
  private object syncRoot = new object();

  public int State // there's still a race condition,
                   // don't do this!
  {
    get { lock (syncRoot) {return state; }}
    set { lock (syncRoot) {state = value; }}
  }
}
```

调用方法 DoTheJob()的线程访问 SharedState 类的 get 存取器，以获得 state 的当前值，接着 get 存取器给 state 设置新值。在调用对象的 get 和 set 存取器期间，对象没有锁定，另一个线程可以获得临时值(代码文件 SynchronizationSamples/Job.cs)。

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
    sharedState.State += 1;
  }
}
```

所以，最好不改变 SharedState 类，让它依旧没有线程安全性(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
  public int State { get; set; }
}
```

然后在 DoTheJob 方法中，将 lock 语句添加到合适的地方(代码文件 SynchronizationSamples/Job.cs)：

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
    lock (sharedState)
    {
      sharedState.State += 1;
    }
  }
}
```

这样，应用程序的结果就总是正确的：

```
summarized 1000000
```

在一个地方使用 lock 语句并不意味着，访问对象的其他线程都正在等待。必须对每个访问共享状态的线程显式地使用同步功能。

当然，还必须修改 SharedState 类的设计，并作为一个原子操作提供递增方式。这是一个设计问题——把什么实现为类的原子功能？下面的代码片段锁定了递增操作(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
  private int state = 0;
  private object syncRoot = new object();

  public int State
  {
    get { return state; }
  }

  public int IncrementState()
  {
    lock (syncRoot)
    {
      return ++state;
    }
  }
}
```

锁定状态的递增还有一种更快的方式，如下节所示。

21.8.2 Interlocked 类

Interlocked 类用于使变量的简单语句原子化。i++不是线程安全的，它的操作包括从内存中获取

一个值，给该值递增 1，再将它存储回内存。这些操作都可能会被线程调度器打断。Interlocked 类提供了以线程安全的方式递增、递减、交换和读取值的方法。

与其他同步技术相比，使用 Interlocked 类会快得多。但是，它只能用于简单的同步问题。

例如，这里不使用 lock 语句锁定对 someState 变量的访问，把它设置为一个新值，以防它是空的，而可以使用 Interlocked 类，它比较快(代码文件 SynchronizationSamples/SharedState.cs)：

```
lock (this)
{
  if (someState == null)
  {
    someState = newState;
  }
}
```

这个功能相同但比较快的版本使用了 Interlocked.CompareExchange()方法：

```
    Interlocked.CompareExchange<SomeState>(ref someState,
newState, null);
```

不是像下面这样在 lock 语句中执行递增操作：

```
public int State
{
  get
  {
    lock (this)
    {
      return ++state;
    }
  }
}
```

而使用较快的 Interlocked.Increment()方法：

```
public int State
{
  get
  {
    return Interlocked.Increment(ref state);
  }
}
```

21.8.3 Monitor 类

lock 语句由 C#编译器解析为使用 Monitor 类。下面的 lock 语句：

```
lock (obj)
{
  // synchronized region for obj
}
```

被解析为调用 Enter()方法，该方法会一直等待，直到线程锁定对象为止。一次只有一个线程能锁定对象。只要解除了锁定，线程就可以进入同步阶段。Monitor 类的 Exit()方法解除了锁定。编译

器把 Exit()方法放在 try 块的 finally 处理程序中，所以如果抛出了异常，就也会解除该锁定(代码文件 SynchronizationSamples/Program.cs)。

try/finally 块详见第 16 章。

```
Monitor.Enter(obj);
try
{
  // synchronized region for obj
}
finally
{
  Monitor.Exit(obj);
}
```

与 C#的 lock 语句相比，Monitor 类的主要优点是：可以添加一个等待被锁定的超时值。这样就不会无限期地等待被锁定，而可以像下面的例子那样使用 TryEnter()方法，其中给它传递一个超时值，指定等待被锁定的最长时间。如果 obj 被锁定，TryEnter()方法就把布尔型的引用参数设置为 true，并同步地访问由对象 obj 锁定的状态。如果另一个线程锁定 obj 的时间超过了 500 毫秒，TryEnter()方法就把变量 lockTaken 设置为 false，线程不再等待，而是用于执行其他操作。也许在以后，该线程会尝试再次获得锁定。

```
bool lockTaken = false;
Monitor.TryEnter(obj, 500, ref lockTaken);
if (lockTaken)
{
  try
  {
    // acquired the lock
    // synchronized region for obj
  }
  finally
  {
    Monitor.Exit(obj);
  }
}
else
{
  // didn't get the lock, do something else
}
```

21.8.4 SpinLock 结构

如果基于对象的锁定对象(Monitor)的系统开销由于垃圾回收而过高，就可以使用 SpinLock 结构。SpinLock 结构是在.NET 4 开始引入的。如果有大量的锁定(例如，列表中的每个节点都有一个锁定)，且锁定的时间总是非常短，SpinLock 结构就很有用。应避免使用多个 SpinLock 结构，也不要调用任何可能阻塞的内容。

除了体系结构上的区别之外，SpinLock 结构的用法非常类似于 Monitor 类。获得锁定使用 Enter()或 TryEnter()方法，释放锁定使用 Exit()方法。SpinLock 结构还提供了属性 IsHeld 和 IsHeldByCurrentThread，指定它当前是否是锁定的。

传送 SpinLock 实例时要小心。因为 SpinLock 定义为结构，把一个变量赋予另一个变量会创建一个副本。总是通过引用传送 SpinLock 实例。

21.8.5 WaitHandle 基类

WaitHandle 是一个抽象基类，用于等待一个信号的设置。可以等待不同的信号，因为 WaitHandle 是一个基类，可以从中派生一些类。

在本章前面描述异步委托时，已经使用了 WaitHandle 基类。异步委托的 BeginInvoke()方法返回一个实现了 IAsycResult 接口的对象。使用 IAsycResult 接口，可以用 AsycWaitHandle 属性访问 WaitHandle 基类。在调用 WaitOne()方法时，线程会等待接收一个与等待句柄相关的信号(代码文件 AsyncDelegate/Program.cs)。

```
static void Main()
{
  TakesAWhileDelegate d1 = TakesAWhile;

  IAsyncResult ar = d1.BeginInvoke(1, 3000, null, null);
  while (true)
  {
    Console.Write(".");
    if (ar.AsyncWaitHandle.WaitOne(50, false))
    {
      Console.WriteLine("Can get the result now");
      break;
    }
  }
  int result = d1.EndInvoke(ar);
  Console.WriteLine("result: {0}", result);
}
```

使用 WaitHandle 基类可以等待一个信号的出现(WaitOne()方法)、等待必须发出信号的多个对象(WaitAll()方法)、或者等待多个对象中的一个(WaitAny()方法)。WaitAll()和 WaitAny()是 WaitHandle 类的静态方法，接收一个 WaitHandle 参数数组。

WaitHandle 基类有一个 SafeWaitHandle 属性，其中可以将一个本机句柄赋予一个操作系统资源，并等待该句柄。例如，可以指定一个 SafeFileHandle 等待文件 I/O 操作的完成，或者指定自定义的 SafeTransactionHandle，参见第 25 章。

因为 Mutex、EventWaitHandle 和 Semaphore 类派生自 WaitHandle 基类，所以可以在等待时使用它们。

21.8.6 Mutex 类

Mutex(mutual exclusion，互斥)是.NET Framework 中提供跨多个进程同步访问的一个类。它非常

类似于 Monitor 类，因为它们都只有一个线程能拥有锁定。只有一个线程能获得互斥锁定，访问受互斥保护的同步代码区域。

在 Mutex 类的构造函数中，可以指定互斥是否最初应由主调线程拥有，定义互斥的名称，获得互斥是否已存在的信息。在下面的示例代码中，第 3 个参数定义为输出参数，接收一个表示互斥是否为新建的布尔值。如果返回的值是 false，就表示互斥已经定义。互斥可以在另一个进程中定义，因为操作系统能够识别有名称的互斥，它由不同的进程共享。如果没有给互斥指定名称，互斥就是未命名的，不在不同的进程之间共享。

```
bool createdNew;
Mutex mutex = new Mutex(false, "ProCSharpMutex", out createdNew);
```

要打开已有的互斥，还可以使用 Mutex.OpenExisting()方法，它不需要用构造函数创建互斥时需要的相同.NET 权限。

由于 Mutex 类派生自基类 WaitHandle，因此可以利用 WaitOne()方法获得互斥锁定，在该过程中成为该互斥的拥有者。调用 ReleaseMutex()方法，即可释放互斥。

```
if (mutex.WaitOne())
{
  try
  {
    // synchronized region
  }
  finally
  {
    mutex.ReleaseMutex();
  }
}
else
{
  // some problem happened while waiting
}
```

由于系统能识别有名称的互斥，因此可以使用它禁止应用程序启动两次。在下面的 Windows Forms 应用程序中，调用了 Mutex 对象的构造函数。接着，验证名称为 SingletonWinAppMutex 的互斥是否存在。如果存在，应用程序就退出。

```
static class Program
{
  [STAThread]
  static void Main()
  {
    bool createdNew;
    var mutex = new Mutex(false, "SingletonWinAppMutex",
      out createdNew);
    if (!createdNew)
    {
      MessageBox.Show("You can only start one instance " +
        "of the application");
      Application.Exit();
      return;
    }
```

```
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new Form1());
    }
  }
```

21.8.7 Semaphore 类

信号量非常类似于互斥，其区别是，信号量可以同时由多个线程使用。信号量是一种计数的互斥锁定。使用信号量，可以定义允许同时访问受旗语锁定保护的资源的线程个数。如果需要限制可以访问可用资源的线程数，信号量就很有用。例如，如果系统有 3 个物理端口可用，就允许 3 个线程同时访问 I/O 端口，但第 4 个线程需要等待前 3 个线程中的一个释放资源。

.NET 4.5 为信号量功能提供了两个类 Semaphore 和 SemaphoreSlim。Semaphore 类可以命名，使用系统范围内的资源，允许在不同进程之间同步。SemaphoreSlim 类是对较短等待时间进行了优化的轻型版本。

在下面的示例应用程序中，在 Main()方法中创建了 6 个任务和一个计数为 3 的信号量。在 Semaphore 类的构造函数中，定义了锁定个数的计数，它可以用信号量(第二个参数)来获得，还定义了最初释放的锁定数(第一个参数)。如果第一个参数的值小于第二个参数，它们的差就是已经分配线程的计数值。与互斥一样，也可以给信号量指定名称，使之在不同的进程之间共享。这里定义信号量时没有指定名称，所以它只能在这个进程中使用。在创建了 SemaphoreSlim 对象之后，启动 6 个任务，它们都获得了相同的信号量。

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      int taskCount = 6;
      int semaphoreCount = 3;
      var semaphore = new SemaphoreSlim(semaphoreCount, semaphoreCount);
      var tasks = new Task[taskCount];

      for (int i = 0; i < taskCount; i++)
      {
        tasks[i] = Task.Run(() => TaskMain(semaphore));
      }

      Task.WaitAll(tasks);

      Console.WriteLine("All tasks finished");
    }
```

在任务的主方法 TaskMain()中，任务利用 Wait()方法锁定信号量。信号量的计数是 3，所以有 3 个任务可以获得锁定。第 4 个任务必须等待，这里还定义了最长的等待时间为 600 毫秒。如果在该

等待时间过后未能获得锁定，任务就把一条消息写入控制台，在循环中继续等待。只要获得了锁定，任务就把一条消息写入控制台，睡眠一段时间，然后解除锁定。在解除锁定时，在任何情况下一定要解除资源的锁定，这一点很重要。这就是在 finally 处理程序中调用 Semaphore 类的 Release()方法的原因。

```
    static void TaskMain(SemaphoreSlim semaphore)
    {
      bool isCompleted = false;
      while (!isCompleted)
      {
        if (semaphore.Wait(600))
        {
          try
          {
            Console.WriteLine("Task {0} locks the semaphore", Task.CurrentId);
            Thread.Sleep(2000);
          }
          finally
          {
            Console.WriteLine("Task {0} releases the semaphore", Task.CurrentId);
            semaphore.Release();
            isCompleted = true;
          }
        }
        else
        {
          Console.WriteLine("Timeout for task {0}; wait again",
             Task.CurrentId);
        }
      }
    }
  }
}
```

运行应用程序，可以看到有4个线程很快被锁定。ID 为 4 和 5 的线程需要等待。该等待会重复进行，直到其中一个被锁定的线程之一解除了信号量。

```
Task 1 locks the semaphore
Task 2 locks the semaphore
Task 3 locks the semaphore
Timeout for task 4; wait again
Timeout for task 4; wait again
Timeout for task 5; wait again
Timeout for task 4; wait again
Task 2 releases the semaphore
Task 5 locks the semaphore
Task 1 releases the semaphore
Task 6 locks the semaphore
Task 3 releases the semaphore
Task 4 locks the semaphore
Task 6 releases the semaphore
Task 5 releases the semaphore
Task 4 releases the semaphore
All tasks finished
```

21.8.8 Events 类

与互斥和信号量对象一样，事件也是一个系统范围内的资源同步方法。为了从托管代码中使用系统事件，.NET Framework 在 System.Threading 名称空间中提供了 ManualResetEvent、AutoResetEvent、ManualResetEventSlim 和 CountdownEvent 类。ManualResetEventSlim 和 CountdownEvent 类是.NET 4 新增的。

第 8 章介绍了 C#中的 event 关键字，它与 System.Threading 名称空间中的 event 类没有关系。event 关键字基于委托，而上述两个 event 类是.NET 封装器，用于系统范围内的本机事件资源的同步。

可以使用事件通知其他任务：这里有一些数据，并完成了一些操作等。事件可以发信号，也可以不发信号。使用前面介绍的 WaitHandle 类，任务可以等待处于发信号状态的事件。

调用 Set()方法，即可向 ManualResetEventSlim 发信号。调用 Reset()方法，可以使之返回不发信号的状态。如果多个线程等待向一个事件发信号，并调用了 Set()方法，就释放所有等待的线程。另外，如果一个线程刚刚调用了 WaitOne()方法，但事件已经发出信号，等待的线程就可以继续等待。

也通过调用 Set()方法向 AutoResetEvent 发信号。也可以使用 Reset()方法使之返回不发信号的状态。但是，如果一个线程在等待自动重置的事件发信号，当第一个线程的等待状态结束时，该事件会自动变为不发信号的状态。这样，如果多个线程在等待向事件发信号，就只有一个线程结束其等待状态，它不是等待时间最长的线程，而是优先级最高的线程。

为了说明 ManualResetEventSlim 类的事件，下面的 Calculator 类定义了 Calculation()方法，这是任务的入口点。在这个方法中，该任务接收用于计算的输入数据，将结果写入变量 result，该变量可以通过 Result 属性来访问。只要完成了计算(在随机的一段时间过后)，就调用 ManualResetEventSlim 类的 Set 方法，向事件发信号(代码文件 EventSample/Calculator.cs)。

```
public class Calculator
{
  private ManualResetEventSlim mEvent;

  public int Result { get; private set; }

  public Calculator(ManualResetEventSlim ev)
  {
     this.mEvent = ev;
  }

  public void Calculation(int x, int y)
  {
    Console.WriteLine("Task {0} starts calculation", Task.Current.Id);
    Thread.Sleep(new Random().Next(3000));
    Result = x + y;

    // signal the event-completed!
    Console.WriteLine("Task {0} is ready", Task.Current.Id);
    mEvent.Set();
```

```
    }
  }
```

程序的 Main()方法定义了包含 4 个 ManualResetEventSlim 对象的数组和包含 4 个 Calculator 对象的数组。每个 Calculator 在构造函数中用一个 ManualResetEventSlim 对象初始化，这样每个任务在完成时都有自己的事件对象来发信号。现在使用 Task 类，让不同的任务执行计算任务(代码文件 EventSample/Program.cs)。

```
class Program
{
  static void Main()
  {
    const int taskCount = 4;

    var mEvents = new ManualResetEventSlim[taskCount];
    var waitHandles = new WaitHandle[taskCount];
    var calcs = new Calculator[taskCount];

    for (int i = 0; i < taskCount; i++)
    {
      int i1 = i;
      mEvents[i] = new ManualResetEventSlim(false);
      waitHandles[i] = mEvents[i].WaitHandle;
      calcs[i] = new Calculator(mEvents[i]);

      Task.Run(() => calcs[i1].Calculation(i1 + 1, i1 + 3));
    }
    //...
```

WaitHandle 类现在用于等待数组中的任意一个事件。WaitAny()方法等待向任意一个事件发信号。与 ManualResetEvent 对象不同，ManualResetEventSlim 对象不派生自 WaitHandle 类。因此有一个 WaitHandle 对象的集合，它在 ManualResetEventSlim 类的 WaitHandle 属性中填充。从 WaitAny()方法返回的 index 值匹配传递给 WaitAny()方法的事件数组的索引，以提供发信号的事件的相关信息，使用该索引可以从这个事件中读取结果。

```
    for (int i = 0; i < taskCount; i++)
    {
      int index = WaitHandle.WaitAny(mEvents);
      if (index == WaitHandle.WaitTimeout)
      {
        Console.WriteLine("Timeout!!");
      }
      else
      {
        mEvents[index].Reset();
        Console.WriteLine("finished task for {0}, result: {1}",
            index, calcs[index].Result);
      }
    }
  }
}
```

启动应用程序，可以看到任务在进行计算，设置事件，以通知主线程，它可以读取结果了。在任意时间，依据是调试版本还是发布版本和硬件的不同，会看到按照不同的顺序，有不同数量的线程在执行任务。

```
Task 2 starts calculation
Task 3 starts calculation
Task 4 starts calculation
Task 1 starts calculation
Task 1 is ready
Task 4 is ready
finished task for 0, result: 4
Task 3 is ready
finished task for 3, result: 10
finished task for 1, result: 6
Task 2 is ready
finished task for 2, result: 8
```

在一个类似的场景中，为了把一些工作分支到多个任务中，并在以后合并结果，使用新的 CountdownEvent 类很有用。不需要为每个任务创建一个单独的事件对象，而只需要创建一个事件对象。CountdownEvent 类为所有设置了事件的任务定义了一个初始数字，在到达该计数后，就向 CountdownEvent 类发信号。

修改 Calculator 类，以使用 CountdownEvent 类替代 ManualResetEvent 类。不使用 Set()方法设置信号，而使用 CountdownEvent 类定义 Signal()方法(代码文件 EventSample/Calculator.cs)。

```
public class Calculator
{
  private CountdownEvent cEvent;

  public int Result { get; private set; }

  public Calculator(CountdownEvent ev)
  {
    this.cEvent = ev;
  }

  public void Calculation(int x, int y)
  {
    Console.WriteLine("Task {0} starts calculation", Task.Current.Id);
    Thread.Sleep(new Random().Next(3000));
    Result = x + y;

    // signal the event-completed!
    Console.WriteLine("Task {0} is ready", Task.Current.Id);
    cEvent.Signal();
  }
}
```

Main()方法现在可以简化，使它只需要等待一个事件。如果不像前面那样单独处理结果，这个新版本就很不错。

```
const int taskCount = 4;
var cEvent = new CountdownEvent(taskCount);
```

```
var calcs = new Calculator[taskCount];

for (int i = 0; i < taskCount; i++)
{
  calcs[i] = new Calculator(cEvent);

  taskFactory.StartNew(calcs[i].Calculation,
                            Tuple.Create(i + 1, i + 3));
}

cEvent.Wait();
Console.WriteLine("all finished");
for (int i = 0; i < taskCount; i++)
{
  Console.WriteLine("task for {0}, result: {1}", i, calcs[i].Result);
}
```

21.8.9 Barrier 类

对于同步，Barrier 类非常适用于其中工作有多个任务分支且以后又需要合并工作的情况。Barrier 类用于需要同步的参与者。激活一个任务时，就可以动态地添加其他参与者，例如，从父任务中创建子任务。参与者在继续之前，可以等待所有其他参与者完成其工作。

下面的应用程序使用一个包含 2 000 000 个字符串的集合。使用多个任务遍历该集合，并统计以 a、b、c 等开头的字符串个数。

FillData()方法创建一个集合，并用随机字符串填充它(代码文件 BarrierSample/Program.cs)：

```
public static IEnumerable<string> FillData(int size)
{
  var data = new List<string>(size);
  var r = new Random();
  for (int i = 0; i < size; i++)
  {
    data.Add(GetString(r));
  }
  return data;
}
private static string GetString(Random r)
{
  var sb = new StringBuilder(6);
  for (int i = 0; i < 6; i++)
  {
    sb.Append((char)(r.Next(26) + 97));
  }
  return sb.ToString();
}
```

CalculationInTask()方法定义了任务执行的作业。通过参数接收一个包含 4 项的元组。第 3 个参数是对 Barrier 实例的引用。任务完成其作业时，任务就会使用 RemoveParticipant()方法从 Barrier 类中删除它自己。

```
static int[] CalculationInTask(int jobNumber, int partitionSize,
  Barrier barrier, IList<string> coll)
```

```
{
  List<string> data = new List<string>(coll);

  int start = jobNumber * partitionSize;
  int end = start + partitionSize;
  Console.WriteLine("Task {0}: partition from {1} to {2}",
     Task.Current.Id, start, end);
  int[] charCount = new int[26];
  for (int j = start; j < end; j++)
  {
    char c = data[j][0];
    charCount[c - 97]++;
  }
  Console.WriteLine("Calculation completed from task {0}. {1} " +
    "times a, {2} times z", Task.Current.Id, charCount[0],
    charCount[25]);

  barrier.RemoveParticipant();
  Console.WriteLine("Task {0} removed from barrier, " +
    "remaining participants {1}", Task.Current.Id,
    barrier.ParticipantsRemaining);
  return charCount;
}
```

在 Main()方法中创建一个 Barrier 实例。在构造函数中，可以指定参与者的数量。在该示例中，这个数量是 3(numberTasks + 1)，因为该示例创建了两个任务，Main()方法本身也是一个参与者。使用 Task.Run 创建两个任务，把遍历集合的任务分为两个部分。启动该任务后，使用 SignalAndWait()方法，Main()方法在完成时发出信号，并等待所有其他参与者或者发出完成的信号，或者从 Barrier 类中删除它们。一旦所有的参与者都准备好，就提取任务的结果，并使用 Zip()扩展方法把它们合并起来。

```
static void Main()
{
  const int numberTasks = 2;
  const int partitionSize = 1000000;
  var data = new List<string>(FillData(partitionSize * numberTasks));

  var barrier = new Barrier(numberTasks + 1);

  var tasks = new Task<int[]>[numberTasks];
  for (int i = 0; i < participants; i++)
  {
    int jobNumber = i;
    tasks[i] = Task.Run(() => CalculationInTask(jobNumber, partitionSize,
      barrier, data));

    barrier.SignalAndWait();
    var resultCollection = tasks[0].Result.Zip(tasks[1].Result, (c1, c2) =>
    {
      return c1 + c2;
    });

    char ch = 'a';
```

```
    int sum = 0;
    foreach (var x in resultCollection)
    {
      Console.WriteLine("{0}, count: {1}", ch++, x);
      sum += x;
    }

    Console.WriteLine("main finished {0}", sum);
    Console.WriteLine("remaining {0}", barrier.ParticipantsRemaining);
  }
```

21.8.10 ReaderWriterLockSlim 类

为了使锁定机制允许锁定多个读取器(而不是一个写入器)访问某个资源，可以使用 ReaderWriterLockSlim 类。这个类提供了一个锁定功能，如果没有写入器锁定资源，就允许多个读取器访问资源，但只能有一个写入器锁定该资源。

ReaderWriterLockSlim 类的属性可获得读取阻塞或不阻塞的锁定，如 EnterReadLock()和 TryEnterReadLock()方法。还可以使用 EnterWriteLock()和 TryEnterWriteLock()方法获得写入锁定。如果任务先读取资源，之后写入资源，它就可以使用 EnterUpgradableReadLock()或 TryEnterUpgradableReadLock()方法获得可升级的读取锁定。有了这个锁定，就可以获得写入锁定，而无须释放读取锁定。

这个类的几个属性提供了当前锁定的相关信息，如 CurrentReadCount、WaitingReadCount、WaitingUpgradableReadCount 和 WaitingWriteCount。

下面的示例程序创建了一个包含 6 项的集合和一个 ReaderWriterLockSlim 对象。ReaderMethod 方法获得一个读取锁定，读取列表中的所有项，并把它们写到控制台中。WriterMethod()方法试图获得一个写入锁定，以改变集合的所有值。在 Main()方法中，启动 6 个线程，以调用 ReaderMethod()或 WriterMethod()方法(代码文件 ReaderWriterSample/Program.cs)。

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    private static List<int> items = new List<int>() { 0, 1, 2, 3, 4, 5};
    private static ReaderWriterLockSlim rwl =
      new ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

    static void ReaderMethod(object reader)
    {
      try
      {
        rwl.EnterReadLock();

        for (int i = 0; i < items.Count; i++)
        {
          Console.WriteLine("reader {0}, loop: {1}, item: {2}",
            reader, i, items[i]);
```

```
          Thread.Sleep(40);
        }
      }
      finally
      {
        rwl.ExitReadLock();
      }
    }

    static void WriterMethod(object writer)
    {
      try
      {
        while (!rwl.TryEnterWriteLock(50))
        {
          Console.WriteLine("Writer {0} waiting for the write lock",
             writer);
          Console.WriteLine("current reader count: {0}",
             rwl.CurrentReadCount);
        }
        Console.WriteLine("Writer {0} acquired the lock", writer);
        for (int i = 0; i < items.Count; i++)
        {
          items[i]++;
          Thread.Sleep(50);
        }
        Console.WriteLine("Writer {0} finished", writer);
      }
      finally
      {
        rwl.ExitWriteLock();
      }
    }

    static void Main()
    {
      var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
        TaskContinuationOptions.None);
      var tasks = new Task[6];
      tasks[0] = taskFactory.StartNew(WriterMethod, 1);
      tasks[1] = taskFactory.StartNew(ReaderMethod, 1);
      tasks[2] = taskFactory.StartNew(ReaderMethod, 2);
      tasks[3] = taskFactory.StartNew(WriterMethod, 2);
      tasks[4] = taskFactory.StartNew(ReaderMethod, 3);
      tasks[5] = taskFactory.StartNew(ReaderMethod, 4);

      for (int i = 0; i < 6; i++)
      {
        tasks[i].Wait();
      }
    }
  }
}
```

运行这个应用程序，可以看到第一个写入器先获得锁定。第二个写入器和所有的读取器需要等

待。接着，读取器可以同时工作，而第二个写入器仍在等待资源。

```
Writer 1 acquired the lock
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 1 finished
reader 4, loop: 0, item: 1
reader 1, loop: 0, item: 1
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 0, item: 1
reader 3, loop: 0, item: 1
reader 4, loop: 1, item: 2
reader 1, loop: 1, item: 2
reader 3, loop: 1, item: 2
reader 2, loop: 1, item: 2
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 2, item: 3
reader 1, loop: 2, item: 3
reader 2, loop: 2, item: 3
reader 3, loop: 2, item: 3
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 3, item: 4
reader 1, loop: 3, item: 4
reader 2, loop: 3, item: 4
reader 3, loop: 3, item: 4
reader 4, loop: 4, item: 5
reader 1, loop: 4, item: 5
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 4, item: 5
reader 3, loop: 4, item: 5
reader 4, loop: 5, item: 6
reader 1, loop: 5, item: 6
reader 2, loop: 5, item: 6
reader 3, loop: 5, item: 6
Writer 2 waiting for the write lock
current reader count: 4
Writer 2 acquired the lock
Writer 2 finished
```

21.9 Timer 类

.NET Framework 提供了几个 Timer 类，用于在某个时间间隔后调用某个方法。表 21-1 列出了 Timer 类及其名称空间和功能。

表 21-1

名 称 空 间	说　明
System.Threading	System.Threading 名称空间中的 Timer 类提供了核心功能。在构造函数中，可以传递一个委托，该委托应按照指定的时间间隔调用
System.Timers	System.Timers 名称空间中的 Timer 类是一个组件，因为它派生自 Component 基类。因此，可以把它从工具箱拖放到服务器应用程序(如 Windows 服务)的设计界面上。这个 Timer 类使用 System.Threading.Timer，但提供了基于事件的机制，而不是基于委托的机制
System.Windows.Forms	使用 System.Threading 和 System.Timers 名称空间中的 Timer 类，可以从不是主调线程的另一个线程中调用回调方法或事件方法。Windows 窗体控件绑定到创建它的线程上。对这个线程的回调通过 System.Windows.Forms 名称空间中的 Timer 类完成
System.Web.UI	System.Web.UI 名称空间中的 Timer 类是一个 AJAX 扩展，该扩展可以用于 Web 页面
System.Windows.Threading	System.Windows.Threading 名称空间中的 DispatcherTimer 类由 WPF 应用程序使用。DispatcherTimer 类运行在 UI 线程上

使用 System.Threading.Timer 类，可以把要调用的方法作为构造函数的第一个参数传递。这个方法必须满足 TimeCallback 委托的要求，该委托定义一个 void 返回类型和一个 object 参数。通过第二个参数，可以传递任意对象，用回调方法中的 object 参数接收对应的对象。例如，可以传递 Event 对象，向调用者发送信号。第 3 个参数指定第一次调用回调方法时的时间段。最后一个参数指定了回调的重复时间间隔。如果计时器应只触发一次，就把第 4 个参数设置为–1。

如果创建 Timer 对象后应改变时间间隔，就可以用 Change()方法传递新值(代码文件 TimerSample/Program.cs)：

```
private static void ThreadingTimer()
{
  var t1 = new System.Threading.Timer(TimeAction, null,
    TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(3));

  Thread.Sleep(15000);

  t1.Dispose();
  }

  static void TimeAction(object o)
  {
    Console.WriteLine("System.Threading.Timer {0:T}", DateTime.Now);
  }
```

System.Timer 名称空间中的 Timer 类的构造函数只需要一个时间间隔。经过该时间间隔后应调用的方法用 Elapsed 事件指定。这个事件需要一个 ElapsedEventHandler 类型的委托，这个委托需要 object 和 ElapsedEventArgs 参数，与下例的 TimeAction()方法相同。AutoReset 属性指定计时器是否重复触发。如果把这个属性设置为 false，事件就只触发一次。调用 Start()方法允许计时器触发事件。除了调用 Start()方法之外，还可以把 Enabled 属性设置为 true。在后台，Start()方法什么也不做。Stop()方法把 Enabled 属性设置为 false，以停止计时器。

```
private static void TimersTimer()
{
  var t1 = new System.Timers.Timer(1000);
  t1.AutoReset = true;
  t1.Elapsed += TimeAction;
  t1.Start();
  Thread.Sleep(10000);
  t1.Stop();

  t1.Dispose();
}

static void TimeAction(object sender, System.Timers.ElapsedEventArgs e)
{
  Console.WriteLine("System.Timers.Timer {0:T}", e.SignalTime );
}
```

21.10 数据流

Parallel 类、Task 类和 Parallel LINQ 为数据并行性提供了很多帮助。但是，这些类不能直接支持数据流的处理，以及并行转换数据。此时，需要使用 Task Parallel Library Data Flow(TPL Data Flow)。这个库必须作为一个 NuGet 包安装。该包中包含 System.Threading.Tasks.DataFlow 名称空间中的 System.Threading.Tasks.DataFlow 程序集。

关于 NuGet 包的安装详见第 17 章。

21.10.1 使用动作块

TPL Data Flow 的核心是数据块，这些数据块作为提供数据的源或者接收数据的目标，或者同时作为源和目标。下面看一个简单的示例，其中用一个数据块来接收一些数据并把数据写入控制台。下面的代码段定义了一个 ActionBlock，它接收一个字符串，并把字符串中的信息写入控制台。Main 方法在一个 while 循环中读取用户输入，然后调用 Post 方法把读入的所有字符串写入 ActionBlock。ActionBlock 异步处理消息，把信息写入到控制台：

```
static void Main()
{
  var processInput = new ActionBlock<string>(s =>
    {
      Console.WriteLine("user input: {0}", s);
    });

  bool exit = false;
  while (!exit)
  {
    string input = Console.ReadLine();
    if (string.Compare(input, "exit", ignoreCase: true) == 0)
    {
```

```
          exit = true;
        }
        else
        {
          processInput.Post(input);
        }
      }
    }
```

21.10.2　源和目标数据块

以前示例中分配给 ActionBlock 的方法执行时，ActionBlock 会使用一个任务来并行执行。通过检查任务和线程标识符并把它们写入控制台可以验证这一点。每个块都实现了 IDataflowBlock 接口，该接口包含了返回一个 Task 的属性 Completion，以及 Complete 和 Fault 方法。调用 Complete 方法后，块不再接受任何输入，也不再产生任何输出。调用 Fault 方法则把块放入失败状态。

如前所述，块既可以是源，也可以是目标，还可以同时是源和目标。在示例中，ActionBlock 是一个目标块，所以实现了 ITargetBlock 接口。ITargetBlock 派生自 IDataflowBlock，除了提供 IDataBlock 接口的成员以外，还定义了 OfferMessage 方法。OfferMessage 发送一条由块处理的消息。Post 是比 OfferMessage 更方便的一个方法，它实现为 ITargetBlock 接口的扩展方法。示例应用程序中也使用了 Post 方法。

ISourceBlock 接口由作为数据源的块实现。除了 IDataBlock 接口的成员以外，ISourceBlock 还提供了链接到目标块以及处理消息的方法。

BufferBlock 同时作为数据源和数据目标，它实现了 ISourceBlock 和 ITargetBlock。在下一个示例中，就使用这个 BufferBlock 来收发消息：

```
static BufferBlock<string> buffer = new BufferBlock<string>();
```

Producer 方法从控制台读取字符串，并通过调用 Post 方法把字符串写入到 BufferBlock 中：

```
static void Producer()
{
  bool exit = false;
  while (!exit)
  {
    string input = Console.ReadLine();
    if (string.Compare(input, "exit", ignoreCase: true) == 0)
    {
      exit = true;
    }
    else
    {
      buffer.Post(input);
    }
  }
}
```

Consumer 方法在一个循环中调用 ReceiveAsync 方法来接收 BufferBlock 中的数据。ReceiveAsync 是 ISourceBlock 接口的一个扩展方法：

```
static async void Consumer()
{
  while (true)
  {
```

```
        string data = await buffer.ReceiveAsync();
        Console.WriteLine("user input: {0}", data);
    }
}
```

现在，只需要启动消息的产生者和使用者。在 Main 方法中通过两个独立的任务完成启动操作：

```
static void Main()
{
    Task t1 = Task.Run(() => Producer());
    Task t2 = Task.Run(() => Consumer());
    Task.WaitAll(t1, t2);
}
```

运行应用程序时，产生者从控制台读取数据，使用者接收数据并把它们写入控制台。

21.10.3 连接块

本节将连接多个块，创建一个管道。首先，创建由块使用的 3 个方法。GetFileNames 方法接收一个目录作为参数，得到以.cs 为扩展名的文件名：

```
static IEnumerable<string> GetFileNames(string path)
{
    foreach (var fileName in Directory.EnumerateFiles(path, "*.cs"))
    {
        yield return fileName;
    }
}
```

LoadLines 方法以一个文件名列表作为参数，得到文件中的每一行：

```
static IEnumerable<string> LoadLines(IEnumerable<string> fileNames)
{
    foreach (var fileName in fileNames)
    {
        using (FileStream stream = File.OpenRead(fileName))
        {
            var reader = new StreamReader(stream);
            string line = null;
            while ((line = reader.ReadLine()) != null)
            {
                // Console.WriteLine("LoadLines {0}", line);
                yield return line;
            }
        }
    }
}
```

GetWords 方法接收一个 lines 集合作为参数，将其逐行分割，从而得到并返回一个单词列表：

```
static IEnumerable<string> GetWords(IEnumerable<string> lines)
{
    foreach (var line in lines)
    {
        string[] words = line.Split(' ', ';', '(', ')', '{', '}', '.', ',');
        foreach (var word in words)
        {
            if (!string.IsNullOrEmpty(word))
```

```
            yield return word;
        }
    }
}
```

为了创建管道，SetupPipeline 方法创建了 3 个 TransformBlock 对象。TransformBlock 是一个源和目标块，通过使用委托来转换源。第一个 TransformBlock 被声明为将一个字符串转换为 IEnumerable<string>。这种转换是通过 GetFileNames 方法完成的，GetFileNames 方法在传递给第一个块的构造函数的 lambda 表达式中调用。类似地，接下来的两个 TransformBlock 对象用于调用 LoadLines 和 GetWords 方法：

```
static ITargetBlock<string> SetupPipeline()
{
  var fileNamesForPath = new TransformBlock<string, IEnumerable<string>>(
    path =>
    {
      return GetFileNames(path);
    });

  var lines = new TransformBlock<IEnumerable<string>, IEnumerable<string>>(
    fileNames =>
    {
      return LoadLines(fileNames);
    });

  var words = new TransformBlock<IEnumerable<string>, IEnumerable<string>>(
    lines2 =>
      {
        return GetWords(lines2);
      });
```

定义的最后一个块是 ActionBlock。这个块只是一个用于接收数据的目标块，前面已经用过：

```
var display = new ActionBlock<IEnumerable<string>>(
  coll =>
  {
    foreach (var s in coll)
    {
      Console.WriteLine(s);
    }
  });
```

最后，将这些块彼此连接起来。fileNamesForPath 被链接到 lines 块，其结果被传递给 lines 块。lines 块链接到 words 块，words 块链接到 display 块。最后，返回用于启动管道的块：

```
  fileNamesForPath.LinkTo(lines);
  lines.LinkTo(words);
  words.LinkTo(display);

  return fileNamesForPath;
}
```

现在，Main 方法只需要启动管道。调用 Post 方法传递目录时，管道就会启动，并最终将单词从 C#源代码写入控制台。这里可以发出多个启动管道的请求，传递多个目录，并行执行这些任务：

```
static void Main()
```

```
{
  var target = SetupPipeline();
  target.Post("../..");
  Console.ReadLine();
}
```

通过对 TPL Data Flow 库的简单介绍，可以看到这种技术的主要用法。该库还提供了其他许多功能，例如以不同方式处理数据的不同块。BroadcastBlock 允许向多个目标传递输入源(例如将数据写入一个文件并显示该文件)，JoinBlock 将多个源连接到一个目标，BatchBlock 将输入作为数组进行批处理。使用 DataflowBlockOptions 选项可以配置块，例如一个任务中可以处理的最大项数，还可以向其传递取消标记来取消管道。使用链接技术，可以对消息进行筛选，只传递指定的消息，还可以进行配置，使得消息被传递到目标的开头而不是结尾，以便更加快速地处理最新的消息。

21.11 小结

本章介绍了如何通过 System.Threading 名称空间编写多线程应用程序，和如何通过 System.Threading.Tasks 名称空间编写多任务应用程序。在应用程序中使用多线程要仔细规划。太多的线程会导致资源问题，线程不足又会使应用程序执行缓慢，执行效果也不好。使用任务可以获得线程的抽象。这个抽象有助于避免创建过多的线程，因为线程是在池中重用的。

我们探讨了创建多个任务的各种方法，如 Parallel 类。通过使用 Parallel.Invoke、Parallel.ForEach 和 Parallel.For，可以实现任务和数据的并行性。还介绍了如何使用 Task 类来获得对并行编程的全面控制。任务可以在主调线程中异步运行，使用线程池中的线程，以及创建独立的新线程。任务还提供了一个层次结构模型，允许创建子任务，并且提供了一种取消完整层次结构的方法。

取消架构提供了一种标准机制，不同的类可以以相同的方法使用它来提前取消某个任务。

本章介绍了任务的后台原理，特别是 ThreadPool 类和 Thread 类，也可以使用自己的类。使用 Thread 类能够定义线程的前台和后台行为，以及为线程分配优先级。

.NET Framework 中的 System.Threading 名称空间提供了处理线程的多种方式，但这并不意味着.NET Framework 完成多线程中所有困难的任务。我们必须考虑线程的优先级和同步问题。本章讨论了这些问题，并介绍了如何在 C#应用程序中为它们编码。还论述了与死锁和争用条件相关的问题。如果要在 C#应用程序中使用多线程功能，就必须仔细规划。

下面是有关线程的几条规则：

- 尽力使同步要求最低。同步很复杂，且会阻塞线程。如果尝试避免共享状态，就可以避免同步。当然，这不总是可行。
- 类的静态成员应是线程安全的。通常，.NET Framework 中的类满足这个要求。
- 实例状态不需要是线程安全的。为了得到最佳性能，最好在类的外部使用同步功能，且不对类的每个成员使用同步功能。.NET Framework 类的实例成员一般不是线程安全的。在 MSDN 库中，对于.NET Framework 的每个类在“线程安全性”部分中可以找到相应的归档信息。

第 22 章介绍另一个.NET 核心主题：安全性。

第22章 安全性

本章要点

- 身份验证和授权
- 加密
- 资源的访问控制
- 代码访问安全性

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 身份验证示例
  - Windows Principal
  - 基于角色的安全
  - 应用程序服务
- 加密示例
  - 签名
  - 安全传输
- 文件访问控制
- 代码访问安全
  - 许可

22.1 概述

为了确保应用程序的安全，安全性有几个重要方面需要考虑。一是应用程序的用户，访问应用程序的是一个真正的用户，还是伪装成用户的某个人？如何确定这个用户是可以信任的？如本章所述，确保应用程序安全的用户方面是一个两阶段过程：用户首先需要进行身份验证，再进行授权，

以验证该用户是否可以使用需要的资源。

对于在网络上存储或发送的数据呢？例如，有人可以通过网络嗅探器访问这些数据吗？这里，数据的加密很重要。一些技术，如 Windows Communication Foundation(WCF)，通过简单的配置提供了加密功能，所以可以看到后台执行了什么操作。

另一方面是应用程序本身。如果应用程序驻留在 Web 提供程序上，如何禁止应用程序执行对服务器有害的操作？

本章将讨论.NET 中有助于管理安全性的一些特性，其中包括.NET 怎样避开恶意代码、怎样管理安全性策略，以及怎样通过编程访问安全子系统等。

22.2 身份验证和授权

安全性的两个基本支柱是身份验证和授权。身份验证是标识用户的过程，授权在验证了所标识用户是否可以访问特定资源之后进行。

22.2.1 标识和 Principal

使用标识可以验证运行应用程序的用户。WindowsIndentity 类表示一个 Windows 用户。如果没有用 Windows 账户标识用户，也可以使用实现了 IIdentity 接口的其他类。通过这个接口可以访问用户名、该用户是否通过身份验证，以及验证类型等信息。

principal 是一个包含用户的标识和用户的所属角色的对象。IPrincipal 接口定义了 Identity 属性和 IsInRole()方法，Identity 属性返回 IIdentity 对象；在 IsInRole()方法中，可以验证用户是否是指定角色的一个成员。角色是有相同安全权限的用户集合，同时它是用户的管理单元。角色可以是 Windows 组或自己定义的一个字符串集合。

.NET 中的 Principal 类有 WindowsPrincipal、GenericPrincipal 和 RolePrincipal。从.NET 4.5 开始，这些 Principal 类型派生于基类 ClaimsPrincipal。还可以创建实现了 IPrincipal 接口或派生于 ClaimsPrincipal 的自定义 Principal 类。

下面创建一个控制台应用程序，它可以访问某个应用程序中的主体，以便允许用户访问底层的 Windows 账户。这里需要导入 System.Security.Principal 和 System.Security.Claims 名称空间。首先，必须指定.NET 使用底层的 Windows 账户自动挂起主体。因为.NET 默认为只用通用的 Principal 填充主体。完成这项工作的代码如下所示(代码段 WindowsPrincipal/Program.cs)：

```
using System;
using System.Security.Claims;
using System.Security.Principal;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
```

使用 SetPrincipalPolicy()方法指定当前线程中的主体应保存一个 WindowsIdentity 对象。SetPrincipalPolicy()方法可以指定的其他选项有 NoPrincipal 和 UnauthenticatedPrincipal。所有的标识类，例如 WindowsIdentity，都实现了 IIdentity 接口，该接口包含 3 个属性(AuthenticationType、IsAuthenticated 和 Name)，便于所有的派生标识类实现它们。

下面添加一些代码，访问主体的属性：

```
var principal = WindowsPrincipal.Current as WindowsPrincipal;
var identity = principal.Identity as WindowsIdentity;
Console.WriteLine("IdentityType: {0}", identity.ToString());
Console.WriteLine("Name: {0}", identity.Name);
Console.WriteLine("'Users'?: {0} ",
    principal.IsInRole(WindowsBuiltInRole.User));
Console.WriteLine("'Administrators'? {0}",
    principal.IsInRole(WindowsBuiltInRole.Administrator));
Console.WriteLine("Authenticated: {0}", identity.IsAuthenticated);
Console.WriteLine("AuthType: {0}", identity.AuthenticationType);
Console.WriteLine("Anonymous? {0}", identity.IsAnonymous);
Console.WriteLine("Token: {0}", identity.Token);
```

从控制台应用程序中输出的结果如下所示。当然，根据计算机的配置和与账户相关的角色(在该账户下用户进行签名)，实际输出的结果也不尽相同。这里的账户是一个映射到 Windows 8 账户上的 Windows Live 账户，因此 AuthType 是 LiveSSP：

```
IdentityType: System.Security.Principal.WindowsIdentity
Name: THEOTHERSIDE\Christian
'Users'?: True
'Administrators'? False
Authenticated: True
AuthType: LiveSSP
Anonymous? False
Token: 488
```

很明显，如果能很容易访问当前用户及其角色的详细信息，然后使用那些信息决定允许或拒绝用户执行某些动作，这就非常有好处。利用角色和 Windows 用户组，管理员可以完成使用标准用户管理工具所能完成的工作，这样，在用户的角色改变时，通常可以避免更改代码。下面将详细讨论角色。

22.2.2 角色

基于角色的安全性可以很好地解决资源的访问问题。例如，在金融行业中，员工的角色决定了他们能够访问的信息和他们能够执行的操作。

此外，基于角色的安全性最好也与 Windows 账户或自定义用户目录一起使用，以便管理基于 Web 的资源的访问权限。例如，Web 站点可以限制用户对其内容的访问，直到用户用那个站点注册为止，而且只有用户成为那个 Web 站点的付费订阅者之后，才能访问站点上的特殊内容。在众多的方法中，只有 ASP.NET 能更容易实现基于角色的安全性，因为许多代码都基于服务器。

例如，要实现一个需要身份验证的 Web 服务，就可以使用 Windows 的账户子系统，并以这种方式编写一个 Web 方法。但是，要确保用户在访问该方法的功能之前，成为某一特定 Windows 用户组的成员。

设想有一个依赖于 Windows 账户的内联网应用程序的情景。系统有一个 Manager 组和一个 Assistant 组，根据用户在公司中的角色，把用户分配到其中的一个组中。假设应用程序包含一个显示员工信息的特性，且只允许 Managers 组中的成员访问这个特性。很容易使用代码检查当前的用户是否是 Manager 组的成员，以此来决定是否允许该用户访问该特性。

但是，如果以后决定重新安排账户组，并引入一个新组 Personnel，这个组的成员也可以访问员工的详细信息，就会出问题。此时就必须仔细检查并更新所有代码，以包括这个新组的规则。

更好的解决方案是创建权限(如 ReadEmployeeDetails)，把它赋予需要权限的组。如果代码对 ReadEmployeeDetails 权限应用了某项检查，更新应用程序以允许 Personnel 组中的成员访问员工信息就变得非常简单，只需要创建一个组，并把用户放到那个组中，然后把 ReadEmployeeDetails 权限赋予那个组即可。

22.2.3 声明基于角色的安全性

如同代码访问的安全性一样，可以使用命令式的请求，方法是调用 IPincipal 接口的 IsInRole() 方法或使用属性，实现基于角色的安全性请求(用户必须是 Administrator 组中的成员)。在类或方法级别上，可以使用 PincipalPermissional 特性声明性地说明权限的需求，其代码如下所示(代码文件 RoleBasedSecurity/Program.cs)：

```
using System;
using System.Security;
using System.Security.Principal;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      AppDomain.CurrentDomain.SetPrincipalPolicy(
          PrincipalPolicy.WindowsPrincipal);
      try
      {
        ShowMessage();
      }
      catch (SecurityException exception)
      {
        Console.WriteLine("Security exception caught ({0})",
                          exception.Message);
        Console.WriteLine("The current principal must be in the local" +
                          "Users group");
      }
    }

    [PrincipalPermission(SecurityAction.Demand, Role = "BUILTIN\\Users")]
    static void ShowMessage()
    {
      Console.WriteLine("The current principal is logged in locally ");
      Console.WriteLine("(member of the local Users group)");
    }
```

```
    }
}
```

除非在 Windows 本地 Users 组的用户环境中执行应用程序，否则 ShowMessage()方法将抛出一个异常。对于 Web 应用程序，运行 ASP.NET 代码的账户必须处于组中，但在实际应用中一定不会把这个账户添加到管理员组中。

如果使用本地 User 组中的账户运行上面的代码，则输出结果如下所示：

```
The current principal is logged in locally
(member of the local Users group)
```

22.2.4 声称

除了使用角色之外，还可以使用声称访问用户的信息。声称与实体相关，描述了实体的能力。实体通常是用户，也可以是应用程序。能力描述了实体允许执行的操作。这样，声称比角色模型灵活得多。

自从.NET 4.5 开始，所有的 principal 类都派生于基类 ClaimsPrincipal。这样，就可以使用 principal 对象的 Claims 属性访问用户的声称了。使用下面的代码段，可以把所有声称的消息写到控制台上：

```
Console.WriteLine();
Console.WriteLine("Claims");
foreach (var claim in principal.Claims)
{
  Console.WriteLine("Subject: {0}", claim.Subject);
  Console.WriteLine("Issuer: {0}", claim.Issuer);
  Console.WriteLine("Type: {0}", claim.Type);
  Console.WriteLine("Value type: {0}", claim.ValueType);
  Console.WriteLine("Value: {0}", claim.Value);
  foreach (var prop in claim.Properties)
  {
    Console.WriteLine("\tProperty: {0} {1}", prop.Key, prop.Value);
  }
  Console.WriteLine();
}
```

下面是从 Windows Live 账户中提取的一个声称，它提供了名称、主 ID 和组标识符等信息。

```
Claims
Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
Value type: http://www.w3.org/2001/XMLSchema#string
Value: THEOTHERSIDE\Christian

Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-5-21-1413171500-312083878-1364686672-1001
      Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
      windowssubauthority NTAuthority
```

```
Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-1-0
      Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
      windowssubauthority WorldAuthority

Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-5-21-1413171500-312083878-1364686672-1008
      Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
      windowssubauthority NTAuthority

...
```

22.2.5 客户端应用程序服务

Visual Studio 很容易对 ASP.NET Web 应用程序使用以前建立的身份验证服务。使用这个服务可以对 Windows 应用程序和 Web 应用程序使用相同的身份验证机制。这是一个提供程序模型，它主要基于 System.Web.Security 名称空间中的 Membership 和 Roles 类。使用 Membership 类可以验证、创建、删除和查找用户，改变密码以及与用户相关的许多其他操作。使用 Roles 类可以添加和删除角色，给用户获取角色，以及改变用户的角色。

角色和用户的存储位置取决于提供程序。ActiveDirectoryMembershipProvider 访问 Active Directory 中的用户和角色，SqlMembershipProvider 使用 SQL Server 数据库。对于客户端应用程序服务，.NET 4.5.1 有两个提供程序：ClientFormsAuthenticationMembershipProvider 和 ClientWindowsAuthenticationMembershipProvider。

下面使用客户端应用程序服务和 Forms 身份验证。为此，首先需要启动一个应用程序服务器，然后才能从 Windows 窗体或 WPF 中使用这个服务。

1. 应用程序服务

要使用客户端应用程序服务，可以创建一个 ASP.NET Web 服务项目，它提供了应用程序服务。该项目需要一个成员提供程序。可以使用已有的成员提供程序，也可以创建自定义提供程序。这里的示例代码定义了 SampleMembershipProvider 类，它派生自基类 MembershipProvider，该基类在 System.Web.ApplicationServices 程序集的 System.Web.Security 名称空间中定义。必须重写基类中的所有抽象方法。对于登录，只需要实现 ValidateUser()方法。所有其他方法都可以用 ApplicationName 属性抛出一个 NotSupportedException 异常。这里的示例代码使用包含用户名和密码的 Dictionary<string, string>。当然，也可以改为使用自己的实现代码，如从数据库中读取用户名和密码(代码文件 AppServices/SampleMembershipProvider.cs)。

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Web.Security;
```

```
namespace Wrox.ProCSharp.Security
{
  public class SampleMembershipProvider: MembershipProvider
  {
    private Dictionary<string, string> users =
        new Dictionary<string, string>();
    internal static string ManagerUserName = "Manager".ToLowerInvariant();
    internal static string EmployeeUserName = "Employee".ToLowerInvariant();

    public override void Initialize(string name, NameValueCollection config)
    {
      users.Add(ManagerUserName, "secret@Pa$$w0rd");
      users.Add(EmployeeUserName, "s0me@Secret");

      base.Initialize(name, config);
      }

      public override string ApplicationName
      {
        get
        {
          throw new NotImplementedException();
        }
        set
        {
          throw new NotImplementedException();
        }
      }

      // override abstract Membership members
      // ...

      public override bool ValidateUser(string username, string password)
      {
        if (users.ContainsKey(username.ToLowerInvariant()))
        {
          return password.Equals(users[username.ToLowerInvariant()]);
        }
        return false;
      }
    }
  }
```

为了使用角色，还需要实现一个角色提供程序。SampleRoleProvider类派生自基类RoleProvider，并实现GetRolesForUser()和IsUserInRole()方法(代码文件AppServices/SampleRoleProvider.cs):

```
using System;
using System.Collections.Specialized;
using System.Web.Security;

namespace Wrox.ProCSharp.Security
{
  public class SampleRoleProvider: RoleProvider
  {
    internal static string ManagerRoleName = "Manager".ToLowerInvariant();
```

```
    internal static string EmployeeRoleName = "Employee".ToLowerInvariant();

    public override void Initialize(string name, NameValueCollection config)
    {
       base.Initialize(name, config);
    }

    public override void AddUsersToRoles(string[] usernames,
        string[] roleNames)
    {
      throw new NotImplementedException();
    }

    // override abstract RoleProvider members
    // ...

    public override string[] GetRolesForUser(string username)
    {
      if (string.Compare(username, SampleMembershipProvider.ManagerUserName,
          true) == 0)
      {
        return new string[] { ManagerRoleName };
      }
      else if (string.Compare(username,
          SampleMembershipProvider.EmployeeUserName, true) == 0)
      {
        return new string[] { EmployeeRoleName };
      }
      else
      {
        return new string[0];
      }
    }

    public override bool IsUserInRole(string username, string roleName)
    {
      string[] roles = GetRolesForUser(username);
      foreach (var role in roles)
      {
        if (string.Equals(role, roleName))
        {
          return true;
        }
      }
      return false;
    }
  }
}
```

身份验证服务必须在Web.Config文件中配置。在产品系统上，从安全性角度来看，最好用包含应用程序服务的服务器配置SSL(配置文件AppServices/web.config)：

```
<system.web.extensions>
    <scripting>
       <webServices>
```

```
            <authenticationService enabled="true" requireSSL="false"/>
            <roleService enabled="true"/>
        </webServices>
    </scripting>
</system.web.extensions>
```

在<system.web>部分中，membership 和 rolemanager 元素必须配置为引用实现了成员和角色提供程序的类：

```
<system.web>
    <membership defaultProvider="SampleMembershipProvider">
        <providers>
            <add name="SampleMembershipProvider"
                    type="Wrox.ProCSharp.Security.SampleMembershipProvider"/>
        </providers>
    </membership>
    <roleManager enabled="true" defaultProvider="SampleRoleProvider">
        <providers>
            <add name="SampleRoleProvider"
                  type="Wrox.ProCSharp.Security.SampleRoleProvider"/>
        </providers>
    </roleManager>
```

在调试时，可以用项目属性的 Web 选项卡指定端口号和虚拟路径。示例应用程序使用端口 55555 和虚拟路径/AppServices。如果使用其他值，就需要修改客户端应用程序的配置。

现在就可以从客户端应用程序中使用应用程序服务。

2. 客户端应用程序

通过客户端应用程序使用 WPF。Visual Studio 有一个项目设置 Services，它允许使用客户端应用程序服务。这里可以设置 Forms 身份验证，把身份验证和角色服务的位置设置为前面定义的地址 http://localhost:55555/AppServices。这个项目配置只引用 System.Web 和 System.Web.Extensions 程序集，并修改应用程序的配置文件，以配置那些使用 ChientAuthenticationMembershipProvider 和 ClientRoleProvider 类的成员提供程序和角色提供程序，以及这些提供程序使用的 Web 服务的地址(配置文件 AuthenticationServices/app.config)。

```
<?xml version="1.0" encoding="utf - 8"?>
  <configuration>
    <system.web>
        <membership defaultProvider="ClientAuthenticationMembershipProvider">
            <providers>
                <add name="ClientAuthenticationMembershipProvider"
                     type="System.Web.ClientServices.Providers.
                     ClientFormsAuthenticationMembershipProvider,
                     System.Web.Extensions, Version=4.0.0.0, Culture=neutral,
                     PublicKeyToken=31bf3856ad364e35" serviceUri=
  "http://localhost:55555/AppServices/Authentication_JSON_AppService.axd" />
            </providers>
        </membership>
        <roleManager defaultProvider="ClientRoleProvider" enabled="true">
            <providers>
          <add name="ClientRoleProvider"
```

```
                     type="System.Web.ClientServices.Providers.ClientRoleProvider,
                     System.Web.Extensions, Version=4.0.0.0, Culture=neutral,
                     PublicKeyToken=31bf3856ad364e35" serviceUri=
                     "http://localhost:55555/AppServices/Role_JSON_AppService.axd"
                     cacheTimeout="86400" />
        </providers>
      </roleManager>
    </system.web>
  </configuration>
```

Windows 应用程序仅使用标签、文本框、密码框和按钮控件，如图 22-1 所示。只有登录成功，才会显示标签的内容 User Validated。

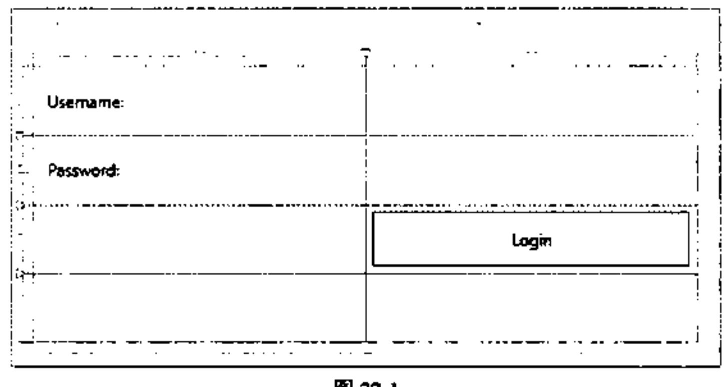


图 22-1

Button.Click 事件的处理程序调用 Membership 类的 ValidateUser()方法。对于成员 API，配置了 ClientAuthenticationMembershipProvider。这个提供程序首先调用 Web 服务，接着调用 SampleMembershipProvider 类的 ValidateUser()方法，验证登录是否成功。如果成功，标签 labelValidatedInfo 就可见；否则弹出一个消息框(代码文件 AuthenticationServices/MainWindow.xaml.cs)：

```
private void OnLogin(object sender, RoutedEventArgs e)
{
   try
   {
      if (Membership.ValidateUser(textUsername.Text,
           textPassword.Password))
      {
        // user validated!
        labelValidatedInfo.Visibility = Visibility.Visible;
      }
      else
      {
        MessageBox.Show("Username or password not valid",
             "Client Authentication Services", MessageBoxButton.OK,
             MessageBoxImage.Warning);
      }
   }
   catch (WebException ex)
   {
      MessageBox.Show(ex.Message, "Client Application Services",
```

```
                MessageBoxButton.OK, MessageBoxImage.Error);
        }
    }
```

22.3 加密

机密数据应得到保护，从而使未授权的用户不能读取它们。这对于在网络中发送的数据或存储的数据都有效。可以用对称或不对称密钥来加密这些数据。

通过对称密钥，可以使用同一个密钥进行加密和解密。与不对称的加密相比，加密和解密使用不同的密钥：公钥/私钥。如果使用一个公钥进行加密，就应使用对应的私钥进行解密，而不是使用公钥解密。同样，如果使用一个私钥加密，就应使用对应的公钥解密，而不是使用私钥解密。

公钥/私钥总是成对创建。公钥可以由任何人使用，它甚至可以放在 Web 站点上，但私钥必须安全地加锁。为了说明加密过程，下面看看使用公钥和私钥的例子。

如果 Alice 给 Bob 发了一封电子邮件，如图 22-2 所示，并且 Alice 希望能保证除了 Bob 外，其他人都不能阅读该邮件，所以她就使用 Bob 的公钥。邮件是使用 Bob 的公钥加密的。Bob 打开该邮件，并使用他秘密存储的私钥解密。这种方式可以保证除了 Bob 外，其他人都不能阅读 Alice 的邮件。

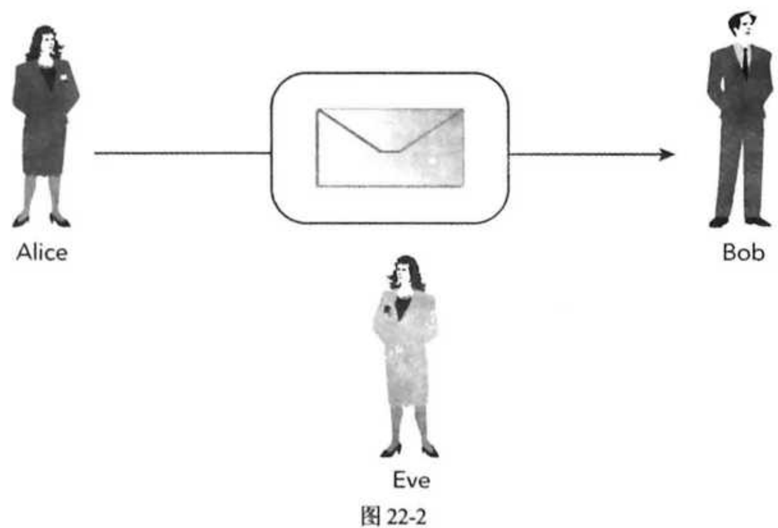


图 22-2

但这还有一个问题：Bob 不能确保邮件是 Alice 发送来的。Eve 可以使用 Bob 的公钥加密发送给 Bob 的邮件并假装是 Alice。我们使用公钥/私钥把这条规则扩展一下。下面再次从 Alice 给 Bob 发送电子邮件开始。在 Alice 使用 Bob 的公钥加密邮件之前，她添加了自己的签名，再使用自己的私钥加密该签名。然后使用 Bob 的公钥加密邮件。这样就保证了除 Bob 外，其他人都不能阅读该邮件。在 Bob 解密邮件时，他检测到一个加密的签名。这个签名可以使用 Alice 的公钥来解密。而 Bob 可以访问 Alice 的公钥，因为这个密钥是公钥。在解密了签名后，Bob 就可以确定是 Alice 发送了电子邮件。

使用对称密钥的加密和解密算法比使用非对称密钥的算法快得多。对称密钥的问题是密钥必须

以安全的方式互换。在网络通信中，一种方式是先使用非对称的密钥进行密钥互换，再使用对称密钥加密通过网络发送的数据。

在.NET Framework 中，可以使用 System.Security.Cryptography 名称空间中的类来加密。它实现了几个对称算法和非对称算法。有几个不同的算法类用于不同的目的。一些类以 Cng 作为前缀或后缀。Cng 是 Cryptography Next Generation 的简称，是本机 Crypto API 的更新版本，这个 API 可以使用基于提供程序的模型，编写独立于算法的程序。

表 22-1 列出了 System.Security.Cryptography 名称空间中的加密类及其功能。没有 Cng、Managed 或 CryptoServiceProvider 后缀的类是抽象基类，如 MD5。Managed 后缀表示这个算法用托管代码实现，其他类可能封装了本地 Windows API 调用。CryptoServiceProvider 后缀用于实现了抽象基类的类，Cng 后缀用于利用新 Cryptography CNG API 的类。

表 22-1

类 别	类	说 明
散列	MD5、MD5Cng SHA1、SHA1Managed、SHA1Cng SHA256、SHA256Managed、SHA256Cng SHA384、SHA384Managed、SHA384Cng SHA512、SHA512Managed、SHA512Cng，RIPEMD160、RIPEMD160Managed	散列算法的目标是从任意长度的二进制字符串中创建一个长度固定的散列值。这些算法和数字签名一起用于保证数据的完整性。如果再次散列相同的二进制字符串，会返回相同的散列结果。MD5(Message Digest Algorithm 5，消息摘要算法 5)由 RSA 实验室开发，比 SHA1 快。SHA1 在抵御暴力攻击方面比较强大。SHA 算法由美国国家安全局(NSA)设计。MD5 使用 128 位的散列长度，SHA1 使用 160 位。其他 SHA 算法在其名称中包含了散列长度。SHA512 是这些算法中最强大的，其散列长度为 512 位，它也是最慢的。RIPEMD160 使用 160 位的散列长度，替代了 128 位的 MD4 和 MD5。RIPEMD 在 EU 项目 RIPE(Race Integrity Primitives Evaluation)中开发
对称	DES、DESCryptoServiceProvider TripleDES、TripleDESCryptoServiceProvider Aes, AesCryptoServiceProvider、AesManaged RC2、RC2CryptoServiceProvider Rijandel、RijandelManaged	对称密钥算法使用相同的密钥进行数据的加密和解密。现在认为 DES(Data Encryption Standard，数据加密标准)是不安全的，因为它只使用 56 位的密钥长度，可以在不超过 24 小时的时间内破解。Triple-DES 是 DES 的继承者，其密钥长度是 168 位，但它提供的有效安全性只有 112 位。AES(Advanced Encryption Standard，高级加密标准)的密钥长度是 128、192 或 256 位。Rijandel 非常类似于 AES，它只是在密钥长度方面的选项较多。AES 是美国政府采用的加密标准
非对称	DSA、DSACryptoServiceProvider ECDsa、ECDsaCng ECDiffieHellman、ECDiffieHellmanCng RSA、RSACryptoServiceProvider	非对称算法使用不同的密钥进行加密和解密。RSA(Rivest，Shamir，Adleman)是第一个用于签名和加密的算法。这个算法广泛用于电子商务协议。DSA(Digital Signature Algorithm，数字签名算法)是用于数字签名的一个美国联邦政府标准。ECDSA(Elliptic Curve DSA，椭圆曲线数字签名算法)和 ECDiffieHellman 使用基于椭圆曲线组的算法。这些算法比较安全，且使用较短的密钥长度。例如，DSA 的密钥长度为 1024 位，其安全性类似于 ECDSA 的 160 位。因此，ECDSA 比较快。ECDiffieHellman 算法用于以安全的方式在公共信道中交换私钥

下面用例子说明如何通过编程使用这些算法。

22.3.1 签名

第一个例子说明了如何使用 ECDSA 算法进行签名。Alice 创建了一个签名，它用 Alice 的私钥加密，可以使用 Alice 的公钥访问。因此保证该签名来自于 Alice。

首先，看看 Main()方法中的主要步骤：创建 Alice 的密钥，给字符串“Alice”签名，最后使用公钥验证该签名是否真的来自于 Alice。要签名的消息使用 Encoding 类转换为一个字节数组。要把加密的签名写入控制台，包含该签名的字节数组应使用 Convert.ToBase64 String()方法转换为一个字符串(代码文件 SigningDemo/Program.cs)。

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Wrox.ProCSharp.Security
{
   class Program
   {
      internal static CngKey aliceKeySignature;
      internal static byte[] alicePubKeyBlob;

      static void Main()
      {
         CreateKeys();

         byte[] aliceData = Encoding.UTF8.GetBytes("Alice");
         byte[] aliceSignature = CreateSignature(aliceData, aliceKeySignature);
         Console.WriteLine("Alice created signature: {0}",
               Convert.ToBase64String(aliceSignature));

         if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))
         {
            Console.WriteLine("Alice signature verified successfully");
         }
      }
```

千万不要使用 Encoding 类把加密的数据转换为字符串。Encoding 类验证和转换 Unicode 不允许使用的无效值，因此把字符串转换回字节数组会得到另一个结果。

CreateKeys()方法为 Alice 创建新的密钥对。因为这个密钥对存储在一个静态字段中，所以可以从其他方法中访问它。CngKey 类的 Create()方法把该算法作为一个参数，为算法定义密钥对。通过 Export()方法，导出密钥对中的公钥。这个公钥可以提供给 Bob，来验证签名。Alice 保留其私钥。除了使用 CngKey 类创建密钥对之外，还可以打开存储在密钥存储器中的已有密钥。通常 Alice 在其私有存储器中有一个证书，其中包含了一个密钥对，该存储器可以用 CngKey.Open()方法访问。

```
static void CreateKeys()
{
   aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP256);
```

```
    alicePubKeyBlob = aliceKeySignature.Export
        (CngKeyBlobFormat.GenericPublicBlob);
}
```

有了密钥对，Alice就可以使用ECDsaCng类创建签名了。这个类的构造函数从Alice那里接收包含公钥和私钥的CngKey类。再使用私钥，通过SignData()方法给数据签名。

```
static byte[] CreateSignature(byte[] data, CngKey key)
{
  byte[] signature;
  using (var signingAlg = new ECDsaCng(key))
  {
    signature = signingAlg.SignData(data);
    signingAlg.Clear();
  }
  return signature;
}
```

要验证签名是否真的来自于Alice，Bob使用Alice的公钥检查签名。包含公钥blob的字节数组可以用静态方法Import()导入CngKey对象。然后使用ECDsaCng类，调用VerifyData()方法来验证签名。

```
    static bool VerifySignature(byte[] data, byte[] signature, byte[] pubKey)
    {
      bool retValue = false;
      using (CngKey key = CngKey.Import(pubKey,
          CngKeyBlobFormat.GenericPublicBlob))
      using (var signingAlg = new ECDsaCng(key))
      {
        retValue = signingAlg.VerifyData(data, signature);
        signingAlg.Clear();
      }
      return retValue;
    }
  }
}
```

22.3.2 交换密钥和安全传输

下面是一个比较复杂的例子，它使用Diffie Hellman算法交换一个对称密钥，以进行安全的传输。Main()方法包含了其主要功能。Alice创建了一条加密的消息，并把它发送给Bob。在此之前，要先为Alice和Bob创建密钥对。Bob只能访问Alice的公钥，Alice也只能访问Bob的公钥(代码文件SecureTransfer/Program.cs)。

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Security
{
  class Program
```

```
{
  static CngKey aliceKey;
  static CngKey bobKey;
  static byte[] alicePubKeyBlob;
  static byte[] bobPubKeyBlob;

  static void Main()
  {
    Run();
    Console.ReadLine();
  }

  private async static void Run()
  {
    try
    {
      CreateKeys();
      byte[] encrytpedData = await AliceSendsData("secret message");
      await BobReceivesData(encrytpedData);
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

在 CreateKeys()方法的实现代码中，使用 EC Diffie Hellman 256 算法创建密钥。

```
private static void CreateKeys()
{
    aliceKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
    bobKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
    alicePubKeyBlob = aliceKey.Export(CngKeyBlobFormat.EccPublicBlob);
    bobPubKeyBlob = bobKey.Export(CngKeyBlobFormat.EccPublicBlob);
}
```

在 AliceSendsData()方法中，包含文本字符的字符串使用 Encoding 类转换为一个字节数组。创建一个 ECDiffieHellmanCng 对象，用 Alice 的密钥对初始化它。Alice 调用 DeriveKeyMaterial()方法，从而使用其密钥对和 Bob 的公钥创建一个对称密钥。返回的对称密钥使用对称算法 AES 加密数据。AesCryptoServiceProvider 需要密钥和一个初始化矢量(IV)。IV 从 GenerateIV()方法中动态生成，对称密钥用 EC Diffie Hellman 算法交换，但还必须交换 IV。从安全性角度来看，在网络上传输未加密的 IV 是可行的——只是密钥交换必须是安全的。IV 存储为内存流中的第一项内容，其后是加密的数据，其中，CryptoStream 类使用 AesCryptoServiceProvider 类创建的 encryptor。在访问内存流中的加密数据之前，必须关闭加密流。否则，加密数据就会丢失最后的位。

```
private async static Task<byte[]> AliceSendsData(string message)
{
  Console.WriteLine("Alice sends message: {0}", message);
  byte[] rawData = Encoding.UTF8.GetBytes(message);
  byte[] encryptedData = null;

  using (var aliceAlgorithm = new ECDiffieHellmanCng(aliceKey))
```

```
    using (CngKey bobPubKey = CngKey.Import(bobPubKeyBlob,
          CngKeyBlobFormat.EccPublicBlob))
    {
      byte[] symmKey = aliceAlgorithm.DeriveKeyMaterial(bobPubKey);
      Console.WriteLine("Alice creates this symmetric key with " +
          "Bobs public key information: {0}",
          Convert.ToBase64String(symmKey));

      using (var aes = new AesCryptoServiceProvider())
      {
        aes.Key = symmKey;
        aes.GenerateIV();
        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        using (MemoryStream ms = new MemoryStream())
        {
          // create CryptoStream and encrypt data to send
          var cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write);

          // write initialization vector not encrypted
          await ms.WriteAsync(aes.IV, 0, aes.IV.Length);
          cs.Write(rawData, 0, rawData.Length);
          cs.Close();
          encryptedData = ms.ToArray();
        }
        aes.Clear();
      }
    }
    Console.WriteLine("Alice: message is encrypted: {0}",
        Convert.ToBase64String(encryptedData));;
    Console.WriteLine();
    return encryptedData;
  }
```

Bob 从 BobReceivesData()方法的参数中接收加密数据。首先，必须读取未加密的初始化矢量。AesCryptoServiceProvider 类的 BlockSize 属性返回块的位数。位数除以 8，就可以计算出字节数。最快的方式是把数据右移 3 位。右移 1 位就是除以 2，右移 2 位就是除以 4，右移 3 位就是除以 8。在 for 循环中，包含未加密 IV 的原字节的前几个字节写入数组 iv 中。接着用 Bob 的密钥对实例化一个 ECDiffieHellmanCng 对象。使用 Alice 的公钥，从 DeriveKeyMaterial()方法中返回对称密钥。

比较 Alice 和 Bob 创建的对称密钥，可以看出所创建的密钥值相同。使用这个对称密钥和初始化矢量，来自 Alice 的消息就可以用 AesCryptoServiceProvider 类解密。

```
  private static void BobReceivesData(byte[] encryptedData)
  {
    Console.WriteLine("Bob receives encrypted data");
    byte[] rawData = null;

    var aes = new AesCryptoServiceProvider();

    int nBytes = aes.BlockSize 3;
    byte[] iv = new byte[nBytes];
    for (int i = 0; i < iv.Length; i++)
      iv[i] = encryptedData[i];
```

```
using (var bobAlgorithm = new ECDiffieHellmanCng(bobKey))
using (CngKey alicePubKey = CngKey.Import(alicePubKeyBlob,
      CngKeyBlobFormat.EccPublicBlob))
{
  byte[] symmKey = bobAlgorithm.DeriveKeyMaterial(alicePubKey);
  Console.WriteLine("Bob creates this symmetric key with " +
      "Alices public key information: {0}",
      Convert.ToBase64String(symmKey));

  aes.Key = symmKey;
  aes.IV = iv;

  using (ICryptoTransform decryptor = aes.CreateDecryptor())
  using (MemoryStream ms = new MemoryStream())
  {
    var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Write);
    cs.Write(encryptedData, nBytes, encryptedData.Length - nBytes);
    cs.Close();

    rawData = ms.ToArray();

    Console.WriteLine("Bob decrypts message to: {0}",
        Encoding.UTF8.GetString(rawData));
  }
  aes.Clear();
}
}
```

运行应用程序，会在控制台上看到如下输出。来自 Alice 的消息被加密，Bob 用安全交换的对称密钥解密它。

```
Alice sends message: secret message
Alice creates this symmetric key with Bobs public key information:
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=
Alice: message is encrypted: 3C5U9CpYxnoFTk3Ew2V0T5Po0Jgryc5R7Te8ztau5N0=

Bob receives encrypted message
Bob creates this symmetric key with Alices public key information:
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=
Bob decrypts message to: secret message
```

22.4 资源的访问控制

在操作系统中，资源(如文件和注册表键，以及命名管道的句柄)都使用访问控制列表(ACL)来保护。图 22-3 显示了这个映射的结构。资源有一个关联的安全描述符。安全描述符包含了资源拥有者的信息，并引用了两个访问控制列表：自由访问控制列表(DACL)和系统访问控制列表(SACL)。DACL 用来确定谁有访问权；SACL 用来确定安全事件日志的审核规则。ACL 包含一个访问控制项(ACE)列表。ACE 包含类型、安全标识符和权限。在 DACL 中，ACE 的类型可以是允许访问或拒绝访问。可以用文件设置和获得的权限是创建、读取、写入、删除、修改、改变许可和获得拥有权。

读取和修改访问控制的类在 System.Security.AccessControl 名称空间中。下面的程序说明了如何

从文件中读取访问控制列表。

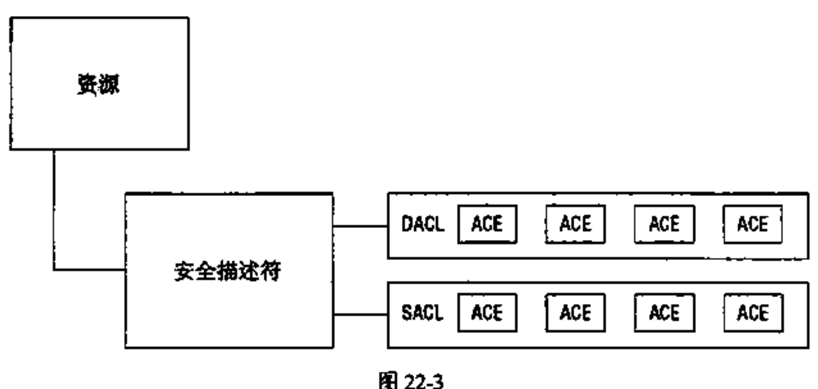


图 22-3

FileStream 类定义了 GetAccessControl()方法，该方法返回一个 FileSecurity 对象。FileSecurity 是一个.NET 类，它表示文件的安全描述符。FileSecurity 类派生自基类 ObjectSecurity、CommonObjectSecurity、NativeObjectSecurity 和 FileSystemSecurity。其他表示安全描述符的类有 CryptoKeySecurity、EventWaitHandleSecurity、MutexSecurity、RegistrySecurity、SemaphoreSecurity、PipeSecurity 和 ActiveDirectorySecurity。所有这些对象都可以使用访问控制列表来保护。一般情况下，对应的.NET 类定义了 GetAccessControl()方法，返回相应的安全类；例如，Mutex.GetAccessControl()方法返回一个 MutexSecurity 类，PipeStream.GetAccessControl()方法返回一个 PipeSecurity 类。

FileSecurity 类定义了读取、修改 DACL 和 SACL 的方法。GetAccessRules()方法以 AuthorizationRuleCollection 类的形式返回 DACL。要访问 SACL，可以使用 GetAuditRules 方法。

在 GetAccessRules()方法中，可以确定是否应使用继承的访问规则(不仅仅是用对象直接定义的访问规则)。最后一个参数定义了应返回的安全标识符的类型。这个类型必须派生自基类 IdentityReference。可能的类型有 NTAccount 和 SecurityIdentifier。这两个类都表示用户或组。NTAccount 类按名称查找安全对象，SecurityIdentifier 类按唯一的安全标识符查找安全对象。

返回的 AuthorizationRuleCollection 包含 AuthorizationRule 对象。AuthorizationRule 对象是 ACE 的.NET 表示。在这里的例子中，因为访问一个文件，所以 AuthorizationRule 对象可以强制转换为 FileSystemAccessRule 类型。在其他资源的 ACE 中，存在不同的.NET 表示，例如 MutexAccessRule 和 PipeAccessRule。在 FileSystemAccessRule 类中，AccessControlType、FileSystemRights 和 IdentityReference 属性返回 ACE 的相关信息(代码文件 FileAccessControl/Program.cs)。

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main(string[] args)
    {
      string filename = null;
      if (args.Length == 0)
        return;
```

```
        filename = args[0];

        using (FileStream stream = File.Open(filename, FileMode.Open))
        {
          FileSecurity securityDescriptor = stream.GetAccessControl();
          AuthorizationRuleCollection rules =
              securityDescriptor.GetAccessRules(true, true,
              typeof(NTAccount));

          foreach (AuthorizationRule rule in rules)
          {
            var fileRule = rule as FileSystemAccessRule;
            Console.WriteLine("Access type: {0}", fileRule.AccessControlType);
            Console.WriteLine("Rights: {0}", fileRule.FileSystemRights);
            Console.WriteLine("Identity: {0}", fileRule.IdentityReference.Value);
            Console.WriteLine();
          }
        }
      }
    }
  }
```

运行应用程序，并传递一个文件名，就可以看到文件的访问控制列表。这里的输出列出了管理员和系统的全部控制权限、通过身份验证的用户的修改权限，以及属于 Users 组的所有用户的读取和执行权限。

```
Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators

Access type: Allow
Rights: FullControl
Identity: NT AUTHORITY\SYSTEM

Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators

Access type: Allow
Rights: FullControl
Identity: TheOtherSide\Christian
```

设置访问权限非常类似于读取访问权限。要设置访问权限，几个可以得到保护的资源类提供了 SetAccessControl()和 ModifyAccessControl()方法。这里的示例代码调用 File 类的 SetAccessControl()方法，以修改文件的访问控制列表。给这个方法传递一个 FileSecurity 对象。FileSecurity 对象用 FileSystemAccessRule 对象填充。这里列出的访问规则拒绝 Sales 组的写入访问权限，给 Everyone 组提供了读取访问权限，并给 Developers 组提供了全部控制权限。

只有定义了 Windows 组 Sales 和 Developers，这个程序才能在系统上运行。可以修改程序，使用自己环境下的可用组。

```
private static void WriteAcl(string filename)
{
  var salesIdentity = new NTAccount("Sales");
  var developersIdentity = new NTAccount("Developers");
  var everyOneIdentity = new NTAccount("Everyone");

  var salesAce = new FileSystemAccessRule(salesIdentity,
      FileSystemRights.Write, AccessControlType.Deny);
  var everyoneAce = new FileSystemAccessRule(everyOneIdentity,
      FileSystemRights.Read, AccessControlType.Allow);
  var developersAce = new FileSystemAccessRule(developersIdentity,
      FileSystemRights.FullControl, AccessControlType.Allow);

  var securityDescriptor = new FileSecurity();
  securityDescriptor.SetAccessRule(everyoneAce);
  securityDescriptor.SetAccessRule(developersAce);
  securityDescriptor.SetAccessRule(salesAce);

  File.SetAccessControl(filename, securityDescriptor);
}
```

打开 Properties 窗口，在 Windows 资源管理器中选择一个文件，选择 Security 选项卡，列出访问控制列表，就可以验证访问规则。

22.5 代码访问安全性

在基于角色的安全性中，可以定义用户允许做什么。同样，代码访问安全性指定了代码能做什么。.NET 4 简化了这个模型，删除了.NET 4 之前的版本存在的复杂的策略配置，添加了第 2 级安全透明性。第 2 级安全透明性区分开了允许进行授权调用(如调用本地代码)的代码和不允许进行授权调用的代码。代码分为 3 类：

- “security-critical(安全第一)”代码可以运行任意代码，这种代码不能由透明代码调用。
- “safe-critical(安全重要)”代码可以由透明代码调用，安全验证由这种代码执行。
- 透明(Transparent)代码可以执行的操作非常有限。这种代码只能运行在指定的权限集中，且运行在沙盒中。它不能包含不安全或无法验证的代码，也不能调用 security-critical 代码。

如果编写 Windows 应用程序，就不应用受限的代码权限。运行在桌面上的应用程序有完全信任权限，且可以包含任意代码——假定它不是由系统管理员定义。沙盒可用于 Silverlight 应用程序和 ASP.NET 应用程序，这些应用程序驻留在 Web 提供程序中，或者有自定义功能，如用 Managed Add-in Framework 运行插件。

本节将讨论如何应用第 2 级安全透明性，如何通过透明代码使用需要的.NET 权限。

22.5.1 第 2 级安全透明性

可以用 SecurityRules 特性注解程序集，并设置 SecurityRuleSet.Level2，以应用.NET 4 新增的级别(这是自从.NET 4 开始的默认级别)。为了向后兼容，可以把它设置为 Level1。

```
[assembly: SecurityRules(SecurityRuleSet.Level2)]
```

如果设置 SecurityTransparent 特性，完整的程序集就不会执行任何授权或不安全的操作。这个程序集只能调用其他透明代码或 safe-critical 代码。这个特性只能应用于完整的程序集。

```
[assembly: SecurityTransparent()]
```

AllowPartiallyTrustedCallers 特性位于透明代码和其他类别的代码之间。使用这个特性，代码默认是透明的，但个别类型或成员可以有其他特性：

```
[assembly: AllowPartiallyTrustedCallers()]
```

如果没有应用上述任何特性，代码就是 security-critical 代码。但是，可以给个别类型和成员应用 SecuritySafeCritical 特性，从而使它们可以由透明代码调用。

```
[assembly: SecurityCritical()]
```

22.5.2 权限

如果代码运行在沙盒中，沙盒就可以定义.NET 权限，以定义代码允许执行的操作。运行在桌面上的应用程序有完全信任权限，而运行在沙盒中的应用程序只允许执行主机给沙盒授予的权限所定义的操作。还可以为从桌面应用程序中启动的应用程序域定义权限，这需要使用沙盒 API。

应用程序域详见第 19 章。

权限是允许(或禁止)每个代码组执行的动作。例如，权限包括“读取文件系统中的文件”、“写入 Active Directory”和“使用套接字打开网络连接”等。有几个预定义的权限，也可以创建自己的权限。

.NET 权限独立于操作系统权限。.NET 权限仅由 CLR 验证。程序集需要特定操作的权限(例如，File 类需要 FileIOPermission)，CLR 验证该程序集是否被授予了该权限，以便它可以继续执行。

可以应用于程序集或从代码中申请的权限有非常精细的权限列表。表 22-2 列出了 CLR 提供的代码访问权限。从中可以看出，使用这些权限，可以很好地控制代码允许做什么和不允许做什么：

表 22-2

权 限	说 明
DirectoryServicesPermission	通过 System.DirectoryServices 类控制访问 Active Directory 的能力
DnsPermission	控制使用 TCP/IP 域名系统(DNS)的能力
EnvironmentPermission	控制读写环境变量的能力
EventLogPermission	控制读写事件日志的能力
FileDialogPermission	控制访问用户在 Open 对话框中访问已选择的文件的能力。该权限通常用于没有赋予 FileIOPermission 权限，不能对文件进行有限的访问时
FileIOPermission	控制处理文件的能力(其中包括读文件、写文件、向文件追加内容，创建、更改和访问文件夹)

(续表)

权 限	说 明
IsolatedStorageFilePermission	控制访问私有虚拟文件系统的能力
IsolatedStoragePermission	控制访问孤立存储器的能力，存储器与个别用户相关，并具有代码的标识的一些特征，孤立存储器详见第24章
MessageQueuePermission	控制通过 Microsoft Message Queue 使用消息队列的能力
PerformanceCounterPermission	控制利用性能计数器的能力
PrintingPermission	控制打印的能力
ReflectionPermission	控制使用 System.Reflection 在运行时查找类型信息的能力
RegistryPermission	控制读、写、创建、删除注册表键和值的能力
SecurityPermission	控制执行、断言权限、调用非托管的代码、忽略验证和其他权力的能力
ServiceControllerPermission	控制 Windows 服务的能力
SQLClientPermission	控制使用 SQL Server 的.NET 数据提供程序访问 SQL Server 数据库的能力
UIPermission	控制访问用户界面的能力
WebPermission	控制连接 Web 或接受与 Web 之间连接的能力

对于上面的每一个权限类，通常可以指定级别更高的粒度。例如，DirectoryServicesPermission 类可以区分读和写访问权限，也可以确定允许访问或拒绝访问目录服务中的哪些项。

1. 权限集

权限集是权限的集合。在权限集中，不需要把每个权限都应用于代码；权限组合为权限集。例如，有 FullTrust 权限的程序集拥有对所有资源的全部访问权限。有 LocalIntranet 权限的程序集是受限的，也就是说，除了使用隔离的存储器之外，不能写入文件系统。可以创建包含所需权限的自定义权限集。

把权限赋予代码组，就不需要单独处理每个程序集。通常在程序块中应用权限，这就是.NET 提供权限集合的概念的原因。这些是组合为已命名集合的代码访问权限列表，以下列表解释了可以即装即用的 7 个已命名权限集：

- FullTrust——没有权限的限制。
- SkipVerification——不进行验证。
- Execution——运行受保护的资源，但是不能访问它们。
- Nothing——没有授予任何权限，代码不能执行。
- LocalIntranet——指定权限全集的一个子集。例如，文件 I/O 只能在程序集源自的共享上进行读取访问。在.NET 3.5 和以前版本(在.NET 3.5 SP1 之前)中，这个权限集在应用程序运行在网络共享上时使用。
- Internet——未知来源的代码的默认策略，这是限制最严格的策略。例如，在这个权限集合下执行的代码没有文件 I/O 能力，不能读写事件日志，也不能读写环境变量。
- Everything——授予这个集合中列出的所有权限，除了忽略代码验证的权限。管理员可以改变这个权限集合中的权限。默认策略要更严格时，可以使用这个权限集。

只能修改 Everything 权限集的定义，而其他权限集是固定的，不能改变。当然，还可以创建自己的权限集。

2. 通过编程要求权限

程序集可以用声明或编程的方式要求权限。下面的代码段说明了如何使用 DemandFileIOPermission()方法要求权限。如果导入 System.Security.Permissions 名称空间，就可以创建一个 FileIOPermission 对象，并调用这个对象的 Demand()方法，以检查权限。这将验证方法的调用者，这里是 DemandFileIOPermission()方法的调用者，是否具有必要的权限。如果 Demand()方法失败，就抛出一个 SecurityException 类型的异常。可以不捕获异常并让调用者处理它(代码文件 DemandPermissionDemo/DemandPermissions.cs)。

```
using System;
using System.Security;
using System.Security.Permissions;

[assembly: AllowPartiallyTrustedCallers()]

namespace Wrox.ProCSharp.Security
{
  [SecuritySafeCritical]
  public class DemandPermissions
  {
    public void DemandFileIOPermissions(string path)
    {
      var fileIOPermission = new
          FileIOPermission(PermissionState.Unrestricted);
      fileIOPermission.Demand();

      //...
    }
  }
}
```

FileIOPermission 类包含在 System.Security.Permissions 名称空间中。这个名称空间包含了权限的全集，除此之外，它还为声明性的权限特性提供了类，为用于创建权限对象的参数提供了枚举(例如，在创建 FileIOPermission 类时，指定是需要完全访问还是只读访问)。

当代码试图违反被赋予的权限时，要捕获 CLR 抛出的异常，可以捕获 SecurityException 类型的异常，通过它可以访问许多有用的信息，其中包括可读的栈踪迹(SecurityException.StackTrace)和对抛出异常的方法的引用(SecurityException.TargetSite)。SecurityException 异常甚至还能提供 SecurityException.PermissionType 属性，这个属性返回导致发生安全异常的 Permission 对象的类型。

如果仅对文件 I/O 使用.NET 类，就不需要请求 FileIOPermission 权限，因为.NET 类在进行文件 I/O 时会要求该权限。但如果封装了本地 API 调用，如 CreateFileTransacted()方法，就需要自己请求该权限。另外，还可以使用这个机制从调用者那里请求自定义权限。

3. 使用沙盒 API 包含未授权的代码

桌面应用程序默认有完全信任权限。使用沙盒 API，可以创建一个不具备完全信任权限的应用程序域。

为了说明沙盒 API 的用法，先创建一个 C#库项目 RequiredFileIOPermissionDemo。这个库包含 RequiredPermissionDemo 类和 RequiredFilePermissions()方法。这个方法根据代码是否有文件权限，返回 true 或 false。在这段实现代码中，File 类创建了一个文件，其中通过参数变量 path 传送路径。如果文件写入操作失败，就抛出一个 SecurityException 类型的异常。File 类检查 FileIOSecurity，与前面的 DemandPermissionDemo 示例相同。如果安全检查失败，FileIOSecurity 类的 Demand()方法就抛出一个 SecurityException 异常。这里捕获 SecurityException 异常，以从 RequiredFilePermissions() 方法中返回 false(代码文件 RequireFileIOPermissionsDemo/RequirePermissionsDemo.cs)。

```
using System;
using System.IO;
using System.Security;

[assembly: AllowPartiallyTrustedCallers()]

namespace Wrox.ProCSharp.Security
{
  [SecuritySafeCritical]
  public class RequirePermissionsDemo : MarshalByRefObject
  {
    public bool RequireFilePermissions(string path)
    {
      bool accessAllowed = true;

      try
      {
        StreamWriter writer = File.CreateText(path);
        writer.WriteLine("written successfully");
        writer.Close();
      }
      catch (SecurityException)
      {
        accessAllowed = false;
      }

      return accessAllowed;
    }
  }
}
```

使用沙盒 API 的宿主应用程序是 AppDomainHost 项目，它是一个简单的 C#控制台应用程序。沙盒 API 是 AppDomain.CreateDomain 方法的一个重载版本，它在沙盒中创建了一个新的应用程序域。这个方法需要 4 个参数，包括应用程序域的名称、从当前应用程序域中提取的凭证、AppDomainSetup 信息和一个权限集。所创建的权限集只包含 SecurityPermission 和 SecurityPermissionFlag.Execution 标记，以便允许执行代码——除以之外没有别的目的。在新的沙盒应用程序域中，实例化 DemandPermission 程序集中 DemandPermissions 类型的对象。

跨应用程序域的调用需要.NET Remoting。因此 RequirePermissionDemo 类需要派生自基类 MarshalByRefObject。对返回的 ObjectHandle 进行解包，会对其他应用程序域中的对象返回一个透明代理，以调用 RequiredFilePermissions 方法(代码文件 AppDomainHost/Program.cs)。

```
using System;
using System.Runtime.Remoting;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      var permSet = new PermissionSet(PermissionState.None);
      permSet.AddPermission(new SecurityPermission(
          SecurityPermissionFlag.Execution));

      AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
      AppDomain newDomain = AppDomain.CreateDomain("Sandboxed domain",
          AppDomain.CurrentDomain.Evidence, setup, permSet);
      ObjectHandle oh = newDomain.CreateInstance(
          "RequireFileIOPermissionsDemo",
          "Wrox.ProCSharp.Security.RequirePermissionsDemo");
      object o = oh.Unwrap();
      var io = o as RequirePermissionsDemo;
      string path = @"c:\temp\file.txt";
      Console.WriteLine("has {0}permissions to write to {1}",
        io.RequireFilePermissions(path) ? null : "no ", path);
    }
  }
}
```

运行这个应用程序，可以看到，结果是所调用的程序集没有创建文件的权限。如果在所创建的应用程序域中给权限集添加 FileIOPermissionSet，如下面的代码所示，写入文件的操作就会成功。

```
PermissionSet permSet = new PermissionSet(PermissionState.None);
permSet.AddPermission(new SecurityPermission(
   SecurityPermissionFlag.Execution));
permSet.AddPermission(new FileIOPermission(
   FileIOPermissionAccess.AllAccess, "c:/temp"));
```

4. 隐式的权限

在授予权限时，通常有一条隐式的语句也可以赋予其他权限。例如，如果赋予了访问 C:\的权限 FileIOPermission，就有一个也可以访问 C:\的子目录的隐式假设。

如果要检查授予的权限是否以子集的方式隐式地赋予了其他的权限，就可以使用下面的代码(代码文件 ImplicitPermissions/Program.cs)：

```
class Program
{
```

```
        static void Main()
        {
            CodeAccessPermission permissionA =
                new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\");
            CodeAccessPermission permissionB =
                new FileIOPermission(FileIOPermissionAccess.Read, @"C:\temp");
            if (permissionB.IsSubsetOf(permissionA))
            {
                Console.WriteLine("PermissionB is a subset of PermissionA");
            }
        }
    }
```

代码的执行结果如下：

```
PermissionB is a subset of PermissionA
```

22.6 使用证书发布代码

可以利用数字证书来对程序集进行签名，让软件的消费者验证软件发布者的身份。根据使用应用程序的地点，可能需要证书。例如，用户利用 ClickOnce 安装应用程序，可以验证证书，以信任发布者。Microsoft 通过 Windows Error Reporting，使用证书来找出哪个供应商映射到错误报告。

ClickOnce 参见第 18 章。

在商业环境中，可以从 Verisign 或 Thawte 之类的公司中获取证书。从软件厂商购买证书(而不是创建自己的证书)的优点是，那些证书可以证明软件的真实性有很高的可信度，软件厂商是可信的第三方。但是，为了测试，.NET 提供了一个命令行实用程序，使用它可以创建测试证书。创建证书和使用证书发布软件的过程相当复杂，但是本节用一个简单的示例说明这个过程。

设想有一个名叫 ABC 的公司。公司的软件产品(Simple.exe)应该值得信赖。首先，输入下面的命令，创建一个测试证书：

```
>makecert -sv abckey.pvk -r -n "CN=ABC Corporation" abccorptest.cer
```

这条命令为ABC 公司创建了一个测试证书，并把它保存到abccorptest.cer文件中。-sv abckey.pvk参数创建一个密钥文件，来存储私钥。在创建密钥文件时，需要输入一个必须记住的密码。

创建证书后，就可以用软件发布者证书测试工具(Cert2spc.exe)创建一个软件发布者测试证书：

```
>cert2spc abccorptest.cer abccorptest.spc
```

有了存储在 spc 文件中的证书和存储在 pvk 文件中的密钥文件，就可以用 pvk2pfx 实用程序创建一个包含证书和密钥文件的 pfx 文件：

```
>pvk2pfx -pvk abckey.pvk -spc abccorptest.spc -pfx abccorptest.pfx
```

现在可以用 signtool.exe 实用程序标记程序集了。使用 sign 选项来标记，用-f 指定 pfx 文件中的证书，用-v 指定输出详细信息：

```
>signtool sign -f abccorptest.pfx -v simple.exe
```

为了建立对证书的信任，可使用证书管理器 certmgr 或 MMC 插件 Certificates，通过 Trusted Root Certification Authorities 和 Trusted Publishers 安装它。之后就可以使用 signtool 验证签名是否成功：

```
>signtool verify -v -a simple.exe
```

22.7 小结

本章讨论了与.NET 应用程序相关的几个安全性方面。用基于角色的安全性进行身份验证和授权，可以确定哪些用户可以访问应用程序中的特性。用户用标识和主体表示，这些类实现了 IIdentity 和 IPrincipal 接口。角色的验证可以在代码中完成，也可以使用特性以简单的方式完成。

本章介绍了加密方法，说明了数据的签名和加密，以安全的方式交换密钥。.NET 提供了对称加密算法和非对称加密算法。

使用访问控制列表还可以读取和修改对操作系统资源(如文件)的访问权限。ACL 的编程方式与安全管道、注册表键、Active Directory 项以及许多其他操作系统资源的编程方式相同。

如果应用程序在不同的区域和不同的语言中使用，就可以参阅下一章介绍的用本地代码交互操作的功能。

第23章

互　操　作

本章要点

- COM 和.NET 技术
- 在.NET 应用程序中使用 COM 对象
- 在 COM 客户端中使用.NET 组件
- 通过平台调用的方式来调用原生方法

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- COMServer
- DotnetServer
- PInvokeSample

23.1　.NET 和 COM 技术

如果有一些 Windows 应用程序是在.NET 出现以前编写的，就可能没有时间和资源来使用.NET 重新编写这类应用程序的所有代码。有时重新编写代码很有用，可以帮助重构或重新思考应用程序的架构，而且由于使用新技术添加新功能更加方便，所以从长远来看，重写代码对于提高生产率也是有帮助的。但是，没有必要纯粹因为有新技术问世就重写原来的代码。现有的正在运行的代码可能有成千上万行，如果只是把它们迁移到托管环境，重写代码的工作量就太大了。

这也适用于 Microsoft。通过提供 System.DirectoryServices 名称空间，Microsoft 没有重写用于访问数据分层存储的 COM 对象，而是让该名称空间中的类作为包装来访问 ADSI COM 对象。System.Data.OleDb 也是同理，该名称空间中的类使用的 OLE DB 提供程序包含十分复杂的 COM 接口。

用户自己的解决方案也可能存在这样的问题。如果需要在.NET 应用程序中使用现有的 COM 对象，或者反过来，如果想要编写.NET 组件，在原来的 COM 客户端使用它们，就要用到 COM 互操作(interop)。本章将介绍这个概念。

如果没有需要集成到.NET 应用程序中的现有 COM 组件，也没有需要使用.NET 组件的旧 COM 客户端，则可以跳过本章。

本章主要使用 System.Runtime.InteropServices 名称空间。

COM 是.NET 的前一代技术，定义了一个允许使用不同编程语言编写组件的组件模型。例如，使用 C++编写的组件可以在 Visual Basic 客户端使用。组件也可以在进程内、进程间、甚至网络上使用。听起来是不是很熟悉？没错，.NET 也有类似的目标。但是，二者实现这些目标的方式是不同的。COM 的概念变得越来越复杂，其不易扩展的缺点逐渐凸显了出来。.NET 在实现与 COM 相似的目标时，引入了新的概念，使开发人员的工作变得更加轻松。

即使在今天，在使用 COM 互操作时，仍然必须先了解 COM。无论是让 COM 客户端使用.NET 组件，还是让.NET 应用程序使用 COM 组件，都必须了解 COM。因此，本节将比较 COM 和.NET 在功能上的异同。

如果读者已经很熟悉 COM 技术，可以利用本节温习有关 COM 的知识。否则，就通过本节学习 COM 的概念。现在由于使用.NET，在日常工作中已经不再需要使用 COM 了。但是，在把 COM 集成到.NET 应用程序时，COM 原有的所有问题仍然存在。

COM 和.NET 有许多相似的概念，但是在使用上差别很大。具体来说，这些概念包括：

- 元数据
- 释放内存
- 接口
- 方法绑定
- 数据类型
- 注册
- 线程
- 错误处理
- 事件处理

下面的几个小节中将讨论这些概念以及封送机制。

23.1.1 元数据

在 COM 中，组件的所有信息都存储在类型库内。例如，类型库包含接口的名称和 ID、方法及其参数等信息。而在.NET 中，如第 15 章和第 19 章所述，这些信息都可以在程序集自身中找到。COM 的问题在于类型库很难扩展。C++使用 IDL(Interface Definition Language)文件描述接口和方法。但是一些 IDL 修饰符在类型库中是找不到的，因为 Visual Basic(类型库由 Visual Basic 团队负责)不能使用这些 IDL 修饰符。在.NET 中则不存在这个问题，因为.NET 元数据可以通过自定义特性扩展。

由于存在上述行为，一些 COM 组件有类型库，一些 COM 组件没有。没有类型库可用时，可以使用一个描述了接口和方法的 C++头文件。在.NET 中使用有类型库的 COM 组件比较容易，但也可以使用没有类型库的 COM 组件，只不过此时必须使用 C#代码重新定义 COM 接口。

23.1.2 释放内存

在.NET 中，通过垃圾回收器来释放内存，这与依赖引用计数的 COM 完全不同。每个 COM 对象都必须实现 IUnkown 接口，它提供了 3 个方法，其中两个与引用计数有关：AddRef 和 Release。客户端在需要一个接口指针时，必须调用 AddRef 方法，它会增加引用计数。调用 Release 方法会减小引用计数，如果得到的引用计数为 0，对象就销毁自身以释放内存。

23.1.3 接口

接口是 COM 的核心，它们把客户端与对象之间的契约和这种契约的具体实现区分开。接口(契约)定义了组件提供的方法和客户端可以使用的方法。在.NET 中，接口也扮演着一个重要的角色。COM 中有 3 种不同的接口类型：自定义(custom)接口、调度(dispatch)接口和双重(dual)接口。

1. 自定义接口

自定义接口派生自 IUnknown 接口。它在一个虚拟表(vtable)中定义了各个方法的顺序，使客户端能够直接访问接口的方法。因为把 vtable 绑定到方法的操作是使用内存地址实现的，所以这也意味着客户端在开发时需要知道 vtable。因此，脚本客户端不能使用自定义接口。图 23-1 显示了自定义接口 IMath 的 vtable。除了 IUnknown 接口的方法外，这个自定义接口还提供了 Add 方法和 Sub 方法。

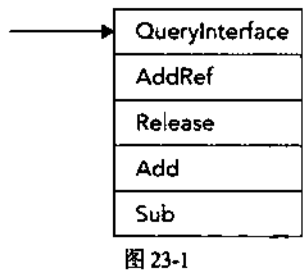


图 23-1

2. 调度接口

因为脚本客户端(以及原来的 Visual Basic 客户端)不支持自定义接口，所以还需要一种不同的接口类型。客户端能够使用的接口总是调度接口 IDispatch，IDispatch 接口派生自 IUnknown 接口，除了 IUnknown 方法以外，还提供了 4 个方法，其中最重要的两个方法是 GetIDsOfNames 和 Invoke。如图 23-2 所示，在调度接口中需要用到两个表。第一个表将方法或属性的名称映射到一个调度 ID，第二个表将调度 ID 映射到方法或属性的实现。

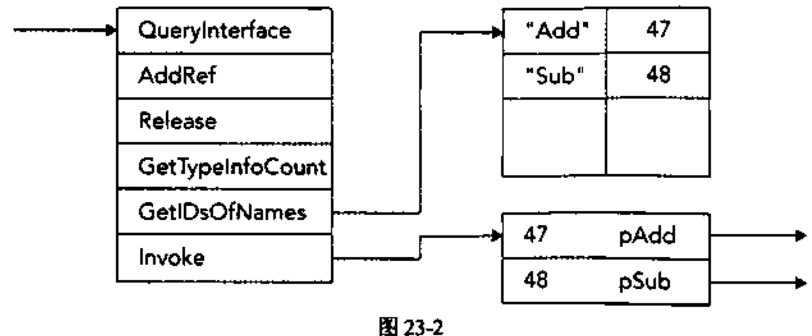


图 23-2

当客户端调用组件中的某个方法时，会首先调用 GetIDsOfNames 方法，并传入想要调用的方法的名称。GetIDsOfNames 方法会在名称-ID 映射表中查找并返回其调度 ID。客户端使用这个 ID 来调用 Invoke 方法。

IDispatch 接口的两个表通常存储在类型库中，但这并不是强制要求，一些组件的表就存储在其他地方。

3. 双重接口

一方面，调度接口比自定义接口慢很多，但另一方面，脚本客户端是不能使用自定义接口的。使用双重接口能够解决这个两难的问题。如图 23-3 所示，双重接口派生自 IDispatch 接口，但是在 vtable 中直接提供了额外的方法。因此，脚本客户端可以使用 IDispatch 接口来调用 Add 和 Sub 方法，而知道 vtable 的客户端可以直接调用 Add 和 Sub 方法。

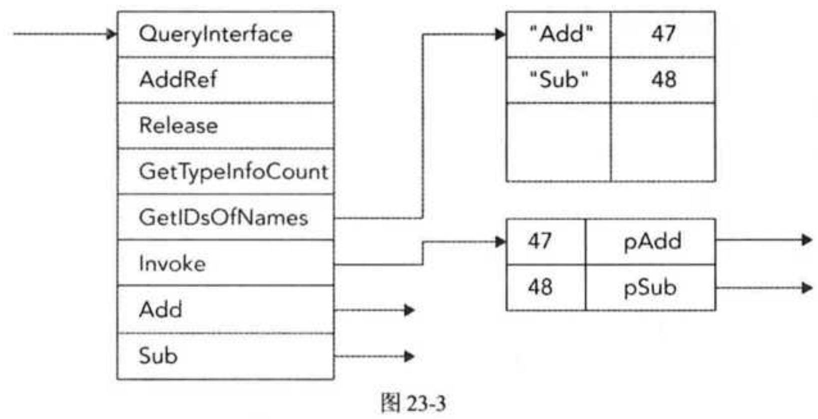


图 23-3

4. 强制转换和 QueryInterface

如果.NET 类实现了多个接口，可以将其强制转换为其中的某个接口。在 COM 中，IUnknown 接口通过 QueryInterface 方法实现了类似的机制。如前一节所述，IUnknown 接口是每个接口的基接口，所以每个接口都可以使用 QueryInterface。

23.1.4 方法绑定

术语早期绑定(early binding)和后期绑定(late binding)定义了客户端映射到方法的方式。后期绑定是指在运行期间才查找要调用的方法。.NET 使用 System.Reflection 名称空间来实现后期绑定(见第 15 章)，COM 则使用前面讨论过的 IDispatch 接口来实现后期绑定。后期绑定只对调度接口和双重接口有效。

在 COM 中，有两个不同的选项可以实现早期绑定。早期绑定的一种方式是直接使用 vtable，所以也叫做 vtable 绑定，这种方式只能用于自定义接口和双重接口。早期绑定的另一种方式叫做 ID 绑定。使用这种方法时，调度 ID 存储在客户端代码内，所以在运行期间只需要调用一次 Invoke。GetIdsOfNames 是在设计期间调用的。对于这类客户端，一定要记住不能改变调度 ID。

23.1.5 数据类型

对于双重接口和调度接口，在 COM 中能够使用的数据类型仅限于一组自动兼容的数据类型。IDispatch 接口的 Invoke 方法接受 VARIANT 类型的一个数组作为参数。VARIANT 是许多不同数据类型的联合，例如 BYTE、SHORT、LONG、FLOAT、DOUBLE、BSTR、IUnknown*、IDispatch*等。在 Visual Basic 中使用 VARIANT 很容易，但是在 C++中使用它们就很复杂了。.NET 用 Object 类代替了 VARIANT 数据类型。

对于自定义接口，在 C++中能够使用的所有数据类型，在 COM 中也都能够使用。但是，这对使用此组件的客户端的编程语言会有所限制。

23.1.6 注册

如第 19 章所述，.NET 区分私有和共享程序集。而在 COM 中，通过注册表配置，所有组件都是全局可用的。

所有的 COM 对象都有一个 128 位的唯一标识符，叫做类 ID(CLSID)。用于创建 COM 对象的 COM API 调用是 CoCreateInstance，它会在注册表中查找 CLSID 和 DLL 或 EXE 的路径，以加载 DLL 或启动 EXE，以及初始化组件。

因为这种包含 128 个位的数字很难记住，所以许多 COM 对象还有一个 ProgID。ProgID 是一个很容易记住的名称，例如 Excel.Application，它映射到一个 CLSID。

COM 对象不只有 CLSID。对于每个接口和类型库，也都有一个唯一标识符，分别叫做 IID 和 typelib ID。本章稍后将更详细地介绍注册表中的信息。

23.1.7 线程

COM 使用单元模型来为程序员处理线程问题。但是，这也增加了一些复杂程度。操作系统的不同版本增加了不同的单元类型。本节将介绍单线程单元和多线程单元。

第 21 章讨论了.NET 中的线程问题。

1. 单线程单元

单线程单元(single-threaded apartment，STA)是在 Windows NT 3.51 中引入的。在 STA 中，只有一个线程(创建实例的线程)能够访问组件。但是，一个进程中可以有多个 STA，如图 23-4 所示。

在图中，与小圆圈相连的矩形代表 COM 组件。COM 组件和单线程(曲线箭头)包含在单线程单元中。最外层的矩形代表一个进程。

在 STA 中，不需要考虑如何防止实例变量被多线程访问，因为 COM 设施提供了这种保护，保证只能有一个线程访问组件。

在编写时明确不考虑线程安全性的 COM 对象会在注册表中将注册表项 ThreadingModel 设为 Apartment，表示需要使用 STA。

2. 多线程单元

Windows NT 4.0 引入了多线程单元(multithreaded apartment，MTA)的概念。在 MTA 中，多个线程可以同时访问组件。图 23-5 显示的进程中有一个 MTA 和两个 STA。

在编写时明确考虑到线程安全性的 COM 对象会在注册表中将注册表项 ThreadingModel 设为 Free，表示需要使用 MTA。对于不明确是否考虑单元类型的线程安全的 COM 对象，将注册表项 ThreadingModel 设为 Both。

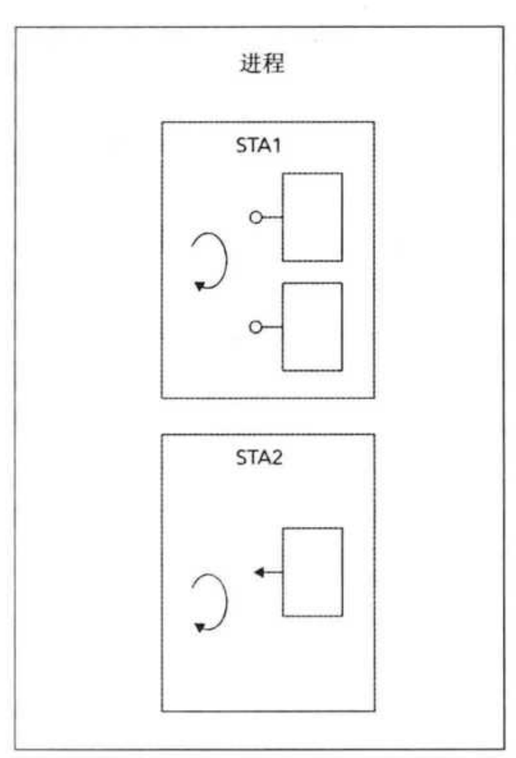


图 23-4

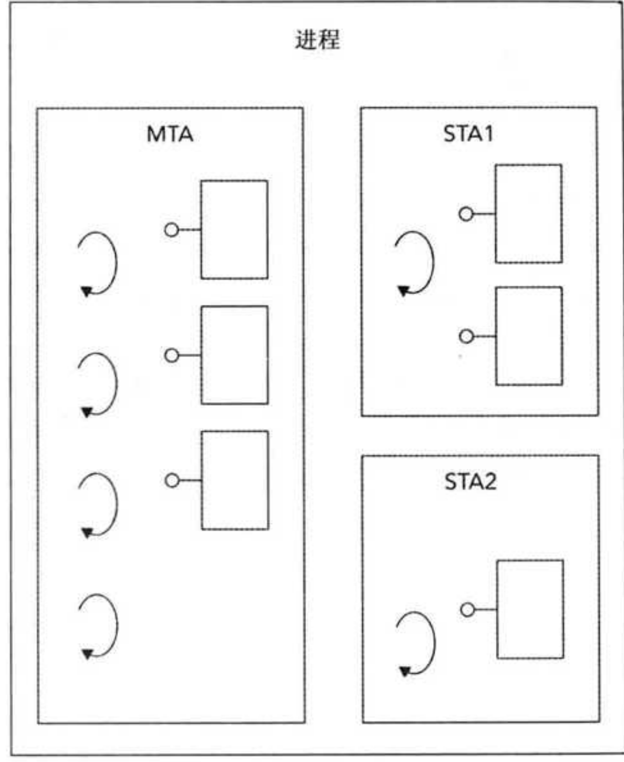


图 23-5

Visual Basic 6.0 不支持多线程单元。如果要使用 VB6 开发的 COM 对象，一定要记住这一点。

Windows 2000 引入了另一个单元模型：线程中性单元(Thread Neutral Apartment，TNA)。这个单元模型只能用于配置为 COM+应用程序的 COM 组件。将 ThreadingModel 设为 Both 可接受 3 种单元：STA、MTA 和 TNA。

23.1.8 错误处理

在.NET 中，在发生错误时会抛出异常。而在 COM 技术中，则通过在方法中返回 HRESULT 值

来定义错误。HRESULT 的值为 S_OK 表示方法成功。

如果 COM 组件提供了更详细的错误信息，则说明该 COM 组件实现了 ISupportErrorInfo 接口，在方法返回的错误信息对象中，不只包含一条错误消息，还包含指向一个帮助文件的链接以及错误源。实现 ISupportErrorInfo 的对象会自动映射到.NET 中包含详细错误信息的异常对象。

第 20 章介绍了如何跟踪和记录错误。

23.1.9 事件

.NET 通过 C#关键字 event 和 delegate 实现了回调机制(见第 8 章)。图 23-6 显示的是 COM 的事件处理结构。对于 COM 事件，组件必须实现 IConnectionPointContainer 接口和一个或多个实现了 IConnectionPoint 接口的连接点对象(connection point object，CPO)。组件还定义了一个由 CPO 调用的传出接口 ICompletedEvents，如图 23-6 所示。客户端必须在 sink 对象(其本身就是一个 COM 对象)中实现这个传出对象。在运行期间，客户端会向 COM 服务查询接口 IConnectionPointContainer，然后客户端调用该接口的 FindConnectionPoint 方法来获得 CPO。FindConnectionPoint 方法会返回一个 IConnectionPoint 的指针。客户端使用此接口指针调用 Advise 方法，向服务器传递 sink 对象的指针。实际上，也就是把组件传递给了服务器。

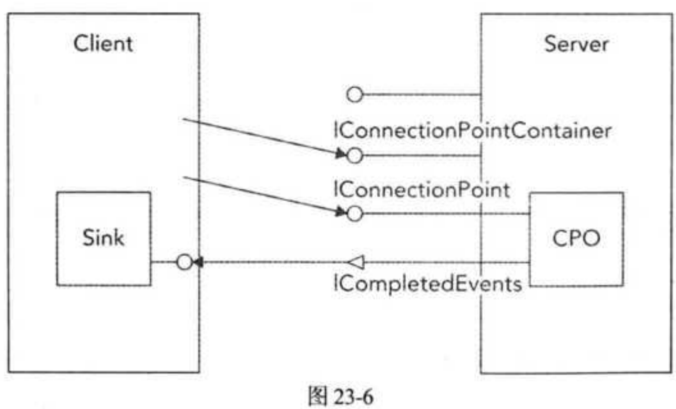


图 23-6

在本章后面将学习如何映射.NET 事件和 COM 事件，以便在.NET 客户端处理 COM 事件，在 COM 客户端处理.NET 事件。

23.1.10 封送

从.NET 传递给 COM 组件的数据和从 COM 组件传递给.NET 的数据必须转换为对应的表示形式，这就是封送机制。具体的操作取决于所传递数据的类型能否按位块传输。

能按位块传输的数据类型在.NET 和 COM 中有相同的表示，所以不需要进行转换。简单的数据类型，如 byte、short、int、long，以及只包含这些简单数据类型的类和数组都是能按位块传输的数据类型。只有一维数组才能按位块传输。

不能按位块传输的数据类型，就需要进行转换了。表 23-1 列出了一些不能按位块传输的 COM

数据类型及其对应的.NET 数据类型。由于需要进行转换，所以不能按位块传输的数据类型的性能开销更高。

表 23-1

COM 数据类型	.NET 数据类型
SAFEARRAY	Array
VARIANT	Object
BSTR	String
IUnknown* IDispatch*	Object

23.2 在.NET 客户端中使用 COM 组件

在.NET 应用程序中使用 COM 组件之前，首先需要创建一个 COM 组件。使用 C#或 Visual Basic 2012 无法创建 COM 组件，而需要使用 Visual Basic 6.0 或 C++(或其他任何支持 COM 的语言)。本章将在 Visual Studio 2013 中使用活动模板库(Active Template Library，ATL)和 C++创建 COM。

我们首先创建一个简单的 COM 组件，并在一个运行库可调用包装(runtime callable wrapper，RCW)中使用该组件。我们还会通过新的 C# 4 动态语言扩展使用该组件，并讨论线程问题，最后将 COM 连接点映射到.NET 事件。

使用 Visual Basic 11 和 C#构建 COM 组件时需要注意：把一个真实的 COM 组件作为包装，在 Visual Basic 11 和 C# 5 中可以构建一个类似于 COM 组件的.NET 组件。让.NET 客户端通过 COM 互操作来调用这个类似于 COM 的.NET 组件是很不合理的。

因为本书的主题不是 COM，所以讨论不会涉及 COM 的方方面面，而只会讨论建立一个示例所需的知识。

23.2.1 创建 COM 组件

为了使用 ATL 和 C++创建一个 COM 组件，首先要新建一个 ATL 项目。在 Visual C++ Projects 组中，选择 File | New Project，可看到 ATL 项目向导。将项目名称设为 COMServer。在 Application Settings 中，选择 Dynamic Link Library 并单击 Finish 按钮。

因为生成步骤会在注册表中注册 COM 组件，而这需要管理员权限，所以在编写 ATL COM 对象时，必须以提升模式运行 Visual Studio。

ATL 项目向导为 COM 服务搭建了基础。但是，还需要创建一个 COM 对象。在 Solution Explorer 中添加一个类，并选择 ATL Simple Object。在对话框的 Short name 文本框中，输入 COMDemo。其他文本框将自动填入，但是我们需要把接口名改为 IWelcome，将 ProgID 改为 COMServer.COMDemo，如图 23-7 所示。单击 Finish，创建类和接口的存根(stub)代码。

图 23-7

该 COM 组件提供了两个接口和 3 个简单的方法。这两个接口有助于理解如何从.NET 映射 QueryInterface，而这 3 个方法有助于理解互操作是如何发生的。在类视图下，选择接口 IWelcome，并使用下面的参数添加方法 Greeting，如图 23-8 所示：

图 23-8

```
HRESULT Greeting([in] BSTR name, [out, retval] BSTR* message);
```

IDL 文件 COMServer.idl 定义了 COM 的接口。向导在 COMServer.idl 文件中生成的代码如下所示，唯一标识符(uuid)会不同。IWelcome 接口定义了 Greeting 方法。关键字 interface 之前的方括号定义了该接口的一些特性。uuid 定义了接口 ID，dual 标记了接口类型(代码文件 COMServer/COMServer.idl)：

```
[
  object,
  uuid(AF05C6E6-BF95-411F-B2FA-531D911C5C5C),
  dual,
  nonextensible,
  pointer_default(unique)
]
interface IWelcome : IDispatch{
   [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
```

IDL 文件也定义了类型库的内容，它是实现了 IWelcome 接口的 COM 对象(coclass)：

```
[
  uuid(8FCA0342-FAF3-4481-9D11-3BC613A7F5C6),
  version(1.0),
]
library COMServerLib
{
  importlib("stdole2.tlb");
  [
    uuid(9015EDE5-D106-4005-9998-DE44849EFA3D)
  ]
  coclass COMDemo
  {
    [default] interface IWelcome;
  };
};
```

通过使用 custom 特性，可以修改由.NET 包装类生成的类和接口的名称。只需要添加 custome 特性和标识符 0F21F359-AB84–41e8–9A78–36D110E6D2F9，以及在.NET 中出现的名称即可。

在 IWelcome 接口的头部，使用上述标识符和名称 Wrox.ProCSharp.Interop.Server.IWelcome 添加 custome 特性。在 coclass COMDemo 中使用对应的名称添加类似的 custom 特性：

```
[
  object,
  uuid(EB1E5898-4DAB-4184-92E2-BBD8F9341AFD),
  dual,
  nonextensible,
  pointer_default(unique),
  custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
```

```
  "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome : IDispatch{
   [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
[
  uuid(8C123EAE-F567-421F-ACBE-E11F89909160),
  version(1.0),
]
library COMServerLib
{
  importlib("stdole2.tlb");
  [
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.COMDemo")
  ]
  coclass COMDemo
  {
     [default] interface IWelcome;
  };
};
```

现在在COMServer.idl文件中再添加一个接口。可以把IWelcome接口的头部复制到新建的IMath接口的头部，但是要记着修改uuid关键字定义的唯一标识符。使用guidgen实用工具可以创建这样的ID。IMath接口提供了方法Add和Sub：

```
// IMath
[
    object,
    uuid(2158751B-896E-461d-9012-EF1680BE0628),
    dual,
    nonextensible,
    pointer_default(unique),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.IMath")
]
interface IMath: IDispatch
{
    [id(1)] HRESULT Add([in] LONG val1, [in] LONG val2,
                        [out, retval] LONG* result);
    [id(2)] HRESULT Sub([in] LONG val1, [in] LONG val2,
                        [out, retval] LONG* result);
};
```

还必须修改coclass COMDemo，使其同时实现接口IWelcome和IMath。IWelcome接口设为默认接口：

```
importlib("stdole2.tlb");
[
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
    helpstring("COMDemo Class"),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.COMDemo")
```

```
]
coclass COMDemo
{
    [default] interface IWelcome;
    interface IMath;
};
```

现在可以将注意力从 IDL 文件转移到 C++代码。COMDemo.h 文件包含 COM 对象的类定义。类 CCOMDemo 使用多继承从模板类 CComObjectRootEx、CComCoClass 和 IdispatchImpl 中派生。CComObjectRootEx 类实现了 IUnknown 接口的功能，例如 AddRef 和 Release 方法。CComCoClass 类创建了一个工厂来实例化模板参数对象，在这里是 CComDemo。IDispatchImple 实现了 IDispatch 接口的方法。

包含在 BEGIN_COM_MAP 和 END_COM_MAP 中的宏创建了一个映射，用于定义 COM 类实现的所有 COM 接口。QueryInterface 方法的实现会使用此映射(代码文件 COMServer/COMDemo.h)：

```
class ATL_NO_VTABLE CCOMDemo:
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
   public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
     /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
  CCOMDemo()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

BEGIN_COM_MAP(CCOMDemo)
  COM_INTERFACE_ENTRY(IWelcome)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

  DECLARE_PROTECT_FINAL_CONSTRUCT()

  HRESULT FinalConstruct()
  {
    return S_OK;
  }

  void FinalRelease()
  {
  }

public:
  STDMETHOD(Greeting)(BSTR name, BSTR* message);
};

OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)
```

在这个类定义中，还需要添加第二个接口 IMath，以及 IMath 接口定义的方法：

```
class ATL_NO_VTABLE CCOMDemo:
```

```
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
    public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>
    public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>
{
public:
    CCOMDemo()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)
BEGIN_COM_MAP(CCOMDemo)
    COM_INTERFACE_ENTRY(IWelcome)
    COM_INTERFACE_ENTRY(IMath)
    COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
    STDMETHOD(Greeting)(BSTR name, BSTR* message);
    STDMETHOD(Add)(long val1, long val2, long* result);
    STDMETHOD(Sub)(long val1, long val2, long* result);
};

OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)
```

现在可以使用下面的代码来实现 COMDemo.cpp 文件中的 3 个方法了。CComBSTR 是一个 ATL 类，用于简化 BSTR 的处理。Greeting 方法只返回一条欢迎消息，它将第一个参数中传递的 name 添加到返回的消息中。Add 方法将两个数值相加，Sub 方法则执行减法，并返回结果(代码文件 COMServer/COMDemo.cpp)：

```
STDMETHODIMP CCOMDemo::Greeting(BSTR name, BSTR* message)
{
    CComBSTR tmp("Welcome, ");
    tmp.Append(name);
    *message = tmp;
    return S_OK;
}

STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
    *result = val1 + val2;
    return S_OK;
```

```
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
  *result = val1 - val2;
  return S_OK;
}
```

现在就可以生成组件了。生成过程会在注册表中配置组件。

23.2.2 创建运行库可调用包装

为了在.NET 中使用 COM 组件，必须创建一个运行库可调用包装(runtime callable wrapper，RCW)。使用 RCW 后，.NET 客户端看到的是.NET 对象而不是 COM 组件，所以不需要处理 COM 特性，包装会完成这些工作。RCW 会隐藏 IUnknown 和 IDispatch 接口(如图 23-9 所示)，并处理 COM 对象的引用计数。

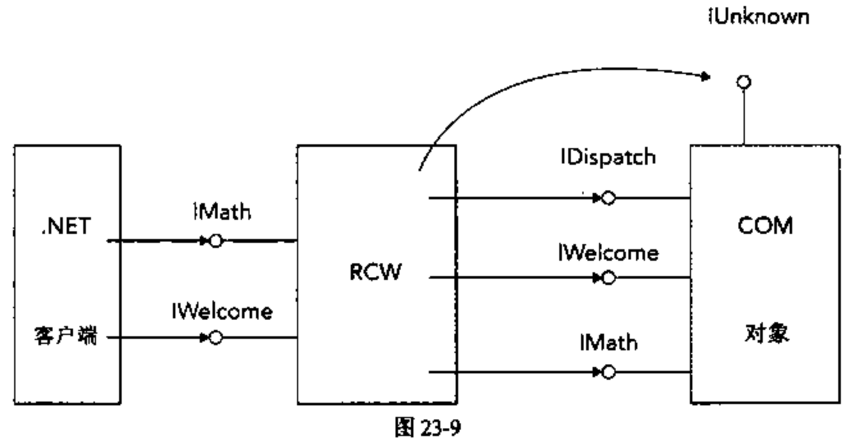


图 23-9

使用命令行工具 tlbimp 或使用 Visual Studio 都可以创建 RCW。使用如下命令：

```
tlbimp COMServer.dll /out:Interop.COMServer.dll
```

创建文件 Interop.COMServer.dll，它包含一个带包装类的.NET 程序集。在生成的这个程序集中，可以找到名称空间 COMWrapper，其中包含 CCOMDemoClass 类和 CCOMDemo、IMath 和 IWelcome 接口。使用 tlbimp 工具的选项可以修改该名称空间的名称。使用选项/namespace 可以指定一个不同的名称空间，使用选项/asmversion 可以定义程序集的版本号。

该命令行工具的另一个重要选项是/keyfile，用于为生成的程序集分配一个强名称。第 19 章讨论了强名称。

也可以使用 Visual Studio 创建 RCW。为了创建一个简单的示例应用程序，首先创建一个 C#控制台项目。在 Solution Explorer 的 Add Reference 对话框中选择 COM 选项卡，滚动到 COMServerLib 条目，添加一个对 COM 服务的引用。这里列出了在注册表中配置的所有 COM 对象。从中选择一个 COM 组件会创建一个 RCW 类。在 Visual Studio 2012 中，将 Embed Interop Types 设为 True(默认

值)，可以在项目的主程序集中创建这个包装类。将该选项设为 false 将创建一个单独的互操作程序集，它需要与应用程序一起部署。

23.2.3 使用 RCW

在创建了包装类以后，可以编写代码来实例化和访问组件。因为在 C++文件中使用了 custom 特性，它为 RCW 类生成的名称空间是 Wrox.ProCSharp.COMInterop.Server。在声明中添加这个名称空间，以及 System.Runtime.InteropServices 名称空间。在名称空间 System.Runtime.InteropServices 中，需要使用 Marshal 类来释放 COM 对象(代码文件 DotnetClient/Program.cs)：

```
using System;
using System.Runtime.InteropServices;
using Wrox.ProCSharp.Interop.Server

namespace Wrox.ProCSharp.Interop.Client
{
  class Program
  {
    [STAThread]
    static void Main()
    {
```

现在可以像使用.NET 类一样使用 COM 组件。obj 是 COMDemo 类型的一个变量。COMDemo 是一个.NET 接口，提供了 IWelcome 和 IMath 接口的方法。但是，也可以将它强制转换为其中一个特定的接口，例如 IWelcome。使用一个声明为 IWelcome 类型的变量，就可以调用 Greeting 方法：

```
var obj = new COMDemo();
IWelcome welcome = obj;
Console.WriteLine(welcome.Greeting("Stephanie"));
```

虽然 COMDemo 是一个接口，但是可以实例化 COMDemo 类型的新对象。与普通的接口不同，可以用包装的 COM 接口来实例化这个对象。

如果对象提供了多个接口，如本例中这样，那么也可以声明另一个接口的变量。使用强制转换运算符进行简单的赋值，包装类会对 COM 对象执行 QueryInterface，以返回第二个接口指针。使用 IMath 变量，可以调用 IMath 接口的方法：

```
IMath math;
math = (IMath)welcome;
int x = math.Add(4, 5);
Console.WriteLine(x);
```

如果想在垃圾回收器清理对象之前释放 COM 对象，用静态方法 Marshal.ReleaseComObject 来调用组件的 Release 方法，让组件销毁自身并释放内存：

```
      Marshal.ReleaseComObject(math);
    }
  }
}
```

前面提到，一旦 COM 对象的引用计数为 0，该对象就会释放。Marshal.ReleaseComObject 通过调用 Release 方法将引用计数减 1。因为无论对 RCW 对象引用了多少次，RCW 对象也只调用一次 AddRef 方法来增加引用计数，所以调用 Marshal.ReleaseComObject 一次就足以释放 COM 对象了。

使用 Marshal.ReleaseComObject 方法释放 COM 对象后，就不能再使用任何引用该对象的变量了。在示例中，通过使用变量 math 来释放 COM 对象。之后，就不能再使用引用相同对象的 welcome 变量，否则将得到 InvalidComObjectException 类型的异常。

在不需要 COM 对象后释放它们是极为重要的。COM 对象会使用本地内存堆，而.NET 对象会使用托管内存堆。垃圾回收器只负责托管内存。

可以看到，使用运行库可调用包装后，COM 组件可以像.NET 对象一样使用。

23.2.4 通过动态语言扩展使用 COM 服务

从版本 4 以后，C#新增了一个动态语言扩展。对于使用提供了 IDispatch 接口的 COM 服务，这也是一个好消息。从前面的“调度接口”一节可知，这种接口在运行期间定义了 GetIdsOfNames 和 Invoke 方法。借助于 dynamic 关键字和在后台使用的 COM 绑定器，不必创建 RCW 对象就可以调用 COM 组件。

声明 dynamic 类型的变量并把一个 COM 对象赋值给它会用到 COM 绑定器，可以像下面这样调用默认接口的方法。通过使用 Type.GetTypeFromProgID 来获得 Type 对象，并使用 Activator.CreateInstance 方法来实例化 COM 对象，不需要 RCW 就能够创建 COM 对象的一个实例。使用 dynamic 关键字时无法利用智能感知功能，但是可以使用在 COM 中经常使用的可选参数(代码文件 DynamicDotnetClient/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Interop
{
  class Program
  {
    static void Main()
    {
      Type t = Type.GetTypeFromProgID("COMServer.COMDemo");
      dynamic o = Activator.CreateInstance(t);
      Console.WriteLine(o.Greeting("Angela"));
    }
  }
}
```

第 12 章讨论了 C#的动态语言扩展。

23.2.5 线程问题

如本章前面所述，COM 组件会根据自己是否被实现为线程安全的组件，标记自己应该放在哪个单元中(STA 或 MTA)。但是线程必须连接一个单元。[STAThread]和[MTAThread]特性定义了线程应该连接什么单元，它们可以应用到应用程序的Main方法。[STAThread]特性表示线程连接一个STA，[MTAThread]表示线程连接一个 MTA。如果没有应用特性，默认连接一个 MTA。

使用 Thread 类的 ApartmentState 属性，也可以在程序中设置单元的状态。可以把 ApartmentState 属性设为 ApartmentState 枚举中的一个值，如 STA 或 MTA(如果未设置，则为 Unknown)。需要注意，一个线程的单元状态只能设置一次。如果设置了两次，则第二次设置会被忽略。

如果线程选择了组件不支持的单元，会发生什么？COM 运行库会自动为 COM 组件创建正确的单元，但是如果在调用组件的方法时越过了单元的边界，那么性能会有所下降。

23.2.6 添加连接点

为了了解在.NET 应用程序中如何处理 COM 事件，必须扩展 COM 组件。首先，必须在接口定义文件 COMDemo 中添加另一个接口：_ICompletedEvents。该接口由客户端实现，即.NET 应用程序实现，并由组件调用。在本例中，当完成计算后，组件会调用 Completed 方法。这样的接口也称为传出接口。传出接口必须是一个调度接口或者自定义接口。所有的客户端都支持调度接口。ID 为 0F21F359-AB84-41e8-9A78-36D110E6D2F9 的 custom 特性定义了将在 RCW 中创建的接口的名称。另外，还必须在 coclass 部分将传出接口写入组件支持的接口列表，并将其标记为 source 接口(代码文件 COMServer/COMServer.idl)：

```
library COMServerLib
{
  importlib("stdole2.tlb");
  [
    uuid(5CFF102B-0961-4EC6-8BB4-759A3AB6EF48),
    helpstring("_ICompletedEvents Interface"),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.IcompletedEvents"),
  ]
  dispinterface _ICompletedEvents
  {
    properties:
    methods:
       [id(1)] void Completed(void);
  };
  [
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
    helpstring("COMDemo Class")
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
       "Wrox.ProCSharp.COMInterop.Server.COMDemo")
  ]
  coclass COMDemo
```

```
{
  [default] interface IWelcome;
  interface IMath;
  [default, source] dispinterface _ICompletedEvents;
};
```

可以使用向导创建一个传出接口，它将事件发送给客户端。打开类视图，选择 CComDemo 类，然后打开上下文菜单，选择 Add | Add Connection Point...来启动 Implement Connection Point Wizard(如图 23-10 所示)。选择源接口 ICompletedEvents 来实现连接点。

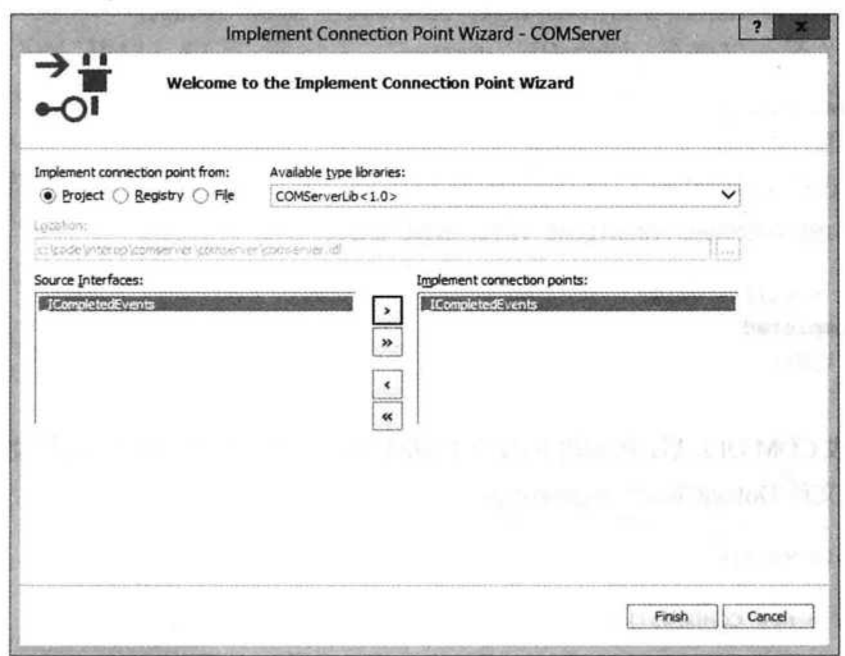


图 23-10

向导会创建代理类 CProxy_ICompletedEvents 来将事件发送给客户端，并改变 CCOMDemo 类。现在，CCOMDemo 类继承自 IConnectionPointContainerImpl 和代理类。接口映射中添加了接口 IConnectionPointContainer，源接口_ICompletedEvents 中添加了一个连接点映射(代码文件 COMServer/COMDemo.h)：

```
class ATL_NO_VTABLE CCOMDemo:
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
  public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
    /*wMajor =*/ 1, /*wMinor =*/ 0>,
  public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>,
  public IConnectionPointContainerImpl<CCOMDemo>,
  public CProxy_ICompletedEvents<CCOMDemo>
{
public:
//...
BEGIN_COM_MAP(CCOMDemo)
  COM_INTERFACE_ENTRY(IWelcome)
  COM_INTERFACE_ENTRY(IMath)
  COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
  COM_INTERFACE_ENTRY(IConnectionPointContainer)
```

```
END_COM_MAP()
//...
public:
  BEGIN_CONNECTION_POINT_MAP(CCOMDemo)
    CONNECTION_POINT_ENTRY(__uuidof(_ICompletedEvents))
  END_CONNECTION_POINT_MAP()
};
```

最后，在 COMDemo.cpp 文件的 Add 和 Sub 方法内，可以调用代理类的 Fire_Completed 方法:

```
STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
  *result = val1 + val2;
  Fire_Completed();
  return S_OK;
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
  *result = val1 - val2;
  Fire_Completed();
  return S_OK;
}
```

在重新生成 COM DLL 后，可以将.NET 客户端改为使用这些 COM 事件，就像使用普通的.NET 事件一样(代码文件 DotnetClient/Program.cs):

```
static void Main()
{
  var obj = new COMDemo();

  IWelcome welcome = obj;
  Console.WriteLine(welcome.Greeting("Stephanie"));

  obj.Completed += () =>Console.WriteLine("Calculation completed");

  IMath math = (IMath)welcome;
  int result = math.Add(3, 5);
  Console.WriteLine(result);

  Marshal.ReleaseComObject(math);
}
```

可以看到，RCW 能自动将 COM 事件映射为.NET 事件。在.NET 客户端，COM 事件可以像.NET 事件一样使用。

23.3 在 COM 客户端中使用.NET 组件

前面介绍了如何从.NET 客户端访问一个 COM 组件。反过来，如何找到一个解决方案，从使用 Visual Basic 6.0、C++和 MFC(Microsoft Foundation Class)或者活动模板库(ATL)的原 COM 客户端访问.NET 组件？这是一个同样有趣的问题。

本节将用.NET代码定义一个.NET组件，然后在一个COM客户端中通过COM可调用包装(COM callable wrapper，CCW)来使用该组件。这样，就可以看到如何通过.NET 程序集创建类型库，使用不同的.NET 特性来指定 COM 互操作行为，以及将.NET 程序集注册为一个 COM 组件。然后，使用 C++创建一个 COM 客户端来使用 CCW。最后，扩展.NET 组件，使其提供 COM 连接点。

23.3.1 COM 可调用包装

如果想在.NET 客户端访问 COM 组件，必须使用 RCW。而为了在 COM 客户端应用程序中访问.NET 组件，就必须使用 CCW。图 23-11 显示的 CCW 包装了一个.NET 类，并提供了供 COM 客户端使用的 COM 接口。CCW 提供了 IUnknown、IDispatch 等接口，还为事件提供了 IConnectionPointContainer 和 IConnectionPoint 等接口。当然，CCW 还提供了.NET 类定义的自定义接口，如 IWelcome 和 IMath。COM 客户端看到的是一个 COM 对象，但是在后台起作用的实际上是一个.NET 组件。包装处理 IUnknown 接口的 AddRef、Release 和 QueryInterface 等方法，而在.NET 对象中，不必处理引用计数，垃圾回收器会完成清理工作。

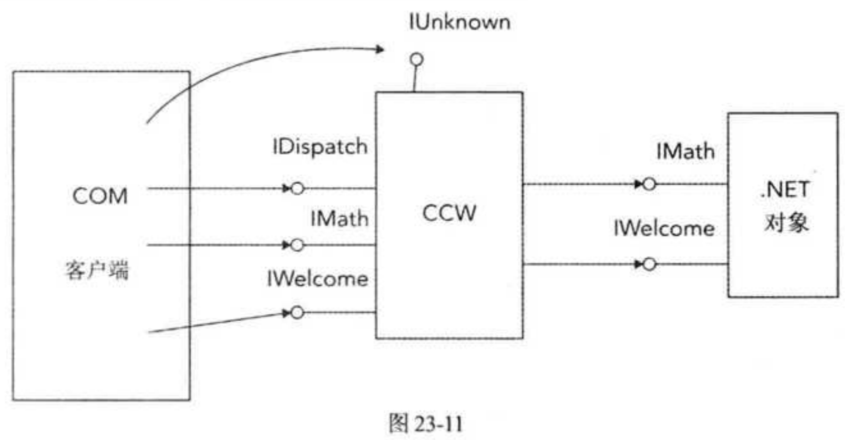


图 23-11

23.3.2 创建.NET 组件

下面的例子将在一个.NET 类中实现与之前的 COM 组件相同的功能。首先创建一个 C#类库，命名为 DotNetServer。然后添加接口 IWelcome 和 IMath，以及实现这些接口的 DotNetComponent 类。ComVisible(true)特性使得类和接口对 COM 可用(代码文件 DotnetServer/DotnetServer.cs)：

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

namespace Wrox.ProCSharp.Interop.Server
{
  [ComVisible(true)]
  public interface IWelcome
  {
    string Greeting(string name);
  }

  [ComVisible(true)]
  public interface IMath
  {
    int Add(int val1, int val2);
```

```
    int Sub(int val1, int val2);
  }

  [ComVisible(true)]
  public class DotnetComponent: IWelcome, IMath
  {
    public DotnetComponent()
    {
    }

    public string Greeting(string name)
    {
      return "Hello " + name;
    }

    public int Add(int val1, int val2)
    {
      return val1 + val2;
    }

    public int Sub(int val1, int val2)
    {
      return val1 - val2;
    }
  }
}
```

生成项目后，可以创建一个类型库。

23.3.3 创建类型库

使用命令行工具 tlbexp 可以创建一个类型库。下面的命令：

```
tlbexp DotnetServer.dll
```

创建了类型库 DotnetServer.tlb。使用 OLE/COM 对象查看器工具 oleview.exe 可以查看类型库。这个工具是 Microsoft SDK 的一部分，可以从 Visual Studio 2012 命令提示符启动。选择 File | View TypeLib 打开类型库。现在看看其接口定义，它与前面用 COM 服务创建的接口很类似。

类型库的名称使用程序集的名称创建。类型库的头部也在一个 custom 特性中定义了程序集的完整名称，并且所有的接口在定义之前进行了声明：

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: <could not determine filename>

[
  uuid(EEA130ED-40E1-4BF8-B06E-6CCA0FD21788),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "DotnetServer, Version=1.0.0.0,
         Culture=neutral, PublicKeyToken=null")
]
library DotnetServer
{
    // TLib : // TLib : mscorlib.dll :
```

```
    // {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-c260-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface IWelcome;
    interface IMath;
    interface _DotnetComponent;
```

在如下的生成代码中，IWelcome 和 IMath 接口定义为 COM 双重接口。在 C#代码中声明的所有方法都会列在这里的类型库定义中。参数发生了变化；.NET 类型映射到 COM 类型(例如，从 String 类映射到 BSTR 类型)；签名也发生了变化，所以会返回一个 HRESULT。因为接口是双重接口，所以还会生成调度 ID：

```
[
  odl,
  uuid(6AE7CB9C-7471-3B6A-9E13-51C2294266F0),
  version(1.0),
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome : IDispatch {
  [id(0x60020000)]
  HRESULT Greeting(
    [in] BSTR name,
    [out, retval] BSTR* pRetVal);
};

[
  odl,
  uuid(AED00E6F-3A60-3EB8-B974-1556096350CB),
  version(1.0),
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
   "Wrox.ProCSharp.Interop.Server.IMath")

]
interface IMath : IDispatch {
  [id(0x60020000)]
  HRESULT Add(
    [in] long val1,
    [in] long val2,
    [out, retval] long* pRetVal);
  [id(0x60020001)]
  HRESULT Sub(
    [in] long val1,
    [in] long val2,
    [out, retval] long* pRetVal);
};
```

coclass部分标记了COM对象本身。头部的uuid是用于实例化对象的CLSID。类DotntComponent支持_DotnetComponent、_Object、IWelcome和IMath接口。_Object在之前的代码部分中包含的mscorlib.tlb文件中定义，并提供了基类Object的方法。组件的默认接口是_DotnetComponent，它在coclass部分之后定义为一个调度接口。在接口声明部分，它被标记为双重接口，但是因为没有包含任何方法，所以它是一个调度接口。使用这个接口时，可以通过后期绑定来访问组件的所有方法：

```
[
  uuid(2F1E78D4-1147-33AC-9233-C0F51121DAAA),
  version(1.0),
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.DotnetComponent")
]
coclass DotnetComponent {
  [default] interface _DotnetComponent;
  interface _Object;
  interface IWelcome;
  interface IMath;
};

[
  odl,
  uuid(2B36C1BF-61F7-3E84-87B2-EAB52144046D),
  hidden,
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.DotnetComponent")
]
interface _DotnetComponent : IDispatch {
};
};
```

生成类型库时，有许多默认值。但是，改变一些默认的.NET到COM的映射常常很有帮助。这可以使用System.Runtime.InteropServices名称空间中的一些特性完成。

23.3.4 COM互操作特性

通过对类、接口或方法应用System.Runtime.InteropServices名称空间中的特性，可以改变CCW的实现。表23-2列出了这些特性：

表23-2

特　性	描　述
Guid	这个特性可应用于程序集、接口和类。使用Guid作为程序集特性定义了类型库ID，将它应用于接口，会定义接口ID(IID)，将它应用于类，则定义类ID(CLSID)。使用guidgen工具，可以创建必须为这个特性定义的唯一ID。CLSID和类型库ID在每个版本中会自动改变。如果不喜欢这种行为，可以使用此特性手动改变。只有当接口的签名改变时，例如添加或删除方法，或者某些参数发生变化，IID才会改变。因为在COM中，接口的每个新版本都应该改变IID，所以这是一种很合理的默认行为，通常不需要使用Guid特性应用IID。唯一需要为一个接口应用固定IID的情况，是.NET接口与现有的COM接口完全相同，所以COM客户端期望使用此固定IID

(续表)

特 性	描 述
ProgId	可以把这个特性应用到一个类，以指定在注册表中配置该对象时使用的名称
ComVisible	在项目的 Assembly Information 属性中设置，可以配置程序集的所有类型都对 COM 是否可见。默认设置是 false。这个默认值很有用，因为这样一来，就必须显式地使用此特性将类、接口和委托标记为可见，以创建对 COM 可见的程序集。如果修改此默认值，使所有类型都对 COM 可见，就对不应创建 COM 表示形式的类型，可以将 ComVisible 特性设为 false
InterfaceType	如果把这个特性设为一个 ComInterfaceType 枚举值，就可以修改为.NET 接口创建的默认双重接口类型。ComInterfaceType 的值包括 InterfaceIsDual、InterfaceIsIDispatch 和 InterfaceIsIUnknown。要向一个.NET 接口应用一个自定义接口类型，只需要像下面这样设置此特性：InterfaceType(ComInterfaceType.InterfaceIsIUnknown)
ClassInterface	使用此特性可以修改为类创建的默认调度接口。可以把 ClassInterface 设置为 ClassInterfaceType 的一个枚举值。取值范围包括 AudoDispatch、AutoDual 和 None。本节前面的示例中，调度接口的默认值是 AutoDispatch，所以创建了一个调度接口。如果类只被已定义的接口访问，则像下面这样对该类应用此特性：ClassInterface(ClassInterfaceType.None)
DispId	这个特性可以应用到双重接口和调度接口，以定义方法和属性的 DispId
In Out	在 COM 中可以指定参数类型的方向。如果参数应该发送给组件，则使用 In 特性。如果要从参数返回一个值，则指定 Out 特性。如果想同时使用两个方向，则同时使用特性 In 和 Out
Optional	COM 方法的参数可以是可选的。使用 Optional 特性可以标记可选参数

现在可以修改 C#代码，为 IWelcome 接口指定一个双重接口类型，为 IMath 接口指定一个自定义接口类型。对类 DotnetComponent 应用 ClassInterface 特性，并将参数设为 ClassInterfaceType.None，这指定了不生成单独的 COM 接口。特性 ProgId 和 Guid 分别指定了一个 ProgID 和一个 GUID(代码文件 DotnetServer/DotnetServer.cs)：

```
[InterfaceType(ComInterfaceType.InterfaceIsDual)]
[ComVisible(true)]
public interface IWelcome
{
  [DispId(60040)]
  string Greeting(string name);
}

[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
[ComVisible(true)]
public interface IMath
{
  int Add(int val1, int val2);
  int Sub(int val1, int val2);
}

[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-4876-8297-35B98A8402C7")]
[ComVisible(true)]
public class DotnetComponent: IWelcome, IMath
{
```

```
public DotnetComponent()
{
}
```

重新生成类库和类型库，会修改接口定义，使用 OleView.exe 可以看到这一点。现在，IWelcome 是一个双重接口，IMath 是一个自定义接口，它派生自 IUnknown 而不是 IDispatch，并且 coclass 部分不再有一个_DotnetComponent 接口。

23.3.5 COM 注册

在把.NET 组件用作 COM 对象之前，先要在注册表中配置它。而且，如果不想把程序集复制到客户端应用程序所在的目录，就必须在全局程序集缓存中安装该程序集。第 19 章讨论过全局程序集缓存。

为了在全局程序集缓存中安装程序集，必须先用一个强名称签名该程序集(使用 Visual Studio 2013 时，可以在解决方案的属性中定义强名)。然后，就可以在全局程序集缓存中注册该程序集：

```
gacutil -i DotnetServer.dll
```

现在可以使用 regasm 工具在注册表中配置该组件。使用/tlb 选项可以提取类型库，并在注册表中配置该类型库：

```
regasm DotnetServer.dll /tlb
```

写入到注册表中的.NET 组件信息如下所示。所有的 COM 配置包含在 HKEY_CLASSES_ROOT (HKCR)配置单元中。ProgID 的项(在本例中是 Wrox.DotnetComponent)和 CLSID 的项是直接写入到这个配置单元的。

HKCR\CLSID\{CLSID}\InProcServer32 项包含如下条目：

- mscoree.dll——代表 CCW。这是一个真正的 COM 对象，负责托管.NET 组件。这个 COM 对象通过访问.NET 组件为客户端提供 COM 行为。在客户端，通过正常的 COM 实例化机制来加载和实例化 mscoree.dll 文件。
- ThreadingModel=Both——这是 mscoree.dll COM 对象的一个特性。这个组件同时支持 STA 和 MTA。
- Assembly=DotnetServer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=5cd57c93b4d9c41a——Assembly 的值存储了程序集的完整名称，包括版本号和公钥标记，以便唯一标识该程序集。在这里注册的程序集将由 mscoree.dll 加载。
- Class=Wrox.ProCSharp.Interop.Server.DotnetComponent——类的名称也由 mscoree.dll 使用。所实例化的就是这个类。
- RuntimeVersion=v4.0.20826——注册表项 RuntimeVersion 指定了将用来托管.NET 程序集的.NET 运行库的版本。

除了这里显示的配置，所有的接口和类型库也都用它们的标识符配置。

如果.NET 组件是使用 AnyCPU(这是 Visual Studio 2013 中类库的默认设置)平台开发的，那么可以把它配置为 32 位或者 64 位的 COM 组件。从 VS2013 x86 Native Tools 命令提示符启动 regasm，会使用<windows>\Microsoft.NET\Framework\v4.0.30319 目录下的 regasm。从 VS2012 x64 Native Tools 命令提示符(如果有 64 位 Windows 的话)启动 regasm 会使用<windows>\Microsoft.NET\Framework64\v4.0.30319 目录下的 regasm。根据使用的工具，组件会注册到 HKCR\CLSID 或 HKCR\WOW6432Node\CLSID 下。

23.3.6 创建 COM 客户端应用程序

现在就创建一个 COM 客户端。首先创建一个简单的 C++ Win32 控制台应用程序，命名为 COMClient。保持项目向导中的默认选项不变，单击 Finish 按钮。

在文件 COMClient.cpp 的开头部分，添加预处理器命令来包含<iostream>头文件，并导入为.NET 组件创建的类型库。import 语句会创建一个智能指针类，使得处理 COM 对象变得更加容易。在生成过程中，import 语句会创建包含智能指针类的.tlh 和.tli 文件，在项目的调试目录下可以找到它们。然后添加 using namespace 指令来打开 std 名称空间和 DotnetServer 名称空间，其中 std 名称空间用来向控制台写输出消息，DotnetServer 名称空间是在智能指针类内创建的(代码文件 COMClient\COMClient.cpp)：

```
// COMClient.cpp: Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#import "./DotNetServer/bin/debug/DotnetServer.tlb" named_guids

using namespace std;
using namespace DotnetServer;
```

在_tmain 方法中进行其他任何 COM 调用之前，首先要调用 CoInitialize 来实例化 COM，它会为线程创建并使线程进入 STA。变量 spWelcome 是 IWelcomePtr 类型，这是一个智能指针。智能指针方法 CreateInstance 以 ProgID 作为参数，通过使用 COM API CoCreateInstance 来创建 COM 对象。运算符->被智能指针重写，以便调用 COM 对象的方法，例如 Greeting：

```
int _tmain(int argc, _TCHAR* argv[])
{
  HRESULT hr;
  hr = CoInitialize(NULL);

  try
  {
    IWelcomePtr spWelcome;

    // CoCreateInstance()
    hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

    cout<< spWelcome->Greeting("Bill") << endl;
```

.NET 组件支持的第二个接口是 IMath，有一个智能指针包装了这个 COM 接口：IMathPtr。可以直接将一个智能指针赋值给另一个智能指针，例如 spMath = spWelcome;。在智能指针的实现中(重载了=运算符)，调用了 QueryInterface 方法。使用 IMath 接口的引用可以调用 Add 方法：

```
  IMathPtr spMath;
  spMath = spWelcome; // QueryInterface()

  long result = spMath->Add(4, 5);
  cout<< "result:" << result << endl;
}
```

如果COM对象返回了一个HRESULT错误值(如果.NET组件生成了一个异常，CCW就会返回一个HRESULT错误)，智能指针就会包装HRESULT错误并生成_com_error异常。错误在catch块中处理。在程序最后，使用CoUninitialize关闭并卸载COM DLL：

```
    catch (_com_error& e)
    {
      cout << e.ErrorMessage() << endl;
    }

    CoUninitialize();
    return 0;
  }
```

运行程序后，在控制台上会得到Greeting和Add方法的输出。试着调试智能指针类，就能够直接看到COM API调用。

如果收到一个无法找到组件的异常，则检查在全局程序集缓存中安装的程序集的版本是否与在注册表中配置的版本相同。

23.3.7 添加连接点

在.NET组件中添加对COM事件的支持需要修改.NET类的实现。提供COM事件并不只是使用event和delegate关键字这么简单，还必须添加其他一些COM互操作特性。

首先，需要向.NET项目添加IMathEvents接口。这是组件的源接口或称作传出接口，在客户端中将由sink对象实现。源接口必须是一个调度接口或者自定义接口。脚本客户端只支持调度接口。通常，首选调度接口作为源接口(代码文件DotnetServer/DotnetServer.cs)：

```
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[ComVisible(true)]
public interface IMathEvents
{
  [DispId(46200)]
  void CalculationCompleted();
}
```

在DotnetComponent类中，必须指定一个源接口。这可以用[ComSourceInterfaces]特性完成：添加此特性，然后指定前面声明的传出接口。使用特性类的不同构造函数，可以指定不只一个源接口，但是唯一支持多个源接口的客户端语言是C++。Visual Basic 6.0客户端只支持一个源接口：

```
[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-4876-8297-35B98A8402C7")]
[ComSourceInterfaces(typeof(IMathEvents))]
[ComVisible(true)]
public class DotnetComponent : IWelcome, IMath
{
```

```
  public DotnetComponent()
  {
  }
```

在 DotnetComponent 类中，必须为源接口的每个方法声明一个事件。方法的类型必须是委托的名称，事件的名称必须与源接口中的方法的名称完全相同。可以把事件调用添加到 Add 和 Sub 方法中。这个步骤是标准的.NET 事件调用方式，如第 8 章所述：

```
  public event Action CalculationCompleted;

  public int Add(int val1, int val2)
  {
    int result = val1 + val2;
    if (CalculationCompleted != null)
      CalculationCompleted();
    return result;
  }

  public int Sub(int val1, int val2)
  {
    int result = val1 - val2;
    if (CalculationCompleted != null)
      CalculationCompleted();
    return result;
  }
}
```

事件的名称必须与源接口内方法的名称完全相同，否则将无法在 COM 客户端映射事件。

23.3.8 使用 sink 对象创建客户端

在生成并注册.NET 程序集，然后把它安装到全局程序集缓存后，就可以使用事件源生成客户端应用程序了。在 Visual Basic 6.0 中，实现一个实现了 IDispatch 接口的回调或 sink 对象只需要添加 With Events 关键字，这与现在 Visual Basic 处理.NET 事件的方式相同。使用 C++时要做的工作更多，但是 ATL 可以提供一些帮助。

打开“创建 COM 客户端应用程序”一节中创建的 C++控制台应用程序，在 stdafx.h 文件中添加下面的 include 语句：

```
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
```

stdafx.cpp 需要包含 ATL 实现文件 atlimpl.cpp：

```
#include <atlimpl.cpp>
```

将新类 CEventHandler 添加到 COMClient.cpp 文件中，该类包含由组件调用的 IDispatch 接口的

实现代码。IDispatch 接口通过基类 IDispEventImpl 来实现。IDispEventImpl 类从类型库中读取与类的方法相匹配的方法 ID 和参数。IDispEventImpl 类的模板参数包括 sink 对象的 ID(这里使用了 4)、实现回调方法的类(CEventHandler)、回调接口的接口 ID(DIID_IMathEvents)、类型库的 ID(LIBID_DotnetComponent)以及类型库的版本号。在#import 语句创建的 dotnetcomponent.tlh 文件中，可以找到 DIID_IMathEvents 和 LIBID_DotnetComponent 这两个 ID。

包含在 BEGIN_SINK_MAP 和 END_SINK_MAP 之间的 sink 映射定义了 sink 对象实现的方法。SINK_ENTRY_EX 将 OnCalcCompleted 方法映射到调度 ID46200。这个调度 ID 是用.NET 组件的 IMathEvents 接口的 CalculationCompleted 方法定义的(代码文件 COMClient/COMClient.cpp)：

```
class CEventHandler: public IDispEventImpl<4, CEventHandler,
    &DIID_IMathEvents, &LIBID_DotnetServer, 1, 0>
{
public:
  BEGIN_SINK_MAP(CEventHandler)
     SINK_ENTRY_EX(4, DIID_IMathEvents, 46200, OnCalcCompleted)
  END_SINK_MAP()

  HRESULT __stdcall OnCalcCompleted()
  {
    cout<< "calculation completed" << endl;
    return S_OK;
  }
};
```

现在需要修改 main 方法，使其通知组件存在事件 sink 对象，这样组件就可以回调 sink 对象。为此，使用 CEventHandler 类的 DispEventAdvise 方法并为其传入一个 IUnknown 接口指针作为参数。DispEventUnadvise 会取消注册 sink 对象：

```
int _tmain(int argc, _TCHAR* argv[])
{
  HRESULT hr;
  hr = CoInitialize(NULL);

  try
  {
    IWelcomePtr spWelcome;
    hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

    IUnknownPtr spUnknown = spWelcome;

    cout<< spWelcome->Greeting("Bill") << endl;

    CEventHandler* eventHandler = new CEventHandler();
    hr = eventHandler->DispEventAdvise(spUnknown);

    IMathPtr spMath;
    spMath = spWelcome; // QueryInterface()

    long result = spMath->Add(4, 5);
    cout<< "result:" << result << endl;
```

```
        eventHandler->DispEventUnadvise(spWelcome.GetInterfacePtr());
        delete eventHandler;
    }
    catch (_com_error& e)
    {
        cout<< e.ErrorMessage() << endl;
    }

    CoUninitialize();
    return 0;
}
```

23.4 平台调用

.NET Framework 并不能使用 Windows API 的全部功能，这不只包括以前的 Windows API 调用，也包括 Windows 8.1 或 Windows Server 2012 R2 的一些非常新的功能。假设编写一些 DLL 来导出非托管方法，现在也想在 C#中使用它们。

要想重用只包含导出的函数而不包含 COM 对象的非托管库，可以使用平台调用(p/invoke)。使用 p/invoke 时，CLR 会加载包含应该调用的函数并封送参数的 DLL。

为了使用非托管函数，首先需要确定该函数在导出时的名称。这可以使用 dumpbin 工具的/exports 选项完成。

例如，下面的命令：

```
dumpbin /exports c:\windows\system32\kernel32.dll | more
```

会列出 kernel32.dll 中所有导出的函数。在本例中，使用 CreateHardLink Windows API 函数来创建已有文件的硬链接。使用这个 API 调用时，只要文件名在一个硬盘上，就可以让几个文件名引用同一个文件。.NET Framework 4.5.1 没有提供这个 API 调用，所以必须使用平台调用。

为了调用本机函数，必须定义一个有相同数量的参数的 C#外部方法，并且在非托管方法中定义的参数类型必须能够映射到托管代码中的某个类型。

CreateHardLink Windows API 调用的 C++定义如下所示：

```
BOOL CreateHardLink(
  LPCTSTR lpFileName,
  LPCTSTR lpExistingFileName,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

这个定义必须映射到.NET 数据类型。非托管代码中的返回类型是 BOOL，它映射到 bool 数据类型。LPCTSTR 定义了指向常量字符串的一个 long 指针。Windows API 对数据类型使用了匈牙利命名法。LP 表示 long 指针，C 表示常量，STR 表示以 NULL 结尾的字符串。T 表示类型为泛型，根据编译器的设置，它会解析为 LPCSTR(ANSI 字符串)或 LPWSTR(Unicode 字符串)。C 字符串映射为.NET 类型 String。LPSECURITY_ATTRIBUTES 是 SECURITY_ATTRIBUTES 结构体的 long 指针。因为可以为这个参数传递 NULL，所以把它映射为 IntPtr 是可以的。在 C#中声明这个方法时需要加上 extern 修饰符，因为 C#代码内没有这个方法的实现。相反，该方法的实现包含在使用特性[DllImport]引用的 DLL kernel32.dll 中。.NET 声明的 CreateHardLink 的返回类型为 bool，但本机方法 CreateHardLink 的返回类型为 BOOL，所以最好再做进一步的说明。C++中有不同的布尔数据类型(例

如本机 bool 和 Windows 定义的 BOOL 的值是不同的)，所以使用[MarshalAs]特性指定.NET 类型 bool 应该映射到哪个本机类型：

```
[DllImport("kernel32.dll", SetLastError="true",
          EntryPoint="CreateHardLink", CharSet=CharSet.Unicode)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool CreateHardLink(string newFileName,
                                          string existingFilename,
                                          IntPtr securityAttributes);
```

在把本机代码转换为托管代码时，http://www.pinvoke.net 网站和 P/Invoke Interop Assistant 工具(可从 http://www.codeplex.com 下载)是很有帮助的。

在[DllImport]特性中可以指定的设置如表 23-3 所示：

表 23-3

[DllImport]的属性或字段	描 述
EntryPoint	允许在 C#中声明非托管库中的函数时使用不同的名称。非托管库中的方法的名称定义在 EntryPoint 字段中
CallingConvention	根据编译非托管函数时使用的编译器或编译器设置，可以使用不同的调用约定。调用约定定义了如何处理参数，以及把它们保存到栈上的什么位置。通过将 CallingConvention 设为一个枚举值，可以自己定义调用约定。在 Windows 操作系统中，Windows API 通常使用 StdCall 调用约定，在 Windows CE 中，则使用 Cdecl 调用约定。将这个值设为 CallingConvention.Winapi 就可以同时工作在 Windows 和 Windows CE 环境中
CharSet	字符串参数可以是 ANSI 或 Unicode。使用 CharSet 可以定义管理字符串的方式。CharSet 枚举中定义的值包括 Ansi、Unicode 和 Auto。CharSet.Auto 在 Windows NT 平台上使用 Unicode，在 Windows 98 和 Windows ME 上使用 ANSI
SetLastError	如果非托管函数使用 Windows API SetLastError 设置了一个错误，就可以把 SetLastError 字段设为 true。这样，以后就可以使用 Marshal.GetLastWin32Error 读取错误号

为了使 CreateHardLink 更容易在.NET 环境中使用，应该遵循下面的指导原则：

- 创建内部类 NativeMethods，使其包装平台调用。
- 创建公有类，使其为.NET 应用程序提供本机方法的功能。
- 使用安全特性来标记必要的安全性。

在下例中(代码文件 PInvokeSample/NativeMethods.cs)，FileUtility 类中的公有方法 CreateHardLink 是.NET 应用程序可以使用的方法。该方法的文件名参数是将本机 Windows API 函数 CreateHardLink 的文件名参数颠倒了过来，第一个参数是已有文件的名称，第二个参数是新文件的名称。这与框架中的其他类相似，如 File.Copy。因为这个实现中没有使用为新文件名传递安全特性的第三个参数，所以这个公有方法只有两个参数。返回类型也被修改。现在不再通过返回值 false 来返回错误，而是抛出一个异常。在发生错误时，非托管方法 CreateHardLink 使用非托管 API SetLastError 来设置错误

号。为了在.NET 中读取这个值，需要把[DllImport]字段 SetLastError 设为 true。在托管方法 CreateHardLink 中通过调用 Marshal.GetLastWin32Error 读取错误号。为了使用这个错误号创建一条错误消息，需要使用 System.ComponentModel 名称空间中的 Win32Exception 类。这个类的构造函数接受一个错误号作为参数，返回一条本地化后的错误消息。在发生错误时，会抛出一个 IOException 类型的异常，它的内部有一个 Win32Exception 类型的异常。对公有方法 CreateHardLink 应用 FileIOPermission 特性，以检查调用者是否有必要的权限。第 22 章详细介绍了.NET 安全性。

```
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Interop
{
  [SecurityCritical]
  internal static class NativeMethods
  {
    [DllImport("kernel32.dll", SetLastError = true,
      EntryPoint = "CreateHardLinkW", CharSet = CharSet.Unicode)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool CreateHardLink(
      [In, MarshalAs(UnmanagedType.LPWStr)] string newFileName,
      [In, MarshalAs(UnmanagedType.LPWStr)] string existingFileName,
      IntPtr securityAttributes);

    internal static void CreateHardLink(string oldFileName,
                                        string newFileName)
    {
      if (!CreateHardLink(newFileName, oldFileName, IntPtr.Zero))
      {
      var ex = new Win32Exception(Marshal.GetLastWin32Error());
      throw new IOException(ex.Message, ex);
      }
    }
  }

  public static class FileUtility
  {
    [FileIOPermission(SecurityAction.LinkDemand, Unrestricted = true)]
    public static void CreateHardLink(string oldFileName,
                                      string newFileName)
    {
      NativeMethods.CreateHardLink(oldFileName, newFileName);
    }
  }
}
```

现在，使用这个类创建硬链接很容易(代码文件 PInvokeSample/Program.cs)。如果在第一个参数中传递的文件不存在，就会得到一个异常，消息为："The system cannot find the file specified."。如果文件存在，就会有一个新的文件名引用原来的文件。为了进行验证，可以修改一个文件中的文本，

所做修改也会出现在另一个文件中：

```
using System;
using System.IO;

namespace Wrox.ProCSharp.Interop
{
  class Program
  {
    static void Main(string[] args)
    {
      if (args.Length != 2)
      {
        Console.WriteLine("usage: PInvokeSample " +
          "existingfilename newfilename");
        return;
      }
      try
      {
        FileUtility.CreateHardLink(args[0], args[1]);
      }
      catch (IOException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

在本机方法调用中，经常要用到Windows句柄。Windows句柄是一个32位的值，有些类型的句柄对可能的取值有所限制。在.NET 1.0中，通常为句柄使用IntPtr结构，因为使用这个结构可以设置所有的32位值。但是，对于一些句柄类型，这会带来安全问题，可能还会造成线程争用，以及在最终完成阶段泄露句柄。因此，.NET 2.0引入了SafeHandle类。SafeHandle类是每个Windows句柄的抽象基类。它在Microsoft.Win32.SafeHandles名称空间中的派生类包括SafeHandleZeroOrMinusOneIsInvalid和SafeHandleMinusOneIsInvalid。顾名思义，这些类不接受无效的0或-1值。进一步的派生句柄类型有SafeFileHandle、SafeWaitHandle、SafeNCryptHandle和SafePipeHandle，它们可以被特定的Windows API调用使用。

例如，为了映射Windows API CreateFile，可以使用下面的声明来返回一个SafeFileHandle。当然，通常使用.NET类File和FileInfo就行了。

```
[DllImport("Kernel32.dll", SetLastError = true,
           CharSet = CharSet.Unicode)]
internal static extern SafeFileHandle CreateFile(
  string fileName,
  [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
  [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
  IntPtr securityAttributes,
  [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
  int flags,
  SafeFileHandle template);
```

第25章将介绍如何创建自定义的SafeHandle类，以使用从Windows Vista后开始提供的Windows事务处理文件API。

23.5 小结

本章提到，不同代的COM和.NET应用程序可以彼此交互。在.NET应用程序中，可以像使用.NET类一样使用COM组件，并不需要重写应用程序或者组件。这需要用到tlbimp工具，它创建的运行库可调用包装(RCW)用.NET表示隐藏了COM对象。

类似地，使用tlbexp工具可以从.NET组件创建一个类型库，供COM可调用包装(CCW)使用。CCW在COM表示下隐藏了.NET组件。将.NET类用作COM组件时，必须使用System.Runtime.InteropServices名称空间中的一些特性来定义COM客户端需要的特定COM特征。

通过平台调用，可以使用C#调用原生方法。平台调用要求使用C#和.NET数据类型重新定义本机方法。在重新定义后，就可以像调用C#方法一样调用本机方法。

下一章将介绍如何通过文件和流的方式来访问文件系统。

第24章

文件和注册表操作

本章要点

- 介绍目录结构
- 移动、复制、删除文件和文件夹
- 读写文本文件
- 读写注册表键
- 读写独立存储器

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- BinaryFileReader
- DriveViewer
- FileProperties
- FilePropertiesAndMovement
- MappedMemoryFiles
- ReadingACLs
- ReadingACLsFromDirectory
- ReadingFiles
- ReadWriteText

24.1 文件和注册表

本章将介绍如何在 C#中执行读写文件和系统注册表的任务。Microsoft 提供了非常直观的对象模型，这些模型包括所有这些领域。本章还将介绍如何使用.NET 基类执行上面的任务。对于文件系统操作，相关的类几乎都在 System.IO 名称空间中，而注册表操作由 System.Win32 名称空间中的类

来处理。

注意，在修改文件或注册表项时，安全性显得更为重要。第 22 章介绍了安全性的各个方面。但在本章中，仅假定用户有足够的访问权限运行修改文件或注册表项的所有示例，如果在拥有管理员权限的账户下运行，就是这种情况。

.NET 基类也包含 System.Runtime.Serialization 名称空间中的许多类和接口，它们都与串行化有关。串行化是把一些数据(例如，文档的内容)转换为字节流并存储在某个地方的过程。本章不讨论这些类，而主要讨论可直接访问文件的类。

24.2 管理文件系统

图 24-1 中的类可以用于浏览文件系统和执行操作，如移动、复制和删除文件。

这些类的作用是：

- System.MarshalByRefObject ——这是.NET 类中用于远程操作的基对象类，它允许在应用程序域之间编组数据。这个列表中的其他项都在 System.IO 名称空间中。
- FileSystemInfo ——这是表示任何文件系统对象的基类。
- FileInfo 和 File ——这些类表示文件系统上的文件。
- DirectoryInfo 和 Directory ——这些类表示文件系统上的文件夹。
- Path ——这个类包含的静态成员可以用于处理路径名。
- DriveInfo ——它的属性和方法提供了指定驱动器的信息。

在 Windows 上，包含文件并用于组织文件系统的对象称为文件夹。例如，在路径 C:\My Documents\ReadMe.txt 中，ReadMe.txt 是一个文件，My Documents 是一个文件夹。文件夹是一个 Windows 专用的术语：在其他操作系统上，用术语“目录”代替文件夹，Microsoft 为了使.NET 具有平台无关性，对应的.NET 基类都称为 Directory 和 DirectoryInfo。因为它有可能与 LDAP 目录混淆，而且本书与 Windows 有关，所以本章仍使用文件夹。

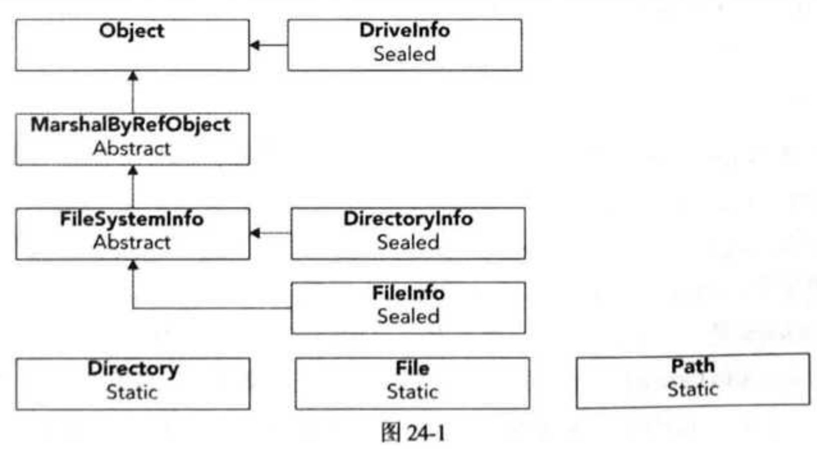


图 24-1

24.2.1 表示文件和文件夹的.NET 类

注意，上面的列表有两个用于表示文件夹的类，和两个用于表示文件的类。使用哪个类主要依赖于访问该文件夹或文件的次数：

- Directory 类和 File 类只包含静态方法，不能被实例化。只要调用一个成员方法，提供合适文件系统对象的路径，就可以使用这些类。如果只对文件夹或文件执行一个操作，使用这些类就很有效，因为这样可以省去实例化.NET 类的系统开销。
- DirectoryInfo 类和 FileInfo 类实现与 Directory 类和 File 类大致相同的公共方法，并拥有一些公共属性和构造函数，但它们都是有状态的，并且这些类的成员都不是静态的。需要实例化这些类，之后把每个实例与特定的文件夹或文件关联起来。如果使用同一个对象执行多个操作，使用这些类就比较有效。这是因为在构造时它们将读取合适文件系统对象的身份验证和其他信息，无论对每个对象(类实例)调用了多少方法，都不需要再次读取这些信息。比较而言，在调用每个方法时，相应的无状态类需要再次检查文件或文件夹的详细内容。

本节主要使用 FileInfo 类和 DirectoryInfo 类，但我们调用的许多方法(不是全部)也可以由 File 类和 Directory 类实现(尽管这些方法需要一个额外的参数——文件系统对象的路径名，这两个方法的名称略有不同)。例如：

```
FileInfo myFile = new FileInfo(@"C:\Program Files\My Program\ReadMe.txt");
myFile.CopyTo(@"D:\Copies\ReadMe.txt");
```

与下面的代码有相同的效果：

```
File.Copy(@"C:\Program Files\My Program\ReadMe.txt", @"D:\Copies\ReadMe.txt");
```

第一个代码段执行的时间略长，因为需要实例化一个 FileInfo 对象 myFile，但 myFile 可以对同一个文件执行进一步的操作。第二个示例不需要实例化对象来复制文件。

把包含对应文件系统对象的路径字符串传递给构造函数，就可以实例化 FileInfo 类或 DirectoryInfo 类。刚才已经介绍了文件的处理，对于文件夹，代码如下：

```
DirectoryInfo myFolder = new DirectoryInfo(@"C:\Program Files");
```

如果路径表示一个不存在的对象，在构造时就不会抛出异常。但如果是第一次调用方法，而该方法需要有一个对应的文件系统对象，就会抛出一个异常。检查 Exists 属性，可以确定对象是否存在，其类型是否合适，FileInfo 类和 DirectoryInfo 类都会实现该属性：

```
FileInfo test = new FileInfo(@"C:\Windows");
Console.WriteLine(test.Exists.ToString());
```

注意，这个属性要返回 true，对应的文件系统对象必须是合适的类型。换言之，如果实例化 FileInfo 对象时提供了文件夹的路径，或者实例化 DirectoryInfo 对象时它提供了文件的路径，Exists 的值就是 false。如果可能，这些对象的大多数属性和方法都会返回一个值——它们不会因为调用了错误类型的对象而抛出异常，除非要求它们完成一些确实不可能的任务。例如，上面的代码段会先显示 false(因为 C:\Windows 是一个文件夹)，但接着就会显示创建文件夹的时间，因为文件夹仍拥有该信息。但如果使用 FileInfo.Open()方法，以打开文件的方式打开文件夹，就会产生一个异常。

在确定了是否存在对应的文件系统对象后，就可以(如果用户正在使用 FileInfo 类或 DirectoryInfo

类)使用许多属性来确定该对象的信息，这些属性如表 24-1 所示。

表 24-1

名　称	作　用
CreationTime	创建文件或文件夹的时间
DirectoryName (仅用于 FileInfo)	包含文件夹的完整路径名
Parent (仅用于 DirectoryInfo)	指定子目录的父目录
Exists	文件或文件夹是否存在
Extension	文件的扩展名，对于文件夹它返回空白
FullName	文件或文件夹的完整路径名
LastAccessTime	最后一次访问文件或文件夹的时间
LastWriteTime	最后一次修改文件或文件夹的时间
Name	文件或文件夹的名称
Root(仅用于 DirectoryInfo)	路径的根部分
Length(仅用于 FileInfo)	返回文件的大小(以字节为单位)

也可以使用表 24-2 所示的方法对文件系统对象执行操作。

表 24-2

名　称	作　用
Create()	创建给定名称的文件夹或空文件。对于 FileInfo，该方法会返回一个流对象，以便写入文件。本章后面讨论流
Delete()	删除文件或文件夹。对于文件夹，有一个可以递归的 Delete 选项
MoveTo()	移动和/或重命名文件或文件夹
CopyTo()	(只适用于 FileInfo)复制文件，注意文件夹没有复制方法。如果复制完整的目录树，需要单独复制每个文件，创建对应于旧文件夹的新文件夹
GetDirectories()	(只适用于 DirectoryInfo)返回 DirectoryInfo 对象数组，该数组表示文件夹中包含的所有文件夹
GetFiles()	(只适用于 DirectoryInfo)返回 FileInfo 对象数组，该数组表示文件夹中包含的所有文件
EnumerateFiles()	返回文件名的 IEnumerable<string>。在返回整个列表之前，可以对列表中的项执行操作
GetFileSystemInfos()	(只适用于 DirectoryInfo)返回 FileInfo 和 DirectoryInfo 对象，它把文件夹中包含的所有对象表示为一个 FileSystemInfo 引用数组

注意，表 24-2 给出了主要的属性和方法，但没有列出所有的属性和方法。

表 24-2 没有列出读写文件数据的大多数属性或方法。读写文件数据实际上使用流对象完成，本章后面会介绍流对象。FileInfo 也可以实现 Open()、OpenRead()、OpenText()、OpenWrite()、Create()和 CreateText()等方法，为此它们都返回流对象。

有趣的是，创建时间、最后一次访问时间和最后一次写入时间都是可写入的。

```
// displays the creation time of a file,
// then changes it and displays it again
FileInfo test = new FileInfo(@"C:\MyFile.txt");
Console.WriteLine(test.Exists.ToString());
Console.WriteLine(test.CreationTime.ToString());
test.CreationTime = new DateTime(2010, 1, 1, 7, 30, 0);
Console.WriteLine(test.CreationTime.ToString());
```

运行这个应用程序，结果如下所示：

```
True
2/5/2009 2:59:32 PM
1/1/2010 7:30:00 AM
```

能手动修改这些属性首先看起来很奇怪，但它相当有效。例如，如果有一个程序可以通过读取、删除文件，用新内容创建新文件，来有效地修改文件，则可以修改创建日期，以匹配旧文件的最初创建日期。

24.2.2 Path 类

Path 类不能实例化。然而，它提供了一些静态方法，可以更容易地对路径名执行操作。例如，假定要显示文件夹 C:\My Documents 中 ReadMe.txt 文件的完整路径名，可以用下述代码查找文件的路径：

```
Console.WriteLine(Path.Combine(@"C:\My Documents", "ReadMe.txt"));
```

使用 Path 类要比手动处理各个符号容易得多。尤其因为 Path 类在处理不同操作系统上的路径名时，能够识别不同的格式。在编写本书时，Windows 是.NET 唯一支持的操作系统，但如果.NET 以后要移植到 UNIX 上，Path 就要处理 UNIX 路径，Unix 把“/”(并不是“\”)用作路径名中的分隔符。Path.Combine()是这个类常常使用的一个方法，Path 类还实现了其他方法，这些方法提供路径的信息，或者以要求的格式显示信息。

表 24-3 列出了 Path 类的一些静态字段。

表 24-3

属 性	说 明
AltDirectorySeparatorChar	提供一种与平台无关的方式，来指定分隔目录级别的另一个字符。在 Windows 上使用“/”符号，而在 UNIX 上使用“\”符号
DirectorySeparatorChar	提供一种与平台无关的方式，来指定分隔目录级别的另一个字符。在 Windows 上使用“/”符号，在 UNIX 上使用“\”符号
PathSeparator	提供一种与平台无关的方式，来指定划分环境变量的路径字符串，默认为分号
VolumeSeparatorChar	提供一种与平台无关的方式，来指定容量分隔符，默认为冒号

下一节用一个示例来说明如何浏览目录，查看文件的属性。

24.2.3 FileProperties 示例

本节要创建一个 C#示例应用程序 FileProperties。该应用程序显示了一个简单的用户界面，可以浏览文件系统，查看文件的创建时间、最后一次访问时间，最后一次写入时间和文件的大小。这个

应用程序的代码可以从 Wrox 网站 www.wrox.com 中下载。

FileProperties 应用程序如下所述。在窗口顶部的主文本框中输入文件夹或文件的名称，再单击 Display 按钮。如果输入了文件夹的路径，其内容就显示在列表框中。如果输入文件的路径，则其详细信息显示在窗体底部的文本框中，父文件夹的内容显示在列表框中。图 24-2 显示了 FileProperties 示例应用程序。

在定位文件系统时，单击右边列表框中的任何一个文件夹，就可以查看它下面的文件夹，或者单击 Up 按钮，查看其父文件夹。图 24-2 显示了 users 文件夹的内容。用户还可以单击列表框中的一个文件名，选择该文件。此时文件的属性就会显示在应用程序底部的文本框中，如图 24-3 所示。

图 24-2

图 24-3

注意，根据需要，还可以使用 DirectoryInfo 属性显示文件夹的创建时间、最后一次访问时间和最后一次修改时间。我们只显示已选定文件的这些属性，使本例简单一些。

在 Visual Studio 2013 中创建一个标准的 C# Windows 应用程序项目，从工具箱的 Windows Forms 区域添加各种文本框和列表框。用更直观的名称来重命名这些控件：textBoxInput、textBoxFolder、buttonDisplay、buttonUp、listBoxFiles、listBoxFolders、textBoxFileName、textBoxCreationTime、textBoxLastAccessTime、textBoxLastWriteTime 和 textBoxFileSize。

然后需要指出，将使用 System.IO 名称空间：

```
using System;
using System.IO;
using System.Windows.Forms;
```

本章中所有与文件系统相关的示例都要使用该名称空间，但我们没有显式说明其余示例中的这部分代码。然后在主窗体中添加一个成员字段：

```
public partial class Form1: Form
  {
     private string currentFolderPath;
```

CurrentFolderPath 字符串存储了文件夹的路径，其内容显示在列表框中。

现在需要为用户生成的事件添加事件处理程序。用户可能输入的是：

- 用户单击 Display 按钮——此时，需要确定用户在主文本框中输入的内容是文件的路径还是文件夹的路径。如果是文件夹，列表框就会列出该文件夹中的文件和子文件夹。如果是文件，则仍要对包含该文件的文件夹进行上述操作，还要在下面的文本框中显示文件的属性。
- 用户单击 Files 列表框中的一个文件名——此时，在下面的文本框中显示文件的属性。
- 用户单击 Folders 列表框中的一个文件夹名——此时，将清空所有控件，并且在列表框中显示这个子文件夹的内容。
- 用户单击 Up 按钮——此时，将清空所有控件，并且在列表框中显示当前选中的文件夹的父文件夹的内容。

在查看事件处理程序的代码前，先列出实际完成所有任务的方法的代码。首先，需要清空所有控件的内容，这个方法很容易理解：

```
protected void ClearAllFields()
{
   listBoxFolders.Items.Clear();
   listBoxFiles.Items.Clear();
   textBoxFolder.Text = "";
   textBoxFileName.Text = "";
   textBoxCreationTime.Text = "";
   textBoxLastAccessTime.Text = "";
   textBoxLastWriteTime.Text = "";
   textBoxFileSize.Text = "";
}
```

其次，定义一个方法 DisplayFileInfo()，该方法用于在文本框中显示给定文件的信息。它接受一个字符串参数，即文件的完整路径名，它根据该路径创建一个 FileInfo 对象：

```
protected void DisplayFileInfo(string fileFullName)
{
   FileInfo theFile = new FileInfo(fileFullName);

   if (!theFile.Exists)
   {
      throw new FileNotFoundException("File not found: " + fileFullName);
   }

   textBoxFileName.Text = theFile.Name;
   textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
   textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
   textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
   textBoxFileSize.Text = theFile.Length.ToString() + " bytes";
}
```

注意，如果在指定位置定位文件时有任何问题，我们将采取措施，防止抛出异常。异常在主调例程(一个事件处理程序)中处理。最后，定义一个方法 DisplayFolderList()，在两个列表框中显示给定文件夹的内容。该文件夹的完整路径名作为参数传递给该方法：

```
protected void DisplayFolderList(string folderFullName)
{
   DirectoryInfo theFolder = new DirectoryInfo(folderFullName);

   if (!theFolder.Exists)
   {
```

```
      throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
   }

   ClearAllFields();
   textBoxFolder.Text = theFolder.FullName;
   currentFolderPath = theFolder.FullName;

   // list all subfolders in folder
   foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
      listBoxFolders.Items.Add(nextFolder.Name);

   // list all files in folder
   foreach(FileInfo nextFile in theFolder.GetFiles())
      listBoxFiles.Items.Add(nextFile.Name);
}
```

用户单击 Display 按钮时会触发相应的事件，管理该事件的事件处理程序最复杂，因为它需要处理用户输入的 3 种不同的文本。用户可能输入文件夹的路径名、文件的路径名或什么也不输入：

```
protected void OnDisplayButtonClick(object sender, EventArgs e)
{
   try
   {
      string folderPath = textBoxInput.Text;
      DirectoryInfo theFolder = new DirectoryInfo(folderPath);

      if (theFolder.Exists)
      {
         DisplayFolderList(theFolder.FullName);
         return;
      }

      FileInfo theFile = new FileInfo(folderPath);

      if (theFile.Exists)
      {
         DisplayFolderList(theFile.Directory.FullName);
         int index = listBoxFiles.Items.IndexOf(theFile.Name);
         listBoxFiles.SetSelected(index, true);
         return;
      }

      throw new FileNotFoundException("There is no file or folder with "
                                       + "this name: " + textBoxInput.Text);
   }
   catch(Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
}
```

在上面的代码中，如果用户提供的文本表示一个文件夹或文件，就应实例化 DirectoryInfo 和 FileInfo 实例，并检查每个对象的 Exists 属性。如果它们都不存在，就抛出一个异常。如果它是一个文件夹，就调用 DisplayFolderList()方法，给列表框填充数据。如果它是一个文件，就需要给列表框填充数据，给显示文件属性的文本框填充文本。具体处理过程是，首先给列表框填充数据，然后在

Files 列表框中以编程方式选择合适的文件名，这与用户选择该项的效果完全相同——它引发选中项对应的事件。然后退出当前事件处理程序，调用选中项的事件处理程序，来显示文件属性。

下面的代码是一个事件处理程序，当用户选中或以编程方式选中 Files 列表框中的一项时，就可以由用户或上面编写的代码调用该事件处理程序。它仅构造所选文件的完整路径名，并把该路径传递给前面给出的 DisplayFileInfo()方法：

```
protected void OnListBoxFilesSelected(object sender, EventArgs e)
{
   try
   {
      string selectedString = listBoxFiles.SelectedItem.ToString();
      string fullFileName = Path.Combine(currentFolderPath, selectedString);
      DisplayFileInfo(fullFileName);
   }
   catch(Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
}
```

处理 Folders 列表框中文件夹选择操作的事件处理程序以非常类似的方式实现。但此时调用 DisplayFolderList 方法来更新列表框的内容：

```
protected void OnListBoxFoldersSelected(object sender, EventArgs e)
{
   try
   {
      string selectedString = listBoxFolders.SelectedItem.ToString();
      string fullPathName = Path.Combine(currentFolderPath, selectedString);
      DisplayFolderList(fullPathName);
   }
   catch(Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
}
```

最后，在单击 Up 按钮时，必须调用 DisplayFolderList()方法，但这次需要获得当前显示的文件夹的父文件夹的路径。这可以通过 FileInfo.DirectoryName 属性来得到，该属性返回父文件夹的路径：

```
protected void OnUpButtonClick(object sender, EventArgs e)
{
   try
   {
      string folderPath = new FileInfo(currentFolderPath).DirectoryName;
      DisplayFolderList(folderPath);
   }
   catch(Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
}
```

24.3　移动、复制和删除文件

前面已经提到，移动和删除文件或文件夹可以使用 FileInfo 和 DirectoryInfo 类的 MoveTo()和 Delete()方法来完成。File 和 Directory 类的这两个对应方法是 Move()和 Delete()。FileInfo 和 File 类也分别实现 CopyTo()和 Copy()方法。没有复制完整文件夹的方法，而应复制文件夹中的每个文件。

这些方法的使用非常直观——SDK 文档提供了详细的解释。本节介绍在特定情况下，调用 File 类的静态方法 Move()、Copy()和 Delete()的作用。为此，把前面的 FileProperties 示例扩展为一个新示例 FilePropetiesAndMovement。这个示例有一个额外的功能：无论什么时候显示文件的属性，该应用程序都会给出删除该文件或把该文件移动或者复制到另一个地方的选项。

24.3.1　FilePropertiesAndMovement 示例

图 24-4 是新示例应用程序的用户界面。

可以看出，FilePropetiesAndMovement 的外观非常类似于 FileProperties 示例，但在窗口的底部添加了 3 个按钮组成的一组和一个文本框。这些控件仅在示例显示了文件的属性时才启用，在所有其他情况下，它们都是禁用的。我们还压缩了现有的控件，防止主窗体过大。在显示所选中文件的属性时，该示例会自动把文件的完整路径名放在底部的文本框中，供用户编辑。用户可以单击底部的任何一个按钮，执行相应的操作。此时，会显示一个相应的信息框，由用户确认该操作，如图 24-5 所示。

当用户单击 Yes 按钮后，就可以开始执行某些动作。用户在窗体上执行的某些动作会使显示不正确。例如，在移动和删除文件时，显然不能在同一个地方显示该文件的内容。而且，如果改变同一个文件夹中的文件名，显示的信息也会过时。此时，FilePropertiesAndMovement 示例会重置其控件，在文件的操作结束后，只显示其中驻留文件的文件夹。

图 24-4

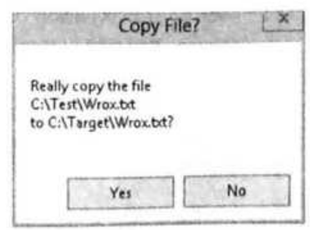


图 24-5

24.3.2 FilePropertiesAndMovement示例的代码

为此，需要在FileProperties示例中添加相关的控件，及其事件处理程序代码。我们添加的控件是buttonDelete、buttonCopyTo、buttonMoveTo和 txtBoxNewPath。

首先看看用户单击Delete按钮时调用的事件处理程序：

```
protected void OnDeleteButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath,
                                   textBoxFileName.Text);
    string query = "Really delete the file\n" + filePath + "?";
    if (MessageBox.Show(query,
       "Delete File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
      File.Delete(filePath);
      DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to delete file. The following exception"
                    + " occurred:\n" + ex.Message, "Failed");
  }
}
```

这个方法的代码包含在一个try块中，这是因为很显然会抛出一个异常，例如，在用户单击Delete按钮前，如果没有删除该文件的权限，或者在文件显示之后另一个进程移动了该文件，就会抛出一个异常。在CurrentParentPath字段中构造要删除文件的路径，其中包含父文件夹的路径，和textBoxFileName文本框中的文本(其中包含文件名)。

移动和复制文件的方法以类似的方式构造：

```
protected void OnMoveButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath,
                                   textBoxFileName.Text);
    string query = "Really move the file\n" + filePath + "\nto "
                   + textBoxNewPath.Text + "?";
    if (MessageBox.Show(query,
       "Move File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
       File.Move(filePath, textBoxNewPath.Text);
       DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to move file. The following exception"
                    + " occurred:\n" + ex.Message, "Failed");
```

```
  }
}

protected void OnCopyButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath,
                                   textBoxFileName.Text);
    string query = "Really copy the file\n" + filePath + "\nto "
                   + textBoxNewPath.Text + "?";
    if (MessageBox.Show(query,
       "Copy File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
      File.Copy(filePath, textBoxNewPath.Text);
      DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to copy file. The following exception"
                   + " occurred:\n" + ex.Message, "Failed");
  }
}
```

此外，还需要确保新按钮和文本框在合适的时间是启用的或禁用的。要在显示文件的内容时启用它们，需要把下述代码添加到 DisplayFileInfo()方法中：

```
protected void DisplayFileInfo(string fileFullName)
{
  FileInfo theFile = new FileInfo(fileFullName);

  if (!theFile.Exists)
  {
     throw new FileNotFoundException("File not found: " + fileFullName);
  }

  textBoxFileName.Text = theFile.Name;
  textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
  textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
  textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
  textBoxFileSize.Text = theFile.Length.ToString() + " bytes";

  // enable move, copy, delete buttons
  textBoxNewPath.Text = theFile.FullName;
  textBoxNewPath.Enabled = true;
  buttonCopyTo.Enabled = true;
  buttonDelete.Enabled = true;
  buttonMoveTo.Enabled = true;
}
```

还需要修改 DisplayFolderList()方法：

```
protected void DisplayFolderList(string folderFullName)
{
```

```
    DirectoryInfo theFolder = new DirectoryInfo(folderFullName);

    if (!theFolder.Exists)
    {
       throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
    }

    ClearAllFields();
    DisableMoveFeatures();
    textBoxFolder.Text = theFolder.FullName;
    currentFolderPath = theFolder.FullName;

    // list all subfolders in folder
    foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
       listBoxFolders.Items.Add(NextFolder.Name);

    // list all files in folder
    foreach(FileInfo nextFile in theFolder.GetFiles())
       listBoxFiles.Items.Add(NextFile.Name);
}
```

DisableMoveFeatures()方法是禁用新控件的一个小实用程序函数：

```
void DisableMoveFeatures()
{
  textBoxNewPath.Text = "";
  textBoxNewPath.Enabled = false;
  buttonCopyTo.Enabled = false;
  buttonDelete.Enabled = false;
  buttonMoveTo.Enabled = false;
}
```

还需要给 ClearAllFields()方法添加额外的代码，以清除额外的文本框：

```
protected void ClearAllFields()
{
  listBoxFolders.Items.Clear();
  listBoxFiles.Items.Clear();
  textBoxFolder.Text = "";
  textBoxFileName.Text = "";
  textBoxCreationTime.Text = "";
  textBoxLastAccessTime.Text = "";
  textBoxLastWriteTime.Text = "";
  textBoxFileSize.Text = "";
  textBoxNewPath.Text = "";
}
```

24.4 读写文件

读写文件在原则上非常简单，但它不是通过 DirectoryInfo 或 FileInfo 对象完成。在.NET Framework 4.5 中，可以通过 File 对象读写文件。本章后面将学习如何使用许多其他类来读写文件，这些类表示一个通用的概念：流。

在.NET Framework 2.0 推出之前，读写文件比较费劲，可以使用 Framework 中的类来读写文件，但不是很简单。.NET Framework 2.0 扩展了 File 类，只需要编写一行代码，就可以读写文件。.NET Framework 4.5 也有这个功能。

24.4.1　读取文件

对于下面读取文件的示例，创建一个 Windows 窗体应用程序，它包含一个常规的文本框、一个按钮和一个多行文本框。最后，窗体如图 24-6 所示。

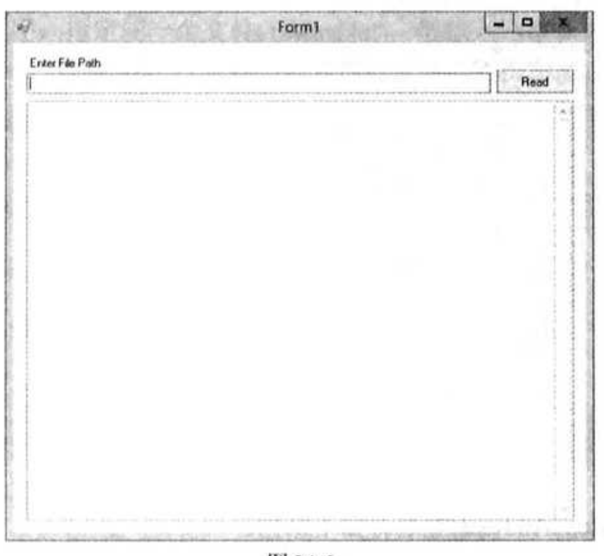


图 24-6

这个窗体的作用是，最终用户在第一个文本框中输入某个特定文件的路径，并单击 Read 按钮。此时应用程序就应读取指定的文件，并在多行文本框中显示文件的内容。其代码如下：

```
using System;
using System.IO;
using System.Windows.Forms;

namespace ReadingFiles
{
  public partial class Form1: Form
  {
    public Form1()
    {
       InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
       textBox2.Text = File.ReadAllText(textBox1.Text);
    }
  }
}
```

在构建这个示例时，第一步是添加 using 语句，引入 System.IO 名称空间。之后，使用窗体上 Send 按钮的 button1_Click 事件，用文件中的内容填充文本框。现在，就可以使用 File.ReadAllText()

方法获得文件的内容。显然，可以使用一条语句读取文件。ReadAllText()方法会打开指定的文件，读取内容，然后关闭文件。ReadAllText()方法的返回值是包含文件全部内容的字符串。最终结果如图 24-7 所示。

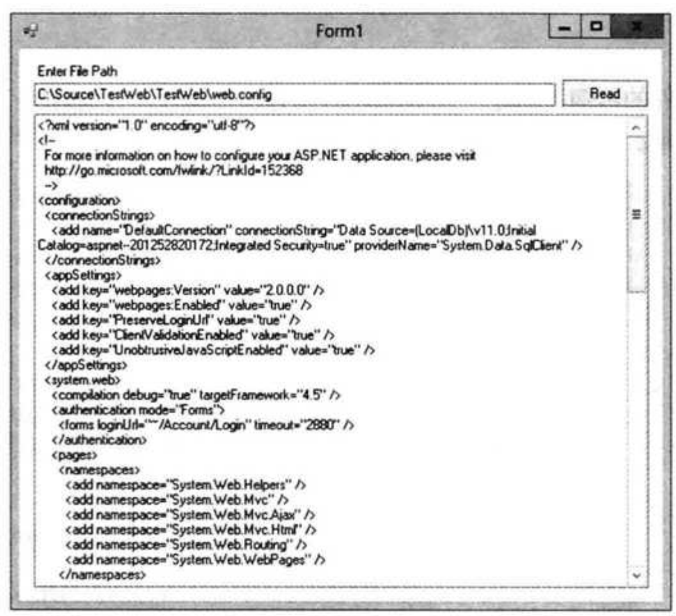


图 24-7

上面示例中的 File.ReadAllText()签名有如下结构：

```
File.ReadAllText(FilePath);
```

另一个选项指定了要读取的文件的编码格式：

```
File.ReadAllText(FilePath, Encoding);
```

使用这个签名可以指定在打开和读取文件内容时使用的编码格式。因此，可以编写如下代码：

```
File.ReadAllText(textBox1.Text, Encoding.ASCII);
```

打开并处理文件的其他选项有 ReadAllBytes()和 ReadAllLines()方法。ReadAllBytes()方法可以打开二进制文件，把其内容读入一个字节数组中。前面列出的 ReadAllText()方法会在一个字符串实例中提供指定文件的全部内容。我们对此不感兴趣，而关注以逐行方式处理文件中内容的方式。此时，应使用 ReadAllLines()方法，因为它拥有这个功能，并返回一个字符串数组，以进行处理。

24.4.2 写入文件

在.NET Framework 中，基类库除了使读取文件是一个非常简单的过程之外，也使写入文件一样简单。基类库除了提供 ReadAllText()、ReadAllLines()和 ReadAllBytes()方法，以几种不同的方式读取文件之外，还提供了写入文件的方法 WriteAllText()、WriteAllBytes()和 WriteAllLines()。

为了说明如何写入文件，虽然使用同一个 Windows 窗体应用程序，但把窗体中的多行文本框用于将数据输入文件中。button1_Click 事件处理程序的代码如下所示：

```
private void button1_Click(object sender, EventArgs e)
{
```

```
        File.WriteAllText(textBox1.Text, textBox2.Text);
    }
```

构建窗体，并启动，在第一个文本框中输入 C:\Testing.txt，在第二个文本框中输入一些随机内容，然后单击按钮。从视觉上什么也没有发生，但如果查看 C 盘，就会看到包含指定内容的 Testing.txt 文件。

如果在保存和关闭文件之前，给文件指定了一些内容，WriteAllText()方法就会进入指定的位置，新建一个文本文件。只需一行代码即可完成文件的写入操作。

如果再次运行应用程序，指定同一个文件(Testing.txt)，但输入一些新内容，再次单击按钮，应用程序就会执行相同的任务，但注意，这次新内容不是添加到原有内容的后面，而是完全覆盖以前的内容。事实上，WriteAllText()、WriteAllBytes()和 WriteAllLines()方法都会覆盖以前的文件，所以在使用这些方法时要小心。

上面示例中的 WriteAllText()方法使用如下签名：

```
File.WriteAllText(FilePath, Contents)
```

还可以指定新文件的编码格式：

```
File.WriteAllText(FilePath, Contents, Encoding)
```

WriteAllBytes()方法可以使用字节数组把内容写入文件，WriteAllLines()方法可以把字符串数组写入文件，如下面的事件处理程序所示：

```
private void button1_Click(object sender, EventArgs e)
{
    string[] movies =
        {"Grease",
        "Close Encounters of the Third Kind",
        "The Day After Tomorrow"};

    File.WriteAllLines(@"C:\Testing.txt", movies);
}
```

现在单击按钮，应用程序就会提供 Testing.txt 文件，其内容如下：

```
Grease
Close Encounters of the Third Kind
The Day After Tomorrow
```

WriteAllLines()方法把字符串数组中的每个元素单独放在文件的一行上。

因为数据不仅可以写入磁盘，还可以放在其他地方(如命名管道或内存)，所以必须理解如何在.NET 中使用流处理文件输入输出，作为一种移动文件内容的方式，如下一节所述。

24.4.3　流

流的概念已经存在很长时间了。流是一个用于传输数据的对象，数据可以向两个方向传输：

- 如果数据从外部源传输到程序中，这就是读取流。
- 如果数据从程序传输到外部源中，这就是写入流。

外部源常常是一个文件，但也不完全都是文件。它还可能是：

- 使用一些网络协议读写网络上的数据，其目的是选择数据，或从另一个计算机上发送数据。

- 读写到命名管道上。
- 把数据读写到一个内存区域上。

在这些示例中，Microsoft 提供了一个.NET 基类 System.IO.MemoryStream 对象来读写内存，而 System.Net.Sockets.NetworkStream 对象处理网络数据。读写管道没有基本的流类，但有一个泛型流类 System.IO.Stream，如果要编写一个这样的类，可以从这个基类继承。流对外部数据源不做任何假定。

外部源甚至可以是代码中的一个变量。这听起来很荒谬，但使用流在变量之间传输数据的技术是一个非常有用的技巧，可以在数据类型之间转换数据。C 语言使用它在整型和字符串之间转换数据类型，或者使用函数 sprintf()格式化字符串。

使用一个独立的对象来传输数据，比使用 FileInfo 或 DirectoryInfo 类更好，因为把传输数据的概念与特定数据源分离开来，可以更容易交换数据源。流对象本身包含许多通用代码，可以在外部数据源和代码中的变量之间移动数据，把这些代码与特定数据源的概念分离开来，就更容易实现不同环境下代码的重用(通过继承)。例如，前面提到的 StringReader 和 StringWriter 类，与本章后面用于读写文本文件的两个类 StreamReader 和 StreamWriter 一样，都是同一继承树的一部分，这些类几乎一定在后台共享许多代码。在 System.IO 名称空间中，与流相关的类的层次结构如图 24-8 所示。

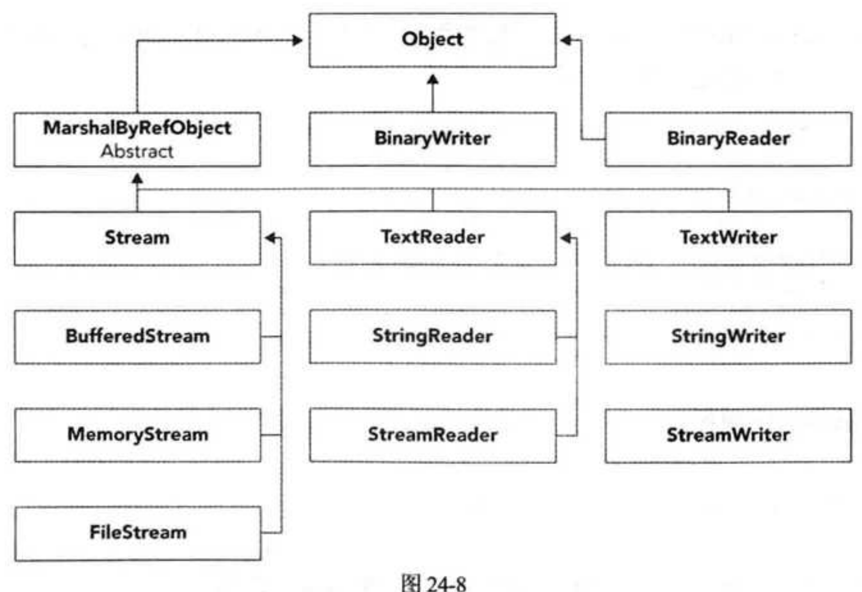


图 24-8

对于文件的读写，最常用的类如下：

- FileStream(文件流)——这个类主要用于在二进制文件中读写二进制数据——也可以使用它读写任何文件。
- StreamReader(流读取器)和 StreamWriter(流写入器) ——这两个类专门用于读写文本文件。

虽然示例没有使用另外两个类 BinaryReader 和 BinaryWriter，但它们也很有用。BinaryReader 和 BinaryWriter 这两个类本身并不实现流，但它们能够提供其他流对象的包装器。BinaryReader 和 BinaryWriter 还可以对二进制数据进行额外的格式化，直接从相关的流中读写 C#变量的内容。最简单的方式是把 BinaryReader 和 BinaryWriter 放在流和代码之间，进行额外的格式化，如图 24-9 所示。

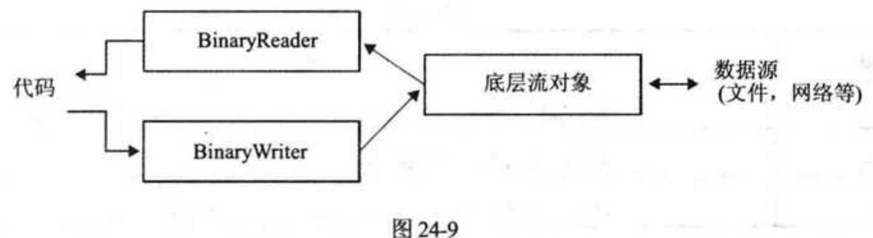


图 24-9

使用这些类和直接使用底层的流对象之间的区别是，基本流是按照字节来工作的。例如，在保存某个文档时，需要把类型为 long 的变量的内容写入一个二进制文件中，每个 long 型变量都占用 8 个字节，如果使用一般的二进制流，就必须显式地写入内存的 8 个字节中。

在 C#代码中，必须执行一些按位操作，从 long 值中提取这 8 个字节。使用 BinaryWriter 实例，可以把整个操作封装在 BinaryWriter.Write()方法的一个重载方法中，该方法的参数是 long 型，它把 8 个字节写入流中(如果流指向一个文件，就写入该文件)。对应的 BinaryReader.Read()方法则从流中提取 8 个字节，恢复 long 的值。BinaryReader 和 BinaryWriter 类的更多信息详见 SDK 文档。

24.4.4　缓存的流

从性能原因上看，在读写文件时，输出结果会被缓存。如果程序要求读取文件流中下面的两个字节，该流会把请求传递给 Windows，则 Windows 不会连接文件系统，再定位文件，并从磁盘中读取文件，仅读取 2 个字节。而是在一次读取过程中，检索文件中的一个大块，把该块保存在一个内存区域，即缓冲区上。以后对流中数据的请求就会从该缓冲区中读取，直到读取完该缓冲区为止。此时，Windows 会从文件中再获取另一个数据块。

写入文件的方式与此相同。对于文件，操作系统会自动完成读写操作，但需要编写一个流类，从其他没有缓存的设备中读取数据。如果是这样，就应从 BufferedStream 派生一个类，它实现一个缓冲区(但 BufferedStream 并不用于应用程序频繁切换读数据和写数据的情形)。

24.4.5　使用 FileStream 类读写二进制文件

读写二进制文件通常要使用 FileStream 类。

1. FileStream 类

FileStream 实例用于读写文件中的数据。要构造 FileStream 实例，需要以下 4 条信息：

- 要访问的文件。
- 表示如何打开文件的模式。例如，新建一个文件或打开一个现有的文件。如果打开一个现有的文件，写入操作是覆盖文件原来的内容，还是追加到文件的末尾？
- 表示访问文件的方式——是只读、只写，还是读写？
- 共享访问——表示是否独占访问文件。如果允许其他流同时访问文件，则这些流是只读、只写，还是读写文件？

第一条信息通常用一个包含文件的完整路径名的字符串来表示，本章只考虑需要该字符串的那些构造函数。除了这些构造函数外，一些其他的构造函数用老式的 Windows-API 风格的 Windows 句柄来处理文件。其余 3 条信息分别由 3 个.NET 枚举 FileMode、FileAccess 和 FileShare 来表示，这些枚举的值很容易理解，如表 24-4 所示。

表 24-4

枚 举	值
FileMode	Append、Create、CreateNew、Open、OpenOrCreate 和 Truncate
FileAccess	Read、ReadWrite 和 Write
FileShare	Delete、Inheritable、None、Read、ReadWrite 和 Write

注意，对于 FileMode，如果要求的模式与文件的现有状态不一致，就会抛出一个异常。如果文件不存在，Append、Open 和 Truncate 就会抛出一个异常；如果文件存在，CreateNew 就会抛出一个异常。Create 和 OpenOrCreate 可以处理这两种情况，但 Create 会删除任何现有的文件，新建一个空文件。因为 FileAccess 和 FileShare 枚举是按位标志，所以这些值可以与 C#的按位 OR 运算符“|”合并使用。

FileStream 类有许多构造函数，其中 3 个最简单的构造函数如下所示。

```
// creates file with read-write access and allows other streams read access
FileStream fs = new FileStream(@"C:\C# Projects\Project.doc",
                               FileMode.Create);
// as above, but we only get write access to the file
FileStream fs2 = new FileStream(@"C:\C# Projects\Project2.doc",
                                FileMode.Create, FileAccess.Write);
// as above but other streams don't get access to the file while
// fs3 is open
FileStream fs3 = new FileStream(@"C:\C# Projects\Project3.doc",
                                FileMode.Create, FileAccess.Write, FileShare.None);
```

从这段代码中可以看出，这些构造函数的重载版本会根据 FileMode 的值，把 FileAccess.ReadWrite 和 FileShare.Read 的默认值作为第 3 个和第 4 个参数，也可以以多种方式从 FileInfo 实例中创建一个文件流：

```
FileInfo myFile4 = new FileInfo(@"C:\C# Projects\Project4.doc");
FileStream fs4 = myFile4.OpenRead();
FileInfo myFile5= new FileInfo(@"C:\C# Projects\Project5doc");
FileStream fs5 = myFile5.OpenWrite();
FileInfo myFile6= new FileInfo(@"C:\C# Projects\Project6doc");
FileStream fs6 = myFile6.Open(FileMode.Append, FileAccess.Write,
                FileShare.None);
FileInfo myFile7 = new FileInfo(@"C:\C# Projects\Project7.doc");
FileStream fs7 = myFile7.Create();
```

FileInfo.OpenRead()方法提供的流只能读取现有文件，而 FileInfo.OpenWrite()方法可以进行读写访问。FileInfo.Open()方法允许显式地指定模式、访问方式和文件共享参数。

当然，使用完一个流后，就应关闭它：

```
fs.Close();
```

关闭流会释放与它相关联的资源，允许其他应用程序为同一个文件设置流。这个操作也会刷新缓冲区。在打开和关闭流之间，可以读写其中的数据。FileStream 类实现了许多方法以进行这样的读写。

ReadByte()是读取数据的最简单的方式，它从流中读取一个字节，把结果转换为一个 0~255 之间的整数。如果到达该流的末尾，它就返回-1：

```
int NextByte = fs.ReadByte();
```

如果要一次读取多个字节，就可以调用 Read()方法，它可以把特定数量的字节读入一个数组中。Read()方法返回实际读取的字节数——如果这个值是 0，就表示到达了流的末尾。下面的示例读入一个字节数组 ByteArray：

```
int nBytesRead = fs.Read(ByteArray, 0, nBytes);
```

Read 方法的第二个参数是一个偏移值，使用它可以要求 Read 操作的数据从数组的某个元素开始填充，而不是从第一个元素开始。第 3 个参数是要读入数组的字节数。

如果要给文件写入数据，就可以使用两个并行方法 WriteByte 和 Write。WriteByte 方法把一个字节写入流中：

```
byte NextByte = 100;
fs.WriteByte(NextByte);
```

而 Write 方法写入一个字节数组。例如，如果用一些值初始化前面的 ByteArray 数组，就可以使用下面的代码输出数组的前 nBytes 个字节：

```
fs.Write(ByteArray, 0, nBytes);
```

与 Read()方法一样，第二个参数可以从数组的某个元素开始写入，而不是从第一个元素开始。WriteByte()方法和 Write()方法都没有返回值。

除了这些方法以外，FileStream 还实现了其他关于记账任务的方法和属性，如确定流中有多少字节，锁定流或刷新缓冲区等。其他方法通常不是基本读写数据所必需的，但如果需要它们，就可以参阅 SDK 文档。

2. BinaryFileReader 示例

下面编写一个示例 BinaryFileReader 说明 FileStream 类的用法。这个示例可以读取和显示任何文件。在 Visual Studio 2013 中为创建一个 Windows 应用程序项目，添加一个菜单项，它可以打开一个标准的 OpenFileDialog 对话框，要求用户指定要读取的文件，然后把该文件显示为二进制码。在读取二进制文件时，需要显示非打印字符。此时可以在多行文本框中逐个显示文件中的每个字节，每行显示 16 个字节。如果字节表示一个可打印的 ASCII 字符，就显示该字符；否则就以十六进制的格式显示该字节的值。在这两种情况下，显示的文本之间都用空格隔开，这样每个显示的字节都占用 4 列，于是显得它们的排列非常整齐。

图 24-10 是查看文本文件时 BinaryFileReader 应用程序的外观(因为 BinaryFileReader 可以查看任何文件，所以可以在文本文件和二进制文件中使用它)。对于本示例，应用程序读取一个基本的 ASP.NET 页面(.aspx)。

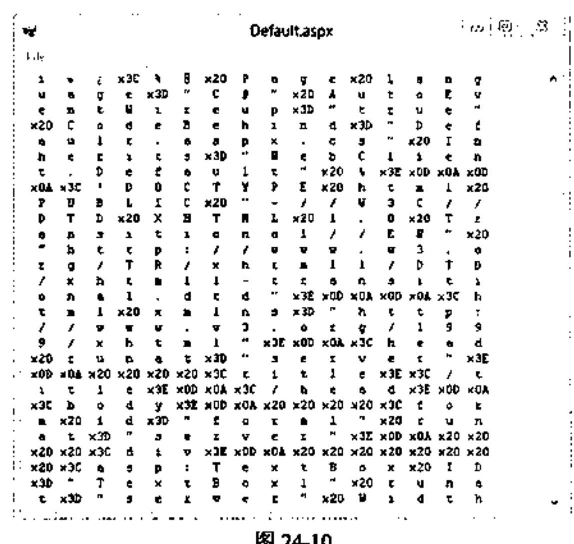


图 24-10

显然，这种格式更适合于查看单个字节的值，而不适合显示文本。本章后面会开发一个专门用于读取文本文件的示例——然后就可以查看文件中的内容。这个示例的优点是可以查看任何文件的内容。

这个示例没有说明如何写入文件。这是因为我们不希望把文本框中的内容(如图 24-10 所示)转换为二进制流，增加程序的复杂性。在后面开发可读写文本文件的示例中，将介绍如何写入文件，但只限于文本文件。

下面看看用于获得这些结果的代码。首先，需要使用 using 语句，引入 System.IO 名称空间：

```
using System.IO;
```

接着，给主窗体类添加两个字段，一个字段表示文件对话框，另一个字符串表示当前查看的文件的路径：

```
partial class Form1: Form

{
  private readonly OpenFileDialog chooseOpenFileDialog =
     new OpenFileDialog();
  private string chosenFile;
}
```

还需要添加一些标准的 Windows 窗体代码，来处理菜单和文件对话框：

```
public Form1()
{
  InitializeComponent();
  menuFileOpen.Click += OnFileOpen;
  chooseOpenFileDialog.FileOk += OnOpenFileDialogOK;
}

void OnFileOpen(object Sender, EventArgs e)
{
   chooseOpenFileDialog.ShowDialog();
}
```

```
void OnOpenFileDialogOK(object Sender, CancelEventArgs e)
{
  chosenFile = chooseOpenFileDialog.FileName;
  this.Text = Path.GetFileName(chosenFile);
  DisplayFile();
}
```

可以看出，用户单击 OK 按钮，在文件对话框中选择一个文件后，就会调用 DisplayFile()方法，该方法读取所选中的文件：

```
void DisplayFile()
{
  int nCols = 16;
  FileStream inStream = new FileStream(chosenFile, FileMode.Open,
                                               FileAccess.Read);
  long nBytesToRead = inStream.Length;
  if (nBytesToRead > 65536/4)
     nBytesToRead = 65536/4;

  int nLines = (int)(nBytesToRead/nCols) + 1;
  string [] lines = new string[nLines];
  int nBytesRead = 0;

  for (int i=0; i<nLines; i++)
  {
    StringBuilder nextLine = new StringBuilder();
    nextLine.Capacity = 4*nCols;

    for (int j = 0; j<nCols; j++)
    {
      int nextByte = inStream.ReadByte();
      nBytesRead++;
      if (nextByte < 0 || nBytesRead > 65536)
         break;
      char nextChar = (char)nextByte;
      if (nextChar < 16)
         nextLine.Append(" x0" + string.Format("{0,1:X}",
                                             (int)nextChar));
      else if
         (char.IsLetterOrDigit(nextChar) ||
                             char.IsPunctuation(nextChar))
         nextLine.Append(" " + nextChar + " ");
      else
         nextLine.Append(" x" + string.Format("{0,2:X}",
                                (int)nextChar));
    }
    lines[i] = nextLine.ToString();
  }
  inStream.Close();
  this.textBoxContents.Lines = lines;
}
```

在这个方法中进行了许多工作，首先为所选文件实例化一个 FileStream 对象，指定要打开一个

现有文件进行读取。然后确定要读取多少个字节，和应该显示多少行。这个字节数一般是文件中的字节数。文本框中的内容最多只能显示 65 536 个字符，但以我们选择的格式，文件中的每个字节会显示 4 个字符。

用户可能希望使用 System.Windows.Forms 名称空间中的 RichTextBox 类。RichTextBox 类似于文本框，但它有更高级的格式化功能，此处使用 TextBox，是为了让示例比较简单，用户可以只考虑读取文件的过程。

在该方法中，有相当多的部分是两个嵌套的 for 循环，它们构造要显示的每行文本。我们使用 StringBuilder 类来构造每一行文本，其性能方面的原因是：循环 16 次可以把每个字节的合适文本追加到表示每行文本的字符串上。如果一行上有一个新字符串，并复制构造该行时的一半字符，则不仅要花很多时间分配字符串，还会浪费堆上的许多内存。注意，可打印的字符是字母、数字或标点符号，这与相关的静态方法 System.Char 指定的一样。值小于 16 的字符都不是可打印的字符，然而，这意味着回车符(13)和换行符(10)都是二进制字符(如果这些字符单独占一行，多行文本框就不能正确显示它们)。

另外，使用 Properties 窗口，把文本框的 Font 属性改为固定宽度的字体——我们选择 Courier New 9pt regular 字体，并把文本框设置为有水平和垂直滚动条。最后，关闭流，把文本框的内容设置为已构建的字符串数组。

24.4.6 读写文本文件

理论上，可以使用 FileStream 类读取和显示文本文件。前面已经介绍了这个类。虽然上面示例中显示 Default.aspx 文件的格式不太容易理解，但这并不是 FileStream 类的内在问题——而在于我们在文本框中显示结果所使用的方式。

如果知道某个特殊文件包含文本，通常就可以使用 StreamReader 和 StreamWriter 类更方便地读写它们，而不是使用 FileStream 类。这是因为这些类工作的级别比较高，特别适合于读写文本。它们实现的方法可以根据流的内容，自动检测出停止读取文本较方便的位置。特别是：

- 这些类实现的方法(StreamReader.ReadLine()和 StreamWriter.WriteLine())可以一次读写一行文本。在读取文件时，流会自动确定下一个回车符的位置，并在该处停止读取。在写入文件时，流会自动把回车符和换行符追加到文本的末尾。
- 使用 StreamReader 和 StreamWriter 类，就不需要担心文件中使用的编码方式(文本格式)。可能的编码方式是 ASCII(一个字节表示一个字符)或者基于 Unicode 的任何格式，Unicode、UTF7、UTF8 和 UTF32。Windows 9x 系统上的文本文件总是 ASCII 格式，因为 Windows 9x 系统不支持 Unicode，但因为 Windows NT、2000、XP、2003、Vista、Windows Server 2008 、Windows 7 和 Windows 8 都支持 Unicode，所以文本文件除了包含 ASCII 数据之外，理论上可以包含 Unicode、UTF7、UTF8 或 UTF32 数据。其约定是：如果文件是 ASCII 格式，它就只包含文本。如果文件是 Unicode 格式，这就用文件的前两个或三个字节来表示，这几个字节可以设置为表示文件中格式的值的特定组合。

这些字节称为字节码标记。在使用标准 Windows 应用程序打开一个文件时，如 Notepad 或

WordPad，不需要考虑这个问题，因为这些应用程序都能识别不同的编码方法，会自动正确地读取文件。StreamReader 类也是这样，它可以正确地读取任何格式的文件，而 StreamWriter 类可以使用任何一种编码技术格式化它要输出的文本。但如果要使用 FileStream 类读取和显示文本文件，就必须自己处理这个过程。

1. StreamReader 类

StreamReader 实例用于读取文本文件。用某些方式构造 StreamReader 实例要比构造 FileStream 实例更简单，因为使用 StreamReader 时不需要 FileStream 的一些选项。特别是模式和访问类型与 StreamReader 类不相关，因为 StreamReader 只能执行读取操作。除此以外，没有指定共享许可的直接选项，但 StreamReader 有两个新选项：

- 需要指定不同的编码方法所执行的不同操作。可以构造一个 StreamReader 检查文件开头的字节码标记，确定编码方法，或者告诉 StreamReader 假定该文件使用某种指定的编码方法。
- 不提供要读取的文件名，而为另一个流提供一个引用。

最后一个选项需要解释一下，它说明了把读写数据的模型建立在流概念上的另一个优点。因为 StreamReader 工作在相对较高的级别上，如果有另一个流在读取其他源中的数据，就要使用 StreamReader 提供的工具来处理这个流，就好像这个流包含文本，所以此时 StreamReader 类非常有用。可以把这个流的输出传递给 StreamReader，这样，StreamReader 就可以用于读取和处理任何数据源(不仅仅是文件)中的数据。前面在讨论 BinaryReader 类时也讨论了这种情况。但在本书中，只使用 StreamReader 来直接连接文件。

因此，StreamReader 类有非常多的构造函数。而且，还有一个返回 StreamReader 引用的 FileInfo 方法 OpenText()。下面仅说明其中一些构造函数。

最简单的构造函数只带一个文件名参数。StreamReader 会检查字节码标记，以确定编码方法：

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt");
```

另外，如果指定 UTF8 编码方法，就应假定：

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt",
                                   Encoding.UTF8);
```

使用 System.Text.Encoding 类的几个属性之一，就可以指定编码方法。这个类是一个抽象基类，可以从这个类派生许多类，这个类实现了实际上完成文本编码的方法。每个属性都返回相应类的一个实例，可以使用的属性包括：

- ASCII
- Unicode
- UTF7
- UTF8
- UTF32
- BigEndianUnicode

下面的示例解释了如何把 StreamReader 关联到 FileStream 上。其优点是可以指定是否创建文件和共享许可，如果直接把 StreamReader 附加到文件中，就不能这么做：

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
                        FileMode.Open, FileAccess.Read, FileShare.None);
StreamReader sr = new StreamReader(fs);
```

对于本例，指定 StreamReader 查找字节码标记，以确定使用了什么编码方法，以后的示例也是这样，从一个 FileInfo 实例中获得 StreamReader：

```
FileInfo myFile = new FileInfo(@"C:\My Documents\ReadMe.txt");
StreamReader sr = myFile.OpenText();
```

与 FileStream 一样，应在使用后关闭 StreamReader。如果没有这样做，就会致使文件一直锁定，因此不能执行其他进程(除非使用 FileStream 构造 StreamReader，并指定 FileShare. ShareReadWrite)：

```
sr.Close();
```

实例化 StreamReader 后，就可以用该实例做一些工作了。与 FileStream 一样，我们仅指出可以用于读取数据的许多方式，在 SDK 文档中可以查阅到其他不太常用的 StreamReader 类的方法。

最容易使用的方法是 ReadLine()，该方法一次读取一行，但返回的字符串中不包括标记该行结束的回车换行符：

```
string nextLine = sr.ReadLine();
```

另外，还可以在一个字符串中提取文件的所有剩余内容(严格地说，是流的全部剩余内容)：

```
string restOfStream = sr.ReadToEnd();
```

可以只读取一个字符：

```
int nextChar = sr.Read();
```

Read()方法的重载版本可以把返回的字符强制转换为一个整数。如果到达流的末尾，它就返回 - 1。最后，可以用一个偏移值，把给定个数的字符读到数组中：

```
// to read 100 characters in.

int nChars = 100;
char [] charArray = new char[nChars];
int nCharsRead = sr.Read(charArray, 0, nChars);
```

如果要求读取的字符数多于文件中剩余的字符数，nCharsRead 就应小于 nChars。

2. StreamWriter 类

StreamWriter 类的工作方式与 StreamReader 类似，但 StreamWriter 只能用于写入文件(或另一个流)。构造 StreamWriter 的方法包括：

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt");
```

上面的代码使用了 UTF8 编码方法，.NET 把这种编码方法设置为默认的编码方法。还可以指定其他的编码方法：

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt", true,
   Encoding.ASCII);
```

在这个构造函数中，第二个参数是布尔型，表示文件是否应以追加方式打开。奇怪的是，构造函数的参数不能仅是一个文件名和一个编码类。

当然，可以把 StreamWriter 关联到一个文件流上，以获得打开文件的更多控制选项：

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
   FileMode.CreateNew, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
```

FileInfo 不执行返回 StreamWriter 类的任何方法。

另外，如果要新建一个文件，并开始向它写入数据，就可以使用下面的代码：

```
FileInfo myFile = new FileInfo(@"C:\My Documents\NewFile.txt");
StreamWriter sw = myFile.CreateText();
```

与其他流类一样，在使用完后，要关闭 StreamWriter 类：

```
sw.Close();
```

写入流可以使用 StreamWriter.Write()方法的 17 个重载版本来完成。最简单的方式是输出一个字符串：

```
string nextLine = "Groovy Line";
sw.Write(nextLine);
```

也可以输出单个字符：

```
char nextChar = 'a';
sw.Write(nextChar);
```

也可以输出一个字符数组：

```
char [] charArray = new char[100];

// initialize these characters

sw.Write(charArray);
```

甚至可以输出字符数组的一部分：

```
int nCharsToWrite = 50;
int startAtLocation = 25;
char [] charArray = new char[100];

// initialize these characters

sw.Write(charArray, startAtLocation, nCharsToWrite);
```

3. ReadWriteText 示例

ReadWriteText 示例说明了 StreamReader 和 StreamWriter 类的用法。它非常类似于前面的 ReadBinaryFile 示例，但假定要读取的文件是一个文本文件，并显示其内容。它还可以保存文件(包括在文本框中对文本进行的修改)。它将以 Unicode 格式保存任何文件。

图 24-11 所示的 ReadWriteText 用于显示前面的同一个 Default.aspx 文件。但这次读取内容会更

容易一些。

Default.aspx

File

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebClient.Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:TextBox ID="TextBox1" runat="server" Width="225px"></asp:TextBox>
    </div>
        <div>
        <asp:TextBox ID="TextBox2" runat="server" Width="225px"></asp:TextBox>
    </div>
    <div>
        <asp:Button ID="Button1" runat="server" Text="Button" onclick="Button1_Click" />
    </div>
    </form>
</body>
</html>
```

图 24-11

这里不打算介绍给 Open File 对话框添加事件处理程序的详细内容，因为它们基本上与前面的 BinaryFileReader 示例相同。与这个示例相同，打开一个新文件，将调用 DisplayFile()方法。其唯一的区别是 DisplayFile 的实现方式，以及本例有一个保存文件的选项。这由另一个菜单项 Save 来表示，这个选项的处理程序调用添加到代码中的另一个方法 SaveFile()(注意，这个新文件总是覆盖原始文件——这个示例没有写入另一个文件的选项)。

首先看看 SaveFile()方法，因为它是最简单的函数。首先利用 StreamReader.WriteLine()方法把文本框中的每行文本依次写入 StreamWriter 流，以在每行文本的最后加上回车换行符：

```
void SaveFile()
{
  StreamWriter sw = new StreamWriter(chosenFile, false, Encoding.Unicode);

  foreach (string line in textBoxContents.Lines)
     sw.WriteLine(line);

  sw.Close();
}
```

chosenFile 是主窗体的一个字符串字段，它包含已经读取的文件的名称(与前面的示例一样)。注意在打开流时指定 Unicode 编码方式。如果要以其他格式写入文件，则只需要改变该参数的值。如果要把文本追加到文件中，这个构造函数的第二个参数就设置为 true，但本例不是这样。在构造时必须为 StreamWriter 设置编码方式，它随后可以用作只读属性 Encoding。

下面介绍文件的读取方式。读取过程比较复杂，因为我们不知道要读取的文件中包含多少行文本(例如，不知道文件中包含多少个(char)13 (char)10 序列，因为 char(13) char(10)是行尾的回车换行符)。解决这个问题的方式是，先把文件读入 StringCollection 类的一个实例中，该类在 System.Collections.Specialized 名称空间中，主要用于保存可动态扩展的一组字符串。它实现的两个方法是

我们感兴趣的：Add()方法把字符串添加到集合中，和 CopyTo()方法把字符串集合复制到一个正常数组(一个 System.Array 实例)中。StringCollection 对象的每个元素包含文件中的一行文本。

DisplayFile()方法调用另一个方法 ReadFileIntoStringCollection()，后一个方法实际上读取文件。之后，就知道文件中有多少行文本，因此把 StringCollection 复制到大小固定的正常数组中，并把数组中的内容填充到文本框中。在进行复制时，因为只复制了字符串的引用，没有复制字符串本身，所以该过程的执行效率很高：

```
void DisplayFile()
{
  StringCollection linesCollection = ReadFileIntoStringCollection();
  string [] linesArray = new string[linesCollection.Count];
  linesCollection.CopyTo(linesArray, 0);
  this.textBoxContents.Lines = linesArray;
}
```

StringCollection.CopyTo()方法的第二个参数表示目标数组中的下标，我们从该下标指定的位置开始复制集合。

下面看看 ReadFileIntoStringCollection()方法。使用 StreamReader 读取每一行文本。编译时需要计算读取的字符数，以确保不超出文本框的容量：

```
StringCollection ReadFileIntoStringCollection()
{
  const int MaxBytes = 65536;
  StreamReader sr = new StreamReader(chosenFile);
  StringCollection result = new StringCollection();
  int nBytesRead = 0;
  string nextLine;
  while ( sr.Peek != 0 )
  {
    nextLine = sr.ReadLine()
    nBytesRead += nextLine.Length;
    if (nBytesRead > MaxBytes)
       break;
    result.Add(nextLine);
  }
  sr.Close();
  return result;
}
```

这就是该示例的完整代码。

如果运行 ReadWriteText，读取 Default.aspx 文件，然后保存它，该文件的格式就是 Unicode。任何常用的 Windows 应用程序(Notepad，Wordpad)都不能分辨这种格式，记事本、写字板，甚至 ReadWriteText 示例也只能在 Windows 的大多数版本下正确地读取和显示文件，尽管因为 Windows 9x 不支持 Unicode，像 Notepad 这样的应用程序不能识别这些平台上的 Unicode 文件(如果从 Wrox 网站 www.wrox.com 上下载了这个示例，就可以试试)，但是，如果使用前面的 BinaryFileReader 示例显示文件，就会立即看出它们的区别，如图 24-12 所示。最前面的两个字节表示 Unicode 格式的文件是可见的，之后，每个字符都用两个字节来表示。这非常明显，因为在这个文件中，每个字符的高位字节都是 0，所以每隔一个字节就显示 x00。

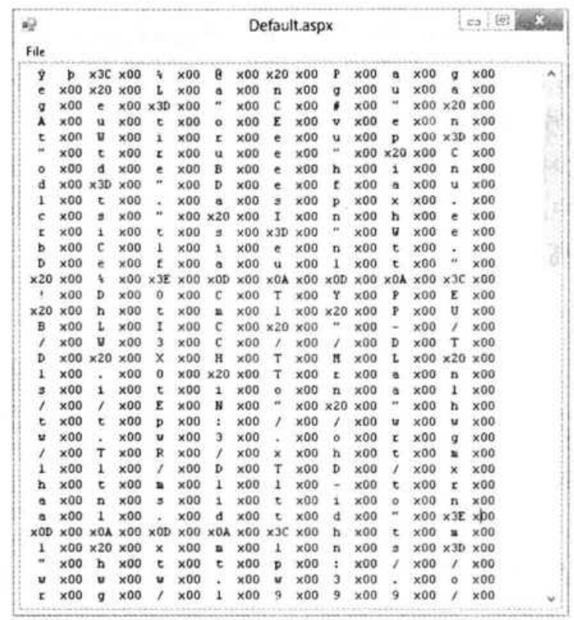


图 24-12

24.5 映射内存的文件

如果用户仅使用托管代码来编程，映射内存的文件就可能是一个全新的概念。.NET Framework 4.5 引入了 System.IO.MemoryMappedFiles 名称空间，把映射内存的文件包含为构建应用程序的工具集的一部分。

对底层的 Windows API 执行一些平台调用操作时，总是会使用映射内存的文件这个概念。但现在引入了 System.IO.MemoryMappedFiles 名称空间后，就可以使用托管代码，而不是操作繁琐的平台调用。

应用程序需要频繁地或随机地访问文件时，最好使用映射内存的文件和这个名称空间。使用这种方式允许把文件的一部分或者全部加载到一段虚拟内存上，这些文件内容会显示给应用程序，就好像这个文件包含在应用程序的主内存中一样。

有趣的是，可以把内存中的这个文件用作多个进程的共享资源。在此之前，用户可能使用 WCF 或命名管道与多个进程之间的共享资源通信，但现在可以在使用共享名称的进程之间共享映射内存的文件。

为了使用映射内存的文件，必须使用两个对象。第一个是映射内存的文件实例，它用于加载文件。第二个是访问器对象。下面的代码写入映射内存的文件对象，再读取它。在释放对象时也会执行写入操作：

```
using System;
using System.IO.MemoryMappedFiles;
using System.Text;

namespace MappedMemoryFiles
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            using (var mmFile = MemoryMappedFile.CreateFromFile(@"c:\users\bill\
                      documents\visual studio 11\
                      Projects\MappedMemoryFiles\MappedMemoryFiles\TextFile1.txt",
                   System.IO.FileMode.Create, "fileHandle", 1024 * 1024))
            {
                string valueToWrite = "Written to the mapped-memory file on " +
                    DateTime.Now.ToString();
                var myAccessor = mmFile.CreateViewAccessor();

                myAccessor.WriteArray<byte>(0,
                   Encoding.ASCII.GetBytes(valueToWrite), 0,
                   valueToWrite.Length);

                var readOut = new byte[valueToWrite.Length];
                myAccessor.ReadArray<byte>(0, readOut, 0, readOut.Length);
                var finalValue = Encoding.ASCII.GetString(readOut);

                Console.WriteLine("Message: " + finalValue);
                Console.ReadLine();
            }
        }
    }
}
```

在这个例子中，使用 CreateFromFile()方法从物理文件中创建一个映射内存的文件。除了映射内存的文件之外，还需要为这个映射创建一个访问器对象，如下面的代码所示：

```
var myAccessor = mmFile.CreateViewAccessor();
```

有了访问器之后，就可以读写这个映射内存的位置，如代码示例所示。

也可以给同一个映射内存的位置创建多个访问器，如下面的代码所示：

```
var myAccessor1 = mmFile.CreateViewAccessor();
var myAccessor2 = mmFile.CreateViewAcces
```

24.6　读取驱动器信息

除了处理文件和目录之外，.NET Framework 还可以从指定的驱动器中读取信息。这使用 DriveInfo 类实现。DriveInfo 类可以扫描系统，提供可用驱动器的列表，还可以进一步提供任何驱动器的大量细节。

为了举例说明 DriveInfo 类的用法，创建一个简单的 Windows 窗体，列出计算机上的所有可用驱动器，再提供用户选择的驱动器的详细信息。Windows 窗体包含一个简单的 ListBox，如图 24-13 所示。

图 24-13

设置好窗体后，其代码包含两个事件，一个在窗体加载时引发，另一个在最终用户从列表框中选择驱动器时引发。这个窗体的代码如下所示：

```
using System;
using System.IO;
using System.Windows.Forms;

namespace DriveViewer
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            DriveInfo[] di = DriveInfo.GetDrives();

            foreach (DriveInfo itemDrive in di)
            {
                 listBox1.Items.Add(itemDrive.Name);
            }
        }

        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            DriveInfo di = new DriveInfo(listBox1.SelectedItem.ToString());

            MessageBox.Show("Available Free Space: "
                + di.AvailableFreeSpace + "\n" +
                "Drive Format: " + di.DriveFormat + "\n" +
                "Drive Type: " + di.DriveType + "\n" +
                "Is Ready: " + di.IsReady + "\n" +
                "Name: " + di.Name + "\n" +
                "Root Directory: " + di.RootDirectory + "\n" +
                "ToString() Value: " + di + "\n" +
                "Total Free Space: " + di.TotalFreeSpace + "\n" +
                "Total Size: " + di.TotalSize + "\n" +
                "Volume Label: " + di.VolumeLabel, di.Name +
                " DRIVE INFO");
        }
    }
}
```

首先，用 using 关键字导入 System.IO 名称空间。在 Form1_Load 事件中使用 DriveInfo 类获取系统上所有可用驱动器的列表。这需要使用 DriveInfo 对象数组，再用 DriveInfo.GetDrives()方法填充这个数组。接着使用 foreach 循环遍历找到的每个驱动器，用循环的结果填充列表框。这会生成如图 24-14 所示的结果。

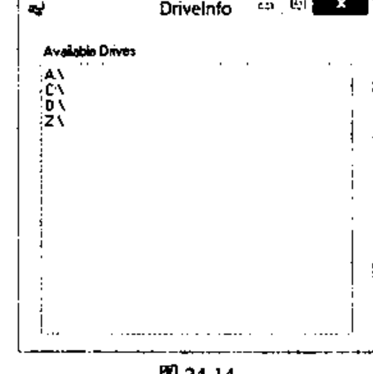


图 24-14

这个窗体允许最终用户在列表框中选择一个驱动器。选择好驱动器后，就显示一个消息框，其中包含所选驱动器的信息。如图 24-14 所示，在当前的计算机上有 4 个驱动器。选择其中几个驱动器会生成如图 24-15 所示的消息框。

图 24-15

在这里可以看出，这些消息框列出了 3 个完全不同的驱动器的详细信息。第一个驱动器 C:\是硬盘，因为消息框显示它的驱动器类型是 Fixed。第二个驱动器 D:\是 CD/DVD 驱动器，第三个驱动器 F:\是 USB 笔，其驱动器类型是 Removable。

24.7　文件的安全性

在第一次引入.NET Framework 1.0/1.1 时，它不能方便地访问和使用文件、目录和注册表键的访问控制列表(ACL)。那时，要访问 ACL，通常要进行一些 COM 交互操作，所以需要使用 ACL 的高级编程技巧。

自.NET Framework 2.0 发布以来，这有了很大的变化，因为这个版本通过 System.Security.AccessControl 名称空间，使 ACL 的使用变得非常容易。利用这个名称空间，可以为文件、注册表键、网络共享、Active Directory 对象等处理安全设置。

24.7.1　从文件中读取 ACL

对于一个使用 System.Security.AccessControl 名称空间的示例，本节将为文件和目录使用 ACL。首先了解如何查看指定文件的 ACL。这个示例在一个控制台应用程序中完成，如下所示：

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ReadingACLs
{
    internal class Program
    {
        private static string myFilePath;

        private static void Main()
        {
            Console.Write("Provide full file path: ");
```

```
            myFilePath = Console.ReadLine();

            try
            {
                using (FileStream myFile =
                    new FileStream(myFilePath, FileMode.Open, FileAccess.Read))
                    {
                        FileSecurity fileSec = myFile.GetAccessControl();

                        foreach (FileSystemAccessRule fileRule in
                            fileSec.GetAccessRules(true, true,
                               typeof (NTAccount)))
                        {
                            Console.WriteLine("{0} {1} {2} access for {3}",
                               myFilePath,
                               fileRule.AccessControlType ==
                               AccessControlType.Allow
                                  ? "provides": "denies",
                                  fileRule.FileSystemRights,
                                  fileRule.IdentityReference);
                    }
                }
            }
            catch
            {
               Console.WriteLine("Incorrect file path given!");
            }

            Console.ReadLine();
        }
    }
}
```

为了使这个示例正常工作，首先引用 System.Security.AccessControl 名称空间，这样就可以在程序的后面访问 FileSecurity 和 FileSystemAccessRule 类。

在检索到指定的文件并把它放在 FileStream 对象中后，就使用 File 对象上的 GetAccessControl() 方法获取该文件的 ACL。GetAccessControl()方法中的这些信息放在 FileSecurity 类中。这个类有对引用项的访问权限。每个访问权限都用一个 FileSystemAccessRule 对象表示。所以要使用 foreach 循环遍历在 FileSecurity 对象中找到的所有访问权限。

对根目录的一个简单文本文件运行这个示例，结果如下所示：

```
Provide full file path: C:\Sample.txt
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides ReadAndExecute, Synchronize access for BUILTIN\Users
C:\Sample.txt provides Modify, Sychronize access for
     NT AUTHORITY\Authenticated Users
```

24.7.2 从目录中读取 ACL

读取目录(而不是文件)的 ACL 信息，与前面的示例没有什么区别，其代码如下所示：

```
using System;
```

```
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ConsoleApplication1
{
    internal class Program
    {
        private static string mentionedDir;

        private static void Main()
        {
            Console.Write("Provide full directory path: ");
            mentionedDir = Console.ReadLine();

            try
            {
                DirectoryInfo myDir = new DirectoryInfo(mentionedDir);

                if (myDir.Exists)
                {
                    DirectorySecurity myDirSec = myDir.GetAccessControl();

                    foreach (FileSystemAccessRule fileRule in
                         myDirSec.GetAccessRules(true, true,
                                                typeof (NTAccount)))
                    {
                        Console.WriteLine("{0} {1} {2} access for {3}",
                           mentionedDir, fileRule.AccessControlType ==
                              AccessControlType.Allow
                              ? "provides": "denies",
                              fileRule.FileSystemRights,
                              fileRule.IdentityReference);
                    }
                }
            }
            catch
            {
                Console.WriteLine("Incorrect directory provided!");
            }

            Console.ReadLine();
        }
    }
}
```

这个示例的不同之处是，它使用 DirectoryInfo 类，该类还包含一个 GetAccessControl()方法，来提取目录的 ACL 信息。在 Windows 8 下运行这个例子的结果如下所示：

```
Provide full directory path: C:\Test
C:\Test provides FullControl access for BUILTIN\Administrators
C:\Test provides 268435456 access for BUILTIN\Administrators
C:\Test provides FullControl access for NT AUTHORITY\SYSTEM
C:\Test provides 268435456 access for NT AUTHORITY\SYSTEM
C:\Test provides ReadAndExecute, Synchronize access for BUILTIN\Users
```

```
C:\Test provides Modify, Synchronize access for
        NT AUTHORITY\Authenticated Users
C:\Test provides -536805376 access for NT AUTHORITY\Authenticated Users
```

ACL的最后一个用法是使用System.Security.AccessControl名称空间添加和删除文件中的ACL项。

24.7.3 添加和删除文件中的 ACL 项

还可以使用与前面示例相同的对象处理资源的 ACL。下面的示例修改了前面读取文件的 ACL 信息的代码。在这个示例中，读取指定文件的 ACL，修改后再次读取它：

```
try
{
  using (FileStream myFile = new FileStream(myFilePath,
      FileMode.Open, FileAccess.ReadWrite))
  {
     FileSecurity fileSec = myFile.GetAccessControl();

     Console.WriteLine("ACL list before modification:");

     foreach (FileSystemAccessRule fileRule in
       fileSec.GetAccessRules(true, true,
         typeof(System.Security.Principal.NTAccount)))
     {
       Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
         fileRule.AccessControlType == AccessControlType.Allow ?
         "provides": "denies",
         fileRule.FileSystemRights,
         fileRule.IdentityReference);
     }

     Console.WriteLine();
     Console.WriteLine("ACL list after modification:");

     FileSystemAccessRule newRule = new FileSystemAccessRule(
       new System.Security.Principal.NTAccount(@"PUSHKIN\Tuija"),
       FileSystemRights.FullControl,
       AccessControlType.Allow);

     fileSec.AddAccessRule(newRule);
     File.SetAccessControl(myFilePath, fileSec);

     foreach (FileSystemAccessRule fileRule in
       fileSec.GetAccessRules(true, true,
       typeof(System.Security.Principal.NTAccount)))
     {
       Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
         fileRule.AccessControlType == AccessControlType.Allow ?
         "provides": "denies",
         fileRule.FileSystemRights,
         fileRule.IdentityReference);
     }
  }
}
```

在这个示例中，给文件的 ACL 添加了一条新的访问规则。这使用 FileSystemAccessRule 对象完成。FileSystemAccessRule 类是一个抽象的访问控制项(ACE)实例。ACE 定义了要使用的用户账户，这个用户账户可以处理的访问类型，以及是允许还是拒绝这个访问。在新建这个对象的实例时，新建一个 NTAccount 对象，并给文件赋予 Full Control 权限。即使新建了 NTAccount 对象，它仍必须引用已有用户。接着使用 FileSecurity 类的 AddAccessRule 方法赋予新规则。之后，使用 FileSecurity 对象引用，通过 File 类的 SetAccessControl 方法给当前文件设置访问控制。

接着，再次列出文件的 ACL。下面是上述代码的执行结果的一个示例：

```
Provide full file path: C:\Users\Bill\Sample.txt
ACL list before modification:
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for PUSHKIN\Bill

ACL list after modification:
C:\Sample.txt provides FullControl access for PUSHKIN\Tuija
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for PUSHKIN\Bill
```

要从ACL列表中删除规则，并不需要对代码进行很多修改。在上面的代码示例中，只需要把下面的一行

```
fileSec.AddAccessRule(newRule);
```

改为：

```
fileSec.RemoveAccessRule(newRule);
```

就可以删除刚才添加的规则。

24.8　读写注册表

自 Windows 95 以来的所有 Windows 版本中，注册表是包含 Windows 安装、用户首选项，以及已安装软件和设备的所有配置信息的核心存储库。目前，几乎所有的商用软件都使用注册表来存储这些信息，COM 组件必须把它们的信息存储在注册表中，才能由客户端调用。.NET Framework 及其无干扰安装概念略微减弱了注册表对于应用程序的重要性，因为程序集是完全自包含的，所以特定程序集的信息不需要放在注册表中，即使是共享程序集也是这样。另外，.NET Framework 引入了独立存储器的概念，通过它应用程序可以在文件中存储专用于每个用户的信息，.NET Framework 将确保为每个在机器上注册的用户单独地存储数据。

应用程序现在使用 Windows Installer 来安装，开发人员不再需要直接操作以前涉及安装应用程序的注册表。但是，如果发布任何完整的应用程序，则应用程序也可能要使用注册表来存储其配置信息。例如，如果应用程序要显示在控制面板的 Add/Remove Programs 对话框中，则仍需要使用相应的注册表项。还需要使用注册表处理与遗留代码的向后兼容性。

注册表的库和.NET 库一样复杂，它包括访问注册表的类。其中有两个类涉及注册表，即 Registry

和 RegistryKey，这两个类都在 Microsoft.Win32 名称空间中。在介绍这两个类前，先简要介绍一下注册表本身的结构。

24.8.1 注册表

注册表的层次结构非常类似于文件系统的层次结构。查看和修改注册表内容的一般方式是使用 regedit 或 regedt32 实用程序。当然，自从 Windows 95 以后，regedit 在所有的 Windows 版本中都有。而 regedt32 则在 Windows NT 和 Windows 2000 中才有，其用户友好性不如 regedit，但可以访问 regedit 不能查看的安全性信息。Windows Server 2003 把 regedit 和 regedt32 合并为一个新的编辑器 regedit。这里只讨论 Windows 7 中的 regedit，在 Run 对话框或命令行提示符中输入 regedit，即可启动它。

图 24-16 显示了第一次启动 regedit 的屏幕。

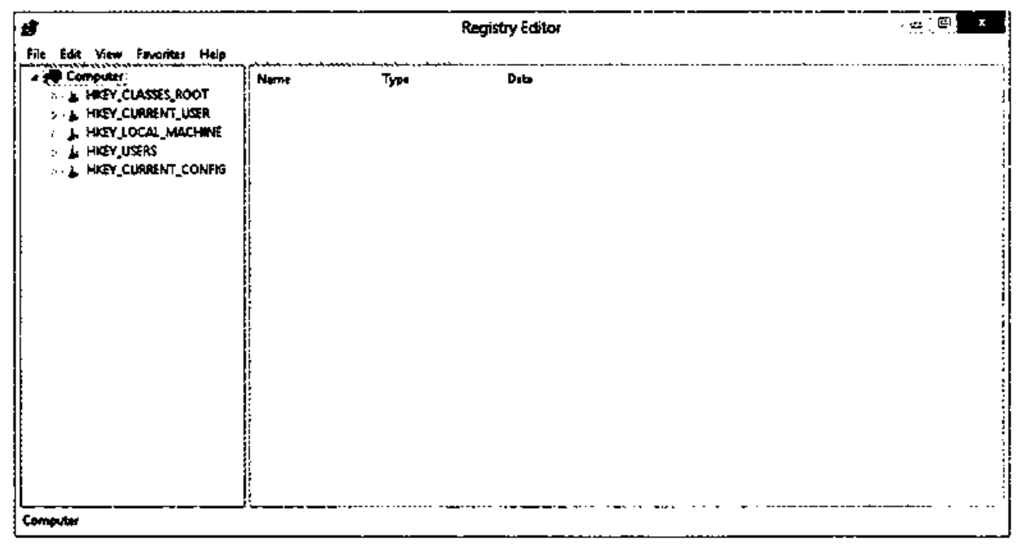


图 24-16

regedit 的树型视图/列表视图风格的用户界面非常类似于 Windows Explorer，与注册表本身的层次结构相匹配。但它们有一些重要区别。

在文件系统中，最上面的节点是磁盘的分区 C:\、D:\等。在注册表中，最上面的节点是注册表配置单元(registry hive)，已有的配置单元是不能改变的——它们是固定的，有 7 个注册表配置单元(但使用 regedit 只能看到其中的 5 个)：

- HKEY_CLASSES_ROOT(HKCR)包含系统上文件类型的细节(.txt、.doc 等)，以及使用哪些应用程序可以打开每种文件。它还包含所有 COM 组件的注册信息(后者通常是注册表中最大的一个区域，因为目前的 Windows 带有非常多的 COM 组件)。
- HKEY_CURRENT_USER(HKCU)包含用户目前登录的计算机的用户配置。这些设置包括桌面设置、环境变量、网络和打印机连接，以及其他定义用户的操作环境的设置。
- HKEY_LOCAL_MACHINE(HKLM)是一个很大的配置单元，其中包含所有安装到计算机上的软件和硬件信息，这些设置不是用户特有的，而是可用于所有登录到计算机上的用户。它还包含 HKCR 配置单元：HKCR 实际上并不是一个独立的配置单元，而只是一个对注册表键 HKLM/SOFTWARE/ Classes 的方便映射。

- HKEY_USERS(HKUSR)包含所有用户的用户首选项。它还包含 HKCU 配置单元，HKCU 配置单元是对 HKEY_USERS 中一个键的映射。
- HKEY_CURRENT_CONFIG(HKCF)包含计算机上硬件的详细信息。

其余的两个键包含临时信息，这些信息常常会更改：

- HKEY_DYN_DATA 是一个一般容器，包含需要存储在注册表中的任何易失性数据。
- HKEY_PERFORMANCE_DATA 包含与运行应用程序的性能相关的信息。

在这些配置单元中，具有注册表键的一个树型结构。每个键在许多方面都类似于文件系统中的文件夹或文件。但是，它们有一个重要区别：文件系统可以区分文件(文件中包含数据)和文件夹(其中主要包含其他文件夹或文件)，但注册表中只有键。键可以包含数据和其他键。

如果键中包含数据，这个键就表示为一系列值。每个值都有一个相关的名称、数据类型和数据，另外，键还可以有默认值，这个值没有名称。

使用 regedit 可以查看这个结构，了解其中的注册表键。图 24-17 显示了 HKCU/Control Panel/Appearance 键中的内容，其中包含当前登录用户所选的颜色主题的详细信息。regedit 在树型视图中用一个打开的文件夹图标来显示要查看的键。

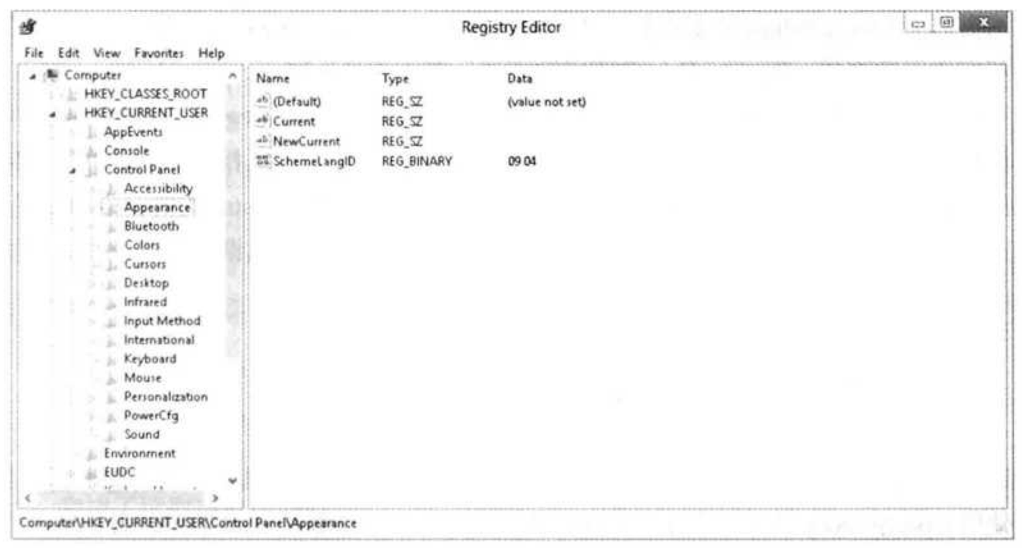


图 24-17

HKCU/Control Panel/Appearance 键有 3 个命名的值集，但其默认值不包含任何数据。在图 24-17 中，标记为 Type 的列显示了每个值的数据类型。注册表项可以格式化为这 3 个数据类型中的一个。这些类型分别是：

- REG_SZ(大致相当于.NET 字符串实例——这种匹配并不精确，因为注册表项的数据类型不是.NET 数据类型)
- REG_DWORD(大致相当于 uint)
- REG_BINARY(大致相当于字节数组)

为此，要在注册表中存储数据的应用程序会创建许多注册表键，通常把它们存储在 HKLM\Software\<CompanyName>键中。注意，这些键不一定包含数据。有时键的存在就可以给应用程序提供所需的足够信息。

24.8.2 .NET 注册表类

要访问注册表，可以使用 Microsoft.Win32 名称空间中的两个类 Registry 和 RegistryKey。RegistryKey 实例表示一个注册表键。这个类实现的方法可以浏览子键、创建新键、读取或修改键中的值。换言之，该类通常可以完成对注册表键进行的所有操作，包括设置键的安全级别。RegistryKey 是处理注册表用得最多的类。Registry 类只能对注册表键进行单一的访问，以执行简单的操作。Registry 类的另一个作用是提供表示顶级键的 RegistryKey 实例(不同的配置单元)，以便开始在注册表中定位。Registry 类通过静态属性来提供这些实例，这些属性共有 7 个，分别是 ClassesRoot、CurrentConfig、CurrentUser、DynData、LocalMachine、PerformanceData 和 Users。用户可以很快猜出它们分别与哪个配置单元相对应。

例如，要获得一个表示 HKLM 键的 RegistryKey 实例，可以编写下面的代码：

```
RegistryKey hklm = Registry.LocalMachine;
```

获得 RegistryKey 对象的引用的过程，称为打开对应键。

用户可能会认为，因为注册表的层次结构类似于文件系统的层次结构，所以 RegistryKey 类提供的方法类似于 DirectoryInfo 类实现的方法，但实际上并非如此。访问注册表的方式通常不同于使用文件和文件夹的方式，RegistryKey 类实现的方法反映了这种不同。

最明显的区别是如何在注册表的给定位置上打开一个注册表键。Registry 类没有用户可以使用的公共构造函数，也没有直接通过键的名称来访问键的方法。但可以从相关的配置单元中从上至下浏览该键。如果要实例化一个 RegistryKey 对象，唯一的方式就是从 Registry 类的对应静态属性开始，向下浏览。例如，如果要读取 HKLM/Software/Microsoft 键中的一些数据，就可以使用下面的代码获得它的一个引用：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft");
```

以这种方式访问注册表键是只读访问。如果要写入该键(包括写入其值，或创建和删除其子键)，就需要使用 OpenSubKey 的另一个重写方法，该方法的第二个参数是 bool 类型，表示是否要对该键进行读写访问。例如，如果要修改 Microsoft 键(并假定用户是一个系统管理员，有修改该键的许可)，就应编写如下代码：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft", true);
```

因为这个键包含 Microsoft 的应用程序使用的信息，所以在大多数情况下不应修改这个特殊的键。

如果希望这个键存在，就应调用 OpenSubKey()方法。如果这个键不存在，它就返回一个空引用。如果要创建一个键，就应使用 CreateSubKey()方法(该方法会通过返回的引用，自动提供该键的读写访问)：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMine = hkSoftware.CreateSubKey("MyOwnSoftware");
```

CreateSubKey()方法的工作方式非常有趣：如果键不存在，它就创建这个键。但如果键已经存在，它就会返回一个表示该键的 RegistryKey 实例。这个方法采用这样的工作方式，其原因与用户使用这个键的通常方式有关。注册表整体上包含长期数据，如 Windows 和各种应用程序的配置信息。因此用户并不需要经常显式地创建键。

更常见的是，应用程序需要确保某些数据存在于注册表中。换言之，如果这些数据不存在，就要创建相关的键，但如果它们存在，就不需要做任何事。CreateSubKey()方法就可以完成这项任务。与 FileInfo.Open()方法的情况不同，例如，CreateSubKey()不会偶然地删除任何数据。如果要删除注册表键，就需要调用 RegistryKey.Delete()方法，因为注册表对于 Windows 非常重要。当调试 C# 注册表调用时，如果删除一些重要的键，就会不经意间完全中断 Windows。

定位了要读取或修改的注册表键后，就可以使用 SetValue()或 GetValue()方法设置或获取该键中的数据。这两个方法的参数都是一个字符串，其中字符串给出了参数值的名称，SetValue()方法还需要另一个包含值的详细信息的对象引用。因为这个参数定义为对象引用，所以它实际上可以是任何一个类的引用。SetValue()方法根据所提供的类的类型，确定把值设置为 REG_SZ、REG_DWORD，还是 REG_ BINARY。例如：

```
RegistryKey hkMine = HkSoftware.CreateSubKey("MyOwnSoftware");
hkMine.SetValue("MyStringValue", "Hello World");
hkMine.SetValue("MyIntValue", 20);
```

这段代码用两个值设置该键：MyStringValue 的类型是 REG_SZ，而 MyIntValue 的类型是 REG_DWORD，这里只考虑这两种类型，在后面的示例中会使用它们。

RegistryKey.GetValue()方法的工作方式也是这样。它返回一个对象引用，如果该方法检测到值的类型为 REG_SZ，它实际上就返回一个字符串引用；如果值的类型为 REG_DWORD，它就返回一个 int 型值。

```
string stringValue = (string)hkMine.GetValue("MyStringValue");
int intValue = (int)hkMine.GetValue("MyIntValue");
```

最后，完成了读取或修改数据后，应关闭该键：

```
hkMine.Close();
```

RegistryKey 类实现了许多方法和属性。表 24-5 和表 24-6 列出了其中最有用的方法和属性。

表 24-5

属　性	作　用
Name	键的名称(只读)
SubKeyCount	键的子键个数
ValueCount	键包含的值的个数

表 24-6

方　法	作　用
Close()	关闭键
CreateSubKey()	创建给定名称的子键(如果该子键已经存在，就打开它)

(续表)

方　法	作　用
DeleteSubKey()	删除指定的子键
DeleteSubKeyTree()	递归删除子键及其所有子键
DeleteValue()	从键中删除一个指定的值
GetAccessControl()	返回指定注册表键的 ACL，该方法是.NET Framework 2.0 中增加的
GetSubKeyNames()	返回包含子键名称的字符串数组
GetValue()	返回指定的值
GetValueKind()	指定注册表数据类型，返回指定的值，该方法是.NET Framework 2.0 中增加的
GetValueNames()	返回一个包含所有键值名称的字符串数组
OpenSubKey()	返回表示给定子键的 RegistryKey 实例的引用
SetAccessControl()	把 ACL 应用于指定的注册表键
SetValue()	设置指定的值

24.9　读写独立存储器

除了读写注册表之外，还可以读写独立存储器中的值。如果在写入注册表或磁盘时一般有问题，就可以使用独立存储器。它可以用于轻松地存储应用程序状态或用户设置。

独立存储器可以看作一个虚拟磁盘，在其中可以保存只能由创建它们的应用程序或与其他应用程序实例共享的数据项。独立存储器的访问类型有两种。第一种是由用户和程序集访问。

在用户和程序集访问独立存储器时，计算机上有一个存储器位置，它可以由多个应用程序实例访问。这种访问通过用户身份和应用程序(或程序集)身份来保证。这说明，同一个应用程序的多个实例可以在同一个存储器上工作。

独立存储器的第二种访问类型是由用户、程序集和域来访问。在这种访问类型中，每个应用程序实例都在它自己的独立存储器中工作。在这种情况下，每个应用程序实例都在它自己的存储器中工作，每个应用程序实例记录的设置都只与它自己相关。这是独立存储器的一种更精细的方法。

为了举例说明如何在 Windows 窗体应用程序中使用独立存储器(也可以在 ASP.NET 应用程序中使用它)，可以使用 ReadSettings()和 SaveSettings()方法从独立存储器中读写值，而不是直接在注册表中读写值。

这里只列出了 ReadSettings()和 SaveSettings()方法中的代码，该应用程序还有更多的代码，参见下载代码文件中示例 SelfPlacingWindow 的其他代码。

首先，需要重写 SaveSettings()方法。为了使代码正常工作，需要添加下面的 using 指令：

```
using System.IO;
using System.IO.IsolatedStorage;
using System.Text;
```

SaveSettings()方法详见下面的代码示例：

```
void SaveSettings()
{
  IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
  IsolatedStorageFileStream storStream = new
     IsolatedStorageFileStream("SelfPlacingWindow.xml",

     FileMode.Create, FileAccess.Write);

  System.Xml.XmlTextWriter writer = new
     System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
  writer.Formatting = System.Xml.Formatting.Indented;

  writer.WriteStartDocument();
  writer.WriteStartElement("Settings");

  writer.WriteStartElement("BackColor");
  writer.WriteValue(BackColor.ToKnownColor().ToString());
  writer.WriteEndElement();

  writer.WriteStartElement("Red");
  writer.WriteValue(BackColor.R);
  writer.WriteEndElement();

  writer.WriteStartElement("Green");
  writer.WriteValue(BackColor.G);
  writer.WriteEndElement();

  writer.WriteStartElement("Blue");
  writer.WriteValue(BackColor.B);
  writer.WriteEndElement();

  writer.WriteStartElement("Width");
  writer.WriteValue(Width);
  writer.WriteEndElement();

  writer.WriteStartElement("Height");
  writer.WriteValue(Height);
  writer.WriteEndElement();

  writer.WriteStartElement("X");
  writer.WriteValue(DesktopLocation.X);
  writer.WriteEndElement();

  writer.WriteStartElement("Y");
  writer.WriteValue(DesktopLocation.Y);
  writer.WriteEndElement();

  writer.WriteStartElement("WindowState");
  writer.WriteValue(WindowState.ToString());
  writer.WriteEndElement();

  writer.WriteEndElement();

  writer.Flush();
  writer.Close();
```

```
    storStream.Close();
    storFile.Close();
  }
```

这些代码比使用注册表时略多一些，这主要是因为需要构建放在独立存储器中的 XML 文档。在这段代码中，第一个重要的地方是：

```
IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
IsolatedStorageFileStream storStream = new
  IsolatedStorageFileStream("SelfPlacingWindow.xml",
  FileMode.Create, FileAccess.Write);
```

这段代码使用访问的用户、程序集和域类型创建了 IsolatedStorageFile 的一个实例，使用 IsolatedStorageFileStream 对象创建了一个流，该对象创建虚拟文件 SelfPlacingWindow.xml。

之后创建一个 XmlTextWriter 对象，以构建 XML 文档，将 XML 内容写入 IsolatedStorage-FileStream 对象实例中。

```
System.Xml.XmlTextWriter writer = new
   System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
```

创建 XmlTextWriter 对象之后，将所有的值逐个节点地写入 XML 文档中。当把所有内容写入 XML 文档中时，关闭所有对象，现在所有数据就都存储在独立存储器中。

从存储器中读取数据通过 ReadSettings()方法实现。这个方法如下面的代码示例所示：

```
bool ReadSettings()
{
  IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
  string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

  foreach (string userFile in userFiles)
  {
    if(userFile == "SelfPlacingWindow.xml")
    {
      listBoxMessages.Items.Add("Successfully opened file " +
                                userFile.ToString());

      StreamReader storStream =
        new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
        FileMode.Open, storFile));
      System.Xml.XmlTextReader reader = new
       System.Xml.XmlTextReader(storStream);

      int redComponent = 0;
      int greenComponent = 0;
      int blueComponent = 0;

      int X = 0;
      int Y = 0;

      while (reader.Read())
      {
        switch (reader.Name)
        {
```

```
          case "Red":
            redComponent = int.Parse(reader.ReadString());
            break;
          case "Green":
            greenComponent = int.Parse(reader.ReadString());
            break;
          case "Blue":
            blueComponent = int.Parse(reader.ReadString());
            break;
          case "X":
            X = int.Parse(reader.ReadString());
            break;
          case "Y":
            Y = int.Parse(reader.ReadString());
            break;
          case "Width":
            this.Width = int.Parse(reader.ReadString());
            break;
          case "Height":
            this.Height = int.Parse(reader.ReadString());
            break;
          case "WindowState":
            this.WindowState = (FormWindowState)FormWindowState.Parse
                (WindowState.GetType(), reader.ReadString());
            break;
          default:
            break;
        }
      }

      this.BackColor =
         Color.FromArgb(redComponent, greenComponent, blueComponent);
      this.DesktopLocation = new Point(X, Y);

      listBoxMessages.Items.Add("Background color: " + BackColor.Name);
      listBoxMessages.Items.Add("Desktop location: " +
         DesktopLocation.ToString());
      listBoxMessages.Items.Add("Size: " + new Size(Width, Height).ToString());
      listBoxMessages.Items.Add("Window State: " + WindowState.ToString());

      storStream.Close();
      storFile.Close();
    }
  }
  return true;
}
```

使用 GetFileNames()方法，SelfPlacingWindow.xml 文档会从独立存储器中弹出，然后放在一个流中，再使用 XmlTextReader 对象分析它。

```
IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

foreach (string userFile in userFiles)
{
```

```
if(userFile == "SelfPlacingWindow.xml")
{
  listBoxMessages.Items.Add("Successfully opened file " +
                            userFile.ToString());

  StreamReader storStream =
    new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
    FileMode.Open, storFile));
```

XML 文档包含在 IsolatedStorageFileStream 对象中后，就使用 XmlTextReader 对象分析它：

```
  System.Xml.XmlTextReader reader = new
     System.Xml.XmlTextReader(storStream);
```

之后，通过XmlTextReader对象从流中读取它。然后元素值会放回到应用程序中。SelfPlacing-Window示例使用注册表记录和检索应用程序状态值，而使用独立存储器与使用注册表同样有效。应用程序会与以前一样记录颜色、大小和位置。

24.10 小结

本章介绍了如何在 C#代码中使用.NET 基类来访问注册表和文件系统。在这两种情况下，基类提供的对象模型比较简单，但功能强大，从而很容易执行这些领域中几乎所有的操作。对于文件系统，可以复制文件；移动、创建、删除文件和文件夹；读写二进制文件和文本文件。而对于注册表，可以创建、修改或读取键。

本章还介绍了独立存储器以及如何在应用程序中使用它们存储应用程序状态。

本章假定用户在有足够访问权限的账户上运行代码，以完成代码需要执行的任务。显然，安全性对于文件访问非常重要，详见第 22 章。

第25章

事务处理

本章要点

- 事务处理阶段和 ACID 属性
- 传统的事务
- 可提交的事务
- 事务的升级
- 依赖的事务
- 环境事务
- 事务的孤立级别
- 自定义资源管理器
- Windows 文件系统事务

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 事务示例
- 多线程环境事务
- 定制资源
- Windows 8 事务

25.1 简介

事务的主要特征是，任务要么全部完成，要么都不完成。在写入一些记录时，要么写入所有记录，要么什么都不写入。在写入一个记录时即使出现了一次失败，在事务中已写入的所有其他数据也会回滚。

事务常用于数据库，但利用 System.Transactions 名称空间中的类，还可以对不稳定的、基于内

存的对象执行事务处理，如对象列表。对于支持事务的对象列表，如果添加或删除了一个对象时事务处理失败，这个列表的操作会自动撤销。写入一个基于内存的列表与写入数据库一样，也可以在事务中完成。

自从 Windows Vista 之后，文件系统和注册表也支持事务。在注册表中写入一个文件，并做出一些修改的操作可以通过事务来完成。

25.2 概述

为了理解事务，考虑一下在 Web 站点上订购图书。图书订购进程会把客户要购买的图书从仓库中取出，并把它放在客户的订购框中，再从客户的信用卡收取购买图书的费用。这两个动作要么都成功完成，要么都不完成。如果从仓库中取出图书时出现错误，就不应从信用卡中收取费用。这个工作可以用事务来完成。

事务最常见的用途是写入或更新数据库中的数据。在消息队列中写入消息，或将数据写入文件或注册表时，也可以使用事务完成。一个事务可以包含多个操作。

用于消息排队的类和体系结构，以及 System.Messaging 名称空间参见第 47 章。

图 25-1 显示了事务中的主要元素。事务由事务管理器来管理和协调。每个影响事务结果的资源都由一个资源管理器来管理。事务管理器与资源管理器通信，以定义事务的结果。

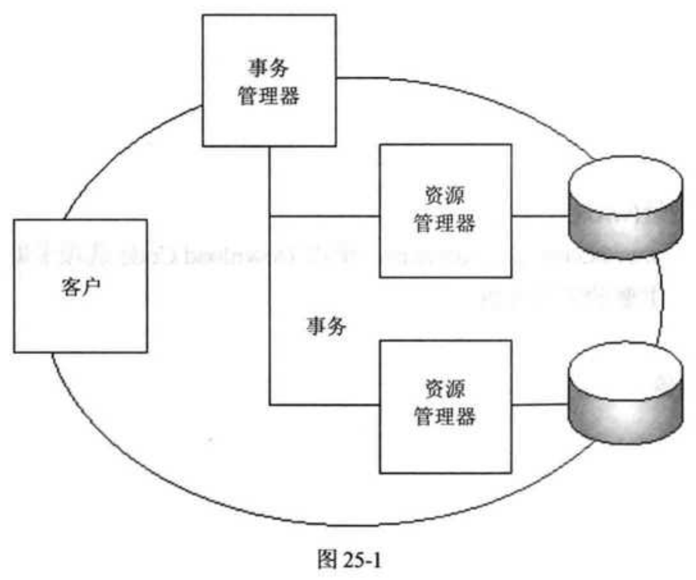


图 25-1

25.2.1 事务处理阶段

事务处理分为激活、准备和提交 3 个阶段。

- **激活阶段**：在这个阶段创建事务。为资源管理事务处理的资源管理器可以用事务进行登记。
- **准备阶段**：在这个阶段，每个资源管理器都可以定义事务的结果。事务的创建者发出结束事务的指令时，就启动这个阶段。事务管理器给所有的资源管理器发出一条“准备”消息。如果

资源管理器可以成功生成事务的结果，就向事务管理器发出一条“已准备好”消息。如果资源管理器未能准备好，就可以终止事务处理，发出一条“回滚”消息，强制事务管理器执行回滚操作。在发出“已准备好”消息后，资源管理器必须保证在提交阶段能成功完成工作。为此，稳定的资源管理器必须将准备状态的信息写入一个日志中，这样，如果在准备和提交过程中出现停电等故障时，就可以从该状态继续执行。

- **提交阶段**：当所有的资源管理器都成功准备好了，就开始这个阶段。即所有资源管理器都发出了“已准备好”消息。接着，事务管理器就可以给所有的参与者发送一条“提交”消息，以完成工作。资源管理器现在可以完成事务中的工作，并返回一条“已提交”消息。

25.2.2 ACID 属性

事务有一些特殊的要求，例如，事务的结果必须处于有效的状态。即使服务器断电了，也需要有有效状态。事务的特征可以用术语 ACID 来定义，ACID 是 Atomicity、Consistency、Isolation 和 Durability 的首字母缩写。

- Atomicity(原子性)：表示一个工作单元。在事务中，要么整个工作单元都成功完成，要么都不完成。
- Consistency(一致性)：事务开始前的状态和事务完成后的状态必须有效。在执行事务的过程中，状态可以有临时值。
- Isolation(隔离性)：表示并发进行的事务独立于状态，而状态在事务处理过程中可能发生变化。在事务未完成时，事务 A 看不到事务 B 中的临时状态。
- Durability(持久性)：在事务完成后，它必须以可持久的方式存储起来。如果关闭电源或服务器崩溃，该状态在重新启动时必须恢复。

并不是每个事务都需要这 4 个属性。例如，基于内存的事务(如将一项写入列表中)就不需要持久性。事务也不总是需要与外界隔离，如后面的事务隔离级别所述。

事务和有效状态很容易用婚礼来解释。新婚夫妇站在事务协调员面前，事务协调员询问一位新人：“你愿意与你身边的男人结婚吗？”如果第一位新人同意，就询问第二位新人：“你愿意与这个女人结婚吗？”如果第二位新人反对，第一位新人就接收到回滚消息。这个事务的有效状态是，要么两个人都结婚，要么两个人都不结婚。如果两个人都同意结婚，事务就会提交，这两个人就都处于已结婚的状态。如果其中一个人反对，事务就会终止，两个人都处于未结婚的状态。无效的状态是：一个人已结婚，而另一个没有结婚。事务确保结果永远不处于无效状态。

25.3 数据库和实体类

本章的事务使用样本数据库 CourseManagement，它由图 25-2 所示的结构定义。Courses 表包含课程信息：课程编号和名称；CourseDates 表包含指定课程的日期，它链接到表 Courses 上。Students 表包含选修某门课程的人。CourseAttendees 表是 Students 表和 CourseDates 表之间的链接，它定义

了哪些学生选修了什么课程。

可以从 Wrox 网站上找到本章的数据库和源代码。

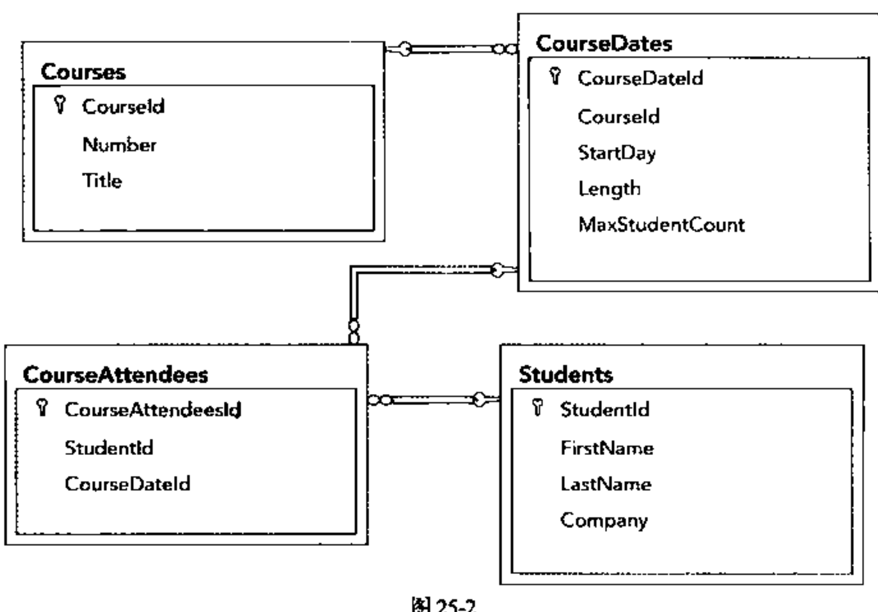


图 25-2

在本章的示例应用程序中，使用了一个包含实体类和数据访问类的库。这个 Student 类包含的属性定义了一个学生，如 Firstname、Lastname 和 Company(代码文件 DataLib/Student.cs)：

```
using System;

namespace Wrox.ProCSharp.Transactions
{
    [Serializable]
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Company { get; set; }
        public int Id { get; set; }

        public override string ToString()
        {
            return String.Format("{0} {1}", FirstName, LastName);
        }
    }
}
```

将学生信息添加到数据库中是在 StudentData 类的 AddStudent()方法中完成的。这里创建一个 ADO.NET 连接，来连接 SQL Server 数据库，用 SqlCommand 对象定义 SQL 语句，并调用 ExecuteNonQuery()方法来执行该命令(代码文件 DataLib/StudentData.cs)：

```
using System.Data.SqlClient;
```

```
using System.Threading.Tasks;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
  public class StudentData
  {
    public async Task AddStudentAsync(Student student)
    {
      var connection = new SqlConnection(
          Properties.Settings.Default.CourseManagementConnectionString);
      await connection.OpenAsync();
      try
      {
        SqlCommand command = connection.CreateCommand();

        command.CommandText = "INSERT INTO Students " +
            "(FirstName, LastName, Company) VALUES " +
            "(@FirstName, @LastName, @Company)";
        command.Parameters.AddWithValue("@FirstName", student.FirstName);
        command.Parameters.AddWithValue("@LastName", student.LastName);
        command.Parameters.AddWithValue("@Company", student.Company);

        await command.ExecuteNonQueryAsync();
      }
      finally
      {
        connection.Close();
      }
    }
  }
}
```

ADO.NET 详见第 32 章。

25.4 传统的事务

在发布 System.Transaction 名称空间之前，可以直接用 ADO.NET 创建事务，也可以通过组件、特性和 COM+运行库(位于 System.EnterpriseServices 名称空间中)进行事务处理。由于 COM+一般不再用于新应用程序，所以本书不介绍它。

25.4.1 ADO.NET 事务

首先看看传统的 ADO.NET 事务。如果没有手动创建事务，每条 SQL 语句就都有一个事务。如果多条语句应参与到同一个事务处理中，就必须手动创建一个事务。

下面的代码段说明了如何使用 ADO.NET 事务。SqlConnection 类定义了 BeginTransaction()方法，它返回一个 SqlTransaction 类型的对象。这个事务对象必须与要参与事务处理的每条命令关联起来。要把命令关联到事务处理上，可将 SqlCommand 类的 Transaction 属性设置为 SqlTransaction 实例。

为了使事务成功完成，必须调用 SqlTransaction 对象的 Commit()方法。如果有错误，就必须调用 Rollback()方法，并撤销每个修改。使用 try/catch 有助于检查错误，并在 catch 块中执行回滚(代码文件 DataLib/CourseData.cs)。

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Transactions
{
  public class CourseData
  {
    public async Task AddCourseAsync(Course course)
    {
      var connection = new SqlConnection(
          Properties.Settings.Default.CourseManagementConnectionString);
      SqlCommand courseCommand = connection.CreateCommand();
      courseCommand.CommandText =
          "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
      await connection.OpenAsync();
      SqlTransaction tx = connection.BeginTransaction();

      try
      {
        courseCommand.Transaction = tx;

        courseCommand.Parameters.AddWithValue("@Number", course.Number);
        courseCommand.Parameters.AddWithValue("@Title", course.Title);
        await courseCommand.ExecuteNonQueryAsync();

        tx.Commit();
      }
      catch (Exception ex)
      {
        Trace.WriteLine("Error: " + ex.Message);
        tx.Rollback();
        throw;
      }
      finally
      {
        connection.Close();
      }
    }
  }
}
```

如果有多条命令要运行在一个事务中，每条命令都必须与该事务关联起来。因为事务还与一个连接关联起来，所以这些命令也必须关联到同一个连接实例上。ADO.NET 事务不支持跨多个连接的事务；它总是关联到一个连接上的本地事务。

如果使用多个对象创建了一个对象持久性模型，例如，Course 和 CourseDate 类应在一个事务处理中持续使用，就很难使用 ADO.NET 事务来实现。这需要把事务传递给参与同一个事务的所有对象。

ADO.NET 事务不是分布式事务。在 ADO.NET 事务中，很难使多个对象参与同一个事务。

25.4.2 System.EnterpriseServices

利用 Enterprise Services，可以免费获得许多服务，其中之一是自动事务处理。Enterprise Services 目前主要由新技术替代，例如 System.Transactions、WCF 和 Windows 应用程序服务器。Enterprise Services 的事务特性影响了 System.Transactions 的功能，因此这里简要介绍 Enterprise Services。

通过 System.EnterpriseServices 使用事务的优点是，不需要显式地进行事务处理，运行库会自动创建事务，只需要给有事务处理要求的类添加[Transaction]特性即可。[AutoComplete]特性把方法标记为自动设置事务的状态位：如果该方法成功，就设置成功位，因此可以提交事务。如果发生异常，就终止事务。

```
using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
  [Transaction(TransactionOption.Required)]
  public class CourseData: ServicedComponent
  {
    [AutoComplete]
    public void AddCourse(Course course)
    {
      var connection = new SqlConnection(
         Properties.Settings.Default.CourseManagementConnectionString);
      SqlCommand courseCommand = connection.CreateCommand();
      courseCommand.CommandText =
         "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
      connection.Open();
      try
      {
        courseCommand.Parameters.AddWithValue("@Number", course.Number);
        courseCommand.Parameters.AddWithValue("@Title", course.Title);
        courseCommand.ExecuteNonQuery();
      }
      finally
      {
        connection.Close();
      }
    }
  }
}
```

用 System.EnterpriseServices 创建事务的一大优点是，多个对象能轻松地运行在同一个事务中，事务还可以自动登记。缺点是它需要 COM+主机模型，使用这个技术的类必须派生自基类 ServicedComponent。

25.5 System.Transactions

自.NET 2.0 以来增加了 System.Transactions 名称空间，为.NET 应用程序带来了一个新的事务编程模型。

这个名称空间提供了几个依赖的 TransactionXXX 类。Transaction 是所有事务处理类的基类，并定义了所有事务类都可以使用的属性、方法和事件。CommittableTransaction 是唯一一个支持提交的事务类。这个类有一个 Commit()方法，所有其他事务类都只能执行回滚。DependentTransaction 类用于依赖于其他事务的事务。依赖的事务可以依赖从可提交的事务中创建的事务。不管事务处理是否成功，都把依赖的事务添加到可提交的事务的结果中。SubordinateTransaction 类和分布式事务协调器(DTC)一起使用。这个类表示非根事务，但可以由 DTC 管理。

表 25-1 列出了 Transaction 类的属性和方法。

表 25-1

Transaction 类的成员	说　明
Current	Current 属性是一个静态属性，不需要有实例。Transaction.Current 返回一个环境事务处理(如果存在)。环境事务处理在后面讨论
IsolationLevel	IsolationLevel 属性返回一个 IsolationLevel 类型的对象。IsolationLevel 是一个枚举，它定义了其他事务必须有什么访问权限才能访问事务的临时结果。它会影响 ACID 中的“I”；并不是所有的事务处理都是隔离的
TransactionInformation	TransactionInformation 属性返回一个 TransactionInformation 对象。该对象提供了事务的当前状态信息、事务的创建时间和事务标识符
EnlistVolatile() EnlistDurable() EnlistPromotableSinglePhase()	使用登记方法 EnlistVolatile()、EnlistDurable()和 EnlistPromotableSinglePhase()，可以登记参与事务处理的自定义资源管理器
Rollback	使用 Rollback()方法，可以终止一个事务，撤销所有的改变，把所有的结果设置为事务处理之前的状态
DependentClone()	使用 DependentClone()方法，可以创建一个依赖当前事务的事务
TransactionCompleted	TransactionCompleted 是一个事件，在事务完成时触发——事务可能成功，也可能失败。在 TransactionCompletedEventHandler 类型的事件处理程序中，可以访问 Transaction 对象，并读取其状态

为了说明 System.Transaction 名称空间的特性，下面在一个独立程序集中的 Utilities 类提供了一些静态方法。AbortTx()方法根据用户的输入返回 true 或 false。DisplayTransactionInformation()方法将一个 TransactionInformation 对象作为参数，显示事务中的所有信息：创建时间、状态、本地和分布式标识符(代码文件 Utilities/Utilities.cs)。

```
public static class Utilities
{
  public static bool AbortTx()
  {
```

```
        Console.Write("Abort the Transaction (y/n)?");
        return Console.ReadLine().ToLower().Equals("y");
      }

      public static void DisplayTransactionInformation(string title,
          TransactionInformation ti)
      {
        Contract.Requires<ArgumentNullException>(ti != null);
        Console.WriteLine(title);
        Console.WriteLine("Creation Time: {0:T}", ti.CreationTime);
        Console.WriteLine("Status: {0}", ti.Status);
        Console.WriteLine("Local ID: {0}", ti.LocalIdentifier);
        Console.WriteLine("Distributed ID: {0}", ti.DistributedIdentifier);
        Console.WriteLine();
      }
    }
```

25.5.1 可提交的事务

Transaction 类不能以编程方式提交，它没有提交事务的方法。基类 Transaction 只支持事务处理的终止。唯一支持提交的事务类是 CommittableTransaction 类。

在 ADO.NET 中，事务可以用连接方式登记。为此，在 StudentData 类中添加 AddStudent()方法，该方法将一个 System.Transactions.Transaction 对象作为第二个参数。调用 SqlConnection 类的 EnlistTransaction()方法，以便用连接登记 tx 对象。这样，ADO.NET 连接就关联到事务上(代码文件 DataLib/StudentData.cs)。

```
public async Task AddStudentAsync(Student student, Transaction tx)
{
  Contract.Requires<ArgumentNullException>(student != null);

  var connection = new SqlConnection(
     Properties.Settings.Default.CourseManagementConnectionString);
  await connection.OpenAsync();
  try
  {
    if (tx != null)
     connection.EnlistTransaction(tx);
    SqlCommand command = connection.CreateCommand();

    command.CommandText = "INSERT INTO Students (FirstName, " +
        "LastName, Company)" +
        "VALUES (@FirstName, @LastName, @Company)";
    command.Parameters.AddWithValue("@FirstName", student.FirstName);
    command.Parameters.AddWithValue("@LastName", student.LastName);
    command.Parameters.AddWithValue("@Company", student.Company);

    await command.ExecuteNonQueryAsync();
  }
  finally
  {
    connection.Close();
  }
}
```

在控制台应用程序 CommittableSamples 的 CommittableTransaction()方法中，先创建一个 CommittableTransaction 类型的事务。之后，将信息显示在控制台上。接着，创建一个 Student 对象，把这个对象从 AddStudent()方法写入数据库中。如果在事务的外部验证数据库中的记录，就看不到刚才添加的学生，除非事务已完成。如果事务失败，就执行回滚，不把学生写入数据库中。

在调用 AddStudent()方法之后，就调用帮助方法 Utilities.AbortTx()，确定事务处理是否要终止。如果用户要终止，就抛出一个 ApplicationException 类型的异常，在 catch 块中，调用 Transaction 类的 Rollback()方法，回滚事务处理。不把记录写入数据库中。如果用户不终止，Commit()方法就提交事务，并提交事务的最终状态(代码文件 TransactionSamples/Program.cs)。

```
static async Task CommittableTransactionAsync()
{
  var tx = new CommittableTransaction();
  Utilities.DisplayTransactionInformation("TX created",
     tx.TransactionInformation);

  try
  {
    var s1 = new Student
    {
      FirstName = "Stephanie",
      LastName = "Nagel",
      Company = "CN innovation"
    };
    var db = new StudentData();
    await db.AddStudentAsync(s1, tx);

    if (Utilities.AbortTx())
    {
      throw new ApplicationException("transaction abort");
    }

    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine();
    tx.Rollback();
  }

  Utilities.DisplayTransactionInformation("TX completed",
    tx.TransactionInformation);

}
```

这里，在应用程序的输出中，事务是激活的，且有一个本地标识符。另外，用户选择终止事务。在完成事务后，会看到终止状态。

```
TX created
Creation Time: 7:30:49 PM
Status: Active
Local ID: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
```

```
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? y
Transaction abort

TX completed
Creation Time: 7:30:49 PM
Status: Aborted
Local ID: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

从应用程序第2次的输出结果可以看出，用户没有终止事务。事务的状态是已提交，数据写入数据库中。

```
TX Created
Creation Time: 7:33:04 PM
Status: Active
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

TX completed
Creation Time: 7:33:04 PM
Status: Committed
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

25.5.2 事务处理的升级

System.Transactions 支持可升级的事务处理。根据参与事务处理的资源，创建本地事务或者分布式事务。SQL Server 自从 2005 版本开始支持可升级的事务，但目前我们只看到本地事务。在前面的例子中，分布式事务 ID 总是设置为 0，且只赋予了本地 ID。对于不支持可升级的事务的资源，会创建分布式事务。如果把多个资源添加到事务中，事务就可以先设置为本地事务，再根据需要升级为分布式事务。当多个 SQL Server 数据库连接添加到事务中时，就会进行这种升级。事务开始时是一个本地事务，之后升级为分布式事务。

现在修改控制台应用程序，使用同一个事务对象 tx 添加第二个学生。因为每个 AddStudent()方法都会打开一个新连接，所以在添加第二个学生后，把两个连接关联到该事务上(代码文件 TransactionSamples/Program.cs)。

```
static void TransactionPromotion()
{
  var tx = new CommittableTransaction();
  Utilities.DisplayTransactionInformation("TX created",
     tx.TransactionInformation);

  try
  {
    var s1 = new Student
    {
      FirstName = "Matthias",
      LastName = "Nagel",
      Company = "CN innovation"
    };
```

```
        var db = new StudentData();
        db.AddStudent(s1, tx);

        var s2 = new Student
        {
          FirstName = "Stephanie",
          LastName = "Nagel",
          Company = "CN innovation"
        };
        db.AddStudent(s2, tx);

        Utilities.DisplayTransactionInformation(
            "2nd connection enlisted", tx.TransactionInformation);

        if (Utilities.AbortTx())
        {
          throw new ApplicationException("transaction abort");
        }

        tx.Commit();
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        tx.Rollback();
      }

      Utilities.DisplayTransactionInformation("TX finished",
          tx.TransactionInformation);
    }
```

现在运行应用程序，就会看到，对于添加的第一个学生，其分布式标识符是 0，但对于添加的第二个学生，升级了事务，所以分布式标识符与该事务关联起来。

```
TX created
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 00000000-0000-0000-0000-0000000000

2nd connection enlisted
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 501abd91-e512-47f3-95d5-f0488743293d

Abort the Transaction (y/n)?
```

事务的升级需要启动分布式事务协调器(DTC)。如果在系统中升级事务时失败，就应验证 DTC 服务是否启动。启动 Component Services MMC 插件，就可以看到运行在系统上的所有 DTC 事务的实际状态。

在树型视图中选择 Transaction List 节点，就可以看到所有激活的事务。在图 25-3 中，有一个激活的事务，其分布式标识符与前面控制台的输出相同。如果验证系统上的输出，就应确保该事务设

置了超时时间，这样在超过指定时间后，事务就终止。在超过该时间后，在事务列表中就看不到该事务了。还可以用这个工具验证事务的统计信息。Transaction Statistics 显示了提交和终止的事务个数。

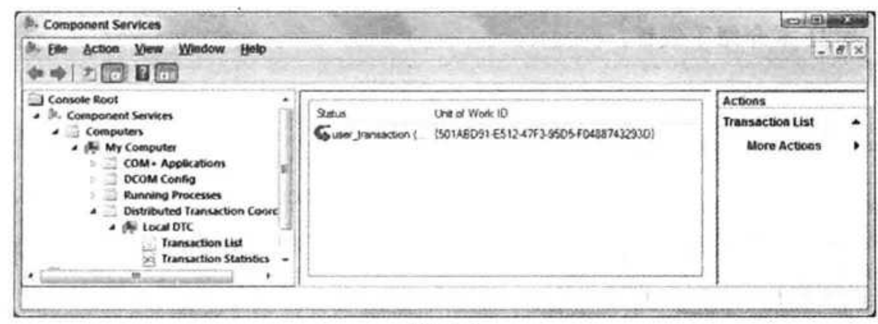


图 25-3

25.5.3 依赖事务

使用依赖事务，可以影响来自多个任务或线程的某个事务。依赖事务会依靠另一个事务，并影响事务的结果。

示例应用程序 DependentTransactions 为一个新任务创建了一个依赖的事务。TxTask()方法是新任务的一个方法，它将一个 DependentTransaction 对象作为参数传递。依赖事务的相关信息用帮助方法 DisplayTransactionInformation()来显示。在任务退出之前，调用依赖事务的 Complete()方法来定义事务的结果。依赖事务可以调用 Complete()或 Rollback()方法来定义事务的结果。Complete 方法设置成功位。如果根事务结束，且所有依赖事务都把成功位设置为 true，就提交事务。如果某个依赖事务调用 Rollback()方法来设置终止位，整个事务就会终止。

```
static void TxTask(object obj)
{
  var tx = obj as DependentTransaction;
  Utilities.DisplayTransactionInformation("Dependent Transaction",
      tx.TransactionInformation);

  Thread.Sleep(3000);

  tx.Complete();

  Utilities.DisplayTransactionInformation("Dependent TX Complete",
      tx.TransactionInformation);
}
```

在 DependentTransaction()方法中，先实例化 CommittableTransaction 类，创建一个根事务，显示事务的信息。接着，tx.DependentClone()方法创建一个依赖的事务。把这个依赖事务传递给 TxTask()方法，它定义为新任务的入口点。

DependentClone()方法需要一个 DependentCloneOption 类型的参数。DependentCloneOption 是一个枚举，其值是 BlockCommitUntilComplete 和 RollbackIfNotComplete。如果根事务在依赖事务之前完成，这个选项就很重要。把该选项设置为 RollbackIfNotComplete，如果依赖事务没有在根事务的 Commit()方法之前调用 Complete()方法，事务就终止。把该选项设置为 BlockCommitUntilComplete，Commit()方法就等待由所有依赖事务定义的结果。

接着，如果用户没有终止事务，就调用 CommittableTransaction 类的 Commit()方法。

第 21 章介绍了线程。

```
static void DependentTransaction()
{
  var tx = new CommittableTransaction();
  Utilities.DisplayTransactionInformation("Root TX created",
      tx.TransactionInformation);

  try
  {
    Task.Factory.StartNew(TxTask, tx.DependentClone(
        DependentCloneOption.BlockCommitUntilComplete));

    if (Utilities.AbortTx())
    {
        throw new ApplicationException("transaction abort");
    }

    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    tx.Rollback();
  }

  Utilities.DisplayTransactionInformation("TX finished",
      tx.TransactionInformation);
}
```

在应用程序的输出中，可以看到根事务及其标识符。由于使用了选项 DependentCloneOption.BlockCommitUntilComplete，根事务在 Commit()方法中等待，直到定义了依赖事务的结果。完成依赖的事务后，就提交事务。

```
Root TX created
Creation Time: 8:35:25 PM
Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-0000000000

Abort the Transaction (y/n)? n

Dependent Transaction
Creation Time: 8:35:25 PM
Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-0000000000
```

```
Dependent TX Complete
Root TX finished
Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-0000000000

Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-0000000000
```

25.5.4 环境事务

System.Transactions 的一大优点是环境事务功能。使用环境事务，就不需要手动用连接登记事务；在支持环境事务的资源中，这是自动实现的。

环境事务与当前的线程关联起来。可以使用静态属性 Transaction.Current 获取和设置环境事务。支持环境事务的 API 会检查这个属性，以获得环境事务，用该事务登记。ADO.NET 连接支持环境事务。

可以创建一个 CommittableTransaction 对象，把它赋予 Transaction.Current 属性，以初始化环境事务。创建环境事务的另一种方式是使用 TransactionScope 类。TransactionScope 类的构造函数会创建一个环境事务。

TransactionScope 类的重要方法有 Complete()和 Dispose()方法。Complete()方法可以设置事务的作用域的成功位。Dispose()方法完成该作用域，如果该作用域与根作用域相关，就提交或回滚事务。

因为 TransactionScope 类实现 IDisposable 接口，所以可以用 using 语句定义事务的作用域。默认构造函数创建了一个新的事务，创建 TransactionScope 实例之后，用 Transaction.Current 属性的 get 访问器访问事务，以在控制台上显示事务信息。

要获得事务何时完成的信息，就应为环境事务的 TransactionCompleted 事件设置 OnTransactionCompleted()方法。

然后，调用 StudentData.AddStudent()方法，以新建一个 Student 对象，并把它写入数据库中。对于环境事务，不再需要给这个方法传递 Transaction 对象，因为 SqlConnection 类支持环境事务，并且会自动通过连接登记它。接着 TransactionScope 类的 Complete()方法设置成功位，在 using 语句的最后删除 TransactionScope 类，完成提交。如果没有调用 Complete()方法，Dispose()就终止事务。

如果 ADO.NET 连接不应使用环境事务来登记，就可以用连接字符串设置值 Enlist=false。

```
static void TransactionScope()
{
  using (var scope = new TransactionScope())
  {
    Transaction.Current.TransactionCompleted +=
        OnTransactionCompleted;
```

```
    Utilities.DisplayTransactionInformation("Ambient TX created",
        Transaction.Current.TransactionInformation);

    var s1 = new Student
    {
      FirstName = "Angela",
      LastName = "Nagel",
      Company = "Kantine M101"
    };
    var db = new StudentData();
    db.AddStudent(s1);

    if (!Utilities.AbortTx())
      scope.Complete();
    else
      Console.WriteLine("transaction will be aborted");

  } // scope.Dispose()
}

static void OnTransactionCompleted(object sender,
                                   TransactionEventArgs e)
{
  Utilities.DisplayTransactionInformation("TX completed",
      e.Transaction.TransactionInformation);
}
```

运行应用程序，可以看到在创建 TransactionScope 类的一个实例后，有一个激活的环境事务。应用程序的最终结果是来自 TransactionCompleted 事件处理程序的输出，以显示最终的事务状态。

```
Ambient TX created
Creation Time: 9:55:40 PM
Status: Active
Local ID: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Distributed ID: 00000000-0000-0000-0000-0000000000

Abort the Transaction (y/n)? n

TX completed
Creation Time: 9:55:40 PM
Status: Committed
Local ID: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Distributed ID: 00000000-0000-0000-0000-0000000000
```

1. 嵌套的作用域和环境事务

使用 TransactionScope 类，还可以嵌套作用域。嵌套的作用域可以直接位于原来的作用域中，或位于从作用域中调用的方法中。嵌套的作用域可以使用与外层作用域相同的事务，抑制事务，或创建独立于外层作用域的新事务。作用域的要求通过 TransactionScopeOption 枚举定义，把该枚举传递给 TransactionScope 类的构造函数。

TransactionScopeOption 枚举的值及其作用如表 25-2 所示。

表 25-2

TransactionScopeOption 的成员	说　明
Required	Required 指定，作用域需要一个事务。如果外层的作用域已经包含了一个环境事务，内层的作用域就使用已有的事务。如果环境事务不存在，就新建一个事务。如果两个作用域共享同一个事务，这两个作用域都会影响事务的结果。只有所有作用域都设置了成功位，事务才能提交。如果在根作用域被删除之前一个作用域没有调用 Complete()方法，事务就终止
RequiresNew	RequiresNew 总是创建一个新的事务。如果外层的作用域已定义了一个事务，内层作用域中的事务就是完全独立的。两个事务都可以独立地提交或终止
Suppress	使用 Suppress，无论外层的作用域是否包含一个事务，作用域都不会包含环境事务

下面的例子定义了两个作用域，其中使用 TransactionScopeOption.RequiresNew 选项，把内层作用域配置为需要一个新的事务：

```
using (var scope = new TransactionScope())
{
  Transaction.Current.TransactionCompleted +=
      OnTransactionCompleted;

  Utilities.DisplayTransactionInformation("Ambient TX created",
      Transaction.Current.TransactionInformation);

  using (var scope2 =
      new TransactionScope(TransactionScopeOption.RequiresNew))
  {
    Transaction.Current.TransactionCompleted +=
        OnTransactionCompleted;

    Utilities.DisplayTransactionInformation(
        "Inner Transaction Scope",
        Transaction.Current.TransactionInformation);

    scope2.Complete();
  }
  scope.Complete();
}
```

运行应用程序，尽管使用相同的线程，但会看到两个作用域有不同的事务标识符。让一个线程因为作用域不同而有不同的环境事务，事务标识符在 GUID 后面的最后一位数字上不同。

GUID 是包含 128 位唯一值的全局唯一标识符。

```
Ambient TX created
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-0000000000
```

```
Inner Transaction Scope
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-0000000000

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-0000000000

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-0000000000
```

如果把内层作用域的设置改为 TransactionScopeOption.Required，就会发现两个作用域使用同一个事务，且都影响事务的结果。

2. 多线程和环境事务

如果多个线程使用同一个环境事务，就需要多做一些工作。因为一个环境事务绑定到一个线程上，所以如果新建了一个线程，它就不会有起始线程中的环境事务。

这个行为在下一个例子中说明。在 Main()方法中，创建了一个 TransactionScope。在这个事务的作用域中，启动一个新线程。新线程的主要方法 ThreadMethod()新建了一个事务的作用域。在创建该作用域的过程中，没有传递任何参数，因此使用默认选项 TransactionScopeOption.Required。如果存在一个环境事务，就使用已有的事务。如果没有环境事务，就新建一个事务(代码文件 MultithreadingAmbientTx/Program.cs)。

```
using System;
using System.Threading.Tasks;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
  class Program
  {
    static void Main()
    {
      try
      {
        using (var scope = new TransactionScope())
        {
          Transaction.Current.TransactionCompleted +=
              TransactionCompleted;

          Utilities.DisplayTransactionInformation("Main task TX",
              Transaction.Current.TransactionInformation);
```

```
          Task.Factory.StartNew(TaskMethod);

          scope.Complete();
        }
      }
      catch (TransactionAbortedException ex)
      {
        Console.WriteLine("Main—Transaction was aborted, {0}",
          ex.Message);
      }
    }

    static void TransactionCompleted(object sender,
       TransactionEventArgs e)
    {
      Utilities.DisplayTransactionInformation("TX completed",
         e.Transaction.TransactionInformation);
    }

    static void TaskMethod()
    {
      try
      {
        using (var scope = new TransactionScope())
        {
          Transaction.Current.TransactionCompleted +=
             TransactionCompleted;

          Utilities.DisplayTransactionInformation("Task TX",
             Transaction.Current.TransactionInformation);
          scope.Complete();
        }
      }
      catch (TransactionAbortedException ex)
      {
        Console.WriteLine("TaskMethod—Transaction was aborted, {0}",
           ex.Message);
      }
    }
  }
}
```

启动应用程序，就会看到两个线程中的事务是完全独立的。新线程中的事务有一个不同的事务ID。事务ID的区别是GUID后面的最后一个数字不同，这与嵌套的作用域相同。

```
Main task TX
Creation Time: 21:41:25
Status: Active
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 21:41:25
Status: Committed
```

```
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Task TX
Creation Time: 21:41:25
Status: Active
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 21:41:25
Status: Committed
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Distributed ID: 00000000-0000-0000-0000-000000000000
```

要在另一个线程中使用同一个环境事务，需要使用依赖事务。现在修改示例，给新线程传递一个依赖事务。调用环境事务上的 DependentClone()方法，以从环境事务中创建依赖事务。在这个方法中，设置了 DependentCloneOption.BlockCommitUntilComplete，以便主调线程会等到新线程完成后，才提交事务。

```
class Program
{
  static void Main()
  {
    try
    {
      using (var scope = new TransactionScope())
      {
        Transaction.Current.TransactionCompleted +=
            TransactionCompleted;

        Utilities.DisplayTransactionInformation("Main thread TX",
            Transaction.Current.TransactionInformation);

        Task.Factory.StartNew(TaskMethod,
          Transaction.Current.DependentClone(
              DependentCloneOption.BlockCommitUntilComplete));

        scope.Complete();
      }
    }
    catch (TransactionAbortedException ex)
    {
      Console.WriteLine("Main—Transaction was aborted, {0}",
          ex.Message);
    }
  }
```

在线程的方法中，使用 Transaction.Current 属性的 set 访问器把所传递的依赖事务赋予环境事务。现在事务的作用域通过依赖事务来使用同一个事务。使用完依赖事务时，需要调用 DependentTransaction 对象的 Complete()方法。

```
static void TaskMethod(object dependentTx)
{
```

```
    var dTx = dependentTx as DependentTransaction;

    try
    {
      Transaction.Current = dTx;

      using (var scope = new TransactionScope())
      {
        Transaction.Current.TransactionCompleted +=
            TransactionCompleted;

        Utilities.DisplayTransactionInformation("Task TX",
            Transaction.Current.TransactionInformation);
        scope.Complete();
      }
    }
    catch (TransactionAbortedException ex)
    {
      Console.WriteLine("TaskMethod - Transaction was aborted, {0}",
          ex.Message);
    }
      finally
      {
        if (dTx != null)
        {
            dTx.Complete();
        }
      }
    }

    static void TransactionCompleted(object sender,
        TransactionEventArgs e)
    {
      Utilities.DisplayTransactionInformation("TX completed",
          e.Transaction.TransactionInformation);
    }
}
```

现在运行应用程序，主线程和新建的线程都正在使用同一个事务，并正在影响该事务。线程列出的事务有相同的标识符。如果一个线程没有调用Complete()方法设置成功位，就终止整个事务。

```
Main task TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Task TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 23:00:57
```

```
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 23:00:57
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

25.6 隔离级别

本章的开头介绍了用于描述成功事务的 ACID 属性。ACID 中的字母 I(Isolation，隔离)并不总是完全需要。出于性能原因，可以降低隔离要求，但必须了解改变隔离级别带来的问题。

如果不完全隔离事务外部的作用域，就可能出问题，这些问题有 3 类。

- **脏读**——在脏读操作中，另一个事务可以读取在一个事务中改变的记录。因为在一个事务中改变的记录可能回滚到最初的状态，所以从另一个事务中读取这个临时状态就称为“脏读”——数据并没有提交。通过锁定要改变的记录，就可以避免这个问题。
- **不可重复读**——当数据在事务中读取，而该事务运行的同时，另一个事务修改了相同的记录，此时，就会出现不可重复读操作。如果该记录在事务中读取多次，结果就会不同——不可重复。锁定读取的记录，即可避免这个问题。
- **幻读**——当读取一个范围内的数据，例如，使用 WHERE 子句读取时，就会出现幻读问题。在一个事务中读取这些记录时，另一个事务可以添加一个属于该范围的新记录。用相同的 WHERE 子句再次读取这些记录，会返回数量不同的记录。幻读一般出现在更新一个范围的记录时。例如，UPDATE Addresses SET Zip=4711 WHERE (Zip=2515)会把所有记录的邮政编码从 2315 更新为 4711。如果在更新过程中，另一个用户添加了一个邮政编码为 2315 的新记录，那么完成更新后，数据库将仍包含邮政编码为 2315 的记录。这个问题可以通过范围锁定来避免。

在定义隔离要求时，可以设置隔离级别。隔离级别用 IsolationLevel 枚举定义，在创建事务时，会配置该枚举(使用 CommittableTransaction 类的构造函数或者 TransactionScope 类的构造函数)。IsolationLevel 枚举定义了锁定操作。表 25-3 列出了 IsolationLevel 枚举的值。

表 25-3

隔离级别	说　明
ReadUncommitted	使用 ReadUncommitted，事务不会相互隔离。使用这个级别，不等待其他事务释放锁定的记录。这样,. 就可以从其他事务中读取未提交的数据——脏读。这个级别通常仅用于读取不管是否读取临时修改都无关紧要的记录，如报表
ReadCommitted	ReadCommitted 等待其他事务释放对记录的写入锁定。这样，就不会出现脏读操作。这个级别为读取当前的记录设置读取锁定，为要写入的记录设置写入锁定，直到事务完成为止。对于要读取的一系列记录，在移动到下一个记录上时，前一个记录都是未锁定的，所以可能出现不可重复的读操作

(续表)

隔离级别	说明
RepeatableRead	RepeatableRead 为读取的记录设置锁定，直到事务完成为止。这样，就避免了不可重复读的问题。但幻读仍可能发生
Serializable	Serializable 设置范围锁定。在运行事务时，不可能添加与所读取的数据位于同一个范围的新记录
Snapshot	Snapshot 用于对实际的数据建立快照。在复制修改的记录时，这个级别会减少锁定。这样，其他事务仍可以读取旧数据，而无须等待解锁
Unspecified	Unspecified 表示，提供程序使用另一个隔离级别值，该值不同于 IsolationLevel 枚举定义的值
Chaos	Chaos 类似于 ReadUncommitted，但除了执行 ReadUncommitted 值的操作之外，它不能锁定更新的记录

表 25-4 总结了设置最常用的事务隔离级别可能导致的问题。

表 25-4

隔离级别	脏读	不可重复读	幻读
ReadUncommitted	Y	Y	Y
ReadCommitted	N	Y	Y
RepeatableRead	N	N	Y
Serializable	N	N	N

下面的代码段说明了如何使用 TransactionScope 类设置隔离级别。在 TransactionScope 类的构造函数中，可以设置前面讨论的 TransactionScopeOption 枚举和 TransactionOptions 类。TransactionOptions 类允许定义 IsolationLevel 和 Timeout 属性。

```
var options = new TransactionOptions
{
  IsolationLevel = IsolationLevel.ReadUncommitted,
  Timeout = TimeSpan.FromSeconds(90)
};
using (var scope = new TransactionScope(
   TransactionScopeOption.Required, options))
{
  // Read data without waiting for locks from other transactions,
  // dirty reads are possible.
}
```

25.7 自定义资源管理器

System.Transactions 命名空间的一个最大优点是，很容易创建参与事务处理的自定义资源管理器。资源管理器不仅管理稳定的资源，还管理不稳定或内存中的资源，例如，简单的 int 值和泛型列表。

图 25-4 显示了资源管理器和事务类之间的关系。资源管理器实现了 IEnlistmentNotification 接口，

该接口定义了 Prepare()、InDoubt()、Commit()和 Rollback()方法。资源管理器实现这个接口，是为了管理资源的事务。作为事务的一部分，资源管理器必须用 Transaction 类登记。不稳定的资源管理器调用 EnlistVolatile()方法，稳定的资源管理器调用 EnlistDurable()方法。根据事务的结果，事务管理器通过资源管理器从 IEnlistmentNotification 接口中调用不同的方法。

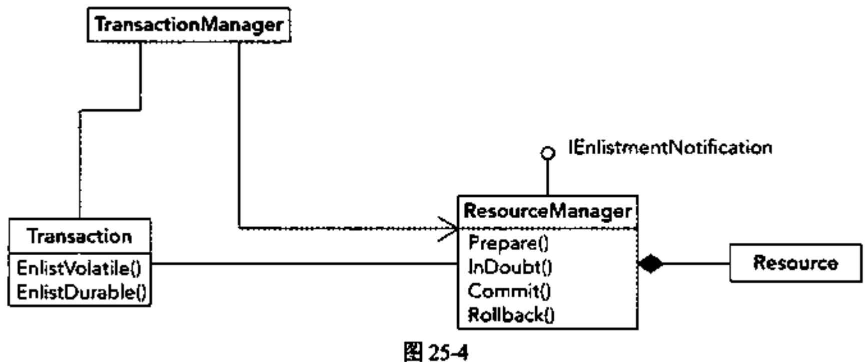


图 25-4

表 25-5 介绍了 IEnlistmentNotification 接口中必须通过资源管理器实现的方法。25.1.1 节解释了激活、准备和提交阶段。

表 25-5

IEnlistmentNotification 接口的成员	说　明
Prepare	事务管理器调用 Prepare()方法准备事务。资源管理器调用 PreparingEnlistment 参数(它传递给 Prepare()方法)的 Prepared()方法，来完成准备阶段。如果工作没有成功完成，资源管理器就调用 ForceRollback()方法，通知事务管理器。稳定的资源管理器必须编写一个日志，以便它在准备阶段之后成功完成事务
Commit	当所有资源管理器都成功准备好事务时，事务管理器就调用 Commit()方法。资源管理器现在可以完成工作，使之可在事务的外部可见，并调用 Enlistment 参数的 Done()方法
Rollback	如果一个资源管理器没有成功地准备好事务，事务管理器就调用所有资源管理器的 Rollback()方法。在状态返回为该事务之前的状态后，资源管理器就调用 Enlistment 参数的 Done()方法
InDoubt	如果事务管理器调用 Commit()方法后出现了一个问题(资源管理器没有用 Done()方法返回完成信息)，事务管理器就调用 InDoubt()方法

事务资源

事务资源必须保存实时值和临时值。实时值从事务的外部读取，并定义了事务回滚时的有效状态。临时值定义了事务提交时事务的有效状态。

为了使非事务类型变成事务类型，泛型类 Transactional<T>封装了一个非泛型类型，从而使它的用法如下：

```
var txInt = new Transactional<int>();
var txString = new Transactional<string>();
```

下面看看 Transactional<T>类的实现代码。托管资源的实时值包含在变量 liveValue 中，与事务相关的临时值存储在 ResourceManager<T>中。变量 enlistedTransaction 与环境事务(假定存在环境事务)关联起来(代码文件 CustomResource/Transactional.cs)。

```
using System.Diagnostics;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
  public partial class Transactional<T>
  {
    private T liveValue;
    private ResourceManager<T> enlistment;
    private Transaction enlistedTransaction;
```

在 Transactional 类的构造函数中，实时值设置为变量 liveValue。如果在环境事务中调用该构造函数，就调用帮助方法 GetEnlistment()。GetEnlistment()方法先检查是否有一个环境事务，并断言是否没有一个环境事务。如果没有登记事务，就实例化 ResourceManager<T>辅助类，并调用 EnlistVolatile()方法，用该事务登记资源管理器。另外，把变量 enlistedTransaction 设置为该环境事务。

如果环境事务不同于已登记的事务，就抛出一个异常。该实现代码不支持在两个不同的事务中修改相同的值。如果有这个要求，就可以创建一把锁，等待一个事务释放该锁，之后在另一个事务中修改它。

```
public Transactional(T value)
{
  if (Transaction.Current == null)
  {
     this.liveValue = value;
  }
  else
  {
    this.liveValue = default(T);
    GetEnlistment().Value = value;
  }
}

public Transactional()
   : this(default(T)) {}

private ResourceManager<T> GetEnlistment()
{
  Transaction tx = Transaction.Current;
  Trace.Assert(tx != null,
     "Must be invoked with ambient transaction");

  if (enlistedTransaction == null)
  {
    enlistment = new ResourceManager<T>(this, tx);
    tx.EnlistVolatile(enlistment, EnlistmentOptions.None);
    enlistedTransaction = tx;
    return enlistment;
  }
```

```
    else if (enlistedTransaction == Transaction.Current)
    {
      return enlistment;
    }
    else
    {
      throw new TransactionException(
          "This class only supports enlisting with one transaction");
    }
  }
```

Value 属性返回所包含的类的值，并设置该值。但是，通过事务，不能只设置和返回 liveValue 变量。只有对象在事务的外部，才能设置和返回它。为了使代码的可读性更强，Value 属性在其实现代码中使用 GetValue()和 SetValue()方法。

```
public T Value
{
    get { return GetValue(); }
    set { SetValue(value); }
}
```

GetValue()方法检查是否存在环境事务。如果不存在，就返回 liveValue 变量。如果有环境事务，前面的 GetEnlistment()方法就返回资源管理器，并使用 Value 属性，返回事务处理中所包含对象的临时值。

SetValue()方法非常类似于 GetValue()方法，其区别是它修改实时值或临时值。

```
protected virtual T GetValue()
{
  if (Transaction.Current == null)
  {
     return liveValue;
  }
  else
  {
     return GetEnlistment().Value;
  }
}

protected virtual void SetValue(T value)
{
  if (Transaction.Current == null)
  {
     liveValue = value;
  }
  else
  {
     GetEnlistment().Value = value;
  }
}
```

在 Transactional<T>类中实现的 Commit()和 Rollback()方法从资源管理器中调用。Commit()方法从第一个变量接收的临时值中设置实时值，并在事务完成时使 enlistedTransaction 变量置空。通过 Rollback()方法，事务也完成了，但这里忽略了临时值，且只使用了实时值。

```
        internal void Commit(T value, Transaction tx)
        {
            liveValue = value;
            enlistedTransaction = null;
        }
        internal void Rollback(Transaction tx)
        {
            enlistedTransaction = null;
        }
    }
```

因为 Transactional<T>类使用的资源管理器仅用于 Transactional<T>类自身，所以它作为一个内部类实现。通过构造函数，把父变量设置为与事务包装类关联起来。在事务中使用的临时值从实时值中复制。注意事务需要隔离(代码文件 CustomResource/ResourceManager.cs)。

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public partial class Transactional<T>
    {
        internal class ResourceManager<T1>: IEnlistmentNotification
        {
            private Transactional<T1>parent;
            private Transaction currentTransaction;

            internal ResourceManager(Transactional<T1>parent, Transaction tx)
            {
                this.parent = parent;
                Value = DeepCopy(parent.liveValue);
                currentTransaction = tx;
            }

            public T1 Value { get; set; }
```

因为临时值可能在事务中变化，所以包装类的实时值可能不会在事务中发生变化。在创建一些类的副本时，可以调用在 ICloneable 接口中定义的 Clone()方法。但是，在定义 Clone()方法时，允许实现代码创建浅表副本或深层副本。如果类型 T 包含引用类型，并实现了浅表副本，改变临时值也会改变初始值。这会与事务的隔离特性和一致特性冲突。这里需要一个深层副本。

为了进行深层复制，DeepCopy()方法可把对象序列化到流中，并从流中反序列化对象。因为在 C# 5.0 中，不能定义对类型 T 的限制，即指定需要序列化，所以 Transactional<T>类的静态构造函数会检查 Type 对象的 IsSerializable 属性，以确定类型是否可以序列化。

```
static ResourceManager()
{
  Type t = typeof(T1);
  Trace.Assert(t.IsSerializable, "Type " + t.Name +
      " is not serializable");
}
```

```
private T1 DeepCopy(T1 value)
{
  using (var stream = new MemoryStream())
  {
    var formatter = new BinaryFormatter();
    formatter.Serialize(stream, value);
    stream.Flush();
    stream.Seek(0, SeekOrigin.Begin);

    return (T1)formatter.Deserialize(stream);
  }
}
```

IEnlistmentNotification 接口由 ResourceManager<T>类实现。这是用事务进行登记的要求。

只有用 preparingEnlistment 调用 Prepared()方法，Prepare()的实现代码才会响应。因为在将临时值赋予实时值时，不应有问题，所以 Prepare()方法会成功。在 Commit()方法的实现代码中，调用父对象的 Commit()方法。其中，把变量 liveValue 设置为在事务中使用的 ResourceManager 的值。Rollback()方法仅完成工作，不改变实时值。对于不稳定的资源，在 InDoubt()方法中不执行太多的操作。写入一个日志项可能会有用。

```
            public void Prepare(PreparingEnlistment preparingEnlistment)
            {
               preparingEnlistment.Prepared();
            }

            public void Commit(Enlistment enlistment)
            {
               parent.Commit(Value, currentTransaction);
               enlistment.Done();
            }

            public void Rollback(Enlistment enlistment)
            {
               parent.Rollback(currentTransaction);
               enlistment.Done();
            }

            public void InDoubt(Enlistment enlistment)
            {
               enlistment.Done();
            }
         }
      }
   }
```

现在，只要类型是可以序列化的，Transactional<T>类就可以用于使非事务类变成事务类，例如，int、string，甚至更复杂的类，如 Student(代码文件 CustomResource/Program.cs)。

```
using System;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
```

```
class Program
{
  static void Main()
  {
    var intVal = new Transactional<int>(1);
    var student1 = new Transactional<Student>(new Student());
    student1.Value.FirstName = "Andrew";
    student1.Value.LastName = "Wilson";

    Console.WriteLine("before the transaction, value: {0}",
       intVal.Value);
    Console.WriteLine("before the transaction, student: {0}",
       student1.Value);

    using (var scope = new TransactionScope())
    {
      intVal.Value = 2;
      Console.WriteLine("inside transaction, value: {0}",
         intVal.Value);

      student1.Value.FirstName = "Ten";
      student1.Value.LastName = "SixtyNine";

      if (!Utilities.AbortTx())
       scope.Complete();
    }
    Console.WriteLine("outside of transaction, value: {0}",
       intVal.Value);
    Console.WriteLine("outside of transaction, student: {0}",
       student1.Value);
  }
 }
}
```

下面的控制台输出显示了应用程序的一次运行及提交的事务。

```
before the transaction, value: 1
before the transaction: student: Andrew Wilson
inside transaction, value: 2

Abort the Transaction (y/n)? n

outside of transaction, value: 2
outside of transaction, student: Ten SixtyNine
```

25.8　文件系统事务

可以编写一个自定义稳定资源管理器，来处理 File 类和 Registry 类。基于文件的稳定资源管理器可以复制原始文件，将对临时文件的修改写入一个临时目录中，使这些改变永久保存起来。在提交事务时，原始文件会用临时文件替代。自 Windows Vista 和 Windows Server 2008 以来，不再需要为文件和注册表编写自定义稳定资源管理器。这两个操作系统及以后的操作系统支持用文件系统和注册表进行本地事务。为此，增加了新的 Windows API 调用，如 CreateFileTransacted()、CreateHardLinkTransacted()、

CreateSymbolicLinkTransacted()、CopyFileTransacted()等。这些API调用的共同之处是，它们都需要一个句柄来把事务作为变量传递，它们都不支持环境事务。不能从.NET 4.5.1中进行事务API调用，但可以使用Platform Invoke创建一个自定义包装器。

Platform Invoke详见第23章。

示例应用程序包装了本地方法CreateFileTransacted()，以在.NET应用程序中创建事务文件流。

在调用本地方法时，本地方法的参数必须映射到.NET数据类型。出于安全考虑，SafeHandle基类可用来映射本地HANDLE类型。SafeHandle基类是一种抽象类型，它包装了操作系统句柄，并支持句柄资源的关键终止操作。根据句柄的允许值，可以使用派生类SafeHandleMinusOneIsInvalid和SafeHandleZeroOrMinusOneIsInvalid包装本地句柄。SafeFileHandle类派生自SafeHandleZeroOrMinusOneIsInvalid。为了把句柄映射到事务上，定义了SafeTransactionHandle类(代码文件FileSystemTransactions/SafeTransactionHandle.cs)。

```
using System;
using System.Runtime.Versioning;
using System.Security.Permissions;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
  [SecurityCritical]
  internal sealed class SafeTransactionHandle:
      SafeHandleZeroOrMinusOneIsInvalid
  {
    private SafeTransactionHandle()
        : base(true) { }

    public SafeTransactionHandle(IntPtr preexistingHandle,
      bool ownsHandle)
      : base(ownsHandle)
    {
      SetHandle(preexistingHandle);
    }

    [ResourceExposure(ResourceScope.Machine)]
    [ResourceConsumption(ResourceScope.Machine)]
    protected override bool ReleaseHandle()
    {
      return NativeMethods.CloseHandle(handle);
    }
  }
}
```

.NET中的所有本地方法都用下面的NativeMethods类定义。在示例中，所需的本地API是CreateFileTransacted()和CloseHandle()方法，它们定义为类的静态成员。这些方法声明为extern，因为它们没有C#实现代码。其实现代码在本地DLL中由DllImport特性定义。这两个方法都可以在本地DLL Kernel32.dll中找到。通过方法的声明，把用Windows API调用定义的参数映射到.NET数据

类型。txHandle 参数表示事务的一个句柄，其类型是以前定义的 SafeTransactionHandle(代码文件 FileSystemTransactions/NativeMethods.cs)。

```
using System;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
  internal static class NativeMethods
  {
    [DllImport("Kernel32.dll",
        CallingConvention = CallingConvention.StdCall,
        CharSet = CharSet.Unicode)]
    internal static extern SafeFileHandle CreateFileTransacted(
        String lpFileName,
        uint dwDesiredAccess,
        uint dwShareMode,
        IntPtr lpSecurityAttributes,
        uint dwCreationDisposition,
        int dwFlagsAndAttributes,
        IntPtr hTemplateFile,
        SafeTransactionHandle txHandle,
        IntPtr miniVersion,
        IntPtr extendedParameter);

    [DllImport("Kernel32.dll", SetLastError = true)]
    [ResourceExposure(ResourceScope.Machine)]
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    [return: MarshalAs(UnmanagedType.Bool)]
    internal static extern bool CloseHandle(IntPtr handle);
  }
}
```

IKernelTransaction 接口用于获得事务句柄，并把它传递给经过事务处理的 Windows API 调用。这是一个 COM 接口，且必须使用 COM Interop 特性包装到.NET 中，如下面所示。GUID 特性必须有与接口定义相同的标识符，因为这是用于 COM 接口定义的标识符(代码文件 FileSystem-Transactions/IKernelTransaction.cs)。

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

namespace Wrox.ProCSharp.Transactions
{
  [ComImport]
  [Guid("79427A2B-F895-40e0-BE79-B57DC82ED231")]
  [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
  internal interface IKernelTransaction
  {
     void GetHandle(out SafeTransactionHandle ktmHandle);
  }
}
```

最后，TransactionFile 类正是将由.NET 应用程序使用的类，这个类定义了 GetTransactedFileStream 方法。这个方法的参数需要一个文件名，并返回一个 System.IO.FileStream。返回的流是一个一般的.NET 流，它只引用一个经过事务处理的文件。

在实现代码中，TransactionInterop.GetDtcTransaction()方法创建了 IKernelTransaction 的一个接口指针，它指向环境事务，环境事务作为一个参数传递给 GetDtcTransaction()方法。使用 IKernelTransaction 接口，创建 SafeTransactionHandle 类型的句柄。然后把这个句柄传递给包装的 API，其名称是 NativeMethods.CreateFileTransacted()。使用返回的文件句柄，新建一个 FileStream 实例，并把它返回调用者(代码文件 FileSystemTransactions/TransactedFile.cs)。

```
using System;
using System.IO;
using System.Security.Permissions;
using System.Transactions;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
  public static class TransactedFile
  {
    internal const short FILE_ATTRIBUTE_NORMAL = 0x80;
    internal const short INVALID_HANDLE_VALUE = -1;
    internal const uint GENERIC_READ = 0x80000000;
    internal const uint GENERIC_WRITE = 0x40000000;
    internal const uint CREATE_NEW = 1;
    internal const uint CREATE_ALWAYS = 2;
    internal const uint OPEN_EXISTING = 3;

    [FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
    public static FileStream GetTransactedFileStream(string fileName)
    {
      IKernelTransaction ktx = (IKernelTransaction)
         TransactionInterop.GetDtcTransaction(Transaction.Current);

      SafeTransactionHandle txHandle;
      ktx.GetHandle(out txHandle);

      SafeFileHandle fileHandle = NativeMethods.CreateFileTransacted(
          fileName, GENERIC_WRITE, 0,
          IntPtr.Zero, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
          IntPtr.Zero,
          txHandle, IntPtr.Zero, IntPtr.Zero);

      return new FileStream(fileHandle, FileAccess.Write);
    }
  }
```

现在，很容易在.NET 代码中使用事务 API 了。可以用 TransactionScope 类创建一个环境事务，在环境事务的作用域内使用 TransactedFile 类。如果终止了事务处理，就不会把数据写入文件中。如果提交了事务，就可以在临时目录下找到该文件(代码文件 Windows8Transactions/Program.cs)。

```
using System;
```

```
using System.IO;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
  class Program
  {
    static void Main()
    {
      using (var scope = new TransactionScope())
      {
        FileStream stream = TransactedFile.GetTransactedFileStream(
            "sample.txt");

        var writer = new StreamWriter(stream);
        writer.WriteLine("Write a transactional file");
        writer.Close();

        if (!Utilities.AbortTx())
            scope.Complete();
      }
    }
  }
}
```

现在就可以在同一个事务中使用数据库、不稳定的资源和文件。

25.9 小结

本章学习了事务的特性，以及如何使用 System.Transactions 名称空间中的类创建和管理事务。

事务用 ACID 属性来描述：原子性、一致性、隔离性和持久性。并不是所有这些属性都是必需的，例如，不稳定的资源就不需要支持持久性但可以包含隔离选项。

进行事务处理的最简单的方式是创建环境事务，并使用 TransactionScope 类。环境事务非常适合于处理没有显式打开和关闭数据库连接的 ADO.NET 数据适配器和 ADO.NET Entity Framework，ADO.NET 详见第 32 章，Entity Framework 详见第 33 章。

在多个线程中使用同一个事务，可以使用 DependentTransaction 类创建对另一个事务的一个依赖关系。登记一个实现了 IEnlistmentNotification 接口的资源管理器，可以创建参与事务处理的自定义资源。

最后，探讨了如何通过.NET Framework 和 C#使用文件系统事务。

第 26 章介绍如何使用 System.Net 名称空间实现在不同系统之间的通信。

第26章

网　络

本章要点

- 使用 HttpClient
- 在 Windows Forms 应用程序中使用 WebBrowser 控件
- 操纵 IP 地址，执行 DNS 查询
- 用 TCP、UDP 和套接字类进行套接字编程

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- HttpClient
- Browser
- DnsLookup
- SocketClient
- SocketServer
- TcpSend
- TcpReceive
- ViewHeaders
- WebSocketSample

26.1　网络

本章将采取非常实用的网络方法，结合示例讨论相关理论和相应的网络概念。本章并不是计算机网络的指南，但会介绍如何使用.NET Framework 进行网络通信。

本章将介绍如何在 Windows Forms 环境中使用 WebBrowser 控件，以及 WebBrower 控件如何更方便地完成某些 Internet 访问任务。但本章会从最简单的示例开始，阐明怎样给服务器发送请求和

在响应中存储返回的信息。

本章将讨论通过.NET 基类提供的工具，便于使用各种网络协议(尤其是 HTTP 和 TCP)作为客户访问网络和 Internet。本章将介绍通过.NET Framework 获得这些协议的一些低级方式，使用 WCF 技术等还可以实现与这些协议通信的其他方式。

在网络环境下，我们最感兴趣的两个名称空间是 System.Net 和 System.Net.Sockets。System.Net 名称空间通常与较高层的操作有关，例如下载和上传文件，使用 HTTP 和其他协议进行 Web 请求等；而 System.Net.Sockets 名称空间包含的类通常与较低层的操作有关。如果要直接使用套接字或 TCP/IP 之类的协议，这个名称空间中的类就非常有用，这些类中的方法与 Windows 套接字(Winsock)API 函数(派生自 Berkeley 套接字接口)非常类似。本章介绍的一些对象位于 System.IO 名称空间中。

后面的章节将讨论怎样使用 C#、ASP.NET 编写功能强、效率高的动态网页。大多数情况下，访问 ASP.NET 页面的客户使用的是 Internet Explorer 或其他 Web 浏览器，如 Chrome、Opera 或 Firefox。但是，有时需要把 Web 浏览特征添加到自己的应用程序中，或者需要让自己的应用程序从某个网站以编程方式获取信息。在后一种情况下，对于站点，比较好的解决方案通常是实现一个 Web 服务。但是，如果访问公共的 Internet 站点，就不能控制站点的实现方式。

26.2 HttpClient 类

HttpClient 类用于发送 HTTP 请求，接收请求的响应。它在 System.Net.Http 名称空间中。System.Net.Http 名称空间中的类有助于简化在客户端和服务器上使用 Web 服务。

HttpClient 类派生于 System.Net.Http.HttpMessageInvoker 类，这个类负责执行 SendAsync 方法。SendAsync 方法是 HttpClient 类的主干。如本节后面所述，这个方法有几个派生物。顾名思义，SendAsync 方法调用是异步的，这样就可以编写一个完全异步的系统来调用 Web 服务。

26.2.1 异步调用 Web 服务

本章的下载代码示例是 HttpClientExample。它异步调用一个 Web 服务：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net.Http;

namespace HttpClientExample
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("In main before call to GetData!");
         GetData();
         Console.WriteLine("Back in main after call to GetData!");
         Console.ReadKey();
      }
```

```
        private static async void GetData()
        {
            HttpClient httpClient = new HttpClient();
            HttpResponseMessage response = null;
            response = await httpClient.GetAsync(
            "http://services.odata.org/Northwind/Northwind.svc/Regions");
            if(response.IsSuccessStatusCode)
            {
                Console.WriteLine("Response Status Code: "
                            + response.StatusCode
                            + " " + response.ReasonPhrase);
                string responseBodyAsText = response
                            .Content
                            .ReadAsStringAsync()
                            .Result;
                Console.WriteLine("Received payload of "
                            + responseBodyAsText.Length
                            + " characters");
                //Console.WriteLine(responseBodyAsText);
            }
        }
    }
}
```

执行这段代码，生成如下结果：

```
In main before call to GetData!
Back in main after call to GetData!
Response Status Code: OK OK
Received payload of 1551 characters
```

因为 HttpClient 类使用 GetAsync 方法调用，且使用了 await 关键字，所以 Main 方法可以在 GetAsync 方法调用 Web 服务的同时执行完毕。因此 Main 方法输出到屏幕上的消息先显示，而 GetData 调用中的数据后显示。

GetData 方法完成主要的工作。首先是实例化一个 HttpClient 对象。这个 HttpClient 对象是线程安全的，所以一个 HttpClient 对象就可以用于处理多个请求。HttpClient 的每个实例都维护它自己的线程池，所以 HttpClient 实例之间的请求会被隔离。

接着调用 GetAsync，给它传递要调用的方法的地址。GetAsync 调用带一个字符串或 URI 对象作为参数。在本例中调用 Microsoft 的 Odata 示例站点，但可以修改这个地址，以调用任意多个 REST Web 服务。

对 GetAsync 的调用返回一个 HttpResponseMessage 对象。HttpResponseMessage 类表示包含标题、状态和内容的响应。检查响应的 IsSuccessfulStatusCode 属性，可以确定请求是否成功。

26.2.2 标题

发出请求时没有设置或改变任何标题，但 HttpClient 的 DefaultRequestHeaders 属性允许设置或改变标题。使用 Add 方法可以给集合添加标题。设置标题值后，标题和标题值会与这个 HttpClient 实例发送的每个请求一起发送。

例如，响应内容默认为 XML 格式。要改变它，可以在请求中添加一个 Accept 标题，以使用 JSON。

在调用 GetAsync 之前添加如下代码，内容就会以 JSON 格式返回：

```
httpClient.DefaultRequestHeaders.Add("Accept",
"application/json;odata=verbose");
```

在 GetData 方法中，取消最后一行注释符号，就会显示响应内容中的数据。添加和删除标题，运行示例，会以 XML 和 JSON 格式显示内容。

从 DefaultHeaders 属性返回的 HttpRequestHeaders 对象有许多辅助属性，可用于许多标准标题。可以从这些属性中读取标题的值，但它们是只读的。要设置其值，需要使用 Add 方法。

HttpClientHeadersExample 项目说明了如何在响应和请求中遍历标题。下面是 GetData 的代码：

```
private static void GetData()
{
   HttpClient httpClient = new HttpClient();
   HttpResponseMessage response = null;
   //uncomment to see Accept Header in request
   //httpClient.DefaultRequestHeaders.Add("Accept",
                                    "application/json;odata=verbose");
   Console.WriteLine("Request Headers:");
   EnumerateHeaders(httpClient.DefaultRequestHeaders);
   Console.WriteLine();
   response = httpClient.GetAsync(
      "http://services.odata.org/Northwind/Northwind.svc/Regions").Result;
   if (response.IsSuccessStatusCode)
   {
      Console.WriteLine("Response Headers:");
      EnumerateHeaders(response.Headers);
   }
}
```

其开头与上一个示例类似。添加了 EnumerateHeaders 方法，它把一个 HttpHeaders 对象作为参数。HttpHeaders 是 HttpRequestHeaders 和 HttpResponseHeaders 的基类。这两个特殊化的类都添加了辅助属性，以直接访问标题。HttpHeader 对象定义为 keyValuePair<string, IEnumerable<string>>。这表示每个标题在集合中都可以有多个值。因此，如果希望改变标题中的值，就需要删除原值，添加新值。

EnumerateHeaders 函数很简单：

```
private static void EnumerateHeaders(HttpHeaders headers)
{
   foreach (var header in headers)
   {
      var value = "";

      foreach (var val in header.Value)
      {
         value = val + " ";
      }
      Console.WriteLine("Header: " + header.Key + " Value: " + value);
   }
}
```

因为标题值可以有多个，标题的值部分也必须迭代。所以在循环的内部还有一个循环，来枚举找到的所有值。

运行这段代码，不会显示请求的任何标题。

如果像上一个示例那样添加了 Accept 标题，再次执行代码，该标题就会出现在输出中。下面是添加了 Accept 标题的输出：

```
Request Headers:
Header: Accept Value: application/json; odata=verbose

Response Headers:
Header: Connection Value: Keep-Alive
Header: Vary Value: *
Header: X-Content-Type-Options Value: nosniff
Header: DataServiceVersion Value: 2.0;
Header: Access-Control-Allow-Origin Value: *
Header: Access-Control-Allow-Methods Value: GET
Header: Access-Control-Allow-Headers Value: Accept, Origin, Content-Type,
        MaxDataServiceVersion
Header: Access-Control-Expose-Headers Value: DataServiceVersion
Header: Cache-Control Value: private
Header: Date Value: Tue, 10 Dec 2013 00:47:07 GMT
Header: Server Value: Microsoft-IIS/7.5
Header: X-AspNet-Version Value: 4.0.30319
Header: X-Powered-By Value: ASP.NET
```

26.2.3 HttpContent

响应中的 Content 属性返回一个 HttpContent 对象。为了获得 HttpContent 对象中的数据，需要使用所提供的一个方法。在例子中，使用了 ReadAsStringAsync 方法。它返回内容的字符串表示。顾名思义，这是一个异步调用，但在这个例子中没有使用异步调用功能。调用 Result 方法会阻塞该调用，直到 ReadAsStringAsync 方法执行完毕，然后继续执行下面的代码。

其他从 HttpContent 对象中获得数据的方法有 ReadAsByteArrayAsync(返回数据的字节数组)和 ReadAsStreamAsync(返回一个流)。也可以使用 LoadIntoBufferAsync 把内容加载到内存缓存中。

Headers 属性返回 HttpContentHeaders 对象。它的工作方式与前面例子中的请求和响应标题相同。

26.2.4 HttpMessageHandler

HttpClient 类可以把 HttpMessageHandler 作为其构造函数的参数，这样就可以定制请求。默认使用 WebRequestHandler 对象。它有许多属性可以设置，例如 ClientCertificates、Pipelining、CachePolity、ImpersonationLevel 等。

HttpClientMessageHandlerRequest 项目是一个非常简单的例子，说明了如何添加定制的处理程序。这里再次使用了 GetData，这次它的代码如下：

```
private static void GetData()
{
   HttpClient httpClient = new HttpClient(new MessageHandler("error"));
   HttpResponseMessage response = null;
   Console.WriteLine();
   response = httpClient.GetAsync(
```

```
        "http://services.odata.org/Northwind/Northwind.svc/Regions").Result;
    Console.WriteLine(response.StatusCode);
}
```

注意创建 HttpClient 对象的调用接受了一个参数——MessageHandler 对象。这个处理程序的作用是把一个字符串作为参数，在控制台上显示它，如果消息是"error"，就把响应的状态码设置为 Bad Request。构造函数中的参数是示例实现的一部分。HttpClientHandler 不需要设置参数。下面是 MessageHandler 的代码：

```
public class MessageHandler : HttpClientHandler
{
    string displayMessage = "";
    public MessageHandler (string message)
    {
        displayMessage = message;
    }
    protected override Task<HttpResponseMessage> SendAsync(
            HttpRequestMessage request,
            System.Threading.CancellationToken cancellationToken)
    {
        Console.WriteLine("In DisplayMessageHandler " + displayMessage);
        if(displayMessage == "error")
        {
            var response = new HttpResponseMessage(
            System.Net.HttpStatusCode.BadRequest);
            var tsc = new TaskCompletionSource<HttpResponseMessage>();
            tsc.SetResult(response);
            return tsc.Task;
        }
        return base.SendAsync(request, cancellationToken);
    }
}
```

该处理程序的有趣之处是：检查 displayMessage 是否是"error"，如果是"error"，就创建要返回的响应，把状态设置为 Bad Request。接下来的两行代码只创建了要返回的 Task。注意响应在 HttpResponseMessage 任务中通过 SetResult 方法设置。

添加定制处理程序有许多理由。设置处理程序管道，是为了添加多个处理程序。除了默认的处理程序之外，还有 DelegatingHandler，它执行一些代码，再把调用委托给内部或下一个处理程序。HttpClientHandler 是最后一个处理程序，它把请求发送到地址。图 26-1 显示了管道。每个添加的 DelegatingHandler 都调用下一个或内部的处理程序，最后一个是基于 HttpClientHandler 的处理程序。

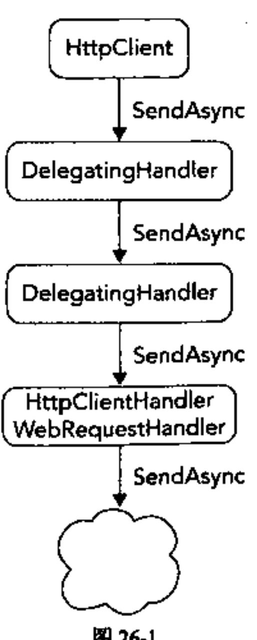


图 26-1

26.3 把输出结果显示为 HTML 页面

前面几个示例说明了.NET 基类如何简化从 Web 上下载和处理数据。但是迄今为止，从 Web 上

下载的文件都是以纯文本显示的。人们总是希望以 Internet Explorer 的界面样式查看 HTML 文件，其中呈现的 HTML 允许用户查看 Web 文档的实际面貌。但是，Microsoft 的 Internet Explorer 并没有.NET 版本，但这并不意味着，不能完成这个任务。

在.NET Framework 2.0 发布之前，可以引用封装了 Internet Explorer 的 COM(Component Object Model，组件对象模型)对象，使用.NET 交互操作功能，把应用程序用作浏览器。自从.NET Framework 2.0 发布以来，就可以在 Windows Forms 应用程序中使用内置的 WebBrowser 控件。

WebBrowser 控件封装了 COM 对象，甚至可以更方便地完成以前复杂的任务。除了使用 WebBrowser 控件之外，另一个选项是使用编程功能，从代码中调用 Internet Explorer 实例。

如果不使用新的 WebBrowser 控件，就可以使用 System.Diagnostics 名称空间中的 Process 类，通过编程打开 Internet Explorer 进程，导航到给定的网页。

```
Process myProcess = new Process();
myProcess.StartInfo.FileName = "iexplore.exe";
myProcess.StartInfo.Arguments = "http://www.wrox.com";
myProcess.Start();
```

但是，上面的代码会把 Internet Explorer 作为单独的窗口打开，而应用程序并没有与新窗口相连接，因此不能控制浏览器。

另一方面，使用 WebBrowser 控件，可以把浏览器作为应用程序的一个集成部分来显示和控制。新的 WebBrowser 控件相当复杂，提供了许多方法、属性和事件。

26.3.1 从应用程序中进行简单的 Web 浏览

为了简单起见，首先创建一个 Windows Forms 应用程序，它只有一个 TextBox 控件和一个 WebBrowser 控件。构建该应用程序，让最终用户在文本框中输入一个 URL，并按回车键。WebBrowser 控件就会提取网页，并显示得到的文档。

在 Visual Studio 2013 设计器中，应用程序如图 26-2 所示。在这个应用程序中，最终用户输入 URL 并按回车键后，这个按键操作就会通过应用程序进行注册，然后 WebBrowser 控件就会开始检索请求的网页，在该控件中显示它。

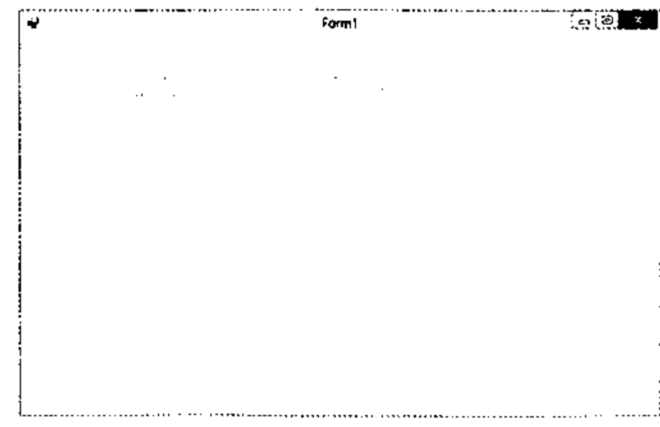


图 26-2

该应用程序的代码如下所示：

```
using System;
```

```
using System.Windows.Forms;

namespace Browser
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)13)
            {
                webBrowser1.Navigate(textBox1.Text);
            }
        }
    }
}
```

在这个示例中，最终用户在文本框中按下的每个键都会被 textBox1_KeyPress 事件捕获。如果输入的字符是一个回车键(按回车键，其键码是(char)13)，就用 WebBrowser 控件采取行动。使用 WebBrowser 控件的 Navigate()方法，并通过 textBox1.Text 属性指定 URI(指定为字符串)，最终结果如图 26-3 所示。

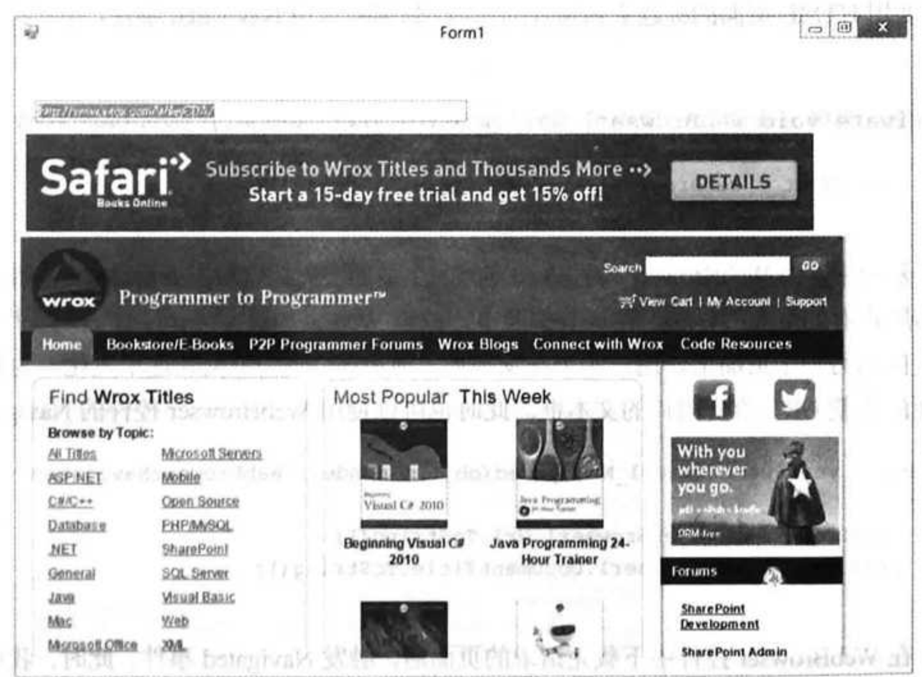


图 26-3

26.3.2　启动 Internet Explorer 实例

读者可能对上一节描述的把浏览器放在应用程序内部不感兴趣，而只对让用户在一般的浏览器中查找网站感兴趣(例如，单击应用程序中的一个链接)。为了演示这个功能，创建一个 Windows Forms

应用程序，其中有一个LinkLabel控件。例如，可以在窗体上放置一个LinkLabel控件，显示“Visit our company website!”。

有了这个控件后，就可以使用下面的代码，在一个单独的浏览器中启动公司的网站，而不是直接在应用程序的窗体中启动：

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
    WebBrowser wb = new WebBrowser();
    wb.Navigate("http://www.wrox.com", true);
}
```

在这个示例中，用户单击LinkLabel控件时，就会新建WebBrowser类的一个实例。然后使用WebBrowser类的Navigate()方法，代码指定了网页的位置和一个布尔值，该布尔值表示是在Windows Forms应用程序内部打开这个端点(其值为false)，还是从一个单独的浏览器中打开这个端点(其值为true)。它默认设置为false。在前面的构造过程中，当最终用户单击Windows应用程序中的链接时，就实例化一个浏览器实例，并启动Wrox网站www.wrox.com。

26.3.3 给应用程序提供更多IE类型的功能

在前面的例子中，直接在Windows Forms应用程序中使用WebBrowser控件时，单击页面中包含的链接，TextBox控件中的文本不会更新，因此不能显示浏览过程中所在的准确位置的URL。要更正这个错误，应侦听WebBrowser控件中的事件，给控件添加处理程序。

为此，要用HTML页面的标题更新窗体的标题。只需要使用Navigated事件，并更新窗体的Text属性即可：

```
private void webBrowser1_Navigated(object sender, EventArgs e)
{
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

在这个示例中，当WebBrowser控件移动到另一个页面上时，就触发Navigated事件，并把窗体的标题改为所查看的页面的标题。在一些情况下，处理Web上的页面时，即使输入了指定的地址，也会被重定向到另一个页面上。用户希望在窗体的文本框(地址栏)中反映这个变化。为此，应根据所查看页面的完整URL改变窗体的文本框。此时也可以使用WebBrowser控件的Navigated事件：

```
private void webBrowser1_Navigated(object sender, WebBrowserNavigatedEventArgs e)
{
    textBox1.Text = webBrowser1.Url.ToString();
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

这里，在WebBrowser控件中下载完请求的页面后，触发Navigated事件。此时，我们只需要把textBox1控件的Text值更新为页面的URL即可。也就是说，页面加载到WebBrowser控件的HTML容器中后，如果URL在这个过程中发生变化(例如有一个重定向过程)，新的URL就会显示在文本框中。如果使用这些步骤并导航到Wrox网站(http://www.wrox.com)，页面的URL会立即改为http://www.wrox.com/WileyCDA/。这个过程也说明，如果最终用户单击了HTML视图中包含的一个链接，就也会在文本框中显示新请求页面的URL。

如果在进行了这些修改后运行应用程序，窗体的标题和地址栏就会像 Microsoft 的 Internet Explorer 一样变化，如图 26-4 所示。

接着，创建 IE 样式的工具栏，让最终用户更多地控制 WebBrowser 控件。这样，就可以使用 Back、Forward、Stop、Refresh 和 Home 等按钮。

图 26-4

这里不使用 ToolBar 控件，而是在窗体顶部地址栏的上面添加一组 Button 控件。在控件的顶部添加 5 个按钮，如图 26-5 所示。

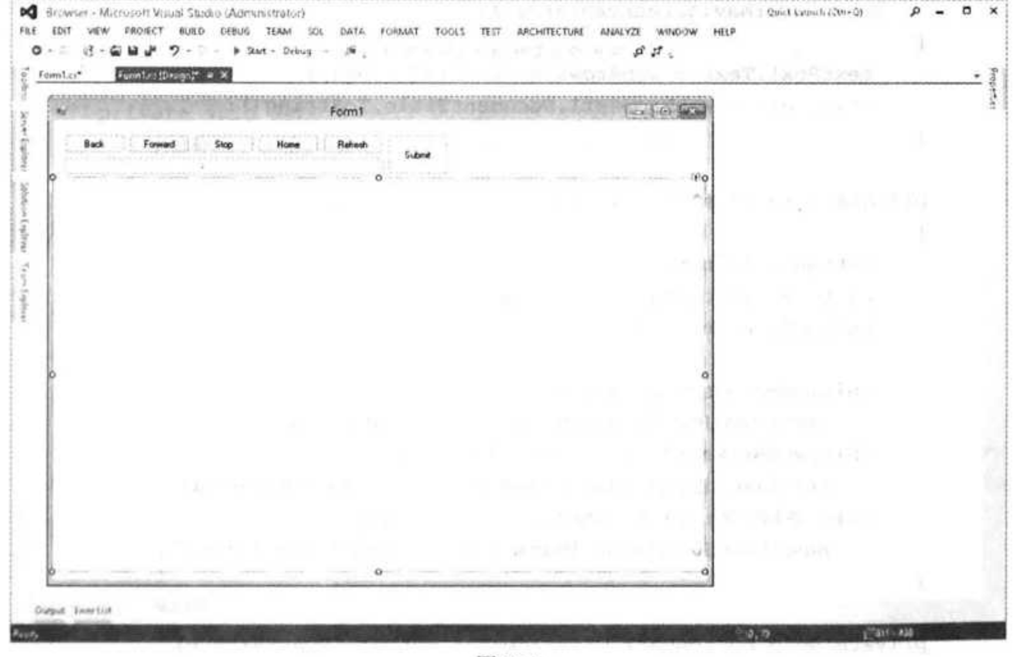


图 26-5

在这个示例中，修改按钮上的文本，以显示按钮的作用。当然，还可以使用屏幕捕捉实用程序，“借用”IE 的按钮图像。按钮应命名为 buttonBack、buttonForward、buttonStop、buttonRefresh 和 buttonHome。为了能重置大小，应确保把右边 3 个按钮的 Anchor 属性设置为 Top, Right。

开始时，buttonBack、buttonForward 和 buttonStop 应是禁用的，因为如果没有在 WebBrowser 控件中加载初始页面，就不能使用这些按钮。以后应告诉应用程序，根据用户在页面栈中的位置，何时启用、禁用 Back 按钮和 Forward 按钮。另外，在加载页面时，需要启用 Stop 按钮，在页面加载完毕后，需要禁用 Stop 按钮。在页面上再添加一个 Submit 按钮，用于提交所请求的 URL。

下面首先给按钮添加功能。因为 WebBrowser 类有我们需要的所有方法，所以这一操作很简单：

```
using System;
using System.Windows.Forms;

namespace Browser
{
    partial class Form1: Form
    {
        public Form1()
        {
           InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)13)
            {
               webBrowser1.Navigate(textBox1.Text);
            }
        }

        private void webBrowser1_Navigated(object sender,
           WebBrowserNavigatedEventArgs e)
        {
            textBox1.Text = webBrowser1.Url.ToString();
            this.Text = webBrowser1.DocumentTitle.ToString();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            buttonBack.Enabled = false;
            buttonForward.Enabled = false;
            buttonStop.Enabled = false;

            this.webBrowser1.CanGoBackChanged +=
               new EventHandler(webBrowser1_CanGoBackChanged);
            this.webBrowser1.CanGoForwardChanged +=
               new EventHandler(webBrowser1_CanGoForwardChanged);
            this.webBrowser1.DocumentTitleChanged +=
               new EventHandler(webBrowser1_DocumentTitleChanged);
        }

        private void buttonBack_Click(object sender, EventArgs e)
        {
```

```
        webBrowser1.GoBack();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonForward_Click(object sender, EventArgs e)
    {
        webBrowser1.GoForward();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonStop_Click(object sender, EventArgs e)
    {
        webBrowser1.Stop();
    }

    private void buttonHome_Click(object sender, EventArgs e)
    {
        webBrowser1.GoHome();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonRefresh_Click(object sender, EventArgs e)
    {
       webBrowser1.Refresh();
    }

    private void buttonSubmit_Click(object sender, EventArgs e)
    {
        webBrowser1.Navigate(textBox1.Text);
    }

    private void webBrowser1_Navigating(object sender,
        WebBrowserNavigatingEventArgs e)
    {
        buttonStop.Enabled = true;
    }

    private void webBrowser1_DocumentCompleted(object sender,
        WebBrowserDocumentCompletedEventArgs e)
    {
        buttonStop.Enabled = false;
        if (webBrowser1.CanGoBack)
        {
           buttonBack.Enabled = true;
        }
        else
        {
           buttonBack.Enabled = false;
        }
        if (webBrowser1.CanGoForward)
        {
           buttonForward.Enabled = true;
        }
        else
        {
           buttonForward.Enabled = false;
```

```
                }
            }
        }
    }
```

在这个示例中要执行许多不同的操作，因为最终用户在使用这个应用程序时有那么多的选项。首先，对于每个按钮单击事件，WebBrowser 类都有一个特定的方法来启动该操作。例如，对于窗体上的 Back 按钮，可以仅使用 WebBrowser 控件的 GoBack()方法；对于 Forward 按钮，要使用 GoForward()方法；对于其他按钮，则要使用 Stop()、Refresh()和 GoHome()方法。所以，很容易创建工具栏，其提供的操作类似于 Microsoft 的 Internet Explorer 中的工具栏。

在第一次加载窗体时，Form1_Load 事件禁用相应的按钮。此时，最终用户可以在文本框中输入一个 URL，并单击 Submit 按钮，让应用程序检索相应的页面。

为了管理按钮的启用和禁用，必须输入一组事件。如前所述，只要开始下载，就需要启用 Stop 按钮。为此，只需要给 Navigating 事件添加事件处理程序，以启用 Stop 按钮：

```
private void webBrowser1_Navigating(object sender,
   WebBrowserNavigatingEventArgs e)
{
   buttonStop.Enabled = true;
}
```

接着，文档加载完毕后，再次禁用 Stop 按钮：

```
private void webBrowser1_DocumentCompleted(object sender,
   WebBrowserDocumentCompletedEventArgs e)
{
   buttonStop.Enabled = false;
}
```

启用、禁用相应的 Back 按钮和 Forward 按钮，实际上依赖于在页面栈中后退或前进的功能。这是使用 CanGoForwardChanged 和 CanGoBackChanged 事件实现的：

```
private void webBrowser1_CanGoBackChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoBack)
    {
       buttonBack.Enabled = true;
    }
    else
    {
       buttonBack.Enabled = false;
    }
}

private void webBrowser1_CanGoForwardChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoForward)
    {
       buttonForward.Enabled = true;
    }
    else
    {
```

```
            buttonForward.Enabled = false;
        }
    }
```

现在运行项目，访问一个网页，并单击几个链接，还应能使用工具栏，增进浏览体验。最终结果如图 26-6 所示。

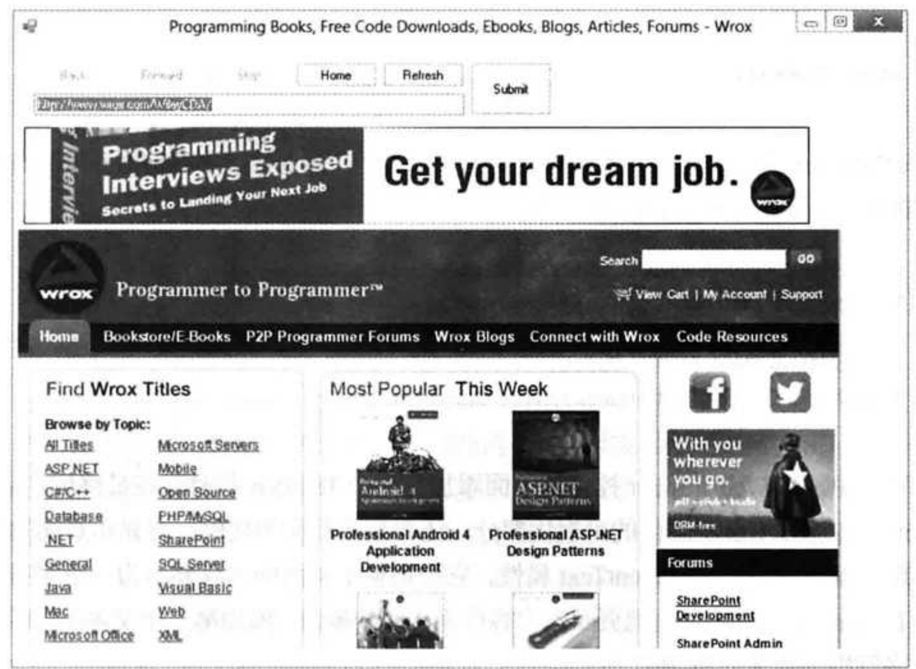


图 26-6

26.3.4　使用 WebBrowser 控件打印

用户不仅可以使用 WebBrowser 控件查看页面和文档，还可以使用 WebBrowser 控件把这些页面和文档发送到打印机上进行打印。要打印在 WebBrowser 控件中查看的页面或文档，只需要使用下面的构造：

```
webBrowser1.Print();
```

与以前相同，不必查看页面或文档，就可以打印它。例如，可以使用 WebBrowser 类加载 HTML 文档并打印它，而无须显示已加载的文档，其代码如下所示：

```
WebBrowser wb = new WebBrowser();
wb.Navigate("http://www.wrox.com");
wb.Print();
```

26.3.5　显示所请求页面的代码

在本章的开头，我们使用 WebRequest 类和 Stream 类访问一个远程页面，以显示所请求页面的代码。下面的代码也可以完成这个任务：

```
public Form1()
{
    InitializeComponent();
    System.Net.WebClient Client = new WebClient();
```

```
        Stream strm = Client.OpenRead("http://www.reuters.com");
        StreamReader sr = new StreamReader(strm);
        string line;

        while ( (line=sr.ReadLine()) != null )
        {
            listBox1.Items.Add(line);
        }

        strm.Close();
    }
```

使用 WebBrowser 控件，这个任务就更容易完成。为此，需要修改本章前面开发的浏览器应用程序，只需要在 Document_Completed 事件中添加一行代码，如下所示：

```
    private void webBrowser1_DocumentCompleted(object sender,
        WebBrowserDocumentCompletedEventArgs e)
    {
        buttonStop.Enabled = false;
        textBox2.Text = webBrowser1.DocumentText;
    }
```

在应用程序中，在 WebBrowser 控件的下面添加另一个 TextBox 控件。在最终用户请求页面时，不仅要在 TextBox 控件中显示页面的可视化部分，还要显示页面的代码。要显示页面的代码，只需要使用 WebBrowser 控件的 DocumentText 属性，它会把整个页面的内容显示为一个字符串。另一个选项是使用 DocumentStream 属性把页面的内容作为一个数据流。添加第二个文本框，可以把页面的内容显示为字符串，结果如图 26-7 所示。

图 26-7

26.3.6 WebRequest 类和 WebResponse 类的层次结构

本节详细讨论 WebRequest 类和 WebResponse 类的底层体系结构。图 26-8 显示了相关类的继承层次结构。

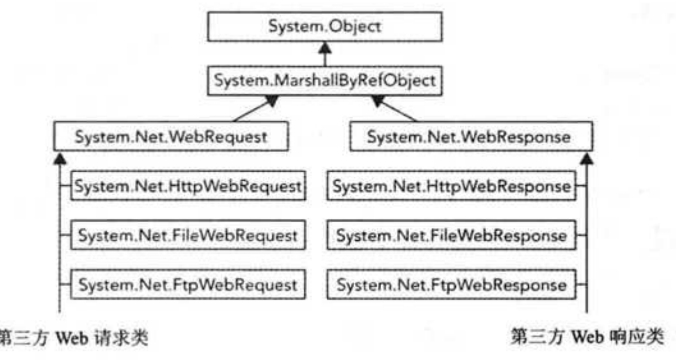


图 26-8

这个层次结构不仅仅包含刚才在代码中使用的两个类。实际上，还应该知道 WebRequest 类和 WebResponse 类都是抽象的，不能进行实例化。这些基类提供了用于处理 Web 请求和响应的通用功能，这些功能独立于给定操作所使用的协议。请求总是通过某一特定协议(如 HTTP、FTP、SMTP 等)实现，并由为该协议编写的派生类处理相应请求。Microsoft 将该方案称为“可插入协议”。

在 26.2 节的代码中，变量定义为对基类的引用，但是 WebRequest.Create()方法实际上给出了一个 HttpWebRequest 对象，而 GetResponse()方法实际上返回一个 HttpWebResponse 对象。这个基于工厂的机制在客户端代码中隐藏了许多细节，以支持基于相同代码的各种协议。

有了 WebRequest.Create()方法，在 URI 中就不需要专门用于处理 HTTP 协议的对象。WebRequest.Create()方法检查 URI 中的协议说明符，以实例化和返回一个适当类的对象。这样代码就不必了解所使用的派生类或特定协议的信息。在需要访问协议的特定功能时，应使用派生类的属性和方法，此时要把 WebRequest 或 WebResponse 引用强制转换为派生类。

有了这个体系结构，就应能使用任意通用协议发送请求。但是，Microsoft 目前提供的派生类只适用于 HTTP、HTTPS、FTP 和 FILE 协议。.NET Framework 自从 2.0 版本以来才支持 FTP 选项。如果要利用其他的协议，如 SMTP，则需要使用 WCF(替代了 Windows API)或 SmtpClient 对象。

26.4 实用工具类

本节将讨论一些实用工具类，它们在处理 URI 和 IP 地址时可简化 Web 编程。

26.4.1 URI

Uri 和 UriBuilder 是 System(注意：不是 System.Net)名称空间中的两个类，它们都用于表示 URI。UriBuilder 类允许把给定的字符串当作 URI 的组成部分，从而构建一个 URI；而 Uri 类允许分析、组合和比较 URI。

对于 Uri 类，构造函数需要一个完整的 URI 字符串：

```
Uri MSPage = new

Uri("http://www.Microsoft.com/SomeFolder/SomeFile.htm?Order=true");
```

Uri 类提供了许多只读属性。当 Uri 对象构造出来之后，它就不能修改了。

```
string Query = MSPage.Query;                      // ?Order=true;
string AbsolutePath = MSPage.AbsolutePath;        // /SomeFolder/SomeFile.htm
string Scheme = MSPage.Scheme;                    // http
int Port = MSPage.Port;                           // 80 (the default for http)
string Host = MSPage.Host;                        // www.microsoft.com
bool IsDefaultPort = MSPage.IsDefaultPort;        // true since 80 is default
```

然而，UriBuilder 类实现的属性较少：只允许构建一个完整的 URI。这些属性可读写。可以给构造函数提供构建 URI 所需的各个组件：

```
UriBuilder MSPage = new
   UriBuilder("http", "www.microsoft.com", 80, "SomeFolder/SomeFile.htm");
```

或者把值赋给属性，以此构建 URI 的组件：

```
UriBuilder MSPage = new UriBuilder();
MSPage.Scheme ="http";
MSPage.Host = "www.microsoft.com";
MSPage.Port = 80;
MSPage.Path = "SomeFolder/SomeFile.htm";
```

在完成 UriBuilder 类的初始化后，就可以使用 Uri 属性获得相应的 Uri 对象：

```
Uri CompletedUri = MSPage.Uri;
```

26.4.2　IP 地址和 DNS 名称

在 Internet 上，服务器和客户端都由 IP 地址或主机名(也称作 DNS 名称)标识。通常，主机名是在 Web 浏览器的窗口中输入的友好名称，如 www.wrox.com 或 www.microsoft.com 等。另一方面，IP 地址是计算机用于互相识别的标识符，它实际上是用于确保 Web 请求和响应到达相应计算机的地址。一台计算机甚至可以有多个 IP 地址。

目前，IP 地址一般是一个 32 位的值，例如 192.168.1.100 就是一个 32 位的 IP 地址。IP 地址的这个格式称为 IPv4。目前有许多计算机和其他设备在竞争 Internet 上的一个地点，所以人们开发了一种较新的地址：IPv6。IPv6 提供了 64 位的 IP 地址。IPv6 至多可以提供 3×10^{28} 个不同的地址。.NET Framework 允许应用程序同时使用 IPv4 和 IPv6。

为了使这些主机名发挥作用，首先必须发送一个网络请求，把主机名翻译成 IP 地址，翻译工作由一个或几个 DNS 服务器完成。DNS 服务器中保存的一个表把主机名映射为它知道的所有计算机的 IP 地址以及其他 DNS 服务器的 IP 地址，这些 DNS 服务器用于在该表中查找它不知道的主机名。本地计算机至少要知道一个 DNS 服务器。网络管理员在设置计算机时配置该信息。

在发送请求之前，计算机首先应要求 DNS 服务器指出与输入的主机名相对应的 IP 地址。找到正确的 IP 地址后，计算机就可以定位请求，并通过网络发送它。所有这些工作一般都在用户浏览 Web 时在后台进行。

1. 用于IP地址的.NET类

.NET Framework提供了许多能够帮助寻找IP地址和主机信息的类。

IPAddress类

IPAddress类代表IP地址。地址本身可以作为GetAddressBytes属性，并使用ToString()方法转化为用小数点隔开的十进制格式。此外，IPAddress类也实现静态的Parse()方法，这个方法的作用与ToString()方法正好相反，把小数点隔开的十进制字符串转化为IP地址：

```
IPAddress ipAddress = IPAddress.Parse("234.56.78.9");
byte[] address = ipAddress.GetAddressBytes();
string ipString = ipAddress.ToString();
```

在上面的示例中，byte整型数address的值是IP地址的二进制表示，字符串ipString的值为文本“234.56.78.9”。

IPAddress类还提供了许多静态的常量字段，以返回特殊的地址。例如，Loopback地址允许计算机给它自己发送消息，而Broadcast地址允许多播到本地网络上。

```
// The following line will set loopback to "127.0.0.1".
// the loopback address indicates the local host.
string loopback = IPAddress.Loopback.ToString();

// The following line will set broadcast address to "255.255.255.255".
// the broadcast address is used to send a message to all machines on
// the local network.
string broadcast = IPAddress.Broadcast.ToString();
```

IPHostEntry类

IPHostEntry类用于封装与某台特定的主机相关的信息。通过这个类的HostName属性(这个属性返回一个字符串)，可以使用主机名；通过AddressList属性返回一个IPAddress对象数组。下一个示例DNSLookupResolver将使用IPHostEntry类。

Dns类

Dns类能够与默认的DNS服务器进行通信，以检索IP地址。Dns类有两个重要的静态方法：Resolve()方法和GetHostByAddress()方法。给Resolve()方法提供主机名，Resolve()方法就可以使用DNS服务器获取主机的详细信息；给GetHostByAddress()方法提供IP地址，GetHostByAddress()方法也可以返回主机的详细信息。这两个方法都返回一个IPHostEntry对象。

```
IPHostEntry wroxHost = Dns.Resolve("www.wrox.com");
IPHostEntry wroxHostCopy = Dns.GetHostByAddress("208.215.179.178");
```

在这段代码中，两个IPHostEntry对象将包含wrox.com服务器的详细信息。

Dns类与IPAddress类和IPHostEntry类的不同之处在于：Dns类实际上可以与服务器进行通信，以获取有关信息；而IPAddress类和IPHostEntry类只是包含许多便利属性的简单数据结构，可以访问底层的数据。

2. DnsLookup 示例

下面通过查找 DNS 名称的示例 DnsLookup 来阐明与 DNS 和 IP 相关的类，如图 26-9 所示。

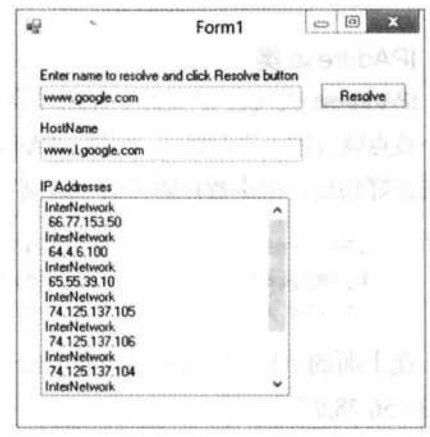


图 26-9

该示例应用程序让用户在主文本框中输入 DNS 名称。当用户单击 Resolve 按钮时，这个示例就使用 Dns.Resolve()方法检索 IPHostEntry 引用，并显示主机名和 IP 地址。注意，显示的主机名也许与输入的名称不同，如果一个 DNS 名称仅作为另一个 DNS 名称的代理，就会发生这种情况。

DnsLookup 应用程序是一个标准的 C# Windows 应用程序。给这个应用程序添加如图 26-9 所示的控件，这些控件分别名为 textBoxInput、btnResolve、txtBoxHostName 和 listboxIPs。然后，把下面的方法添加到 Form1 类中，作为 btnResolve 的 Click 事件处理程序。

```
void btnResolve_Click (object sender, EventArgs e)
{
  try
  {
    IPHostEntry iphost = Dns.GetHostEntry(txtBoxInput.Text);
    foreach (IPAddress ip in iphost.AddressList)
    {
      string ipaddress = ip.AddressFamily.ToString();
      listBoxIPs.Items.Add(ipaddress);
      listBoxIPs.Items.Add(" " + ip.ToString());
    }
    txtBoxHostName.Text = iphost.HostName;
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to process the request because " +
      "the following problem occurred:\n" +
      ex.Message, "Exception occurred");
  }
}
```

注意在这段代码中如何捕获异常。如果用户输入了无效的 DNS 名称，或者网络处于断开状态，就会产生异常。

在检索到 IPHostEntry 实例之后，使用它的 AddressList 属性获取包含 IP 地址的数组，再用 foreach 循环遍历该数组。对于每一项，都使用 IPAddress.AddressFamily.ToString()方法把 IP 地址显示为整数和字符串。

26.5 较低层的协议

本节简要介绍一些用于在较低层次上进行通信的.NET 类。System.Net.Sockets 名称空间包含一

些相关类。例如，这些类允许直接发送 TCP 网络请求或在某个特定端口上侦听 TCP 网络请求。其中主要的类如表 26-1 所示。

表 26-1

类	用 途
Socket	这个低层的类用于管理连接。WebRequest、TcpClient 和 UdpClient 等类在内部使用这个类
NetworkStream	这个类是从 Stream 派生的，它表示来自网络的数据流
SmtpClient	允许通过 SMTP 发送消息(邮件)
TcpClient	允许创建和使用 TCP 连接
TcpListener	允许侦听引入的 TCP 连接请求
UdpClient	用于为 UDP 客户创建连接(UDP 是 TCP 的一种替代协议，但它没有得到广泛的使用，主要用于本地网络)

网络的通信分为几个不同的层次，本章迄今为止讨论的类都工作在最高层，即处理某些特定命令的一层。如果考虑使用 FTP 传输文件，这个概念就非常容易理解。尽管目前的 GUI 应用程序隐藏了许多 FTP 细节，但在命令行提示符上执行 FTP 还是不久之前的事情。在这个环境中，显式地输入一些要发送至服务器的命令，以下载、上传和列出文件。

FTP 并不是依赖于文本命令的唯一高层协议，HTTP、SMTP、POP 和其他协议都基于相似的文本命令。同样，许多现代的图形工具隐藏了命令的传输过程，因此用户一般意识不到这些命令的存在。例如，在 Web 浏览器中输入 URL 和把 Web 请求发送给服务器时，浏览器实际上发送给服务器的是一条纯文本的 GET 命令，这条命令的作用与 FTP 的 get 命令相似。此外，浏览器也可以发送 POST 命令，该命令表示浏览器在请求上附有其他数据。

但是，这些协议本身都不足以实现计算机之间的通信。即使客户和服务器都理解某个协议，如 HTTP，它们仍然不能互相理解，除非另外有协议说明字符是如何传输的，即使用的是什么二进制格式。更进一步说，认真考虑最低层问题，什么电压用于代表二进制数据中的 0 和 1？这些问题都需要通过协议配置和规定它们，因此网络领域的开发人员和硬件工程师通常要查阅协议栈。在列出两个主机进行通信所需的各种协议和机制时，创建一个协议栈，其中既有最高层的协议，也有最底层的协议。这种方法利用模块化和分层的方式获得了高效的通信。

幸运的是，对于大多数的开发工作，我们都不需要使用协议栈或处理电压级别。但是，如果要编写代码，该代码需要在计算机之间进行高效的通信，则需要编写可以直接在计算机之间发送二进制数据包的代码。这是 TCP 等协议的领域，Microsoft 提供的几个类都允许方便地在该层次上操作二进制数据。

26.5.1 使用 SmtpClient

SmtpClient 对象可以通过 SMTP 传送邮件消息。使用 SmtpClient 对象的一个简单示例如下：

```
SmtpClient sc = new SmtpClient("mail.mySmtpHost.com");
sc.Send("evjen@yahoo.com", "editor@wrox.com",
   "The latest chapter", "Here is the latest.");
```

在其最简单的形式中，使用了 SmtpClient 对象的一个实例。在这个例子中，该实例还提供 SMTP 服务器的主机，该 SMTP 服务器用来在 Internet 上发送邮件消息。还可以使用 Host 属性完成相同的任务：

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
sc.Send("evjen@yahoo.com", "editor@wrox.com",
   "The latest chapter", "Here is the latest.");
```

有了 SmtpClient 对象后，就可以调用 Send()方法，提供 From 地址、To 地址、主题以及邮件的消息正文。

在许多情况下，邮件消息都比这里的示例复杂。为此，还可以给 Send()方法传递一个 MailMessage 对象：

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
sc.Send(mm);
```

使用 MailMessage 对象可以细调构建邮件消息的方式。我们可以发送 HTML 消息，添加任意多个 To 和 CC 收件人，修改消息的优先级，使用消息编码，以及添加附件。添加附件的功能在下面的代码片段中定义：

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
Attachment att = new Attachment("myExcelResults.zip",
   MediaTypeNames.Application.Zip);
mm.Attachments.Add(att);
sc.Send(mm);
```

这段代码创建了一个 Attachment 对象，使用 Add()方法给 MailMessage 对象添加该对象，之后调用 Send()方法。

26.5.2　使用 TCP 类

传输控制协议(TCP)类为连接和发送两个端点之间的数据提供了简单的方法。端点是 IP 地址和端口号的组合。已有的协议很好地定义了端口号，例如，HTTP 使用端口 80，而 SMTP 使用端口 25。Internet 地址编码分配机构(即 IANA，http://www.iana.org/)把端口号赋予这些已知的服务。除非实现某个已知的服务，否则应选择大于 1024 的端口号。

TCP 流量构成了目前 Internet 上的主要流量。TCP 通常是首选的协议，因为它提供了有保证的传输、错误校正和缓冲。TcpClient 类封装了 TCP 连接，提供了许多属性来控制连接，包括缓冲、缓冲区的大小和超时。通过 GetStream()方法请求 NetworkStream 对象可以实现读写功能。

TcpListener 类用 Start()方法侦听引入的 TCP 连接。当连接请求到达时，可以使用 AcceptSocket()方法返回一个套接字，与远程计算机通信，或使用 AcceptTcpClient()方法通过高层的 TcpClient 对象进行通信。阐明 TcpListener 类和 TcpClient 类如何协同工作的最简单方式是给出一个示例。

26.5.3　TcpSend 和 TcpReceive 示例

为了说明这两个类的工作原理，需要构建两个应用程序。第一个应用程序是 TcpSend，如图 26-10 所示。这个应用程序打开一个到服务器的 TCP 连接，并发送 C#源代码。

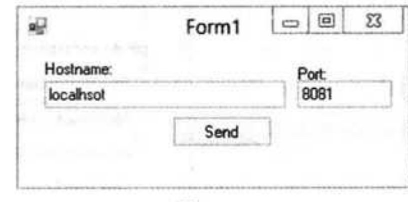


图 26-10

与前面一样，创建一个 C# Windows 应用程序，其中的窗体包含两个文本框(txtHost 和 txtPort)，分别用于主机名和端口。该窗体还有一个按钮(btnSend)，单击它可以启动一个连接。首先，确保包含相关的名称空间：

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Windows.Forms;
```

按钮的 Click 事件处理程序如下面的代码所示：

```
private void btnSend_Click(object sender, System.EventArgs e)
{
  TcpClient tcpClient = new TcpClient(txtHost.Text, Int32.Parse(txtPort.Text));
  NetworkStream ns = tcpClient.GetStream();
  FileStream fs = File.Open("form1.cs", FileMode.Open);

  int data = fs.ReadByte();

  while(data != -1)
  {
    ns.WriteByte((byte)data);
    data = fs.ReadByte();
  }

  fs.Close();
  ns.Close();
  tcpClient.Close();
}
```

这个示例用主机名和端口号创建 TcpClient 类。或者，如果有 IPEndPoint 类的一个实例，就可以把该实例传递给 TcpClient 类的构造函数。在检索到 NetworkStream 类的一个实例后，打开源代码文件，并开始读取字节。与许多二进制流一样，这里也需要将 ReadByte()方法的返回值和－1 相比较，以确定是否到达流的末尾。循环读取所有的字节并把它们发送给网络流后，就应关闭所有打开的文件、连接和流。

在连接的另一端，TcpReceive 应用程序显示传输完成后接收到的文件，如图 26-11 所示。

```
Form1
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace TCPSend
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            TcpClient tcpClient = new TcpClient(textBox1.Text, Int32.Parse(textBox2.Text));
            NetworkStream ns = tcpClient.GetStream();
            FileStream fs = File.Open("form1.cs", FileMode.Open);

            int data = fs.ReadByte();

            while (data != -1)
            {
                ns.WriteByte((byte)data);
                data = fs.ReadByte();
            }
```

图 26-11

该窗体只包含一个 TextBox 控件 txtDisplay。TcpReceive 应用程序使用 TcpListener 等待引入的连接。为了避免应用程序界面的冻结，我们使用一个后台线程来等待，然后从连接中读取。因此还需要包含 System.Threading 名称空间以及下面这些名称空间：

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Windows.Forms;
```

在窗体的构造函数中，添加一个后台线程：

```
public Form1()
{
   InitializeComponent();
   Thread thread = new Thread(new ThreadStart(Listen));
   thread.Start();
}
```

其他重要的代码如下所示：

```
public void Listen()
{
   IPAddress localAddr = IPAddress.Parse("127.0.0.1");
   Int32 port = 2112;
   TcpListener tcpListener = new TcpListener(localAddr, port);
   tcpListener.Start();

   TcpClient tcpClient = tcpListener.AcceptTcpClient();

   NetworkStream ns = tcpClient.GetStream();
   StreamReader sr = new StreamReader(ns);
   string result = sr.ReadToEnd();
   Invoke(new UpdateDisplayDelegate(UpdateDisplay),new object[] {result} );
   tcpClient.Close();
   tcpListener.Stop();
}

public void UpdateDisplay(string text)
{
   txtDisplay.Text= text;
}

protected delegate void UpdateDisplayDelegate(string text);
```

该线程在 Listen()方法中开始执行，并允许在不暂停界面的情况下阻塞对 AcceptTcpClient()方法的调用。注意这里把 IP 地址 127.0.0.1 和端口号 2112 硬编码到应用程序中，因此需要从客户端应用程序中输入相同的端口号。

我们使用 AccepTcpClient()方法返回的 TcpClient 对象打开一个新流，进行读取。与本章前面的示例类似，创建一个 StreamReader，把引入的网络数据转换为字符串。在关闭客户端和停止侦听器前，更新窗体的文本框。因为我们不想从后台线程中直接访问文本框，所以使用窗体的 Invoke()方法和一个委托，把得到的字符串作为 object 参数数组的第一个元素传递。Invoke()方法可确保用户的调用正确地编组到线程中，该线程拥有用户界面中的控制句柄。

26.5.4 TCP 和 UDP

本节要介绍的另一个协议是 UDP(用户数据报协议)。UDP 是一个几乎没有什么功能的简单协议，且几乎没有什么系统开销。开发人员常常在速度和性能要求比可靠性更高的应用程序中使用 UDP，如视频流。相反，TCP 提供了许多功能来确保数据的传输，它还提供了错误校正以及当数据丢失或数据包损坏时重新传输它们的功能。最后，TCP 可缓冲传入和传出的数据，还保证在传输过程中，在把数据包传送给应用程序之前重组杂乱的一系列数据包。即使有一些额外的开销，TCP 仍是在 Internet 上使用最广泛的协议，因为它有非常高的可靠性。

26.5.5 UDP 类

可以看出，与 TcpClient 类相比，UdpClient 类提供了一个较小、较简单的接口。这反映出 UDP 协议相对简单的本质。尽管 TCP 类和 UDP 类都在后台使用套接字，但 UdpClient 类不包含返回一个网络流以读写数据的方法。相反，成员函数 Send()把一个字节数组作为参数，Receive()函数则返回一个字节数组。另外，因为 UDP 是一个无连接的协议，所以可以指定把通信的端点作为 Send()方法和 Receive()方法的一个参数，而不是在前面的构造函数或 Connect()方法中指定。也可以在某个

后续的发送或接收过程中修改端点。

下面的代码段使用 UdpClient 类给回显服务(echo service)发送一条消息。带有回显服务的服务器在端口 7 处接受 TCP 或 UDP 连接。回显服务只把发送给服务器的数据再发送回客户端。这个服务可用于诊断和测试(尽管许多系统管理员从安全的角度考虑而禁用回显服务)。

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace Wrox.ProCSharp.InternetAccess.UdpExample
{

  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      UdpClient udpClient = new UdpClient();
      string sendMsg = "Hello Echo Server";
      byte [] sendBytes = Encoding.ASCII.GetBytes(sendMsg);
      udpClient.Send(sendBytes, sendBytes.Length, "SomeEchoServer.net", 7);
      IPEndPoint endPoint = new IPEndPoint(0,0);
      byte [] rcvBytes = udpClient.Receive(ref endPoint);
      string rcvMessage = Encoding.ASCII.GetString(rcvBytes,
                                                   0,
                                                   rcvBytes.Length);
      // should print out "Hello Echo Server"
      Console.WriteLine(rcvMessage);
    }
  }
}
```

Encoding.ASCII 类常常用于把字符串转换为字节数组，或把字节数组转换为字符串。还要注意，IPEndPoint 应通过引用传递给 Receive()方法。因为 UDP 不是一个面向连接的协议，对 Receive()方法的每次调用都会从不同的端点读取数据，所以 Receive()方法会用发送主机的 IP 地址和端口填充该参数。UdpClient 类和 TcpClient 类在最低层的 Socket 类上提供了一个抽象层。

26.5.6 Socket 类

Socket 类提供了网络编程中最高级的控制。说明该类最简单的方式是用 Socket 类重写 TcpReceive 应用程序。更新后的 Listen()方法如下例所示：

```
public void Listen()
{
  Socket listener = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream,
                               ProtocolType.Tcp);
  listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
  listener.Listen(0);
  Socket socket = listener.Accept();
  Stream netStream = new NetworkStream(socket);
  StreamReader reader = new StreamReader(netStream);
```

```
    string result = reader.ReadToEnd();
    Invoke(new UpdateDisplayDelegate(UpdateDisplay),
           new object[] {result} );
    socket.Close();
    listener.Close();
}
```

Socket类需要再编写几行代码来完成相同的任务。对于初学者，构造函数的参数需要为使用TCP协议的流套接字指定IP寻址方案。这些参数只是可用于Socket类的许多组合之一，TcpClient类会配置这些设置。接着把侦听器的套接字绑定到一个端口上，开始侦听引入的连接。当引入的请求到达时，就可以使用Accept()方法创建一个新的套接字来处理该连接。最后为套接字附加一个StreamReader实例来读取引入的数据，其方式与前面大致相同。

Socket类也包含许多方法，用于异步地接受、连接、发送和接收数据。通过回调委托使用这些方法的方式与前面用WebRequest类请求异步页面的方式相同。如果确实需要了解套接字的内部情况，就可以使用GetSocketOption()方法和SetSocketOption()方法，它们允许查看和配置各种选项，包括超时、生存期和其他低级选项。

1. 构建服务器控制台应用程序

为了进一步探讨Socket类，下一个例子创建一个控制台应用程序，该应用程序作为引入的套接字请求的服务器。再创建第二个例子(另一个控制台应用程序)，它把一条消息发送给服务器控制台应用程序。

构建的第一个应用程序是用作服务器的控制台应用程序，它会在指定的TCP端口上打开一个套接字，侦听传入的消息。该控制台应用程序的代码如下：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsole
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Starting: Creating Socket object");
            Socket listener = new Socket(AddressFamily.InterNetwork,
                                         SocketType.Stream,
                                         ProtocolType.Tcp);
            listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
            listener.Listen(10);

            while (true)
            {
                Console.WriteLine("Waiting for connection on port 2112");
                Socket socket = listener.Accept();
                string receivedValue = string.Empty;
```

```
                while (true)
                {
                    byte[] receivedBytes = new byte[1024];
                    int numBytes = socket.Receive(receivedBytes);
                    Console.WriteLine("Receiving .");
                    receivedValue += Encoding.ASCII.GetString(receivedBytes,
                                         0, numBytes);
                    if (receivedValue.IndexOf("[FINAL]") > -1)
                    {
                        break;
                    }
                }
                Console.WriteLine("Received value: {0}", receivedValue);
                string replyValue = "Message successfully received.";
                byte[] replyMessage = Encoding.ASCII.GetBytes(replyValue);
                socket.Send(replyMessage);
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();
            }
            listener.Close();
        }
    }
}
```

这个例子使用Socket类建立了一个套接字。该套接字使用TCP协议，并通过端口2112接收从任意IP地址引入的消息。把通过打开的套接字接收到的值写入控制台屏幕。这个消费应用程序会继续接收字节，直到接收到[FINAL]字符串为止。这个[FINAL]字符串表示引入的消息的末尾，之后就可以解释消息了。

从客户端上接收到消息的末尾后，就把一条回应消息发送给该客户端。之后，使用Close()方法关闭套接字，控制台应用程序继续等待接收新的消息。

2. 构建客户端应用程序

下一步是构建一个客户端应用程序，该应用程序给第一个控制台应用程序发送一条消息。客户端只要遵循该应用程序建立的规则，就可以给服务器控制台应用程序发送任意消息。第一条规则是服务器控制台应用程序只使用特定的协议侦听。本例中的服务器应用程序使用TCP协议侦听消息。另一条规则是服务器应用程序只侦听特定的端口，对于本例是端口2112。最后一条规则是对于任意正在发送的消息，消息的最后一位必须以字符串[FINAL]结尾。

下面的客户端控制台应用程序遵循这些规则：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsoleClient
{
    class Program
    {
        static void Main()
        {
```

```
            byte[] receivedBytes = new byte[1024];
            IPHostEntry ipHost = Dns.Resolve("127.0.0.1");
            IPAddress ipAddress = ipHost.AddressList[0];
            IPEndPoint ipEndPoint = new IPEndPoint(ipAddress, 2112);
            Console.WriteLine("Starting: Creating Socket object");

            Socket sender = new Socket(AddressFamily.InterNetwork,
                                       SocketType.Stream,
                                       ProtocolType.Tcp);
            sender.Connect(ipEndPoint);
            Console.WriteLine("Successfully connected to {0}",
                  sender.RemoteEndPoint);
            string sendingMessage = "Hello World Socket Test";
            Console.WriteLine("Creating message: Hello World Socket Test");
            byte[] forwardMessage = Encoding.ASCII.GetBytes(sendingMessage
               + "[FINAL]");
            sender.Send(forwardMessage);
            int totalBytesReceived = sender.Receive(receivedBytes);
            Console.WriteLine("Message provided from server: {0}",
                              Encoding.ASCII.GetString(receivedBytes,
                              0, totalBytesReceived));
            sender.Shutdown(SocketShutdown.Both);
            sender.Close();
            Console.ReadLine();
        }
    }
}
```

在这个例子中，使用 localhost 的 IP 地址和服务器控制台应用程序要求的端口 2112 创建一个 IPEndPoint 对象。这里创建了一个套接字，并调用了 Connect()方法。打开套接字并连接到服务器控制台应用程序的套接字实例后，使用 Send()方法把一个文本字符串发送给服务器应用程序。因为服务器应用程序会返回一条消息，所以使用 Receive()方法获取这条消息(把它放在一个字节数组中)。之后，将字节数组转换为一个字符串，并把它显示在控制台应用程序上，最后关闭套接字。

运行这个应用程序，结果如图 26-12 所示。

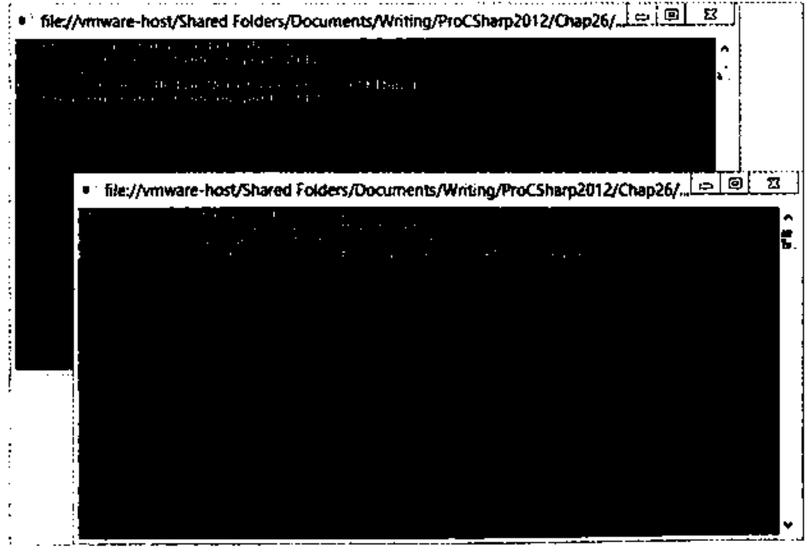


图 26-12

查看图 26-12 中的两个控制台应用程序，会发现服务器应用程序打开并等待引入的消息。引入的消息从客户端应用程序中发送。接着，所发送的字符串由服务器应用程序显示。在接收和显示第一条消息后，服务器应用程序会等待其他消息的传入。关闭客户端应用程序，然后再次运行服务器应用程序，就会看到这一点。接着会看到服务器应用程序再次显示接收到的消息。

26.5.7 WebSocket

WebSocket 协议用于完全双工的双向通信。这种通信一般在浏览器和 Web 服务器之间进行，但仅交流那些支持使用 WebSocket 协议的客户端信息。WebSocket API 由 W3C 标准化，该协议已由 IEIF(Internet Engineering Task Force)在 RFC 6455 中标准化。

与浏览器和 Web 服务器使用的请求/响应模型不同，WebSocket 维持一个打开的连接。TCP 发送的是字节流，而 WebSocket 是在服务器和客户端之间来回发送消息。

并不是所有的浏览器和 Web 服务器都支持 WebSocket 协议。目前，Firefox 11.0(MozWebSocket)、Google Chrome 16 和 Internet Explorer 10 提供了这种浏览器支持。对于服务器，带有 ASP.NET 4.5 的 IIS 8 提供了低级 WebSocket 支持。

Chat 示例

WebSocket 端点可以使用任意类型的处理程序或模块来创建。下例使用一个.ashx 处理程序作为端点。该示例是一个使用浏览器和 Web 服务器的简单聊天程序，每个客户端或用户都连接到 Web 服务器上，提供其名称，以便在聊天服务器上“注册”，接着就可以给在该服务器上注册的其他用户发送简单的文本消息了。

首先是用于浏览器的代码。这是一个非常简单的 HTML 页面，它使用 jQuery 建立 WebSocket。jQuery 提供了在页面上处理 WebSocket 事件的简单方式。

```
<!doctype html>
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <script src="Scripts/jquery-1.7.2.min.js" type="text/javascript"></script>
    <title>WroxChat</title>
    <script type="text/javascript">
        $(document).ready(function () {
            var name = prompt('what is your name?:');
            var url = 'ws://localhost/ws.ashx?name=' + name;
            ws = new WebSocket(url);
            ws.onopen = function () {
                $('#messages').prepend('Connected <br/>');
                $('#cmdSend').click(function () {
                    ws.send($('#txtMessage').val());
                    $('#txtMessage').val('');
                });
            };
            ws.onmessage = function (e) {
                 $('#chatMessages').prepend(e.data + '<br/>');
            };
            $('#cmdLeave').click(function () {
                 ws.close();
            });
```

```
            ws.onclose = function () {
                $('#chatMessages').prepend('Closed <br/>');
            };
            ws.onerror = function (e) {
                $('#chatMessages').prepend('Oops something went wrong <br/>');
            };
        });
    </script>
</head>
<body>
<input id="txtMessage" />
<input id="cmdSend" type="button" value="Send" />
<input id="cmdLeave" type="button" value="Leave" />
<br />
<div id="chatMessages" />
</body>
```

这个示例放在 localhost 上。url 变量可以改为驻留该示例的任意有效 URL。

代码行 ws = new WebSocket(url);建立了浏览器和服务器之间的连接。WebSocket 类触发 onopen 事件时，就定义 cmdSend 单击事件的处理程序，它调用 WebSocket 对象的 Send 方法。

在这个示例中处理的其他 WebSocket 事件有 ommessage、onclose 和 onerror。消息发送给浏览器时调用 ommessage，中断 WebSocket 连接时调用 onclose，发生异常时调用 onerror。

在服务器上事情就复杂多了。对于这个示例，需要创建一个简单的 ChatUser 对象。对于每个在服务器上注册的用户，都要把一个 ChatUser 对象放在 IList<ChatUser>中。把消息发送给服务器时，它会广播给 IList<ChatUser>列表中的每个用户。

IHttpHandler 完成这个工作。处理 ProcessRequest 方法时，会创建新用户，并把该用户添加到列表中。最后，它调用 ChatUser 对象的 HandleWebSocket 方法。这就完成了在浏览器和服务器之间建立连接的工作。

下面是 IHttpHandler 类 ws.ashx 的 HandleWebSocket 方法的代码：

```
public void ProcessRequest(HttpContext context)
{
    if (context.IsWebSocketRequest)
    {
        var chatuser = new ChatUser();
        chatuser.UserName = context.Request.QueryString["name"];
        ChatApp.AddUser(chatuser);
        context.AcceptWebSocketRequest(chatuser.HandleWebSocket);
    }
}
```

如上所示，创建了新的 ChatUser，其名称根据来自浏览器的查询参数进行设置，把该用户添加到列表中，再调用 HandleWebSocket 方法。

HandleWebSocket 方法负责处理消息，其代码如下：

```
public async Task HandleWebSocket(WebSocketContext wsContext)
{
    _context = wsContext;
    const int maxMessageSize = 1024;
    byte[] receiveBuffer = new byte[maxMessageSize];
```

```
        WebSocket socket = _context.WebSocket;
        while (socket.State == WebSocketState.Open)
        {
            WebSocketReceiveResult receiveResult =
                    await socket.ReceiveAsync(new ArraySegment<byte>(receiveBuffer),
                                                    CancellationToken.None);

            if (receiveResult.MessageType == WebSocketMessageType.Close)
            {
                await socket.CloseAsync(WebSocketCloseStatus.NormalClosure,
                                        string.Empty,
                                        CancellationToken.None);
            }
            else if (receiveResult.MessageType == WebSocketMessageType.Binary)
            {
                await socket.CloseAsync(WebSocketCloseStatus.InvalidMessageType,
                                        "Cannot accept binary frame",
                                        CancellationToken.None);
            }
            else
            {
                var receivedString = Encoding.UTF8.GetString(receiveBuffer,
                                                            0,
                                                            receiveResult.Count);
                var echoString = string.Concat(UserName,
                                                " said: ",
                                                receivedString);
                ArraySegment<byte> outputBuffer =
                    new ArraySegment<byte>(Encoding.UTF8.GetBytes(echoString));
                ChatApp.BroadcastMessage(echoString);
            }
        }
    }
```

请求到达时，首先需要确保套接字连接是打开的。从传入的 WebSocketContext 对象中获取该套接字，然后检查其 State 属性。

接着，调用 ReceiveAsync 方法，返回 WebSocketReceiveResult 对象。在该对象中可以确定消息是一个关闭消息还是二进制消息。如果发送的是关闭消息，就关闭连接——本例只能发送文本。

ReceiveAsync 调用中的一个参数是 receiveBuffer，它是一个字节数组，要用消息数据来填充。在功能比较全面的聊天程序中，需要确保消息没有超过最大尺寸限制。

现在该处理消息了。因为这是一个字节数组，所以需要把数据转换为文本格式。为此，要使用 Encoding.UTF8.GetString 方法，它会提取字节数组，返回消息的字符串表示。我们把发送消息的用户名和 ChatApp 类的 Broadcast 方法调用连接起来。

Broadcast 方法迭代所有的 ChatUser 对象，并调用 SendMessage 方法，其代码如下：

```
public async Task SendMessage(string message)
{
    if (_context != null && _context.WebSocket.State == WebSocketState.Open)
    {
        var outputBuffer = new ArraySegment<byte>(
                                            Encoding.UTF8.GetBytes(message));
```

```
            await _context.WebSocket.SendAsync(
                                    outputBuffer,
                                    WebSocketMessageType.Text,
                                    true,
                                    CancellationToken.None);
        }
    }
```

SendMessage 方法分析消息字符串，并把它放回字节数组。接着，这个字节数组作为参数发送给 SendAsync 方法。_context 变量是用户第一次在聊天程序中注册时创建的 WebSocketContext，所以把消息发送回活动的连接。因为页面中的 JavaScript 在监听 DOM 中 WebSocket 对象的 onMessage 事件，所以会接收到该消息，并显示在页面上。

26.6 小结

本章回顾了 System.Net 名称空间中用于网络通信的.NET Framework 类。从中可了解到，某些.NET 基类可处理在网络和 Internet 上打开的客户端连接，如何给服务器发送请求和从服务器接收响应(最常见的应用就是接收 HTML 页面)。利用 WebBrowser 控件，可以轻松地从桌面应用程序中使用 Internet Explorer。

作为经验规则，在使用 System.Net 名称空间中的类编程时，应尽可能一直使用最通用的类。例如，使用 TCPClient 类代替 Socket 类，可以把代码与许多低级套接字细节分离开来。更进一步，WebRequest 类允许利用.NET Framework 中的可插入协议体系结构。代码应利用新的应用程序级协议，因为 Microsoft 和其他第三方开发人员引入了新功能。

最后，我们讨论了网络类中异步功能的使用，该功能给 Windows Forms 应用程序提供了用户响应界面的专业外观。

第27章

Windows 服务

本章要点

- Windows 服务的体系结构
- Windows 服务的安装程序
- Windows 服务的控制程序
- Windows 服务的故障排除

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Quote 服务器
- Quote 客户端
- Quote 服务
- 服务控制

27.1 Windows 服务

Windows 服务是可以在系统启动时自动打开(不需要任何人登录计算机)的程序。如果需要在没有用户交互操作的情况下运行程序，或者在权限比交互式用户更大的用户下运行程序，就可以创建 Windows 服务。Windows 服务的例子有 WCF 宿主(假定由于某些原因不能使用 IIS)、缓存网络服务器中数据的程序，或者在后台重新组织本地磁盘数据的程序。

本章首先讨论 Windows 服务的体系结构。接着创建一个托管网络服务器的 Windows 服务，之后讨论 Windows 服务的启动、监控、控制和故障排除。

如前所述，Windows 服务指的是操作系统启动时可以自动打开的应用程序。Windows 服务可以在没有交互式用户登录系统的情况下运行，在后台进行某些处理。

例如，在 Windows Server 上，系统网络服务应可以从客户端访问，无须用户登录到服务器上。

在客户端系统上，服务可以从 Internet 上获取新软件版本，或在本地磁盘上进行文件清理工作。

可以把 Windows 服务配置为从已经过特殊配置的用户账户或系统用户账户上运行，该用户账户的权限比系统管理员的权限更大。

除非特别说明，否则把 Windows 服务简称为服务。

下面是一些服务的示例：

- Simple TCP/IP Services 是驻留一些小型 TCP/IP 服务器的服务程序，如 echo、daytime 和 quote 等。
- World Wide Publishing Service 是 IIS(Internet Information Server，Internet 信息服务器)的服务。
- Event Log 服务用于把消息记录到事件日志系统中。
- Windows Search 服务用于在磁盘上创建数据的索引。
- SuperFetch 服务可以把常用的应用程序和库预先加载到内存中，因此缩短了这些应用程序的启动时间。

可以使用 Services 管理工具查看系统上的所有服务，如图 27-1 所示。这个程序可以通过控制面板上的管理工具找到。

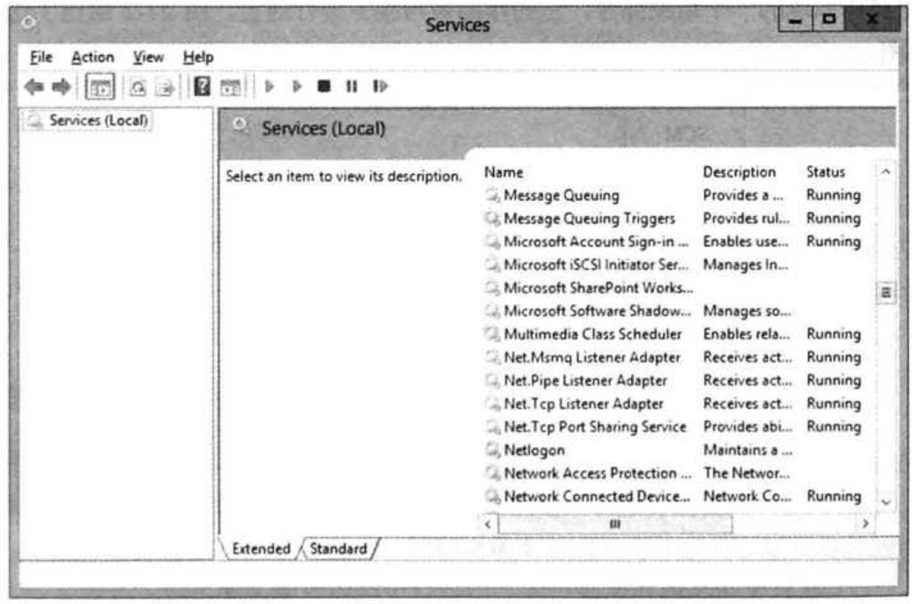


图 27-1

27.2 Windows 服务的体系结构

操作 Windows 服务需要 3 种程序：

- 服务程序
- 服务控制程序
- 服务配置程序

服务程序本身用于提供需要的实际功能。服务控制程序可以把控制请求发送给服务，如开始、停止、暂停和继续。使用服务配置程序可以安装服务，这意味着不但要把服务复制到文件系统中，还要把服务的信息写到注册表中，这个注册信息由服务控制管理器(SCM)用于启动和停止服务。尽管.NET组件可通过xcopy安装——因为.NET组件不需要把信息写入注册表中，所以可以使用xcopy命令安装它们；但是，服务的安装需要注册表配置。此外，服务配置程序也可以在以后改变服务的配置。下面介绍Windows服务的3个组成部分。

27.2.1 服务程序

在讨论服务的.NET实现方式之前，本节首先讨论服务的Windows体系结构和服务的内部功能。

服务程序实现服务的功能。服务程序需要3个部分：

- 主函数
- service-main函数
- 处理程序

在讨论这些部分前，首先需要介绍服务控制管理器(Service Control Manager，SCM)。对于服务，SCM的作用非常重要，它可以把启动服务或停止服务的请求发送给服务。

1. 服务控制管理器

SCM是操作系统的一个组成部分，它的作用是与服务进行通信。图27-2给出了这种通信工作方式的序列图。

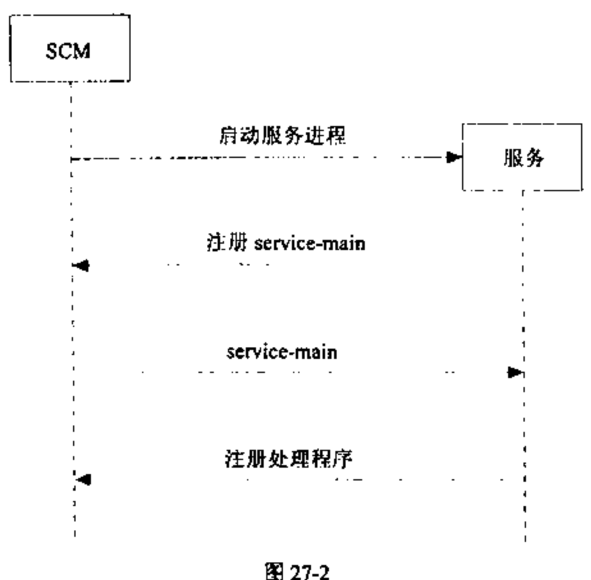


图 27-2

如果将服务设置为自动启动，则在系统启动时，将启动该服务的每个进程，进而调用该进程的主函数。该服务负责为它的每项服务都注册一个service-main函数。主函数是服务程序的入口点，在这里，service-main函数的入口点必须用SCM注册。

2. 主函数、service-main和处理程序

服务的主函数是程序的一般入口点，即Main()方法，它可以注册多个service-main函数，service-main

函数包含服务的实际功能。服务必须为所提供的每项服务注册一个 service-main 函数。服务程序可以在一个程序中提供许多服务，例如，<windows>\system32\services.exe 服务程序就包括 Alerter、Application Management、Computer Browser 和 DHCP Client 等服务项。

SCM 为每一个应该启动的服务调用 service-main 函数。service-main 函数的一个重要任务是用 SCM 注册一个处理程序。

处理程序函数是服务程序的第 3 部分。处理程序必须响应来自 SCM 的事件。服务可以停止、暂停或重新开始，处理程序必须响应这些事件。

使用 SCM 注册处理程序后，服务控制程序可以把停止、暂停和继续服务的请求发送给 SCM。服务控制程序独立于 SCM 和服务本身。在操作系统中有许多服务控制程序，例如以前介绍的 MMC Services 管理单元。也可以编写自己的服务控制程序，一个比较好的服务控制程序是 SQL Server Configuration Manager，如图 27-3 所示。

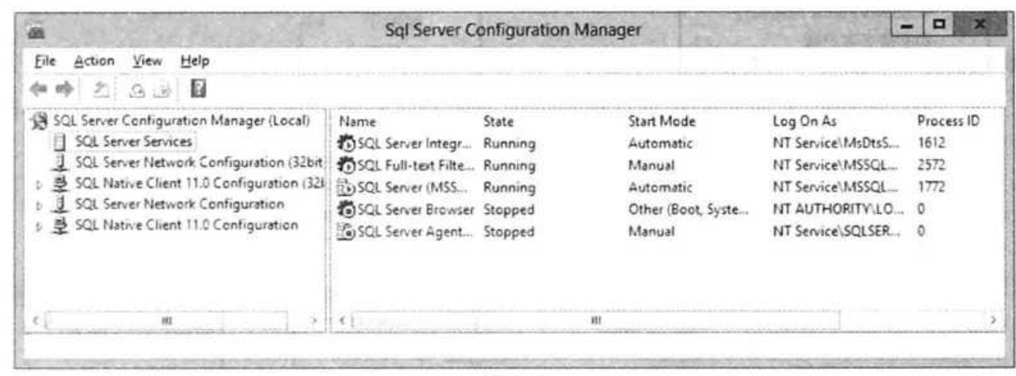


图 27-3

27.2.2 服务控制程序

顾名思义，使用服务控制程序可以控制服务。为了停止、暂停和继续服务，可以把控制代码发送给服务，处理程序应该响应这些事件。此外，还可以询问服务的实际状态(假定服务在运行或挂起，或者在某种错误的状态下)，并实现一个响应自定义控制代码的自定义处理程序。

27.2.3 服务配置程序

不能使用 xcopy 安装服务，服务必须在注册表中配置。注册表包含了服务的启动类型，该启动类型可以设置为自动、手动或禁用。必须配置服务程序的用户、服务的依赖关系(例如，一个服务必须在当前服务开始之前启动)。所有这些配置工作都在服务配置程序中进行。虽然安装程序可以使用服务配置程序配置服务，但是服务配置程序也可以用于在以后改变服务配置参数。

27.2.4 Windows 服务的类

在.NET Framework 中，可以在 System.ServiceProcess 名称空间中找到实现服务的 3 部分的服务类：

- 必须从 ServiceBase 类继承才能实现服务。ServiceBase 类用于注册服务、响应开始和停止请求。
- ServiceController 类用于实现服务控制程序。使用这个类，可以把请求发送给服务。
- 顾名思义，ServiceProcessInstaller 类和 ServiceInstaller 类用于安装和配置服务程序。

下面介绍怎样新建服务。

27.3 创建 Windows 服务程序

本章创建的服务将驻留在引用服务器内。对于客户发出的每一个请求，引用服务器都返回引用文件的一个随机引用。解决方案的第一部分由 3 个程序集完成，一个用于客户端，两个用于服务器，图 27-4 显示了这个解决方案。程序集 QuoteServer 包含实际的功能。服务可以在内存缓存中读取引用，然后在套接字服务器的帮助下响应引用的请求。QuoteClient 是 WPF 胖客户端应用程序。这个应用程序创建客户端套接字，以便与 Quote Server 进行通信。第 3 个程序集是实际的服务。Quote Service 开始和停止 QuoteServer，服务将控制服务器。

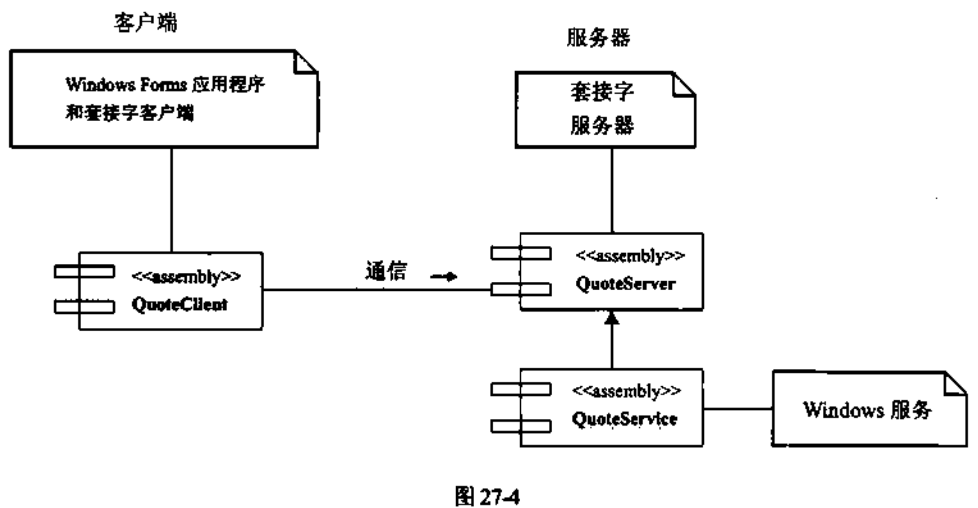


图 27-4

在创建程序的服务部分之前，在额外的 C#类库(在服务进程中使用这个类库)中建立一个简单的套接字服务器。具体步骤参见下一节。

27.3.1 创建服务的核心功能

可以在 Windows 服务中建立任何功能，如扫描文件以进行备份或病毒检查，或者启动 WCF 服务器。但所有服务程序都有一些类似的地方。这种程序必须能启动(并返回给调用者)、停止和暂停。下面讨论用套接字服务器实现的程序。

对于 Windows 8，Simple TCP/IP Services 可以作为 Windows 组件的一个组成部分安装。Simple TCP/IP Services 的一部分是“quote of the day”或 qotd TCP/IP 服务器，这个简单的服务在端口 17 处侦听，并使用文件<windir>\system32\drivers\etc\quotes 中的随机消息响应每一个请求。使用这个示例服务，我们将在这里构建一个相似的服务器，它返回一个 Unicode 字符串，而不是像 qotd 服务器那样返回 ASCII 代码。

首先创建一个 QuoteServer 类库，并实现服务器的代码。下面详细解释 QuoteServer.cs 文件中 QuoteServer 类的源代码：

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
```

```
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.WinServices
{
  public class QuoteServer
  {
    private TcpListener listener;
    private int port;
    private string filename;
    private List<string> quotes;
    private Random random;
    private Task listenerTask;
```

重载 QuoteServer()构造函数，以便把文件名和端口传递给主调程序。只传递文件名的构造函数使用服务器的默认端口 7890。默认的构造函数把引用的默认文件名定义为 quotes.txt：

```
public QuoteServer()
   : this ("quotes.txt")
{
}
public QuoteServer(string filename)
   : this(filename, 7890)
{
}
public QuoteServer(string filename, int port)
{
  Contract.Requires<ArgumentNullException>(filename != null);
  Contract.Requires<ArgumentException>(port >= IPEndpoint.MinPort &&
    port<= IPEndPoint.MaxPort);

  this.filename = filename;
  this.port = port;
}
```

ReadQuotes()是一个辅助方法，它从构造函数指定的文件中读取所有引用，把所有引用添加到 List<string> quotes 中。此外，创建 Random 类的一个实例，Random 类用于返回随机引用：

```
protected void ReadQuotes()
{
  try
  {
    quotes = File.ReadAllLines(filename).ToList();
    if (quotes.Count == 0)
    {
      throw new QuoteException("quotes file is empty");
    }
    random = new Random();
  }
  catch (IOException ex)
  {
```

```
        throw new QuoteException("I/O Error", ex);
    }
}
```

另一个辅助方法是 GetRandomQuoteOfTheDay()，它返回引用集合中的一个随机引用：

```
protected string GetRandomQuoteOfTheDay()
{
  int index = random.Next(0, quotes.Count);
  return quotes[index];
}
```

在 Start()方法中，使用辅助函数 ReadQuotes()在 List<string>引用中读取包含引用的完整文件。在启动新的线程之后，它立即调用 Listener()方法。这类似于第 26 章的 TcpReceive 示例。

这里使用了任务，因为 Start()方法不能停下来等待客户，它必须立即返回给调用者(即 SCM)。如果方法没有及时返回给调用者(30 秒)，SCM 就假定启动失败。侦听任务是一个长时间运行的后台线程，应用程序就可以在不停止该线程的情况下退出。

```
public void Start()
{
  ReadQuotes();

  listenerTask = Task.Factory.StartNew(Listener,
                     TaskCreationOptions.LongRunning);
}
```

任务函数 Listener()创建一个 TcpListener 实例。AcceptSocket()方法等待客户端进行连接。客户端一连接，AcceptSocket()方法就返回一个与客户端相关联的套接字。之后使用 socket.Send()方法，调用 GetRandomQuoteOfTheDay()方法把返回的随机引用发送给客户端：

```
protected void Listener()
{
  try
  {
    IPAddress ipAddress = IPAddress.Any;
    listener = new TcpListener(ipAddress, port);
    listener.Start();
    while (true)
    {
      Socket clientSocket = listener.AcceptSocket();
      string message = GetRandomQuoteOfTheDay();
      var encoder = new UnicodeEncoding();
      byte[] buffer = encoder.GetBytes(message);
      clientSocket.Send(buffer, buffer.Length, 0);
      clientSocket.Close();
    }
  }
  catch (SocketException ex)
  {
    Trace.TraceError(String.Format("QuoteServer {0}", ex.Message));
    throw new QuoteException("socket error", ex);
  }
}
```

除了 Start()方法之外，还需要如下方法来控制服务：Stop()、Suspend()和 Resume()。

```
public void Stop()
{
   listener.Stop();
}
public void Suspend()
{
   listener.Stop();
}
public void Resume()
{
   Start();
}
```

另一个公共方法是 RefreshQuotes()。如果包含引用的文件发生了变化，就要使用这个方法重新读取文件：

```
        public void RefreshQuotes()
        {
           ReadQuotes();
        }
    }
}
```

在服务器上建立服务之前，首先应该建立一个测试程序，这个测试程序仅创建 QuoteServer 类的一个实例，并调用 Start()方法。这样，不需要处理与具体服务相关的问题，就能够测试服务的功能。测试服务器必须手动启动，使用调试器可以很容易调试代码。

测试程序是一个 C#控制台应用程序 TestQuoteServer，我们必须引用 QuoteServer 类的程序集。包含引用的文件必须复制到 c:\ProCSharp\services 目录中(或者必须在构造函数中改动参数，以指定在什么地方复制文件)。在调用构造函数之后，就调用 QuoteServer 实例的 Start()方法。Start()方法在创建线程之后立即返回，因此在按回车键之前，控制台应用程序一直处于运行状态(代码文件 TestQuoteServer/Program.cs)。

```
static void Main()
{
  var qs = new QuoteServer("quotes.txt", 4567);
  qs.Start();
  Console.WriteLine("Hit return to exit");
  Console.ReadLine();
  qs.Stop();
}
```

注意，QuoteServer 示例将运行在使用这个程序的本地主机的 4567 端口上——后面的内容需要在客户端中使用这些设置。

27.3.2 QuoteClient 示例

客户端是一个简单的 WPF Windows 应用程序，可以在此请求来自服务器的引用。客户端应用程序使用 TcpClient 类连接到正在运行的服务器，然后接收返回的消息，并把它显示在文本框中。用户界面仅包含一个按钮和一个文本框。单击按钮，就向服务器请求引用，并显示该引用。

给按钮的 Click 事件指定 OnGetQuote()方法，以向服务器请求引用，并将 IsEnable 属性绑定到 EnableRequest 方法上，在请求激活时禁用按钮。把文本框的 Text 属性绑定到 Quote 属性上，以显示所设置的引用：

```
<Button Margin="3" VerticalAlignment="Stretch" Grid.Row="0"
       IsEnabled="{Binding EnableRequest}" Click="OnGetQuote">
    Get Quote</Button>
<TextBlock Margin="6" Grid.Row="1" TextWrapping="Wrap"
           Text="{Binding Quote}" />
```

在用户界面中绑定的信息是属性 EnableRequest 和 Quote，在 QuoteInformation 类中定义这两个属性。这个类实现了接口 INotifyPropertyChanged，以允许 WPF 接收属性值的改变：

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Wrox.ProCSharp.WinServices
{
  public class QuoteInformation : INotifyPropertyChanged
  {
    public QuoteInformation()
    {
       EnableRequest = true;
    }
    private string quote;
    public string Quote
    {
      get
      {
         return quote;
      }
      internal set
      {
         SetProperty(ref quote, value);
      }
    }

    private bool enableRequest;
    public bool EnableRequest
    {
      get
      {
         return enableRequest;
      }
      internal set
      {
         SetProperty(ref enableRequest, value);
      }
    }
```

```
    private void SetProperty<T>(ref T field, T value,
                                [CallerMemberName] string propertyName = "")
    {
      if (!EqualityComparer<T>.Default.Equals(field, value))
      {
        field = value;
        var handler = PropertyChanged;
        if (handler != null)
        {
          handler(this, new PropertyChangedEventArgs(propertyName));
        }
      }
    }

    public event PropertyChangedEventHandler PropertyChanged;
  }
}
```

接口 INotifyPropertyChanged 的实现代码使用了特性 CallerMemberNameAttribute，这个特性的解释参见第 16 章。

把类 QuoteInformation 的一个实例赋予 Windows 类 QuoteClientWindow 的 DataContext，以便直接进行数据绑定(代码文件 QuoteClient/MainWindow.xaml.cs)：

```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows;
using System.Windows.Input;

namespace Wrox.ProCSharp.WinServices
{
  public partial class QuoteClientWindow : Window
  {
    private QuoteInformation quoteInfo = new QuoteInformation();

    public QuoteClientWindow()
    {
      InitializeComponent();
      this.DataContext = quoteInfo;
    }
```

在项目的属性中，可以用 Settings 选项卡来配置连接到服务器的服务器名称和端口信息，如图

27-5 所示。这里定义了 ServerName 和 PortName 设置的默认值。把 Scope 设置为 User，该设置就会保存到用户特定的配置文件中，因此应用程序的每个用户都可以有不同的设置。Visual Studio 的 Settings 特性也会创建一个 Settings 类，以便用一个强类型化的类来读写设置。

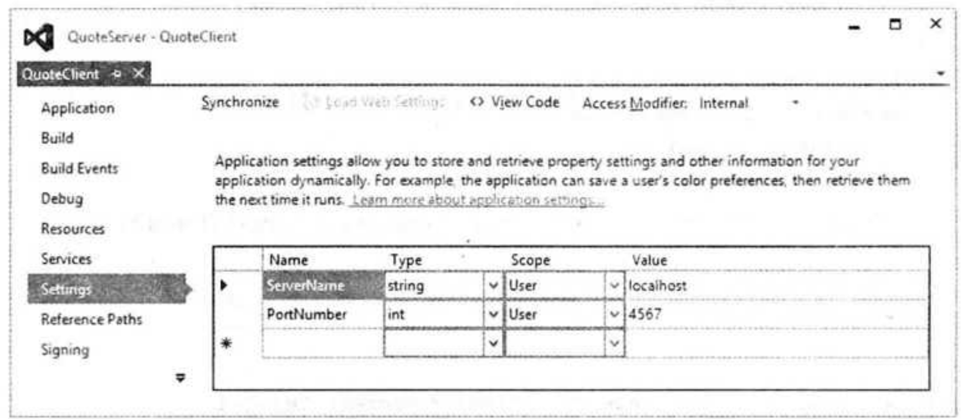


图 27-5

客户端的主要功能体现在 Get Quote 按钮的 Click 事件处理程序中。

```
protected async void OnGetQuote(object sender, RoutedEventArgs e)
{
  const int bufferSize = 1024;
  Cursor currentCursor = this.Cursor;
  this.Cursor = Cursors.Wait;
  quoteInfo.EnableRequest = false;

  string serverName = Properties.Settings.Default.ServerName;
  int port = Properties.Settings.Default.PortNumber;

  var client = new TcpClient();
  NetworkStream stream = null;
  try
  {
    await client.ConnectAsync(serverName, port);
    stream = client.GetStream();
    byte[] buffer = new byte[bufferSize];
    int received = await stream.ReadAsync(buffer, 0, bufferSize);
    if (received <= 0)
    {
      return;
    }
    quoteInfo.Quote = Encoding.Unicode.GetString(buffer).Trim('\0');
  }
  catch (SocketException ex)
  {
    MessageBox.Show(ex.Message, "Error Quote of the day",
        MessageBoxButton.OK, MessageBoxImage.Error);
  }
  finally
  {
    if (stream != null)
    {
```

```
        stream.Close();
      }

      if (client.Connected)
      {
        client.Close();
      }
    }
    this.Cursor = currentCursor;
    quoteInfo.EnableRequest = true;
  }
```

在打开测试服务器和这个 Windows 应用程序的客户端之后，就可以对功能进行测试。如果运行成功，就可以得到如图 27-6 所示的结果。

现在继续在服务器中实现服务功能。程序已经在运行，还需要确保在系统启动时，不需要任何人登录系统，服务器程序就应该自动地启动。为此，可以创建一个服务程序。

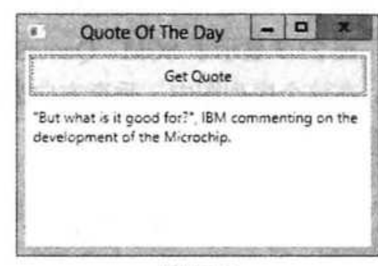


图 27-6

27.3.3 Windows 服务程序

使用 Add New Project 对话框中的 C# Windows Service 模板，就可以创建一个 Windows 服务程序，将该服务命名为 QuoteService。

在单击 OK 按钮开始创建 Windows 服务程序之后，就会出现设计器界面，但是不能在其中插入 UI 组件，因为应用程序不能直接在屏幕上显示任何信息。本章后面将使用设计器界面添加安装对象、性能计数器和事件日志等其他组件。

选择这个服务的属性，可以打开 Properties 对话框。在其中可以配置如下值：

- AutoLog 指定把启动和停止服务的事件自动写到事件日志中。
- CanPauseAndContinue、CanShutdown 和 CanStop 可以指定服务的暂停、继续、关闭和停止请求。
- ServiceName 是写到注册表中的服务的名称，使用这个名称可以控制服务。
- CanHandleSessionChangeEvent 确定服务是否能处理终端服务器会话中的改变事件。
- CanHandlePowerEvent 选项对运行在笔记本电脑或移动设备上的服务有效。如果启用这个选项，服务就可以响应低电源事件，并相应地改变服务的行为。电源事件包括电量低、电源状态改变(因为与 A/C 电源之间的切换)开关和改为断电。

不管项目的名称是什么，默认的服务名称都是 Service1。可以只安装一个 Service1 服务。如果在测试过程中出现了安装错误，就有可能已经安装一个 Service1 服务。因此，在服务开发的初始阶段，一定要用 Properties 对话框把服务的名称改为比较适当的名称。

使用 Properties 对话框改变上述属性，在 InitalizeComponent()方法中设置 ServiceBase 派生类的值。Windows Forms 应用程序也使用 InitalizeComponent()方法，对于服务，这个方法的使用方式与 Windows Forms 应用程序相似。

虽然向导将生成代码，但是应把文件名改为 QuoteService.cs，把名称空间的名称改为 Wrox.ProCSharp.WinServices，把类名改为 QuoteService。后面将详细讨论该服务的代码。

1. ServiceBase 类

ServiceBase 类是所有用.NET Framework 开发的 Windows 服务的基类。QuoteService 类派生自 ServiceBase 类；QuoteService 类使用一个未归档的辅助类 System.ServiceProcess.NativeMethods 与 SCM 进行通信，System.ServiceProcess.NativeMethods 类是 Win32 API 调用的包装类。ServiceBase 是内部类，因此不能在这里的代码中使用它。

图 27-7 显示了 SCM、QuoteService 类和 System.ServiceProcess 名称空间中的类是怎样相互作用的。在这个序列图中，垂直方向为对象的生命线，水平方向为通信情况，通信是按照时间的先后顺序自上而下进行的。

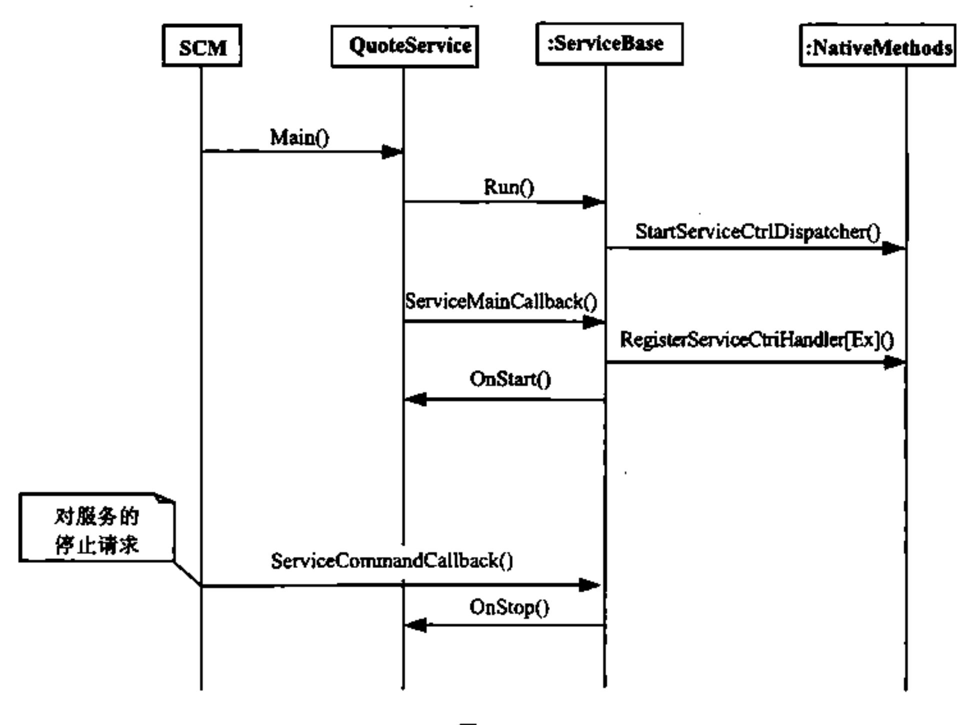


图 27-7

SCM 启动应该启动的服务的进程。在启动时，调用 Main()方法。在示例服务的 Main()方法中，调用 ServiceBase 基类的 Run()方法。Run()方法使用 SCM 中的 NativeMethods.StartServiceCtrlDispatcher()方法注册 ServiceMainCallback()方法，并把记录写到事件日志中。

接下来，SCM 在服务程序中调用已注册的 ServiceMainCallback()方法。ServiceMainCallback()方法本身使用 NativeMethods.RegisterServiceCtrlHandler[Ex]()方法在 SCM 中注册处理程序，并在 SCM 中设置服务的状态。之后调用 OnStart()方法。在 OnStart()方法中，必须实现启动代码。如果 OnStart()方法执行成功，就把字符串“Service Started Sucessful”写到事件日志中。

处理程序是在 ServiceCommandCallback()方法中实现的。当改变对服务的请求时，SCM 就调用 ServiceCommandCallback()方法。ServiceCommandCallback()方法再把请求发送给 OnPause()、OnContinue()、OnStop()、OnCustomCommand()和 OnPowerEvent()方法。

2. 主函数

现在讨论服务进程中由应用程序模板生成的主函数。在主函数中，声明了一个元素为 ServiceBase 类的数组 ServicesToRun。创建 QuoteService 类的一个实例，并将其作为 ServicesToRun 数组的第一个元素传递。如果在这个服务进程中要运行多个服务，就需要把具体服务类的多个实例添加到数组中。然后把 ServicesToRun 数组传递给 ServiceBase 类的静态方法 Run()。使用 ServiceBase 类的 Run()方法，可以把 SCM 引用提供给服务的入口点。服务进程的主线程现在处于阻塞状态，等待服务的结束。

下面是自动生成的代码(代码文件 QuoteService/Program.cs)：

```
/// <summary>
/// The main entry point for the application.
/// </summary>
static void Main()
{
  ServiceBase[] ServicesToRun;
  ServicesToRun = new ServiceBase[]
  {
     new QuoteService()
  };
  ServiceBase.Run(ServicesToRun);
}
```

如果进程中只有一个服务，就可以删除数组。由于 Run()方法接受从 ServiceBase 类派生的单个对象，因此 Main()方法可以简化为：

```
ServiceBase.Run(new QuoteService());
```

服务程序 Services.exe 包含多个服务。如果有类似的服务，其中有多个服务运行在一个进程中，且需要初始化多个服务的某些共享状态，则共享的初始化必须在 Run()方法运行之前完成。在运行 Run()方法时，主线程处于阻塞状态，直到服务进程停止为止，以后的指令在服务结束之前不能执行。

初始化花费的时间不应该超过 30 秒。如果初始化代码所花费的时间过多，SCM 就认为服务启动失败了。初始化时间不应该超过 30 秒必须是针对速度最慢的计算机而言。如果初始化的时间过长，就应该在另一个线程中开始初始化，以便主线程及时地调用 Run()方法。然后，事件对象可以用信号通知线程已经完成了它的工作。

3. 服务的启动

在服务启动时，调用 OnStart()方法。这时，可以启动前面创建的套接字服务器。为了使用 QuoteServer 类，必须引用 QuoteServer 程序集。调用 OnStart()方法的线程不能阻塞，OnStart()方法必须返回给调用者(即 ServiceBase 类的 ServiceMainCallback()方法)。ServiceBase 类注册处理程序，并在调用 OnStart()方法之后把服务成功启动的消息通知给 SCM(代码文件 QuoteService/QuoteService.cs)：

```
protected override void OnStart(string[] args)
```

```
{
  quoteServer = new QuoteServer(Path.Combine(
                    AppDomain.CurrentDomain.BaseDirectory, "quotes.txt"),
                    5678);
  quoteServer.Start();
}
```

把 quoteServer 变量声明为类中的私有成员：

```
namespace Wrox.ProCSharp.WinServices
{
  public partial class QuoteService: ServiceBase
  {
     private QuoteServer quoteServer;
```

4. 处理程序方法

当停止服务时，调用 OnStop()方法。应该在 OnStop()方法中停止服务的功能：

```
protected override void OnStop()
{
   quoteServer.Stop();
}
```

除了 OnStart()和 OnStop()方法之外，还可以重写服务类中的下列处理程序：

- OnPause()——在暂停服务时调用这个方法。
- OnContinue()——当服务从暂停状态返回到正常操作时，调用这个方法。为了调用已重写的 OnPause()方法和 OnContinue()方法，CanPauseAndContinue 属性必须设置为 true。
- OnShutdown()——当 Windows 操作系统关闭时，调用这个方法。通常情况下，OnShutdown()方法的行为应该与 OnStop()方法的实现代码相似。如果需要更多的时间关闭服务，则可以申请更多的时间。与 OnPause()方法和 OnContinue()方法相似，必须设置一个属性启用这种行为，即 CanShutdown 属性必须设置为 true。
- OnPowerEvent()——在系统的电源状态发生变化时，调用这个方法。电源状态发生变化的信息在 PowerBroadcastStatus 类型的参数中，PowerBroadcastStatus 是一个枚举类型，其值是 BatteryLow 和 PowerStatusChange。在这个方法中，还可以获得系统是否要挂起(QuerySuspend)的信息，此时可以同意或拒绝挂起。电源事件详见本章后面的内容。
- OnCustomCommand()——这个处理程序可以为服务控制程序发送过来的自定义命令提供服务。OnCustomCommand()的方法签名有一个用于获取自定义命令编号的 int 参数，编号的取值范围是 128~276，小于 128 的值是为系统预留的值。在我们的服务中，使用自定义命令编号为 128 的命令重新读取引用文件：

```
protected override void OnPause()
{
   quoteServer.Suspend();
}

protected override void OnContinue()
{
   quoteServer.Resume();
```

```
}

public const int commandRefresh = 128;
protected override void OnCustomCommand(int command)
{
  switch (command)
  {
    case commandRefresh:
      quoteServer.RefreshQuotes();
      break;

    default:
      break;
  }
}
```

27.3.4 线程化和服务

如前所述，如果服务的初始化花费的时间过多，则 SCM 就假定服务启动失败。为了解决这个问题，必须创建线程。

服务类中的 OnStart()方法必须及时返回。如果从 TcpListener 类中调用一个 AcceptSocket()之类的阻塞方法，就必须启动一个线程来完成调用工作。使用能处理多个客户端的网络服务器时，线程池也非常有用。AcceptSocket()方法应接收调用，并在线程池的另一个线程中进行处理，这样就不需要等待代码的执行，系统看起来似乎是立即响应的。

27.3.5 服务的安装

服务必须在注册表中配置，所有服务都可以在 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services 中找到。使用 regedit 命令，可以查看注册表项。在注册表中，可以看到服务的类型、显示名称、可执行文件的路径、启动配置以及其他信息。图 27-8 显示了 W3SVC 服务的注册表配置。

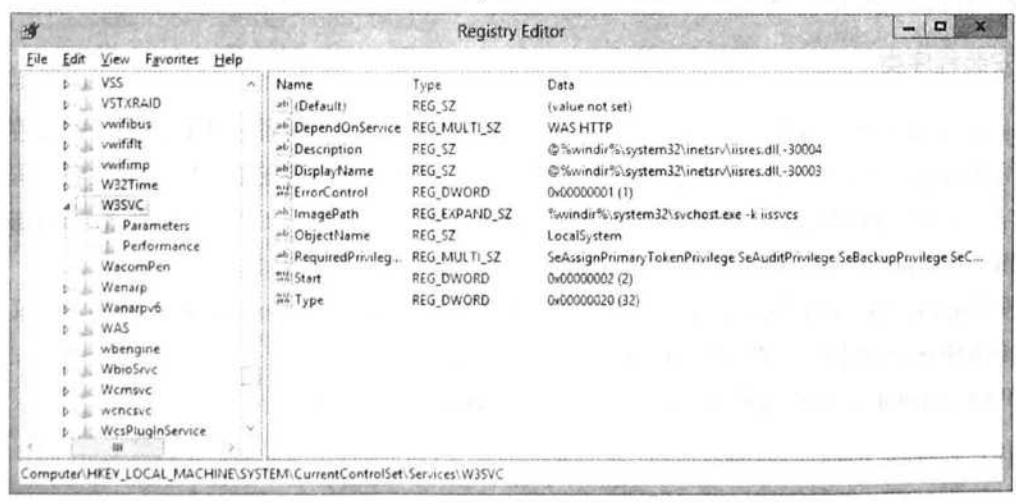


图 27-8

使用 System.ServiceProcess 名称空间中的安装程序类，可以完成服务在注册表中的配置。下面讨论这些内容。

27.3.6 安装程序

切换到 Visual Studio 的设计视图，从弹出的上下文菜单中选择 Add Installer 选项，就可以给服务添加安装程序。使用 Add Installer 选项时，新建一个 ProjectInstaller 类、一个 ServiceInstaller 实例和一个 ServiceProcessInstaller 实例。

图 27-9 显示了服务的安装程序类。

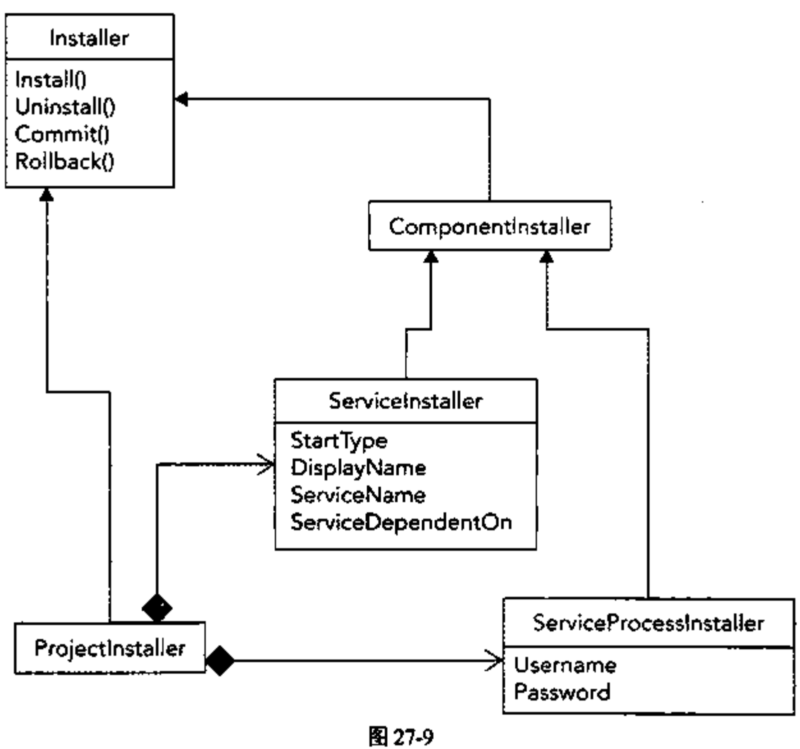


图 27-9

根据图 27-9，下面详细讨论由 Add Installer 选项创建的 ProjectInstaller.cs 文件中的源代码。

1. 安装程序类

ProjectInstaller 类派生自 System.Configuration.Install.Installer，后者是所有自定义安装程序的基类。使用 Installer 类，可以构建基于事务的安装程序。使用基于事务的安装时，如果安装失败，系统就可以回滚到以前的状态，安装程序所做的所有修改都会被取消。如图 27-9 所示，Installer 类中有 Install()、Commit()、Rollback()和 Uninstall()方法，这些方法都从安装程序中调用。

如果 RunInstaller 特性的值为 true，则在安装程序集时会调用 ProjectInstaller 类。自定义安装程序和 installutil.exe(这个程序以后将用到)都能检查该特性。

在 ProjectInstaller 类的构造函数内部调用 InitializeComponent()：

```
using System.ComponentModel;
using System.Configuration.Install;

namespace Wrox.ProCSharp.WinServices
{
  [RunInstaller(true)]
  public partial class ProjectInstaller: Installer
  {
```

```
    public ProjectInstaller()
    {
      InitializeComponent();
    }
  }
}
```

下面看看项目安装程序调用的其他安装程序。

2. ServiceProcessInstaller 类和 ServiceInstaller 类

在 InitializeComponent()方法的实现代码中，创建了 ServiceProcessInstaller 类和 ServiceInstaller 类的实例。这两个类都派生于 ComponentInstaller 类，ComponentInstaller 类本身派生于 Installer 类。

ComponentInstaller 类的派生类可以用作安装进程的一个部分。注意，一个服务进程可以包括多个服务。ServiceProcessInstaller 类用于配置进程，为这个进程中的所有服务定义值，而 ServiceInstaller 类用于服务的配置，因此每个服务都需要 ServiceInstaller 类的一个实例。如果进程中有 3 个服务，则必须添加 3 个 ServiceInstaller 对象：

```
partial class ProjectInstaller
{
  /// <summary>
  /// Required designer variable.
  /// </summary>
  private System.ComponentModel.IContainer components = null;

  /// <summary>
  /// Required method for Designer supportdo not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent()
  {
    this.serviceProcessInstaller1 =
       new System.ServiceProcess.ServiceProcessInstaller();
    this.serviceInstaller1 =
       new System.ServiceProcess.ServiceInstaller();
    //
    // serviceProcessInstaller1
    //
    this.serviceProcessInstaller1.Password = null;
    this.serviceProcessInstaller1.Username = null;
    //
    // serviceInstaller1
    //
    this.serviceInstaller1.ServiceName = "QuoteService";
    //
    // ProjectInstaller
    //
    this.Installers.AddRange(
      new System.Configuration.Install.Installer[]
          {this.serviceProcessInstaller1,
          this.serviceInstaller1});
  }
```

```
    private System.ServiceProcess.ServiceProcessInstaller
        serviceProcessInstaller1;
    private System.ServiceProcess.ServiceInstaller serviceInstaller1;

}
```

ServiceProcessInstaller 类安装一个实现 ServiceBase 类的可执行文件。ServiceProcessInstaller 类包含用于整个进程的属性。由进程中所有服务共享的属性如表 27-1 所示。

表 27-1

属 性	描 述
Username、Password	如果把 Accout 属性设置为 ServiceAccout.User，则 Username 属性和 Password 属性指出服务在哪一个用户账户下运行
Account	使用这个属性，可以指定服务的账户类型
HelpText	HelpText 是只读属性，它返回的帮助文本用于设置用户名和密码

用于运行服务的进程可以用 ServiceProcessInstaller 类的 Accout 属性指定，其值可以是 ServiceAccout 枚举的任意值。Account 属性的不同值如表 27-2 所示。

表 27-2

值	描 述
LocalSystem	设置这个值可以指定服务在本地系统上使用权限很高的用户账户，并用作网络上的计算机
NetworkService	类似于 LocalService，这个值指定把计算机的证书传递给远程服务器。但与 LocalService 不同，这种服务可以以非授权用户的身份登录本地系统。顾名思义，这个账户只能用于需要从网络上获得资源的服务
LocalService	这个账户类型给任意远程服务器提供计算机的匿名证书，其本地权限与 NetworkService 相同
User	把 Accout 属性设置为 ServiceAccout.User，表示可以指定应从服务中使用的账户

ServiceInstaller 是每一个服务都需要的类，这个类的属性可以用于进程中的每一个服务，其属性有 StartType、DisplayName、ServiceName 和 ServicesDependentOn，如表 27-3 所示。

表 27-3

属 性	描 述
StartType	StartType 属性指出服务是手动启动还是自动启动。它的值可以是：ServiceStartMode.Automatic、ServiceStartMode.Manual、ServiceStartMode.Disabled。如果使用 ServiceStartMode.Disabled，服务就不能启动。这个选项可用于不应在系统中启动的服务。例如，如果没有得到需要的硬件控制器，就可以把该选项设置为 Disabled
DelayedAutoStart	如果 StartType 属性没有设置为 Automatic，就忽略这个属性。此时可以指定服务是否应在系统启动时不立即启动，而是在以后启动

(续表)

属　性	描　述
DisplayName	DisplayName 属性是服务显示给用户的友好名称。这个名称也由管理工具用于控制和监控服务
ServiceName	ServiceName 属性是服务的名称。这个值必须与服务程序中 ServiceBase 类的 ServiceName 属性一致。这个名称把 ServiceInstaller 类的配置与需要的服务程序关联起来
ServicesDependentOn	指定必须在服务启动之前启动的一组服务。当服务启动时，所有依赖的服务都自动启动，并且我们的服务也将启动

如果在 ServiceBase 的派生类中改变了服务的名称，则还必须修改 ServiceInstaller 对象中 ServiceName 属性的值。

在测试阶段，最好把 StartType 属性的值设置为 Manual。这样，如果服务因程序中的 bug 不能停止，就仍可以重新启动系统。如果把 StartType 属性的值设置为 Automatic，服务就会在重新启动系统时自动启动！当确信没有问题时，可以在以后改变这个配置。

3. ServiceInstallerDialog 类

System.ServiceProcess.Design 名称空间中的另一个安装程序类是 ServiceInstallerDialog。在安装过程中，如果希望系统管理员输入该服务应使用的账户(具体方法是指定用户名和密码)，就可以使用这个类。

如果把 ServiceProcessInstaller 类的 Account 属性设置为 ServiceAccount.User，Username 和 Password 属性设置为 null，则在安装时将自动显示如图 27-10 所示的 Set Service Login 对话框。此时，也可以取消安装。

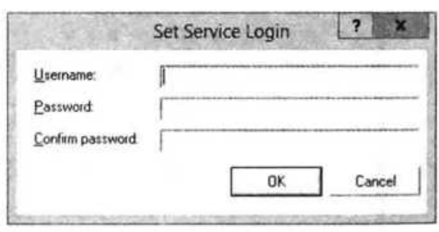


图 27-10

4.installutil

在把安装程序类添加到项目中之后，就可以使用 installutil.exe 实用程序安装和卸载服务。这个实用程序可以用于安装包含 Installer 类的所有程序集。installutil.exe 实用程序调用 Installer 派生类的 Installer()方法进行安装，调用 UnInstaller()方法进行卸载。

安装和卸载示例服务的命令分别是：

```
installutil quoteservice.exe
installutil /u quoteservice.exe
```

如果安装失败了，一定要检查安装日志文件 InstallUtil.InstallLog 和<servicename>. InstallLog。通常，在安装日志文件中可以发现一些非常有用的信息，例如：“指定的服务已存在”。

在成功地安装服务后，就可以从 Services MMC 中手动启动服务(详细内容请参阅 27.4 节)，并启动客户端应用程序。

27.4 Windows 服务的监控和控制

可以使用 Services MMC 管理单元对服务进行监控和控制。Services MMC 管理单元是 Computer Management 管理工具的一部分。每个 Windows 操作系统还有一个命令行实用程序 net.exe，使用这个程序可以控制服务。sc.exe 是另一个命令行实用程序，它的功能比 net.exe 更强大。还可以使用 Visual Studio Server Explorer 直接控制服务。本节将创建一个小型的 Windows 应用程序，它利用 System.ServiceProcess.ServiceController 类监控和控制服务。

27.4.1 MMC 管理单元

如图 27-11 所示，使用 MMC 的 Services 管理单元可以查看所有服务的状态，也可以把停止、启用或禁用服务的控制请求发送给服务，并改变它们的配置。Services 管理单元既是服务控制程序，又是服务配置程序。

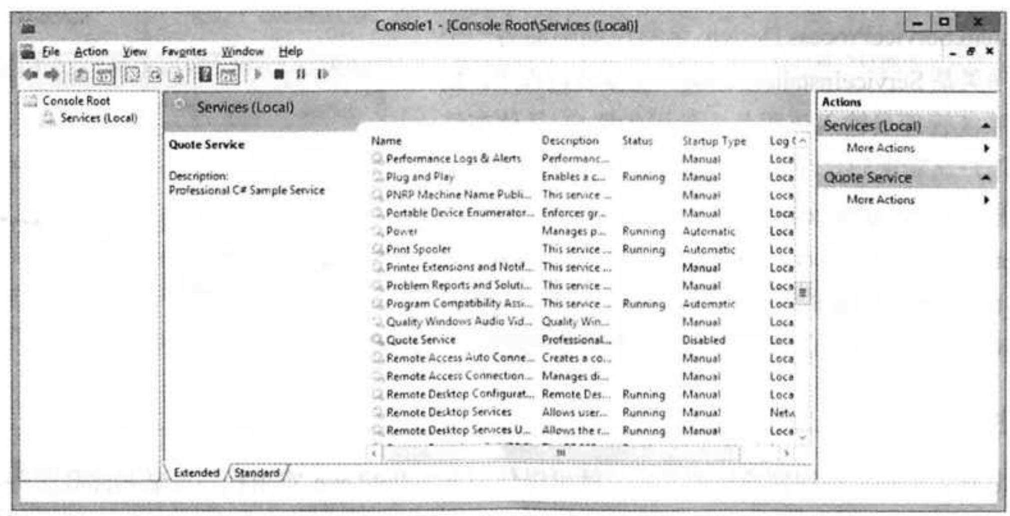


图 27-11

双击 QuoteService，打开如图 27-12 所示的 Quote Service Properties 对话框。在这个对话框中，可以看到服务的名称、描述、可执行文件的路径、启动类型和状态。目前服务已启动。使用这个对话框中的 Log On 选项卡，可以改变服务进程的账户。

图 27-12

27.4.2　net.exe 实用程序

Services 管理单元使用起来很简单，但是系统管理员不能使其自动化，原因是它不能用在管理脚本中。要通过脚本实现的工具自动控制服务，可以用命令行实用程序 net.exe 来完成。net start 命令显示所有正在运行的服务，net start servicename 启动服务，net stop servicename 向服务发送停止请求。此外使用 net pause 和 net continue 可以暂停和继续服务(当然，它们只有在服务允许的情况下才能使用)。

27.4.3　sc.exe 实用程序

sc.exe 是不太出名的一个实用程序，它作为操作系统的一部分发布。sc.exe 是管理服务的一个很有用的工具。与 net.exe 实用程序相比，sc.exe 实用程序的功能更加强大。使用 sc.exe 实用程序，可以检查服务的实际状态，或者配置、删除以及添加服务。当服务的卸载程序不能正常工作时，可以使用 sc.exe 实用程序卸载服务。

27.4.4　Visual Studio Server Explorer

在 Visual Studio 中，要使用 Server Explorer 控制服务，应在树型视图中选择 Services 节点，再选择计算机，最后选择 Services 项和需要的服务。选择一个服务，打开上下文菜单，就可以启动和停止服务。这个上下文菜单也可以用于把 ServiceController 类添加到项目中。

如果要控制应用程序中的具体服务，则可以把服务从 Server Explorer 拖放到设计器中：即把 ServiceController 实例添加到应用程序中，ServiceController 对象的属性自动设置为访问选中的服务，并引用 System.ServiceProcess 程序集。使用 ServiceController 实例控制服务的方式与使用下一节开发的应用程序来控制服务的方式相同。

27.4.5 编写自定义 ServiceController 类

下面创建一个小的 Windows 应用程序，该应用程序使用 ServiceController 类监控和控制 Windows 服务。

创建一个 WPF 应用程序，其用户界面如图 27-13 所示。这个应用程序的主窗口包含一个显示所有服务的列表框、4 个文本框(分别用于显示服务的显示名称、状态、类型和名称)和 6 个按钮，其中 4 个按钮用于发送控制事件，一个按钮用于刷新列表，最后一个按钮用于退出应用程序。

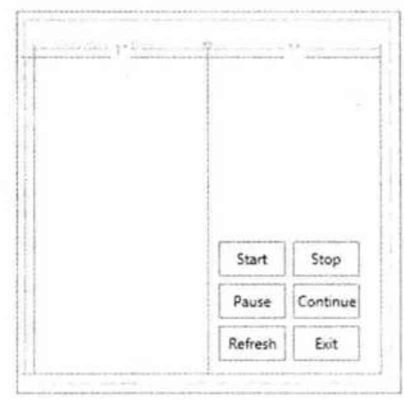


图 27-13

WPF 的介绍详见第 35 章。

1.服务的监控

使用 ServiceController 类，可以获取每一个服务的相关信息。表 27-4 列出了 ServiceController 类的属性。

表 27-4

属　性	描　述
CanPauseAndContinue	如果暂停和继续服务的请求可以发送给服务，则这个属性返回 true
CanShutdown	如果服务有用于关闭系统的处理程序，则它的值为 true
CanStop	如果服务是可以停止的，则它的值为 true
DependentServices	它返回一个依赖服务的集合。如果停止服务，则所有依赖的服务都预先停止
ServicesDependentOn	返回这个服务所依赖的服务集合
DisplayName	指定服务应该显示的名称
MachineName	指定运行服务的计算机名
ServiceName	指定服务的名称
ServiceType	指定服务的类型。服务可以运行在共享的进程中，在共享的进程中，多个服务使用同一进程(Win32ShareProcess)。此外，服务也可以运行在只包含一个服务的进程(Win32OwnProcess)中。如果服务可以与桌面交互，其类型就是 InteractiveProcess
Status	指定服务的状态。状态可以是正在运行、停止、暂停或处于某些中间模式(如启动待决、停止待决)等。状态值在 ServiceControllerStatus 枚举中定义

在示例应用程序中，使用 DisplayName、ServiceName、ServiceType 和 Status 属性显示服务信息。此外，CanPauseAndContinue 和 CanStop 属性用于启用和禁用 Pause、Continue 和 Stop 按钮。

为了得到用户界面的所有必要信息，创建一个 ServiceControllerInfo 类。这个类可以用于数据绑定，并提供状态信息、服务名称、服务类型，以及哪些控制服务的按钮应启用或禁用的信息。

因为使用了 System.ServiceProcess.ServiceController 类，所以必须引用 System.ServiceProcess 程序集。

ServiceControllerInfo 类包含一个嵌入的 ServiceController 类，用 ServiceControllerInfo 类的构造函数设置它。还有一个只读属性 Controller，它用来访问嵌入的 ServiceController 类(代码文件 ServiceControl/ServiceControllerInfo.cs)。

```
public class ServiceControllerInfo
{
  private readonly ServiceController controller;

  public ServiceControllerInfo(ServiceController controller)
  {
    this.controller = controller;
  }

  public ServiceController Controller
  {
    get { return controller; }
  }
```

为了显示服务的当前信息，可以使用 ServiceControllerInfo 类的只读属性 DisplayName、ServiceName、ServiceTypeName 和 ServiceStatusName。DisplayName 和 ServiceName 属性的实现代码只访问底层类 ServiceController 的 DisplayName 和 ServiceName 属性。对于 ServiceTypeName 和 ServiceStatusName 属性的实现代码，需要完成更多的工作：服务的状态和类型不太容易返回，因为要显示一个字符串，而不是只显示 ServiceController 类返回的数字。ServiceTypeName 属性返回一个表示服务类型的字符串。从 ServiceController. ServiceType 属性中得到的 ServiceType 代表一组标记，使用按位 OR 运算符，可以把这组标记组合在一起。InteractiveProcess 位可以与 Win32OwnProcess 和 Win32ShareProcess 一起设置。首先，在检查其他值之前，一定要先检查以前是否设置过 InteractiveProcess 位。使用这些服务，返回的字符串将是“Win32 Service Process”或“Win32 Shared Process”。

```
public string ServiceTypeName
{
  get
  {
    ServiceType type = controller.ServiceType;
    string serviceTypeName = "";
    if ((type & ServiceType.InteractiveProcess) != 0)
    {
      serviceTypeName = "Interactive ";
      type -= ServiceType.InteractiveProcess;
    }
    switch (type)
    {
      case ServiceType.Adapter:
        serviceTypeName += "Adapter";
        break;
```

```
        case ServiceType.FileSystemDriver:
        case ServiceType.KernelDriver:
        case ServiceType.RecognizerDriver:
          serviceTypeName += "Driver";
          break;

        case ServiceType.Win32OwnProcess:
          serviceTypeName += "Win32 Service Process";
          break;

        case ServiceType.Win32ShareProcess:
          serviceTypeName += "Win32 Shared Process";
          break;

        default:
          serviceTypeName += "unknown type " + type.ToString();
          break;
      }
      return serviceTypeName;
    }
  }

  public string ServiceStatusName
  {
    get
    {
      switch (controller.Status)
      {
        case ServiceControllerStatus.ContinuePending:
          return "Continue Pending";
        case ServiceControllerStatus.Paused:
          return "Paused";
        case ServiceControllerStatus.PausePending:
          return "Pause Pending";
        caseServiceControllerStatus.StartPending:
          return "Start Pending";
        case ServiceControllerStatus.Running:
          return "Running";
        case ServiceControllerStatus.Stopped:
          return "Stopped";
        case ServiceControllerStatus.StopPending:
          return "Stop Pending";
        default:
          return "Unknown status";
      }
    }
  }

  public string DisplayName
  {
    get { return controller.DisplayName; }
  }

  public string ServiceName
```

```
    {
      get { return controller.ServiceName; }
    }
```

ServiceControllerInfo 类还有一些属性可以启用 Start、Stop、Pause 和 Continue 按钮：EnableStart、EnableStop、EnablePause 和 EnableContinue，这些属性根据服务的当前状态返回一个布尔值。

```
      public bool EnableStart
      {
        get
        {
          return controller.Status == ServiceControllerStatus.Stopped;
        }
      }

      public bool EnableStop
      {
        get
        {
          return controller.Status == ServiceControllerStatus.Running;
        }
      }

      public bool EnablePause
      {
        get
        {
          return controller.Status == ServiceControllerStatus.Running &&
              controller.CanPauseAndContinue;
        }
      }

      public bool EnableContinue
      {
        get
        {
          return controller.Status == ServiceControllerStatus.Paused;
        }
      }
    }
```

在 ServiceControlWindow 类中，RefreshServiceList()方法使用 ServiceController.GetServices()方法获取在列表框中显示的所有服务。GetServices()方法返回一个 ServiceController 实例数组，它们表示在操作系统上安装的所有 Windows 服务。ServiceController 类还有一个静态方法 GetDevices()，该方法返回一个表示所有设备驱动程序的 ServiceController 数组。返回的数组利用扩展方法 OrderBy()按照 DisplayName 属性来排序，这是传递给 OrderBy()方法的 lambda 表达式定义的属性。使用 Select()方法，将 ServiceController 实例转换为 ServiceControllerInfo 类型。在下面的代码中传递了一个 lambda 表达式，它调用每个 ServiceController 对象的 ServiceControllerInfo()构造函数。最后，将 ServiceControllerInfo 数组赋予窗口的 DataContext 属性，进行数据绑定(代码文件 ServiceControl/ServiceControlWindow.xaml.cs)。

```
protected void RefreshServiceList()
{
  this.DataContext = ServiceController.GetServices().
    OrderBy(sc => sc.DisplayName).
    Select(sc => new ServiceControllerInfo(sc));
}
```

在列表框中获得所有服务的 RefreshServiceList()方法在 ServiceControlWindow 类的构造函数中调用。这个构造函数还为按钮的 Click 事件定义了事件处理程序：

```
public ServiceControlWindow()
{
  InitializeComponent();

  RefreshServiceList();
}
```

现在就可以定义 XAML 代码，把信息绑定到控件上。首先，为显示在列表框中的信息定义一个 DataTemplate。列表框包含一个标签，其 Content 属性绑定到数据源的 DisplayName 属性上。在绑定 ServiceControllerInfo 对象数组时，用 ServiceControllerInfo 类定义 DisplayName 属性：

```
<Window.Resources>
  <DataTemplate x:Key="listTemplate">
    <Label Content="{Binding DisplayName}"/>
  </DataTemplate>
</Window.Resources>
```

放在窗口左边的列表框将 ItemsSource 属性设置为{Binding}。这样，显示在列表中的数据就从 RefreshServiceList()方法设置的 DataContext 属性中获得。ItemTemplate 属性引用了前面用 DataTemplate 定义的资源 listTemplate。把 IsSynchronizedWithCurrentItem 属性设置为 True，从而使位于同一个窗口中的文本框和按钮控件绑定到列表框中当前选择的项上。

```
<ListBox Grid.Row="0" Grid.Column="0" HorizontalAlignment="Left"
  Name="listBoxServices" VerticalAlignment="Top"
  ItemsSource="{Binding}"
  ItemTemplate="{StaticResource listTemplate}"
  IsSynchronizedWithCurrentItem="True">
</ListBox>
```

为了区分按钮控件，使之分别用于启动、停止、暂停、继续服务，定义了下面的枚举(代码文件 ServiceControl/ButtonState.cs)：

```
public enum ButtonState
{
  Start,
  Stop,
  Pause,
  Continue
}
```

对于 TextBlock 控件，Text 属性绑定到 ServiceControllerInfo 实例的对应属性上。按钮控件是启用还是禁用也从数据绑定中定义，即把 IsEnabled 属性绑定到 ServiceControllerInfo 实例的对应属性上，该属性返回一个布尔值。给按钮的 Tag 属性赋予前面定义的 ButtonState 枚举的一个值，以便在

同一个处理程序方法 OnServiceCommand 中区分按钮：

```
<TextBlock Grid.Row="0" Grid.ColumnSpan="2"
  Text="{Binding /DisplayName, Mode=OneTime}" />
<TextBlock Grid.Row="1" Grid.ColumnSpan="2"
  Text="{Binding /ServiceStatusName, Mode=OneTime}" />
<TextBlock Grid.Row="2" Grid.ColumnSpan="2"
  Text="{Binding /ServiceTypeName, Mode=OneTime}" />
<TextBlock Grid.Row="3" Grid.ColumnSpan="2"
  Text="{Binding /ServiceName, Mode=OneTime}" />
<Button Grid.Row="4" Grid.Column="0" Content="Start"
  IsEnabled="{Binding /EnableStart, Mode=OneTime}"
  Tag="{x:Static local:ButtonState.Start}"
  Click="OnServiceCommand" />
<Button Grid.Row="4" Grid.Column="1" Name="buttonStop" Content="Stop"
  IsEnabled="{Binding /EnableStop, Mode=OneTime}"
  Tag="{x:Static local:ButtonState.Stop}"
  Click="OnServiceCommand" />
<Button Grid.Row="5" Grid.Column="0" Name="buttonPause" Content="Pause"
  IsEnabled="{Binding /EnablePause, Mode=OneTime}"
  Tag="{x:Static local:ButtonState.Pause}"
  Click="OnServiceCommand" />
<Button Grid.Row="5" Grid.Column="1" Name="buttonContinue"
  Content="Continue"
  IsEnabled="{Binding /EnableContinue,
  Tag="{x:Static local:ButtonState.Continue}"
  Mode=OneTime}" Click="OnServiceCommand" />
<Button Grid.Row="6" Grid.Column="0" Name="buttonRefresh"
  Content="Refresh"
  Click="OnRefresh" />
<Button Grid.Row="6" Grid.Column="1" Name="buttonExit"
  Content="Exit" Click="OnExit" />
```

2. 服务的控制

使用 ServiceController 类，也可以把控制请求发送给服务，该类的方法如表 27-5 所示。

表 27-5

方　法	说　明
Start()	Start()方法告诉 SCM 应启动服务。在服务程序示例中，调用了 OnStart()方法
Stop()	如果 CanStop 属性在服务类中的值是 true，则在 SCM 的帮助下，Stop()方法调用服务程序中的 OnStop()方法
Pause()	如果 CanPauseAndContinue 属性的值是 true，则 Pause()方法调用 OnPause()方法
Continue()	如果 CanPauseAndContinue 属性的值是 true，则 Continue()方法调用 OnContinue()方法
ExecuteCommand()	使用 ExecuteCommand()可以把定制的命令发送给服务

下面就是控制服务的代码。因为启动、停止、挂起和暂停服务的代码是相似的，所以仅为这 4 个按钮使用一个处理程序：

```
protected void OnServiceCommand(object sender, RoutedEventArgs e)
{
  Cursor oldCursor = this.Cursor;
  try
  {
    this.Cursor = Cursors.Wait;
    ButtonState currentButtonState = (ButtonState)(sender as Button).Tag;

    var si = listBoxServices.SelectedItem as ServiceControllerInfo;
    if (currentButtonState == ButtonState.Start)
    {
      si.Controller.Start();
      si.Controller.WaitForStatus(ServiceControllerStatus.Running,
          TimeSpan.FromSeconds(10));
    }
    else if (currentButtonState == ButtonState.Stop)
    {
      si.Controller.Stop();
      si.Controller.WaitForStatus(ServiceControllerStatus.Stopped,
      TimeSpan.FromSeconds(10));
    }
    else if (currentButtonState == ButtonState.Pause)
    {
      si.Controller.Pause();
      si.Controller.WaitForStatus(ServiceControllerStatus.Paused,
          TimeSpan.FromSeconds(10));
    }
    else if (currentButtonState == ButtonState.Continue)
    {
      si.Controller.Continue();
      si.Controller.WaitForStatus(ServiceControllerStatus.Running,
          TimeSpan.FromSeconds(10));
    }
    int index = listBoxServices.SelectedIndex;
    RefreshServiceList();
    listBoxServices.SelectedIndex = index;
  }
  catch (System.ServiceProcess.TimeoutException ex)
  {
    MessageBox.Show(ex.Message, "Timout Service Controller",
    MessageBoxButton.OK, MessageBoxImage.Error);
  }
  catch (InvalidOperationException ex)
  {
    MessageBox.Show(String.Format("{0} {1}", ex.Message,
      ex.InnerException != null ? ex.InnerException.Message :
        String.Empty), MessageBoxButton.OK, MessageBoxImage.Error);
  }
  finally
  {
   this.Cursor = oldCursor;
  }
}

protected void OnExit(object sender, RoutedEventArgs e)
{
 Application.Current.Shutdown();
}
```

```
protected void OnRefresh_Click(object sender, RoutedEventArgs e)
{
  RefreshServiceList();
}
```

由于控制服务要花费一定的时间，因此光标在第一条语句中切换为等待光标。然后，根据所按下的按钮调用 ServiceController 类的方法。使用 WaitForStatus()方法，表明用户正在等待检查服务把状态改为被请求的值，但是我们最多等待 10 秒。在 10 秒之后，就会刷新列表框中的信息，并把选中的索引设置为与以前相同的值，接着显示这个服务的新状态。

因为应用程序需要管理权限，大多数服务都需要管理权限来启动和停止，所以把一个应用程序清单添加到项目中，并把 requestedExecutionLevel 属性设置为 requireAdministrator(代码文件 ServiceControl/app.manifest)。

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
    xmlns="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges
 xmlns="urn:schemas-microsoft-com:
 asm.v3">
        <requestedExecutionLevel
   level="requireAdministrator"
            uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

运行应用程序的结果如图27-14所示。

图 27-14

27.5　故障排除和事件日志

服务方面的故障排除与其他类型应用程序的故障排除并不相同。本节将讨论一些服务问题、交互式服务特有的问题和事件日志。

创建服务最好的方式就是在实际创建服务之前，先创建一个具有所需功能的程序集和一个测试客户端，以便进行正常的调试和错误处理。只要应用程序运行，就可以使用该程序集创建服务。当然，对于服务，仍然存在下列问题：

- 在服务中，错误信息不显示在消息框中(除了运行在客户端系统上的交互式服务之外)，而是使用事件日志服务把错误写入事件日志中。当然，在使用服务的客户端应用程序中，可以显示一个消息框，以通知用户出现了错误。

- 虽然服务不能从调试器中启动，但是调试器可以与正在运行的服务进程联系起来。打开带有服务源代码的解决方案，并且设置断点。从 Visual Studio 的 Debug 菜单中选择 Processes 命令，关联正在运行的服务进程。
- 性能监视器可以用于监控服务的行为。可以把自己的性能对象添加到服务中，这样可以添加一些有用的信息，以便进行调试。例如，通过 Quote 服务，可以建立一个对象，让它给出返回的引用总数和初始化花费的时间等。

把事件添加到事件日志中，服务就可以报告错误和其他信息。当 AutoLog 属性设置为 true 时，从 ServiceBase 类中派生的服务类可以自动把事件写入日志中。ServiceBase 类检查 AutoLog 属性，并且在启动、停止、暂停和继续请求时编写日志条目。

图27-15是服务中的一个日志条目示例。

图 27-15

事件日志和如何编写自定义事件的内容详见第 20 章。

27.6 小结

本章讨论了 Windows 服务的体系结构和如何使用.NET Framework 创建 Windows 服务。应用程序可以与 Windows 服务一起在系统启动时自动启动，也可以把具有特权的 System 账户用作服务的用户。Windows 服务从主函数、service-main 函数和处理程序中创建。本章还介绍了与 Windows 服务相关的其他程序，如服务控制程序和服务安装程序。

.NET Framework 对 Windows 服务提供了很好的支持。创建、控制和安装服务所需的代码都封装在 System.ServiceProcess 名称空间的.NET Framework 类中。从 ServiceBase 类中派生一个类，就可以重写暂停、继续或停止服务时调用的方法。对于服务的安装，ServiceProcessInstaller 类和 ServiceInstaller 类可以处理服务所需的所有注册表配置。还可以使用 ServiceController 类控制和监控服务。

第 28 章介绍.NET 的全球化和本地化特性，如果应用程序在不同的地区用于不同的语言，这个特性就很有用。

第28章 本地化

本章要点

- 数字和日期的格式化
- 为本地化内容使用资源
- 创建和使用附属程序集
- 本地化桌面应用程序
- 本地化 Web 应用程序
- 本地化 Windows Store 应用程序
- 创建自定义资源读取器
- 创建自定义区域性

本章源代码下载地址(wrox.com):

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- NumberAndDateFormatting
- CreateResource
- CultureDemo
- BookOfTheDay
- DatabaseResourceReader
- CustomCultures

28.1 全球市场

价值 1.25 亿美元的 NASA 的火星气象卫星在 1999 年 9 月 23 日失踪了，其原因是一个工程组为一个关键的太空操作使用了米制单位，而另一个工程组以英寸为单位。当编写的应用程序要在世界各国发布时，必须考虑不同的区域性和区域。

不同的区域性在日历、数字和日期格式上各不相同。按照字母 A～Z 给字符串排序也会导致不同的结果，因为存在不同的文化差异。为了使应用程序可应用于全球市场，就必须对应用程序进行全球化和本地化。

本章将介绍.NET 应用程序的全球化和本地化。全球化(globalization)用于国际化的应用程序：使应用程序可以在国际市场上销售。采用全球化策略，应用程序应根据区域性、不同的日历等支持不同的数字和日期格式。本地化(localization)用于为特定的区域性翻译应用程序。而字符串的翻译可以使用资源，如.NET 资源或 WPF 资源字典。

.NET 支持 Windows 和 Web 应用程序的全球化和本地化。要使应用程序全球化，可以使用 System.Globalization 名称空间中的类；要使应用程序本地化，可以使用 System.Resources 名称空间支持的资源。

28.2 System.Globalization 名称空间

System.Globalization 名称空间包含了所有的区域性类和区域类，以支持不同的日期格式、不同的数字格式，甚至由 GregorianCalendar 类、HebrewCalendar 类和 JapaneseCalendar 类等表示的不同日历。使用这些类可以根据不同的地区显示不同的表示法。

本节讨论使用 System.Globalization 名称空间时要考虑的如下问题：

- Unicode 问题
- 区域性和区域
- 显示所有区域性及其特征的例子
- 排序

28.2.1 Unicode 问题

因为一个 Unicode 字符有 16 位，所以共有 65536 个 Unicode 字符。这对于当前在信息技术中使用的所有语言够用吗？例如，汉语就需要 80000 多个字符。但是，Unicode 可以解决这个问题。使用 Unicode 必须区分基本字符和组合字符。可以给一个基本字符添加若干个组合字符，组成一个可显示的字符或一个文本元素。

例如，冰岛的字符 Ogonek 就可以使用基本字符 0x006F(拉丁小写字母 o)、组合字符 0x0328(组合 Ogonek)和 0x0304(组合 Macron)组合而成，如图 28-1 所示。组合字符在 0x0300~0x0345 之间定义，对于美国和欧洲市场，预定义字符有助于处理特殊的字符。字符 Ogonek 也可以用预定义字符 0x01ED 来定义。

ǭ = o + ̨ + ̄

0x01ED 0x006F 0x0928 0x0904

图 28-1

对于亚洲市场，只有汉语需要 80000 多个字符，但没有这么多的预定义字符。在亚洲语言中，总是要处理组合字符。其问题在于获取显示字符或文本元素的正确数字，得到基本字符而不是组合字符。System.Globalization 名称空间提供的 StringInfo 类可以用于处理这个问题。

表 28-1 列出了 StringInfo 类的静态方法，这些方法有助于处理组合字符。

表28-1

方　法	说　明
GetNextTextElement()	返回指定字符串的第一个文本元素(基本字符和所有的组合字符)
GetTextElementEnumerator()	返回一个允许迭代字符串中所有文本元素的 TextElementEnumerator 对象
ParseCombiningCharacters()	返回一个引用字符串中所有基本字符的整型数组

一个显示字符可以包含多个 Unicode 字符。要解决这个问题，如果编写的应用程序要在国际市场销售，就不应使用数据类型 char，而应使用 string。string 可以包含由基本字符和组合字符组成的文本元素，而 char 不具备该作用。

28.2.2 区域性和区域

世界分为多个区域性和区域，应用程序必须知道这些区域性和区域的差异。区域性是基于用户的语言和区域性习惯的一组首选项。RFC 1766(www.ietf.org/rfc/rfc1766.txt)定义了区域性的名称，这些名称根据语言和国家或区域的不同在世界各地使用。例如，en-AU、en-CA、en-GB 和 en-US 分别用于表示澳大利亚、加拿大、英国和美国的英语。

在 System.Globalization 名称空间中，最重要的类是 CultureInfo。这个类表示区域性，定义了日历、数字和日期的格式，以及和区域性一起使用的排序字符串。

RegionInfo 类表示区域设置(如货币)，说明该区域是否使用米制系统。在某些区域中，可以使用多种语言。例如，西班牙区域就有 Basque(eu-ES)、Catalan(ca-ES)、Spanish(es-ES)和 Galician(gl-ES)区域性。一个区域可以有多种语言，同样，一种语言也可以在多个区域使用；例如，墨西哥、西班牙、危地马拉、阿根廷和秘鲁等都使用西班牙语。

本章的后面将介绍一个示例应用程序，以说明区域性和区域的这些特征。

1. 特定、中立和不变的区域性

在.NET Framework 中使用区域性，必须区分 3 种类型：特定、中立和不变的区域性。特定的区域性与真正存在的区域性相关，这种区域性用上一节介绍的 RFC 1766 定义。特定的区域性可以映射到中立的区域性。例如，de 是特定区域性 de-AT、de-DE、de-CH 等的中立区域性。de 是德语(German)的简写，AT、DE 和 CH 分别是奥地利(Austria)、德国(Germany)和瑞士(Switzerland)等国家的简写。

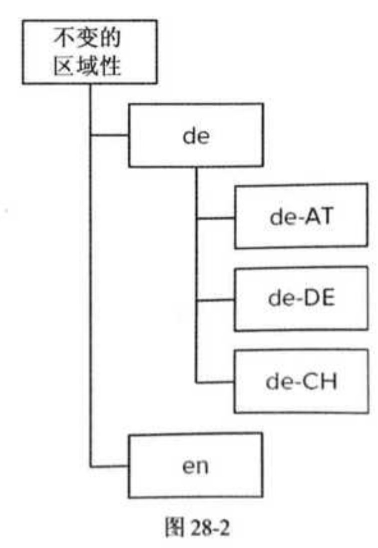


图 28-2

在翻译应用程序时，通常不需要为每个区域进行翻译，因为奥地利和瑞士等国使用的德语没有太大的区别。所以可以使用中立的区域性来本地化应用程序，而不需要使用特定的区域性。

不变的区域性独立于真正的区域性。在文件中存储格

式化的数字或日期，或通过网络把它们发送到服务器上时，最好使用独立于任何用户设置的区域性。

图 28-2 显示了区域性类型的相互关系。

2. CurrentCulture 和 CurrentUICulture

设置区域性时，必须区分用户界面的区域性和数字及日期格式的区域性。区域性与线程相关，通过这两种区域性类型，就可以把两种区域性设置应用于线程。Thread 类提供了 CurrentCulture 和 CurrentUICulture 属性。CurrentCulture 属性用于设置与格式化和排序选项一起使用的区域性，而 CurrentUICulture 属性用于设置用户界面的语言。

使用 Windows 控制面板中的“区域和语言”选项，就可以改变 CurrentCulture 的默认设置，如图 28-3 所示。使用这个配置，还可以改变区域性的默认数字、时间和日期格式。

图 28-3

CurrentUICulture 属性不依赖于这个配置，而依赖于操作系统的语言。在控制面板的 Add Languages 设置中可以添加其他语言，如图 28-4 所示。

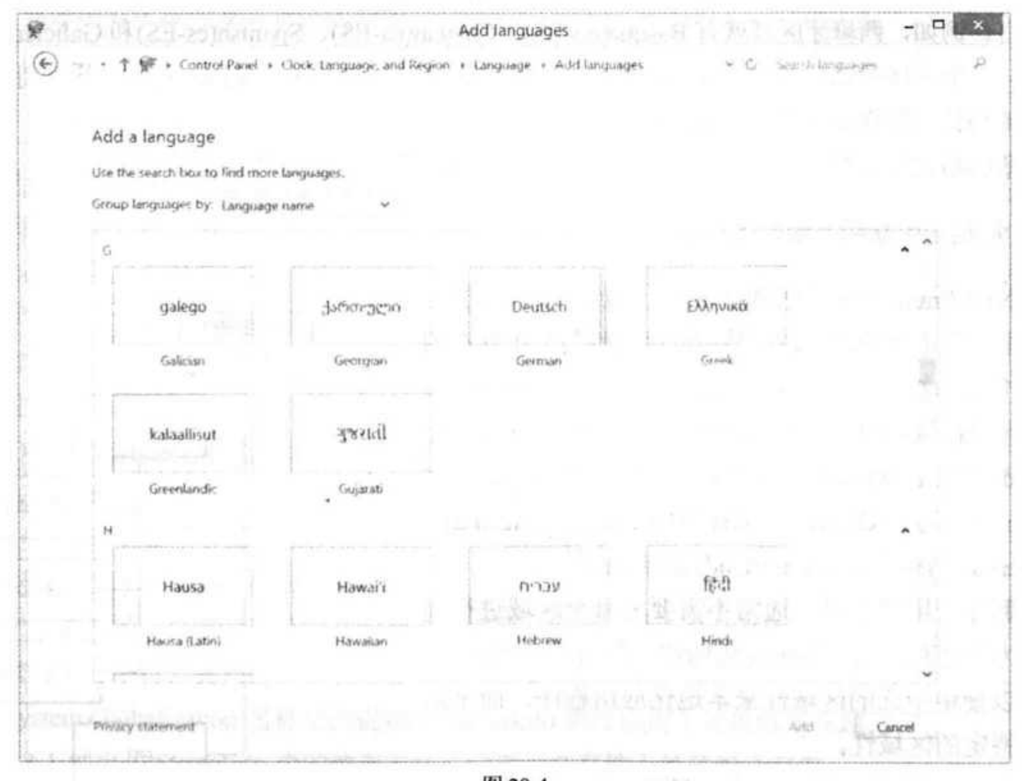


图 28-4

这些设置都使用默认值，在许多情况下，不需要改变默认值。如果需要改变区域性，只需要把线程的两个区域性改为Spanish区域性，如下面的代码片段所示(使用名称空间System.Globalization和System.Threading)：

```
var ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

前面已学习了区域性的设置，下面讨论CurrentCulture设置对数字和日期格式化的影响。

3. 数字格式化

System名称空间中的数字结构Int16、Int32和Int64等都有一个重载的ToString()方法。这个方法可以根据区域设置创建不同的数字表示法。对于Int32结构，ToString()方法有下述4个重载版本：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

不带参数的ToString()方法返回一个没有格式化选项的字符串，也可以给ToString()方法传递一个字符串和一个实现IFormatProvider接口的类。

该字符串指定表示法的格式，而这个格式可以是标准数字格式化字符串或者图形数字格式化字符串。对于标准数字格式化，字符串是预定义的，其中C表示货币符号，D表示输出为小数，E表示输出用科学计数法表示，F表示定点输出，G表示一般输出，N表示输出为数字，X表示输出为十六进制数。对于图形数字格式化字符串，可以指定位数、节和组分隔符、百分号等。图形数字格式字符串###，###表示：两个3位数块被一个组分隔符分开。

IFormatProvider接口由NumberFormatInfo、DateTimeFormatInfo和CultureInfo类实现。这个接口定义了GetFormat()方法，它返回一个格式对象。

NumberFormatInfo类可以为数字定义自定义格式。使用NumberFormatInfo类的默认构造函数，可以创建独立于区域性的对象或不变的对象。使用这个类的属性，可以改变所有格式化选项，如正号、百分号、数字组分隔符和货币符号等。从静态属性InvariantInfo返回一个与区域性无关的只读NumberFormatInfo对象。NumberFormatInfo对象的格式化值取决于当前线程的CultureInfo类，该线程从静态属性CurrentInfo返回。

下一个示例使用一个简单的控制台项目。在这段代码中，第一个示例显示了在当前线程的区域性格式中所显示的数字(这里是English-US，是操作系统的设置)。第二个示例使用了带有IFormatProvider参数的ToString()方法。CultureInfo类实现IFormatProvider接口，所以创建一个使用法国区域性的CultureInfo对象。第3个示例改变了当前线程的区域性。使用Thread实例的CurrentCulture属性，把区域性改为德国(代码文件NumberAndDateFormatting/Program.cs)：

```
using System;
using System.Globalization;
using System.Threading;

namespace NumberAndDateFormatting
{
```

```
    class Program
    {
      static void Main()
      {
        NumberFormatDemo();
      }

      private static void NumberFormatDemo()
      {
        int val = 1234567890;

        // culture of the current thread
        Console.WriteLine(val.ToString("N"));

        // use IFormatProvider
        Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));

        // change the culture of the thread
        Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
        Console.WriteLine(val.ToString("N"));
      }
    }
  }
```

结果如下所示。可以把这个结果与前面列举的美国、英国、法国和德国区域性的结果进行比较。

```
1,234,567,890.00
1 234 567 890,00
1.234.567.890,00
```

4. 日期格式化

对于日期，Visual Studio 也提供了与数字相同的支持。DateTime 结构有一些把日期转换为字符串的方法。公共实例的 ToLongDateString()、ToLongTimeString()、ToShortDateString()和 ToShortTimeString()方法都使用当前区域性来创建字符串表示法。使用 ToString()方法，可以指定另一种区域性：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

使用 ToString()方法的字符串参数，可以指定预定义格式字符或自定义格式字符串，把日期转换为字符串。DateTimeFormatInfo 类指定了可能的值。DateTimeFormatInfo 类指定的格式字符串有不同的含义。例如，D 表示长日期格式，d 表示短日期格式，ddd 表示一星期中某一天的缩写，dddd 表示一星期中某一天的全称，yyyy 表示年份，T 表示长时间格式，t 表示短时间格式。使用 IFormatProvider 参数可以指定区域性。使用不带 IFormatProvider 参数的重载方法，表示所使用的是当前线程的区域性：

```
DateTime d = new DateTime(2013, 09, 27);

// current culture
Console.WriteLine(d.ToLongDateString());
```

```
// use IFormatProvider
Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));

// use culture of thread
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));

ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));
```

这个示例程序的结果说明了使用线程的当前区域性的ToLongDateString()方法，其中给ToString()方法传递一个CultureInfo 实例，则显示其法国版本；把线程的CurrentCulture属性改为es-ES，则显示其西班牙版本，如下所示。

```
Friday, September 27, 2013
vendredi 27 septembre 2013
en-US: Friday, September 27, 2013
es-ES: viernes, 27 de septiembre de 2013
```

28.2.3 使用区域性

为了全面介绍区域性，下面使用一个 WPF 应用程序示例，该应用程序列出所有的区域性，描述区域性属性的不同特征。图 28-5 显示了该应用程序在 Visual Studio 2013 WPF 设计器中的用户界面。

图 28-5

在应用程序的初始化阶段，所有可用的区域性都添加到应用程序左边的TreeView控件中。这个初始化工作在SetupCultures()方法中进行，该方法在 Window 类 CultureDemoForm 的构造函数中调用(代码文件 CultureDemo/MainWindow.xaml.cs)：

```
public CultureDemoWindow()
{
  InitializeComponent();

  SetupCultures();
}
```

对于在用户界面上显示的数据，创建自定义类 CultureData。这个类可以绑定到 TreeView 控件上，因为它的 SubCultures 属性包含一列 CultureData。因此 TreeView 控件可以遍历这个树状结构。

CultureData 不包含子区域性，而包含数字、日期和时间的 CultureInfo 类型以及示例值。数字以适用于特定区域性的数字格式返回一个字符串，日期和时间也以特定区域性的格式返回字符串。CultureData 包含一个 RegionInfo 类来显示区域。对于一些中立区域性(例如 English)，创建 RegionInfo 会抛出一个异常，因为某些区域有特定的区域性。但是，对于其他中立区域性(例如 German)，可以成功创建 RegionInfo，并映射到默认的区域上。这里抛出的异常应这样处理：

```
public class CultureData
{
  public CultureInfo CultureInfo { get; set; }
  public List<CultureData> SubCultures { get; set; }

  double numberSample = 9876543.21;
  public string NumberSample
  {
    get { return numberSample.ToString("N", CultureInfo); }
  }
  public string DateSample
  {
    get { return DateTime.Today.ToString("D", CultureInfo); }
  }
  public string TimeSample
  {
    get { return DateTime.Now.ToString("T", CultureInfo); }
  }
  public RegionInfo RegionInfo
  {
    get
    {
      RegionInfo ri;
      try
      {
        ri = new RegionInfo(CultureInfo.Name);
      }
      catch (ArgumentException)
      {
        // with some neutral cultures regions are not available
        return null;
      }
      return ri;
    }
  }
}
```

在 SetupCultures()方法中，通过静态方法 CultureInfo.GetCultures()获取所有区域性。给这个方法传递 CultureTypes.AllCultures，就会返回所有可用区域性的未排序数组。该数组按区域性名称来排序。有了排好序的区域性，就创建一个 CultureData 对象集合，并分配 CultureInfo 和 SubCultures 属性。之后，创建一个字典，以快速访问区域性名称。

对于应绑定的数据，创建一个 CultureData 对象列表，在执行完 foreach 循环后，该列表将包含树状视图中的所有根区域性。可以验证根区域性，以确定它们是否把不变的区域性作为其父区域性。不变的区域性把 LCID(Locale Identifier)设置为 127，每个区域性都有自己的唯一标识符，可用于快速验证。在代码段中，根区域性在 if 语句块中添加到 rootCultures 集合中。如果一个区域性把不变的区域性作为其父区域性，它就是根区域性。

如果区域性没有父区域性，它就会添加到树的根节点上。要查找父区域性，必须把所有区域性保存到一个字典中。相关内容参见前面章节，其中第10章介绍了字典，第8章介绍了lambda表达式。如果所迭代的区域性不是根区域性，它就添加到父区域性的SubCultures集合中。使用字典可以快速找到父区域性。在最后一步中，把根区域性赋予窗口的DataContext，使根区域性可用于UI：

```
private void SetupCultures()
{
  var cultureDataDict = CultureInfo.GetCultures(CultureTypes.AllCultures)
    .OrderBy(c => c.Name)
    .Select(c => new CultureData
    {
      CultureInfo = c,
      SubCultures = new List<CultureData>()
    })
    .ToDictionary(c => c.CultureInfo.Name);

  var rootCultures = new List<CultureData>();
  foreach (var cd in cultureDataDict.Values)
  {
    if (cd.CultureInfo.Parent.LCID == 127)
    {
      rootCultures.Add(cd);
    }
    else
    {
      CultureData parentCultureData;
      if (cultureDataDict.TryGetValue(cd.CultureInfo.Parent.Name,
        out parentCultureData))
      {
        parentCultureData.SubCultures.Add(cd);
      }
      else
      {
        throw new ParentCultureException(
          "unexpected error - parent culture not found");
      }
    }
  }
  this.DataContext = rootCultures.OrderBy(cd =>
    cd.CultureInfo.EnglishName);
}
```

在用户选择树中的一个节点时，就会调用TreeView类的SelectedItemChanged事件的处理程序。在这里，这个处理程序在TreeCultures_SelectedItemChanged()方法中实现。在这个方法中，把Grid控件的DataContext设置为选中的CultureData对象。在XAML逻辑树中，这个Grid控件是显示所选区域性信息的所有控件的父控件。

```
private void treeCultures_SelectedItemChanged(object sender,
  RoutedPropertyChangedEventArgs<object> e)
{
  CultureData cd = e.NewValue as CultureData;
  if (cd != null)
  {
    itemGrid.DataContext = cd;
  }
}
```

现在看看显示内容的XAML代码。一个树型视图用于显示所有的区域性(代码文件CultureDemo/MainWindow.xaml)。对于在树型视图内部显示的项，使用项模板。这个模板使用一个文本框，该文本框绑定到CultureInfo类的EnglishName属性上。为了绑定树型视图中的项，应使用HierarchicalDataTemplate来递归地绑定CultureData类型的SubCultures属性：

```
<TreeView SelectedItemChanged="treeCultures_SelectedItemChanged" Margin="5"
  ItemsSource="{Binding}" >
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate DataType="{x:Type local:CultureData}"
      ItemsSource="{Binding SubCultures}">
      <TextBlock Text="{Binding Path=CultureInfo.EnglishName}" />
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

为了显示所选项的值，使用了几个TextBlock控件，它们绑定到CultureData类的CultureInfo属性上，从而绑定到从CultureInfo返回的CultureInfo类型的属性上，例如Name、IsNeutralCulture、EnglishName和NativeName等。要把从IsNeutralCulture属性返回的布尔值转换为Visiblility枚举值，并显示日历名称，应使用转换器：

```
<TextBlock Grid.Row="0" Grid.Column="0" Text="Culture Name:" />
<TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding CultureInfo.Name}"
  Width="100" />
<TextBlock Grid.Row="0" Grid.Column="2" Text="Neutral Culture"
  Visibility="{Binding CultureInfo.IsNeutralCulture,
  Converter={StaticResource boolToVisiblity}}" />

<TextBlock Grid.Row="1" Grid.Column="0" Text="English Name:" />
<TextBlock Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{Binding CultureInfo.EnglishName}" />

<TextBlock Grid.Row="2" Grid.Column="0" Text="Native Name:" />
<TextBlock Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{Binding CultureInfo.NativeName}" />

<TextBlock Grid.Row="3" Grid.Column="0" Text="Default Calendar:" />
<TextBlock Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{Binding CultureInfo.Calendar,
  Converter={StaticResource calendarConverter}}" />

<TextBlock Grid.Row="4" Grid.Column="0" Text="Optional Calendars:" />
<ListBox Grid.Row="4" Grid.Column="1" Grid.ColumnSpan="2"
  ItemsSource="{Binding CultureInfo.OptionalCalendars}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding
        Converter={StaticResource calendarConverter}}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

把布尔值转换为Visiblility枚举值的转换器在BooleanToVisiblilityConverter类中定义(代码文件Converters\BooleanToVisiblilityConverter.cs)：

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace CultureDemo.Converters
{
  public class BooleanToVisibilityConverter : IValueConverter
  {
    public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
      bool b = (bool)value;
      if (b)
        return Visibility.Visible;
      else
        return Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
      throw new NotImplementedException();
    }
  }
}
```

转换日历文本以进行显示的转换器有点复杂。下面是 CalendarToCalendarInformationConverter 类中 Convert 方法的实现代码，该实现代码使用类名和日历类型名称，给日历返回一个有用的值：

```
public object Convert(object value, Type targetType, object parameter,
  CultureInfo culture)
{
  if (c == null) return null;
  StringBuilder calText = new StringBuilder(50);
  calText.Append(c.ToString());
  calText.Remove(0, 21); // remove the namespace
  calText.Replace("Calendar", "");

  GregorianCalendar gregCal = c as GregorianCalendar;
  if (gregCal != null)
  {
    calText.AppendFormat(" {0}", gregCal.CalendarType.ToString());
  }
  return calText.ToString();
}
```

CultureData 类包含的属性可以为数字、日期和时间格式显示示例信息，这些属性用下面的 TextBlock 元素绑定：

```
<TextBlock Grid.Row="0" Grid.Column="0" Text="Number" />
<TextBlock Grid.Row="0" Grid.Column="1"
  Text="{Binding NumberSample}" />

<TextBlock Grid.Row="1" Grid.Column="0" Text="Full Date" />
<TextBlock Grid.Row="1" Grid.Column="1"
  Text="{Binding DateSample}" />
```

```
<TextBlock Grid.Row="2" Grid.Column="0" Text="Time" />
<TextBlock Grid.Row="2" Grid.Column="1"
  Text="{Binding TimeSample}" />
```

区域的信息用 XAML 代码的最后一部分显示。如果 RegionInfo 不可用，就隐藏整个 GroupBox。TextBlock 元素绑定了 RegionInfo 类型的 DisplayName、CurrencySymbol、ISOCurrencySymbol 和 IsMetric 属性：

```
<GroupBox x:Name="groupRegion" Header="Region Information" Grid.Row="6"
  Grid.Column="0" Grid.ColumnSpan="3" Visibility="{Binding RegionInfo,
  Converter={StaticResource nullToVisibility}}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="0" Text="Region" />
    <TextBlock Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2"
      Text="{Binding RegionInfo.DisplayName}" />

    <TextBlock Grid.Row="1" Grid.Column="0" Text="Currency" />
    <TextBlock Grid.Row="1" Grid.Column="1"
      Text="{Binding RegionInfo.CurrencySymbol}" />

    <TextBlock Grid.Row="1" Grid.Column="2"
      Text="{Binding RegionInfo.ISOCurrencySymbol}" />

    <TextBlock Grid.Row="2" Grid.Column="1" Text="Is Metric"
      Visibility="{Binding RegionInfo.IsMetric,
      Converter={StaticResource boolToVisiblity}}" />
</Grid>
```

启动应用程序，在树型视图中就会看到所有的区域性，选择一个区域性后，就会列出该区域性的特征，如图 28-6 所示。

图 28-6

28.2.4 排序

排序字符串取决于区域性。在默认情况下，为排序而比较字符串的算法依赖于区域性。例如在芬兰，字符 V 和 W 就是相同的。为了说明芬兰的排序方式，下面的代码创建一个小型控制台应用程序示例，其中对数组中尚未排序的一些美国州名进行排序。因为我们将使用 System.Collections.Generic、System.Threading 和 System.Globalization 名称空间中的类，所以必须声明这些名称空间。下面的 DisplayName()方法用于在控制台上显示数组或集合中的所有元素(代码文件 SortingDemo/Program.cs)：

```
static void DisplayNames(string title, IEnumerable<string> e)
{
  Console.WriteLine(title);
  foreach (string s in e)
  {
    Console.Write(s + "-");
  }
  Console.WriteLine();
  Console.WriteLine();
}
```

在 Main()方法中，在创建了包含一些美国州名的数组后，就把线程的 CurrentCulture 属性设置为 Finnish 区域性，这样，下面的 Array.Sort()方法就使用芬兰的排列顺序。调用 DisplayName()方法在控制台上显示所有的州名：

```
static void Main()
{
  string[] names = {"Alabama", "Texas", "Washington",
                    "Virginia", "Wisconsin", "Wyoming",
                    "Kentucky", "Missouri", "Utah", "Hawaii",
                    "Kansas", "Louisiana", "Alaska", "Arizona"};
  Thread.CurrentThread.CurrentCulture = new CultureInfo("fi-FI");

  Array.Sort(names);
  DisplayNames("Sorted using the Finnish culture", names);
```

在以芬兰排列顺序第一次显示美国州名后，数组将再次排序。如果希望排序独立于用户的区域性，就可以使用不变的区域性。在要将已排序的数组发送到服务器上或存储到某个地方时，就可以采用这种方式。

为此，给 Array.Sort()方法传递第二个参数。Sort()方法希望第二个参数是实现 IComparer 接口的一个对象。System.Collections 名称空间中的 Comparer 类实现 IComparer 接口。Comparer.DefaultInvariant 返回一个 Comparer 对象，该对象使用不变的区域性比较数组值，以进行独立于区域性的排序。

```
  // sort using the invariant culture
  Array.Sort(names, System.Collections.Comparer.DefaultInvariant);
  DisplayNames("Sorted using the invariant culture", names);
}
```

这个程序的输出显示了用 Finnish 区域性进行排序的结果和独立于区域性的排序结果。在使用独立于区域性的排序方式时，Virginia 排在 Washington 的前面。用 Finnish 区域性进行排序时，Virginia

排在 Washington 的后面。

```
Sorted using the Finnish culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Washington-Virginia-Wisconsin-Wyoming -

Sorted using the invariant culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Virginia-Washington-Wisconsin-Wyoming -
```

如果对集合进行的排序应独立于区域性，该集合就必须用不变的区域性进行排序。在把排序结果发送给服务器或存储在文件中时，这种方式尤其有效。

除了依赖区域设置的格式化和测量系统之外，文本和图片也可能因区域性的不同而有所变化。此时就需要使用资源。

28.3 资源

像图片或字符串表这样的资源可以放在资源文件或附属程序集中。在本地化应用程序时，这种资源非常有用，.NET 对本地化资源的搜索提供了内置支持。在说明如何使用资源本地化应用程序之前，先讨论如何创建和读取资源，而无须考虑语言因素。

28.3.1 创建资源文件

资源文件包含图片、字符串表等条目。要创建资源文件，可以使用一般的文本文件，或者使用那些利用 XML 的.resX 文件。下面从一个简单的文本文件开始。

内嵌字符串表的资源可以使用一般的文本文件来创建。该文本文件只是把字符串赋予键。键是可以用来从程序中获取值的名称。键和值都可以包含空格。

这个例子显示了 Wrox.ProCSharp.Localization.MyResources.txt 文件中的一个简单字符串表：

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

在保存带 Unicode 字符的文本文件时，必须将文件和相应的编码一起保存。为此，可以在 Save 对话框中选择 Unicode 编码。

28.3.2 资源文件生成器

可以使用资源文件生成器(Resgen.exe)实用程序在 Wrox.ProCSharp.Localization.MyResources.txt 的外部创建一个资源文件，输入如下代码：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
```

这会创建 Wrox.ProCSharp.Localization.MyResources.resources 文件。得到的资源文件可以作为一个外部文件添加到程序集中，或者内嵌到 DLL 或 EXE 中。Resgen 还可以创建基于 XML 的.resX 资源文件。构建 XML 文件的一种简单方法是使用 Resgen 本身：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
Wrox.ProCSharp.Localization.MyResources.resX
```

这条命令创建了 XML 资源文件 Wrox.ProCSharp.Localization.MyResources.resX。28.3 节将讨论如何使用 XML 资源文件。

Resgen 支持强类型化的资源。强类型化的资源用一个访问资源的类表示。这个类可以用 Resgen 实用程序的/str 选项创建：

```
resgen /str:C#,Wrox.ProCSharp.Localization,MyResources,MyResources.cs
Wrox.ProCSharp.Localization.MyResources.resX
```

在/str 选项中，按照语言、名称空间、类名和源代码文件名的顺序定义资源。

Resgen 实用程序不支持添加图片。在.NET Framework SDK 示例中，有一个带教程的 ResXGen 示例。使用 ResXGen 可以在.resX 文件中引用图片。还可以使用 ResourceWriter 类或 ResXResourceWriter 类，以编程方式把图片添加到资源中，如下所述。

28.3.3 ResourceWriter

除了使用 Resgen 实用程序构建资源文件外，编写程序来创建资源也很简单。ResourceWriter 是来自 System.Resources 名称空间的一个类，它可以用于编写二进制资源文件；ResXResourceWriter 类编写基于 XML 的资源文件。这两个类也支持图片和任何其他可序列化的对象。在使用 ResXResourceWriter 类时，必须引用 System.Windows.Forms 程序集。

下面的代码使用构造函数和文件名 Demo.resx 创建一个 ResXResourceWriter 对象 rw。在创建了一个实例后，使用 ResXResourceWriter 类的 AddResource()方法可以添加至多 2GB 的资源。AddResource()方法的第一个参数指定资源名，第二个参数指定值。可以使用 Image 类的一个实例来添加图片资源。要使用 Image 类，必须引用 System.Drawing 程序集。还要添加 using 指令，以打开 System.Drawing 名称空间。

下面打开 logo.gif 文件，创建一个 Image 对象。必须把图片复制到可执行文件的目录下，或者在 Image.ToFile()方法的参数中指定图片的完整路径。using 语句指定应在 using 块的尾部自动释放图像资源。把其他简单的字符串资源添加到 ResXResourceWriter 对象中。ResXResourceWriter 类的 Close()方法会自动调用 ResXResourceWriter.Generate()方法，把资源写入 Demo.resx 文件中(代码文件 CreateResource/Program.cs)：

```
using System;
using System.Resources;
using System.Drawing;

class Program
{
  static void Main()
  {
```

```
        var rw = new ResXResourceWriter("Demo.resx");
        using (Image image = Image.FromFile("logo.gif"))
        {
          rw.AddResource("WroxLogo", image);
          rw.AddResource("Title", "Professional C#");
          rw.AddResource("Chapter", "Localization");
          rw.AddResource("Author", "Christian Nagel");
          rw.AddResource("Publisher", "Wrox Press");
          rw.Close();
        }
      }
    }
```

启动这个小程序，创建嵌入了图像 logo.gif 的资源文件 Demo.resx，这个文件将用于下面的一个 Windows 应用程序。

28.3.4 使用资源文件

使用 C#命令行编译器 csc.exe 和/resource 选项，或直接使用 Visual Studio，可以把资源文件添加到程序集中。为了说明如何在 Visual Studio 中使用资源文件，下面创建一个控制台应用程序 ResourceDemo。

在 Solution Explorer 窗口的上下文菜单(Add | Add Existing Item 命令)中，把前面创建的资源文件 Demo.resx 添加到这个项目中。默认情况下，把这个资源的 Build Action 设置为 Embedded Resource，这样，这个资源就嵌入到输出程序集中。

在项目设置(Application | Assembly information 命令)中，把应用程序的 Neutral Language 设置为主要语言，如 English(United States)，如图 28-7 所示。改变这个设置，会在 assemblyinfo.cs 文件中添加[NeutralResourceLanguageAttribute]特性：

图 28-7

```
[assembly:
NeutralResourcesLanguageAttribute("en-US")]
```

设置这个选项会提高 ResourceManager 的性能，因为它会更快地找到 en-US 的资源，该资源还会用作默认的回退。使用这个特性也可以通过构造函数的第二个参数指定默认资源的位置。使用 UltimateResourceFallbackLocation 枚举可以指定默认资源将在主程序集或附属程序集(MainAssembly 和 Satellite 值)中存储。

构建项目后，使用 ildasm 查看生成的程序集时，会在程序集清单中看到.mresource 特性，如图 28-8 所示。它声明了程序集中资源的名称。如果把.mresource 声明为 public(与本例一样)，该资源就会从程序集中导出，且可以用于其他程序集的类中。如果把.mresource 声明为 private，则表示该资源不能导出，只能用于该程序集内部。

```
MANIFEST
Find  Find Next
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compil
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.mresource public Wrox.ProCSharp.Localization.Demo.resources
{
  // Offset: 0x00000000 Length: 0x000005B2
}
.module ResourceDemo.exe
// MVID: {440A0B40-F166-4C8E-A9FD-D6126173B048}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000003    //  ILONLY 32BITREQUIRED
// Image base: 0x021F0000
```

图 28-8

要访问嵌入的资源，可以使用 System.Resources 名称空间中的 ResourceManager 类。把以嵌入资源为参数的程序集传递给 ResourceManager 类的构造函数。在本例中，因为把资源嵌入正在执行的程序集中，所以应把 Assembly. GetExecutingAssembly()方法的结果作为构造函数的第二个参数。第一个参数是资源的根名。根名由名称空间和资源文件名(不带资源扩展名)组成。如前所述，使用 ildasm 来显示该名称。为此，只需要删除资源的扩展名 resources 即可。还可以使用 System.Reflection. Assembly 类的 GetManifestResourceNames()方法通过编程方式获取该名称。

```
using System;
using System.Drawing;
using System.Reflection;
using System.Resources;

namespace Wrox.ProCSharp.Localization
{
  class Program
  {
    static void Main()
    {
      var rm = new ResourceManager("Wrox.ProCSharp.Localization.Demo",
                                   Assembly.GetExecutingAssembly());
```

使用 ResourceManager 实例 rm，通过指定 GetObject()和 GetString()方法的键，就可以获得所有的资源：

```
      Console.WriteLine(rm.GetString("Title"));
      Console.WriteLine(rm.GetString("Chapter"));
      Console.WriteLine(rm.GetString("Author"));
      using (Image logo = (Image)rm.GetObject("WroxLogo"))
      {
        logo.Save("logo.bmp");
      }
    }
  }
}
```

通过强类型化的资源，可以简化前面编写的代码；不需要实例化 ResourceManager，也不需要使用索引器访问资源，而只需要使用属性访问资源名·

```
private static void StronglyTypedResources()
{
  Console.WriteLine(Demo.Title);
  Console.WriteLine(Demo.Chapter);
  Console.WriteLine(Demo.Author);
  using (Bitmap logo = Demo.WroxLogo)
  {
    logo.Save("logo.bmp");
  }
}
```

要使用托管资源编辑器创建强类型化的资源，可以把 Access Modifier 从 No Code Generation 重置为 Public 或 Internal。使用 Public 选项，生成的类就使用公共访问修饰符，可以在其他程序集中使用。而使用 Internal 选项，生成的类就使用内部访问修饰符，只能在程序集内部访问。

设置这个选项后，就会创建 Demo 类(它与资源同名)。这个类的静态属性为所有的资源提供了强类型化的资源名。在静态属性的实现代码中，使用了 ResourceManager 对象，该对象在第一次访问时实例化，然后缓存(代码文件 ResourceDemo/Demo.Designer.cs)：

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.33440
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace Wrox.ProCSharp.Localization
{
  using System;

  /// <summary>
  /// A strongly-typed resource class, for looking up localized strings, etc.
  /// </summary>
  // This class was auto-generated by the StronglyTypedResourceBuilder
  // class via a tool like ResGen or Visual Studio.
  // To add or remove a member, edit your .ResX file then rerun ResGen
  // with the /str option, or rebuild your VS project.
  [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Resources.Tools.StronglyTypedResourceBuilder", "4.0.0.0")]
  [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
  [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
  internal class Demo
  {
    private static global::System.Resources.ResourceManager resourceMan;

    private static global::System.Globalization.CultureInfo resourceCulture;
    [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
      "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
    internal Demo()
    {
    }
```

```
/// <summary>
/// Returns the cached ResourceManager instance used by this class.
/// </summary>
[global::System.ComponentModel.EditorBrowsableAttribute(
  global::System.ComponentModel.EditorBrowsableState.Advanced)]
internal static global::System.Resources.ResourceManager ResourceManager
{
  get
  {
    if (object.ReferenceEquals(resourceMan, null))
    {
      global::System.Resources.ResourceManager temp =
        new global::System.Resources.ResourceManager(
        "Wrox.ProCSharp.Localization.Demo", typeof(Demo).Assembly);
      resourceMan = temp;
    }
    return resourceMan;
  }
}

/// <summary>
/// Overrides the current thread's CurrentUICulture property for all
/// resource lookups using this strongly typed resource class.
/// </summary>
[global::System.ComponentModel.EditorBrowsableAttribute(
  global::System.ComponentModel.EditorBrowsableState.Advanced)]
internal static global::System.Globalization.CultureInfo Culture
{
  get
  {
    return resourceCulture;
  }
  set
  {
    resourceCulture = value;
  }
}

/// <summary>
/// Looks up a localized string similar to Chapter.
/// </summary>
internal static string Chapter
{
  get
  {
    return ResourceManager.GetString("Chapter", resourceCulture);
  }
}

//...

internal static System.Drawing.Bitmap WroxLogo
{
  get
```

```
        {
          object obj = ResourceManager.GetObject("WroxLogo", resourceCulture);
          return ((System.Drawing.Bitmap)(obj));
        }
      }
    }
  }
```

28.3.5 System.Resources 名称空间

在举例之前，本节先复习一下 System.Resources 名称空间中处理资源的类：

- ResourceManager 类可以用于从程序集或资源文件中获取当前区域性的资源。使用 ResourceManager 类还可以获取特定区域性的 ResourceSet 类。
- ResourceSet 类表示特定区域性的资源。在创建 ResourceSet 类的实例时，它会枚举一个实现 IResourceReader 接口的类，并在散列表中存储所有的资源。
- IResourceReader 接口用于从 ResourceSet 中枚举资源。ResourceReader 类实现这个接口。
- ResourceWriter 类用于创建资源文件。ResourceWriter 类实现 IResourceWriter 接口。
- ResXResourceSet、ResXResourceReader 和 ResXResourceWriter 类分别类似于 ResourceSet、ResourceReader 和 ResourceWriter 类，但创建的是基于 XML 的资源文件.resX，而不是二进制文件。ResXFileRef 可以用于链接资源，而不是把资源嵌入 XML 文件中。
- System.Resources.Tools 名称空间包含的 StronglyTypedResourceBuilder 类可以从资源中创建类。

28.4 使用 Visual Studio 的 Windows Forms 本地化

下面创建一个简单的 Windows Forms 应用程序，以说明如何使用 Visual Studio 2012 进行本地化。这个应用程序没有使用复杂的 Windows Forms，也没有任何实际的内部功能，因为这里主要是说明本地化功能。在自动生成的源代码中，把名称空间改为 Wrox.ProCSharp.Localization，把类名改为 BookOfTheDayForm。因为名称空间不仅在源文件 BookOfTheDayForm.cs 中作了修改，还在项目设置中进行了修改，所以所有生成的资源文件也都可以获得这个名称空间。为此，需要从 Project | Properties 菜单中选择 Common Properties 命令，为所有新建的条目修改该名称空间。

为了说明本地化的一些问题，这个程序有一幅图像、一些文本、一个日期和一个数字。图像显示的是一幅已进行了本地化的国旗。图 28-9 在 Windows 窗体设计器中显示了该应用程序的这个窗体。

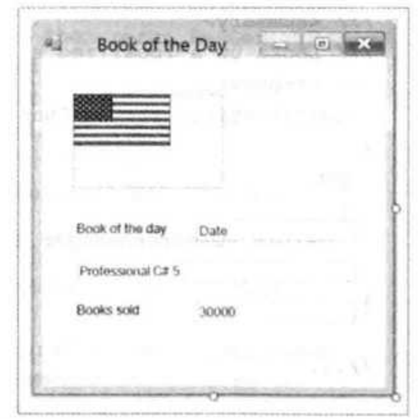


图 28-9

Windows Forms 元素的 Name 和 Text 属性值如表 28-2 所示。

表28-2

名　称	文　本
labelBookOfTheDay	Book of the day
labelItemsSold	Books sold
textDate	Date
textTitle	Professional C# 5
textItemsSold	30000
pictureFlag	

除了这个窗体外，还需要一个消息框，根据当前时间显示不同的问候信息。该示例说明了动态创建的对话框在进行本地化时必须采用不同的方式。在 WelcomeMessage()方法中，使用 MessageBox.Show()方法显示一个消息框，在窗体类BookOfTheDayForm的构造函数中调用WelcomeMessage()方法，之后调用 InitializeComponent()方法。

下面是 WelcomeMessage()方法的代码：

```
public void WelcomeMessage()
{
  DateTime now = DateTime.Now;
  string message;
  if (now.Hour <= 12)
  {
    message = "Good Morning";
  }
  else if (now.Hour <= 19)
  {
    message = "Good Afternoon";
  }
  else
  {
    message = "Good Evening";
  }
  MessageBox.Show(String.Format("{0}\nThis is a localization sample",
  message));
}
```

窗体中的数字和日期应使用格式化选项来设置。我们添加一个新方法 SetDateAnd Number()，用格式选项来设置这些值。在实际应用程序中，这些值应从一个 Web 服务或数据库中得到，但本例把注意力集中在本地化上。日期使用 D 选项来格式化(以显示长日期名)。使用图片数字格式字符串 ###,###,### 来显示该数字，其中"#"表示一个数字，","是组分隔符(代码文件 BookOfTheDay/BookOfTheDayForm.cs)：

```
public void SetDateAndNumber()
{
  DateTime today = DateTime.Today;
  textDate.Text = today.ToString("D");
  int itemsSold = 327444;
  textItemsSold.Text = itemsSold.ToString("###,###,###");
```

```
}
```

在BookOfTheDayForm类的构造函数中，调用了WelcomeMessage()和SetDateAndNumber()方法：

```
public BookOfTheDayForm()
{
  WelcomeMessage();

  InitializeComponent();

  SetDateAndNumber();
}
```

Windows窗体设计器的一个强大功能体现在把窗体的Localizable属性从false改为true时。这个设置的结果是为对话框创建一个基于XML的资源文件，以存储所有资源字符串、属性(包括Windows Forms元素的位置和大小)、嵌入的图片等。另外，对InitializeComponent()方法的实现代码也进行了修改：创建System.Resources.ResourceManager类的一个实例，为了获取文本字段以及图片的值和位置，应使用GetObject()方法，而不是直接在代码中写入值。GetObject()方法使用当前线程的CurrentUICulture属性来查找合适的本地化资源。

下面是从BookOfTheDayForm.Designer.cs文件中把Localizable属性设置为true之前Initalize-Component()方法的部分代码，其中设置了textTitle的所有属性：

```
private void InitializeComponent()
{
  //...
  this.textTitle = new System.Windows.Forms.TextBox();
  //
  // textTitle
  //
  this.textTitle.Anchor = ((System.Windows.Forms.AnchorStyles)
    (((System.Windows.Forms.AnchorStyles.Top
    | System.Windows.Forms.AnchorStyles.Left)
    | System.Windows.Forms.AnchorStyles.Right)));
  this.textTitle.Location = new System.Drawing.Point(29, 164);
  this.textTitle.Name = "textTitle";
  this.textTitle.Size = new System.Drawing.Size(231, 20);
  this.textTitle.TabIndex = 3;
```

把Localizable属性设置为true后，InitalizeComponent()方法的代码会自动修改，如下所示：

```
private void InitializeComponent()
{
  System.ComponentModel.ComponentResourceManager resources =
    new System.ComponentModel.ComponentResourceManager(
      typeof(BookOfTheDayForm));
  //...
  this.textTitle = new System.Windows.Forms.TextBox();
  //
  // textTitle
  //
  resources.ApplyResources(this.textTitle, "textTitle");
  this.textTitle.Name = "textTitle";
```

资源管理器从哪里获取数据？在把 Localizable 属性设置为 true 时，就生成了一个资源文件 BookOfTheDay.resX。在这个文件中，首先找到 XML 资源的架构，接着找到窗体中的所有元素：Type、Text、Location 和 TabIndex 等。

ComponentResourceManager 类派生自 ResourceManager 类，并提供了 ApplyResources()方法。在这个方法中，使用第二个参数定义的资源应用于第一个参数中的对象。

下面的 XML 段说明了 textBoxTitle 的几个属性：Location 属性的值是“29, 164”，Size 属性的值是“231, 20”，Text 属性设置为“ProfessionalC#5”等。对于每个值，还存储了值的类型。例如，Location 属性的类型是 System.Drawing.Point，这个类可在程序集 System.Drawing 中找到。

为什么位置和大小也存储在这个 XML 文件中？在转换时，许多字符串都会有完全不同的大小，且不再适合于原始位置。把位置和大小都存储在资源文件中后，需要进行本地化的全部内容也都存储在这些文件中，从而与 C#代码分开(代码文件 BookOfTheDay/BookOfTheDayForm.resx)：

```
<data name="textTitle.Anchor" type=
   "System.Windows.Forms.AnchorStyles, System.Windows.Forms">
  <value>Top, Left, Right</value>
</data>
<data name="textTitle.Location" type="System.Drawing.Point, System.Drawing">
  <value>29, 164</value>
</data>
<data name="textTitle.Size" type="System.Drawing.Size, System.Drawing">
  <value>231, 20</value>
</data>
<data name="textTitle.TabIndex" type="System.Int32, mscorlib">
  <value>3</value>
</data>
<data name="textTitle.Text" xml:space="preserve">
  <value>Professional C# 2012</value>
</data>
<data name="&gt;&gt;textTitle.Name" xml:space="preserve">
  <value>textTitle</value>
</data>
<data name="&gt;&gt;textTitle.Type" xml:space="preserve">
  <value>System.Windows.Forms.TextBox, System.Windows.Forms, Version=4.0.0.0,
       Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
</data>
<data name="&gt;&gt;textTitle.Parent" xml:space="preserve">
  <value>$this</value>
</data>
<data name="&gt;&gt;textTitle.ZOrder" xml:space="preserve">
  <value>2</value>
</data>
```

在修改其中一些资源值时，不需要直接使用 XML 代码来修改。我们可以在 Visual Studio 设计器中直接修改这些资源。无论何时修改窗体的 Language 属性和一些窗体元素的属性，都会为指定的语言生成一个新资源文件。把 Language 属性设置为 German，就会创建德国版本的窗体。把 Language 属性设置为 French，就会创建法国版本的窗体。对于每种语言，都会生成一个属性已被修改的资源文件：对于本例，即 BookOfTheDayForm.de.resX 和 BookOfTheDayForm.fr.resX。

表 28-3 列出了德国版本的窗体需要进行的修改。

表 28-3

德国名称	值
$this.Text (窗体的标题)	Buch des Tages
labelItemsSold.Text	Bücher verkauft:
labelBookOfTheDay.Text	Buch des Tages:

表 28-4 列出了法国版本的窗体应进行的修改。

表 28-4

法国名称	值
$this.Text (窗体的标题)	Le livre du jour
labelItemsSold.Text	Des livres vendus:
labelBookOfTheDay.Text	Le livre du jour:

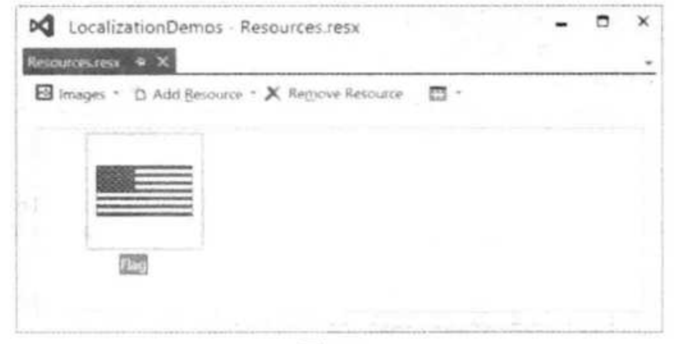


图 28-10

图像不再默认移动到附属程序集中。但在示例应用程序中，国旗应根据国家的不同而改变。为此，必须在 Resources.resx 文件中添加美国国旗的图像。这个文件位于 Visual Studio 的 Solution Explorer 窗口的 Properties 部分中。使用资源编辑器选择 Images 类别，如图 28-10 所示，以添加 americanflag.bmp 文件。为了用图像进行本地化，图像必须在所有的语言中有相同的名称。在这里，Resources.resx 文件中的图像名是 Flag。可以在属性编辑器中给图像重命名。在属性编辑器中，还可以指定图像是链接式还是嵌入式。为了提高资源的性能，图像默认为链接式。对于链接的图像，图像文件必须与应用程序一起发布。如果要在程序集中嵌入图像，就可以把 Persistence 属性改为 Embedded。

把 Resource.resx 文件复制到 Resource.de.resx 和 Resource.fr.resx 中，并用 GermanFlag.bmp 和 FranceFlag.bmp 替代国旗，就可以添加国旗的本地化版本。因为只有中立资源需要强类型化的资源类，所以可以给所有特定语言的资源文件清除 CustomTool 属性。

现在编译该项目，为每种语言创建一个附属程序集。在 debug 目录(根据目前的配置，或者是 release 目录)中，创建语言子目录，如 de 和 fr。在这个子目录下，保存了 BookOfTheDay.resources.dll 文件。这个文件就是只包含本地化资源的附属程序集。使用 ildasm 打开这个程序集，可以看到其已嵌入资源的程序集清单，以及一个定义好的地区。因为这个程序集在程序集属性中有一个地区 de，所以它位于 de 子目录下。其中还有一个带有.mresource 后缀的资源名：它的前缀是名称空间名 Wrox.ProCSharp.Localization，其后是类名 BookOfTheDayForm 和语言代码 de。

28.4.1 通过编程方式修改区域性

在翻译资源并构建附属程序集后，就可以根据为用户配置的区域性获得正确的译文。此时不翻译欢迎消息，它们需要以另一种方式翻译，稍后讨论。

除了系统配置外，还可以把语言代码当作命令行参数传递给应用程序，以便进行测试。修改BookOfTheDayForm构造函数(代码文件BookOfTheDay/BookOfTheDayForm.cs)，以传递区域性字符串，并根据这个字符串设置区域性。创建一个CultureInfo 实例，把它传递给当前线程的CurrentCulture和 CurrentUICulture 属性。注意CurrentCulture属性用于格式化，而CurrentUICulture属性用于加载资源。

```
public BookOfTheDayForm(string culture)
{
  if (!String.IsNullOrEmpty(culture))
  {
    var ci = new CultureInfo(culture);
    // set culture for formatting
    Thread.CurrentThread.CurrentCulture = ci;
    // set culture for resources
    Thread.CurrentThread.CurrentUICulture = ci;
  }

  WelcomeMessage();

  InitializeComponent();
  SetDateAndNumber();
}
```

在Program.cs文件的Main()方法中实例化BookOfTheDayForm()。在这个方法中，把区域性字符串传送给BookOfTheDayForm()构造函数：

```
[STAThread]
static void Main(string[] args)
{
  string culture = String.Empty;
  if (args.Length == 1)
  {
    culture = args[0];
  }

  Application.EnableVisualStyles();
  Application.SetCompatibleTextRenderingDefault(false);
  Application.Run(new BookOfTheDayForm(culture));
}
```

现在可以使用命令行选项启动应用程序。通过正在运行的应用程序，可以查看格式化选项和从Windows窗体设计器中生成的资源。图28-11和图28-12分别显示了使用de-DE和fr-FR命令行选项启动的应用程序。

欢迎消息框还有一个问题，这些字符串在程序中都是硬编码的。因为这些字符串都不是窗体中元素的属性，所以窗体设计器就不能像处理 Windows 控件的属性那样，在改变窗体的 Localizable属性时提取XML资源，而必须自己修改代码。

图 28-11

图 28-12

28.4.2 使用自定义资源消息

对于欢迎消息，必须翻译硬编码的字符串。英语文本翻译成德语和法语的译文如表 28-5 所示。可以在 Resources.resx 文件中直接编写自定义资源消息和与特定语言相关的派生文本。当然，也可以新建资源文件。

表 28-5

名 称	英 语	德 语	法 语
GoodMorning	Good Morning	Guten Morgen	Bonjour
GoodAfternoon	Good Afternoon	Guten Tag	Bonjour
GoodEvening	Good Evening	Guten Abend	Bonsoir
Message1	This is a localization sample.	Das ist ein Beispiel mit Lokalisierung.	C'est un exemple avec la localization.

WelcomeMessage()方法的源代码也必须改为使用资源。在强类型化的资源中，不需要实例化 ResourceMessager 类，而可以使用强类型化资源的属性：

```
public static void WelcomeMessage()
{
  DateTime now = DateTime.Now;
  string message;
  if (now.Hour <= 12)
  {
    message = Properties.Resources.GoodMorning;
  }
  else if (now.Hour <= 19)
  {
    message = Properties.Resources.GoodAfternoon;
  }
  else
  {
    message = Properties.Resources.GoodEvening;
  }
  MessageBox.Show(String.Format("{0}\n{1}", message,
      Properties.Resources.Message1);
}
```

使用英语、德语或法语启动程序时，会得到对应语言的消息框。

28.4.3 资源的自动回退

对于示例中的法国和德国版本，我们已经在附属程序集中包含了所有资源。如果不使用这些版本，则所有标签和文本框的值都要修改，这根本没有问题。只需要把要修改的值放在附属程序集中，其他值放在父程序集中即可。例如，对于 de-at (奥地利)，可以把 Good Afternoon 资源的值改为 GrüßGott，而其他值不应修改。在运行期间，如果在 de-at 附属程序集中找不到 Good Morning 资源的值，就应搜索父程序集。de-at 附属程序集的父程序集是 de。如果 de 程序集中也没有这个资源，就应在 de 的父程序集(即中立程序集)中搜索该值。中立程序集不包含区域性代码。

在主程序集的区域性代码中，不应定义任何区域性！

28.4.4 外包翻译

使用资源文件很容易完成外包翻译(outsource translation)的任务。在翻译资源文件时，不需要安装 Visual Studio，一个简单的 XML 编辑器就足够了。使用 XML 编辑器的缺点是没有机会重新安排 Windows Forms 元素，如果翻译过来的文本不适合标签或按钮的原始边框，则其大小是不能改变的。使用 Windows 窗体设计器进行翻译是很自然的选择。

Microsoft 的.NET Framework SDK 附带的一个工具可以满足所有这些需要：Windows 资源本地化编辑器 winres.exe(参见图 28-13)。使用该工具的用户无须访问 C#源文件，只需要翻译二进制文件或基于 XML 的资源文件。在完成这些翻译工作后，就可以把资源文件导入 Visual Studio 项目中，以构建附属程序集。

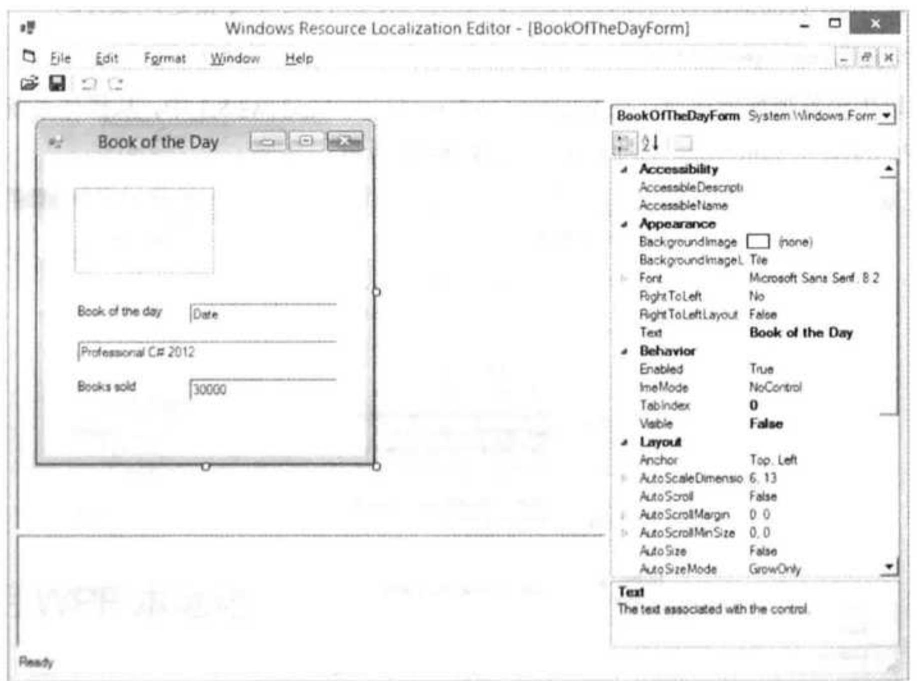


图 28-13

如果不希望翻译局改变标签和按钮的大小和位置，且不能处理 XML 文件，就可以发送一个基于文本的简单文件。使用命令行实用程序 resgen.exe，可以从 XML 文件中创建一个文本文件：

```
resgen myresource.resX myresource.txt
```

在收到翻译局完成的翻译文件后，就可以从返回的文本文件中创建一个 XML 文件。注意要给文件名添加区域性名：

```
resgen myresource.es.txt myresource.es.resX
```

28.5 ASP.NET Web Forms 的本地化

ASP.NET Web Forms 应用程序的本地化与 Windows 应用程序的本地化类似。第 40 章将讨论 ASP.NET Web Forms 应用程序的功能，本节只讨论 ASP.NET 应用程序的本地化问题。ASP.NET 4.5 和 Visual Studio 2013 有许多支持本地化的特性。本地化和全球化的基本概念与前面讨论的一样，但 ASP.NET 的本地化有一些特殊的问题。

如前所述，在 ASP.NET 中，必须区分用户界面的区域性和用于格式化的区域性。这两种区域性都可以在 Web 和页面级上定义，也可以通过编程定义。

要独立于 Web 服务器的操作系统，区域性和用户界面区域性可以用 web.config 配置文件中的 <globalization>元素定义：

```
<configuration>
  <system.web>
    <globalization culture="en-US" uiCulture="en-US" />
  </system.web>
</configuration>
```

如果特定 Web 页面的配置应该有所不同，则可以使用 Page 指令指定区域性：

```
<%Page Language="C#" Culture="en-US" UICulture="en-US" %>
```

用户可以用浏览器配置语言。在 Internet Explorer 11 和 Windows 8.1 中，从操作系统中提取这个设置。IE 中的配置用 Language 选项定义，如图 28-14 所示。

图 28-14

如果页面的语言应根据客户端的语言设置而不同，就可以通过编程把线程的区域性设置为从客户端接收的语言设置。ASP.NET 的一个自动设置可以完成这个任务。把区域性设置为 Auto，就可以根据客户端的设置指定线程的区域性。

```
<%Page Language="C#" Culture="Auto" UICulture="Auto" %>
```

在使用资源时，ASP.NET 会区分用于整个网站的资源和只用于一个页面的资源。

如果在一个页面中使用资源，就可以在设计视图中选择 Visual Studio 的 Tools | Generate Local Resource 命令，为页面创建资源。这会创建一个子目录 App_LocalResources，在这里存储每个页面的资源文件。这些资源可以用与 Windows 应用程序相同的方式进行本地化。Web 控件和本地资源文件之间的关系用 meta:resourcekey 特性指定，如下面的 ASP.NET 标签控件示例。LabelResource1 是资源名，该名称可以在本地资源文件中修改。

```
<asp:Label ID="Label1" Runat="server" Text="English Text"
meta:resourcekey="Label1Resource1"></asp:Label>
```

对于应该在多个页面之间共享的资源，必须创建一个 ASP.NET 文件夹 Appl_GlobalResources。在这个目录中，可以添加资源文件(如 Messages.resx)及其资源。为了把 Web 控件与这些资源关联起来，可以使用属性编辑器中的 Expressions 按钮。单击 Expressions 按钮，打开 Expressions 对话框，如图 28-15 所示。在该对话框中，可以选择表达式类型 Resources，设置 ClassKey 的名称(这是资源文件的名称，这里会生成一个强类型化的资源文件)、ResourceKey 的名称(资源的名称)。

在 ASPX 文件中，用绑定表达式语法<%$关联资源：

```
<asp:Label ID="Label2" Runat="server"
   Text="<%$ Resources:Messages, String1 %>">
</asp:Label>
```

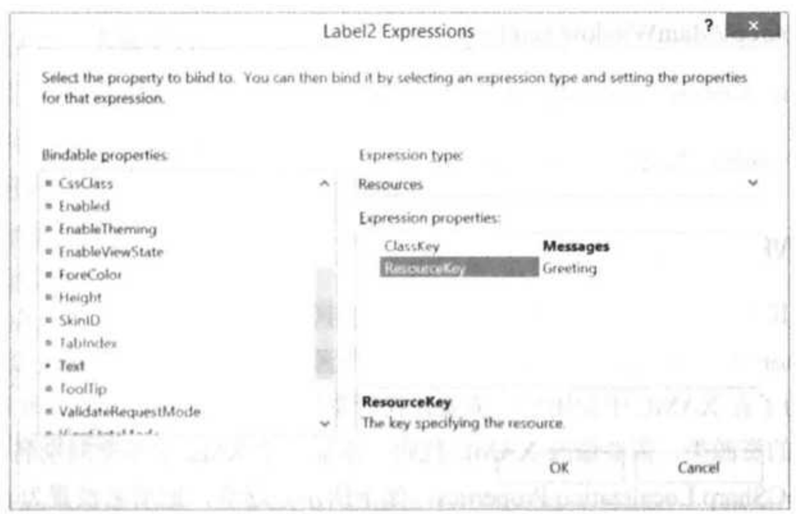


图 28-15

28.6 用 WPF 本地化

Visual Studio 2013 没有给 WPF 应用程序的本地化提供强大的支持，但仍可以本地化 WPF 应用

程序。WPF 内置了本地化支持，可以使用与 Windows Forms 和 ASP.NET 应用程序类似的.NET 资源，也可以使用 XAML(XML for Application Markup Language)资源字典。

下面讨论这些选项。WPF 和 XAML 详见第 35 章和第 36 章。为了说明如何对 WPF 应用程序使用资源，创建一个简单的 WPF 应用程序，它只包含一个按钮，如图 28-16 所示。

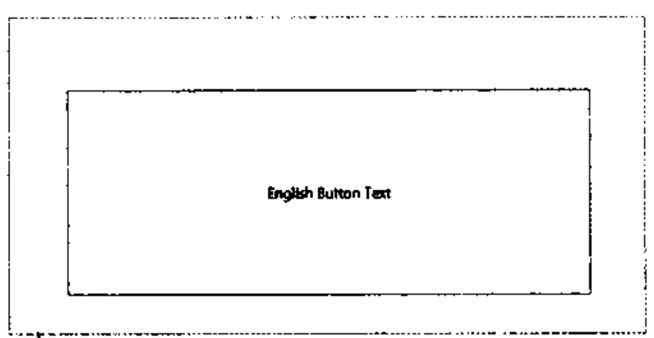


图 28-16

这个应用程序的 XAML 代码如下(代码文件 WPFApplicationUsingResources/MainWindow.xaml)：

```
<Window x:Class="Wrox.ProCSharp.Localization.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Sample" Height="240" Width="500">
  <Grid>
    <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
        Content="English Button" />
  </Grid>
</Window>
```

在按钮的 Click 事件的处理程序代码中，仅弹出一个包含示例消息的消息框(代码文件 WPFApplicationUsingResources/MainWindow.xaml.cs)：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("English Message");
}
```

28.6.1 用于 WPF 的.NET 资源

可以用处理其他应用程序的方式把.NET 资源添加到 WPF 应用程序中。在 Resources.resx 文件中定义资源 Button1Text 和 Button1Message。这个资源文件默认使用 Internal 访问修饰符来创建 Resources 类。为了在 XAML 中使用它，需要在托管资源编辑器中把修饰符改为 Public。

要使用生成的资源类，需要修改 XAML 代码。添加一个 XML 名称空间别名，以引用.NET 名称空间 Wrox.ProCSharp.Localization.Properties，如下所示。这里，把别名设置为 props。在 XAML 元素中，这个类的属性可以用于 x:Static 标记扩展。把 Button 的 Content 属性设置为 Resources 类的 Button1Text 属性(代码文件 WPFApplicationUsingResources/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.Localization.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:props="clr-namespace:Wrox.ProCSharp.Localization.Properties"
```

```
    Title="WPF Sample" Height="300" Width="300">
  <Grid>
    <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
      Content="{x:Static props:Resources.Button1Text}" />
  </Grid>
</Window>
```

要在代码隐藏中使用.NET资源，可以直接访问Button1Message属性，这与Windows Forms应用程序的使用方式相同(代码文件WPFApplicationUsingResources/MainWindow.xaml.cs)：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(Properties.Resources.Button1Message);
}
```

现在资源可以以前面介绍的方式本地化了。

使用.NET资源本地化WPF应用程序的两个主要优点是：

- .NET资源很容易管理。
- x:Static绑定由编译器检查。

当然，.NET资源也有缺点：

- 需要在XAML文件中添加x:Static绑定，它没有设计器支持。
- 绑定到使用ResourceManager生成的资源类。需要执行一些额外的工作，才能支持其他资源管理器，如本章后面讨论的DatabaseResourceManager。
- 其他XAML元素没有类型转换器的支持。

28.6.2 XAML资源字典

除了使用.NET资源本地化WPF应用程序之外，还可以直接使用XAML创建本地化内容。它也有自己的优缺点。本地化过程的步骤如下：

(1) 从主要内容中创建一个附属程序集。

(2) 对可以本地化的内容使用资源字典。

(3) 给将进行本地化的元素添加x:Uid特性。

(4) 从程序集中提取本地化内容。

(5) 翻译相应内容。

(6) 为每种语言创建附属程序集。

下面描述这些步骤。

1. 创建附属程序集

在编译WPF应用程序时，XAML代码编译为二进制格式BAML，BAML存储在一个程序集中。要把BAML代码从主程序集移动到一个独立的附属程序集中，可以修改.csproj生成文件，添加如下的<UICulture>元素，作为<propertyGroup>元素的一个子元素。这里的区域性是en-US，它定义了项目的默认区域性。用这个生成设置构建项目，会创建一个子目录en-US，并创建一个附属程序集，其中包含了默认语言的BAML代码(项目文件WPFApplicationUsingXAMLDictionaries/WPFApplicationUsingXAMLDictionaries.csproj)。

```
<UICulture>en-US</UICulture>
```

要修改在UI中不可用的项目设置，最简单的方法是在Solution Explorer窗口中选择该项目，从弹出的上下文菜单中选择Unload Project命令以卸载项目，然后从弹出的上下文菜单中选择Edit project-file命令。修改了项目文件后，就可以再次加载项目。

把BAML分离到一个附属程序集中，还应该应用NeutralResourcesLanguage特性，给附属程序集提供资源回退位置。如果决定把BAML保存在主程序集中(不给.csproj文件定义<UICulture>)，UltimateResourcesFallbackLocation特性就应设置为MainAssembly(代码文件WPFApplicationUsingXAMLDictionaries/AssemblyInfo.cs)。

```
[assembly: NeutralResourcesLanguage("en-US",
UltimateResourceFallbackLocation.Satellite)]
```

2. 添加资源字典

对于需要本地化的代码隐藏内容，可以添加一个资源字典。使用XAML可以在<ResourceDictionary>元素中定义资源，如下所示。在Visual Studio中，可以添加一个新的资源字典项，并定义文件名，以新建资源字典。在这里的例子中，资源字典包含一个字符串项。要访问System名称空间中的String类型，需要定义一个XML名称空间别名。这里把别名system设置为程序集mscorlib中的clr-namespace System。所定义的字符串可以用message1键访问。这个资源字典在LocalizaedStrings.xaml文件中定义。

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">
  <system:String x:Key="Message1">English Message</system:String>
</ResourceDictionary>
```

为了使资源字典可用于应用程序，必须把它添加到资源中。如果资源字典只需要在一个窗口或一个特定的WPF元素中使用，就可以把它添加到该窗口或WPF元素的资源集合中。如果多个窗口需要同一个资源字典，就可以把该资源字典添加到<Application>元素的App.xaml文件中，从而它可用于整个应用程序。这里把资源字典添加到主窗口的资源中(代码文件WPFApplicationUsingXAMLDictionaries/MainWindow.xaml)：

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="LocalizationStrings.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

要从代码隐藏中使用XML资源字典，可以使用Resources属性的索引器、FindResource()方法或者TryFindResource()方法。因为资源是用窗口定义的，所以使用Windows类的Resources属性的索引器访问该资源。FindResource()方法以层次结构的方式来搜索资源。在这个简单的应用程序中，

如果使用 Button 的 FindResource()方法，且通过 Button 资源没有找到它，就可以在 Grid 中搜索资源。如果还没有找到资源，就查找 Window 资源，然后查找 Application 资源(代码文件 WPFApplicationUsingXAMLDictionaries/MainWindow.xaml.cs)。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(this.Resources["Message1"] as string);
  MessageBox.Show(this.FindResource("Message1") as string);
}
```

3. 用于本地化的 Uid 特性

对于自定义资源字典文件，可以引用应从代码中本地化的文本。为了本地化 XAML 代码和 WPF 元素，将 x:Uid 特性用作需要本地化的元素的唯一标识符。不必把这个特性手动应用于 XAML 内容，而可以使用 msbuild 命令和如下选项：

```
msbuild /t:updateuid
```

在项目文件所在的目录下调用这条命令时，项目的 XAML 文件会被修改，给每个元素添加一个 x:Uid 属性和一个唯一标识符。如果控件已经应用 Name 或 x:Name 特性，x:Uid 就有相同的值；否则就生成一个新值。前面的相同 XAML 现在就应用了新特性：

```
<Window x:Uid="Window_1"
       x:Class="WPFApplicationUsingXAMLDictionaries.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Main Window" Height="240" Width="500">
  <Window.Resources>
    <ResourceDictionary x:Uid="ResourceDictionary_1">
      <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary x:Uid="ResourceDictionary_2"
                          Source="LocalizationStrings.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Window.Resources>
  <Grid x:Uid="Grid_1">
    <Button x:Uid="button1" Name="button1" Margin="30,20,30,20"
      Click="Button_Click" Content="English Button" />
  </Grid>
</Window>
```

如果在添加了 x:Uid 特性后修改 XAML 文件，就可以用选项/t:checkuid 验证 x:Uid 特性的正确性。现在可以编译项目，以创建包含 x:Uid 特性的 BAML 代码，并使用一个工具来提取这个信息。

4. 用于本地化的 LocBaml 工具

编译项目，会创建一个包含 BAML 代码的附属程序集。在这个附属程序集中，可以用 System.Windows.Markup.Localizer 名称空间中的类提取需要本地化的内容。在 Windows SDK 中包含了示例程序 LocBaml。这个程序可以用于从 BAML 中提取本地化内容。我们需要把这个可执行的、包含默认内容的附属程序集和 LocBaml.exe 复制到一个目录下，并启动示例程序，用本地化内容生成一

个.csv 文件。

```
LocBaml /parse WPFApplicationUsingXAMLDictionaries.resources.dll /out: trans.csv
```

要给通过.NET 4.5 构建的 WPF 应用程序使用 LocBaml，还必须用.NET 4 或较新的版本构建这个工具。如果 LocBaml 工具是用.NET 2.0 构建的旧版本，它就不能加载.NET 4 程序集。因为 Windows SDK 包含这个工具的源代码，所以可以使用.NET 的最新版本重建它。

可以使用 Microsoft Excel 打开.csv 文件，并翻译其内容。从.csv 文件中提取的内容包含了按钮的内容和资源字典中的消息，如下所示：

```
WPFandXAMLResources.g.en-US.resources:localizationstrings.baml,
system:String_1:System.String.$Content,None,True,True,,English Message
WPFandXAMLResources.g.en-US.resources:window1.baml,
button1:System.Windows.Controls.ContentControl.Content,Button,True,True,,
English Button
```

这个文件包含如下字段：

- BAML 的名称
- 资源的标识符
- 提供内容类型的资源类别
- 资源对于翻译是否可见(可读)的布尔值
- 资源对于翻译是否可修改的布尔值
- 本地化注释
- 资源的值

本地化资源后，就可以为新语言创建一个新目录(如用于德语的 de 目录)。目录结构遵循与本章前面的附属程序集相同的约定。使用 LocBaml 工具，可以用翻译过来的内容创建附属程序集：

```
LocBaml /generate WPFandXAMLResources.resources.dll /trans:trans_de.csv
/out: ./de /cul:de-DE
```

现在，这里应用了用来设置线程区域性和查找附属程序集的相同规则，以前通过 Windows Forms 应用程序来说明该规则。

如前所述，使用 XAML 字典进行本地化有许多工作要做。这是一个缺点。幸好，不必每天都进行本地化。那么其优点是什么？

- 可以把 XAML 文件中的本地化过程延迟到应用程序完成之后。不需要特别的标记或资源映射语法。本地化过程可以与开发过程分开。
- 使用 XAML 资源字典在运行期间的效率很高。
- 本地化很容易使用 CSV 编辑器完成。

它的缺点是：

- 在 SDK 的示例中不支持 LocBaml 工具。
- 本地化是一次性过程，很难修改已配置的本地化。

28.7 自定义资源读取器

资源读取器是.NET Framework 4.5 的一部分，利用资源读取器，可以从资源文件和附属程序集中读取资源。如果要把资源放在另一个存储器(如数据库)中，就可以创建一个自定义资源读取器来读取这些资源。

要使用自定义资源读取器，还必须创建自定义资源集和自定义资源管理器。但是，这些任务都不难，因为可以从已有的类中派生自定义类。

对于该示例应用程序，需要创建一个简单的数据库，其中只有一个表，用于存储消息；该表对于每种支持的语言有一列。表 28-6 列出了这些列及其相应的值。

表 28-6

键	默　认	DE	ES	FR	IT
Welcome	Welcome	Willkommen	Recepcion	Bienvenue	Benvenuto
Good Morning	Good Morning	Guten Morgen	Buonas diaz	Bonjour	Buona mattina
Good Evening	Good Evening	Guten Abend	Buonas noches	Bonsoir	Buona sera
Thank you	Thank you	Danke	Gracias	Merci	Grazie
Goodbye	Goodbye	Auf Wiedersehen	Adiós	Au revoir	Arrivederci

对于自定义资源读取器，用 3 个类创建一个组件库，这些类分别是 DatabaseResourceReader、DatabaseResourceSet 和 DatabaseResourceManager。

28.7.1 创建 DatabaseResourceReader 类

DatabaseResourceReader 类定义了两个字段，即访问数据库所需要的连接字符串和读取器应返回的语言。这些字段在这个类的构造函数中填充。language 字段设置为区域性名称，这个区域性名称将与 CultureInfo 对象一起传递给构造函数(代码文件 DatabaseResourceReader/DatabaseResource-Reader.cs)。

```
public class DatabaseResourceReader: IResourceReader
{
private string connectionString;
private string language;

public DatabaseResourceReader(string connectionString,
  CultureInfo culture)
{
  this.connectionString = connectionString;
  this.language = culture.Name;
}
```

资源读取器必须实现 IResourceReader 接口，这个接口定义了 Close()和 GetEnumerator()方法，GetEnumerator()方法返回一个 IDictionaryEnumerator，它为资源返回键和值。在 GetEnumerator()方法的实现代码中，将创建一个散列表，其中存储了特定语言的所有键和值。接着，使用 System.Data.

SqlClient 名称空间中的 SqlConnection 类在 SQL Server 中访问数据库。Connection.CreateCommand()方法创建一个 SqlCommand 对象，该对象用于指定 SQL 的 SELECT 语句，以访问数据库中的数据。如果把语言设置为 de，Select 语句就是 SELECT [key], [de] FROM Messages。接着，使用 SqlDataReader 对象从数据库中读取所有的值，并把它们放在一个散列表中。最后，返回散列表的枚举器。

有关 ADO.NET 数据访问的更多信息请参阅第 32 章。

```
public System.Collections.IDictionaryEnumerator GetEnumerator()
{
  var dict = new Dictionary<string, string>();

  var connection = new SqlConnection(connectionString);
  SqlCommand command = connection.CreateCommand();
  if (string.IsNullOrEmpty(language))
  {
    language = "Default";
  }

  command.CommandText = "SELECT [key], [" + language + "] " +
                        "FROM Messages";

  try
  {
    connection.Open();

    SqlDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
      if (reader.GetValue(1) != System.DBNull.Value)
      {
        dict.Add(reader.GetString(0).Trim(), reader.GetString(1));
      }
    }

    reader.Close();
  }
  catch (SqlException ex)
  {
    if (ex.Number != 207) // ignore missing columns in the database
      throw; // rethrow all other exceptions
  }
  finally
  {
    connection.Close();
  }
  return dict.GetEnumerator();
}

public void Close()
```

```
    {
    }
```

因为IResourceReader接口派生自IEnumerable和IDisposable接口，所以必须实现GetEnumerator()方法，该方法返回IEnumerator接口；同时也必须实现Dispose()方法。

```
  IEnumerator IEnumerable.GetEnumerator()
  {
    return this.GetEnumerator();
  }

  void IDisposable.Dispose()
  {
  }
}
```

28.7.2 创建 DatabaseResourceSet 类

DatabaseResourceSet类可以使用基类ResourceSet的几乎所有实现代码。我们只需要另一个构造函数，它用我们的资源读取器DatabaseResourceReader初始化基类。ResourceSet类的构造函数通过实现IResourceReader接口来传递一个对象，DatabaseResourceReader类可以满足这个要求(代码文件DatabaseResourceReader/DatabaseResourceSet.cs)。

```
public class DatabaseResourceSet: ResourceSet
{
  internal DatabaseResourceSet(string connectionString, CultureInfo culture)
    : base(new DatabaseResourceReader(connectionString, culture))
  {
  }

  public override Type GetDefaultReader()
  {
    return typeof(DatabaseResourceReader);
  }
}
```

28.7.3 创建 DatabaseResourceManager 类

第3个要创建的类是自定义资源管理器DatabaseResourceManager，它派生自ResourceManager类，我们只需要实现一个新的构造函数，并重写InternalGetResourceSet()方法。

在构造函数中，新建一个Dictionary<string, DatabaseResourceSet>，以存储所有查询到的资源集，并把它放在基类定义的ResourceSets字段中(代码文件DatabaseResourceReader/DatabaseResourceManager.cs)。

```
public class DatabaseResourceManager: ResourceManager
{
  private string connectionString;
  private Dictionary<string, DatabaseResourceSet> resourceSets;

  public DatabaseResourceManager(string connectionString)
  {
    this.connectionString = connectionString;
```

```
    resourceSets = new Dictionary<string, DatabaseResourceSet>();
  }
```

可以使用 ResourceManager 类的方法(如 GetString()和 GetObject())来访问资源，这些方法调用 InternalGetResourceSet()方法来访问资源集，在资源集中可以返回适当的值。

在 InternalGetResourceSet()方法的实现代码中，首先检查查询到的区域性的资源集是否已在散列表中，如果已经存在，就把它们返回给调用者。如果资源集不可用，就用查询到的区域性新建一个对象 DatabaseResourceSet，把它添加到字典中，再将它返回给调用者。

```
  protected override ResourceSet InternalGetResourceSet(
     CultureInfo culture, bool createIfNotExists, bool tryParents)
  {
    DatabaseResourceSet rs = null;

    if (resourceSets.ContainsKey(culture.Name))
    {
      rs = resourceSets[culture.Name];
    }
    else
    {
      rs = new DatabaseResourceSet(connectionString, culture);
      resourceSets.Add(culture.Name, rs);
    }
    return rs;
  }
}
```

28.7.4 DatabaseResourceReader 的客户端应用程序

在客户端应用程序中使用 ResourceManager 类的方式与该类在前面的使用方式没有太大的区别。唯一的区别是要使用自定义类 DatabaseResourceManager 代替 ResourceManager 类。下面的代码片段说明了如何使用自己的资源管理器。

通过把数据库连接字符串传递给构造函数，可以新建一个 DatabaseResourceManager 对象。接着，像以前一样，调用在基类中实现的 GetString()方法，给它传递键和一个 CultureInfo 类型的可选对象，以指定某种区域性。然后，从数据库中获取一个资源值，因为这个资源管理器正在使用 DatabaseResourceSet 和 DatabaseResourceReader 类(代码文件 DatabaseResourceReaderClient/Program.cs)。

```
var rm = new DatabaseResourceManager(
             @"server=(local)\sqlexpress;database=LocalizationDemo;" +
              "trusted_connection=true");

string spanishWelcome = rm.GetString("Welcome", new CultureInfo("es-ES"));
string italianThankyou = rm.GetString("ThankYou", new CultureInfo("it"));
string threadDefaultGoodMorning = rm.GetString("GoodMorning");
```

28.8 创建自定义区域性

随着时间的推移，.NET Framework 支持的语言越来越多。但并不是所有语言都可用于.NET。对于不可用于.NET 的语言，可以创建自定义区域性。例如，为了给一个区域的少数民族创建自定义区

域性，或者给不同的方言创建子区域性，创建自定义区域性就很有用。

自定义区域性和区域可以用 System.Globalization 名称空间中的 CultureAndRegionInfoBuilder 类创建，这个类位于 sysglobl 程序集中。

在 CultureAndRegionInfoBuilder 类的构造函数中，可以传递区域性名。该构造函数的第二个参数需要 CultureAndRegionModifiers 类型的一个枚举。这个枚举有 3 个值：Neutral 表示中立区域性；如果应替换已有的 Framework 区域性，就使用值 Replacement；第 3 个值是 None。

在实例化 CultureAndRegionInfoBuilder 对象后，就可以设置属性来配置区域性。使用这个类的一些属性，可以定义所有的区域性和区域信息，如名称、日历、数字格式、米制信息等。如果区域性应基于已有的区域性和区域，就可以使用 LoadDataFromCultureInfo()和 LoadDataFromRegionInfo()方法设置实例的属性，之后通过设置属性来修改不同的值。

调用 Register()方法，给操作系统注册新区域性。描述区域性的文件位于<windows>\Globalization 目录中，其扩展名是.nlp(代码文件 CustomCultures/Program.cs)。

```
using System;
using System.Globalization;

namespace CustomCultures
{
  class Program
  {
    static void Main()
    {
      try
      {
        // Create a Styria culture
        var styria = new CultureAndRegionInfoBuilder("de-AT-ST",
          CultureAndRegionModifiers.None);
        var cultureParent = new CultureInfo("de-AT");
        styria.LoadDataFromCultureInfo(cultureParent);
        styria.LoadDataFromRegionInfo(new RegionInfo("AT"));
        styria.Parent = cultureParent;
        styria.RegionNativeName = "Steiermark";
        styria.RegionEnglishName = "Styria";
        styria.CultureEnglishName = "Styria (Austria)";
        styria.CultureNativeName = "Steirisch";

        styria.Register();
      }
      catch (UnauthorizedAccessException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

因为在系统上注册自定义语言需要管理员权限，所以需要使用应用程序清单文件来指定请求的执行权限。在项目属性中，清单文件必须在 Application 设置中设置：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1"
xmlsn:asmv1="urn:schemas-microsoft-com:asm.v1" xmlns:asmv2="urn:schemas-microsoftcom:
asm.v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="requireAdministrator"
          uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

现在新建的区域性就可以像其他区域性那样使用了：

```
var ci = new CultureInfo("de-AT-ST");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

区域性可以用于格式化和资源。如果再次启动本章前面编写的 Cultures In Action 应用程序，就可以看到自定义区域性。

28.9 用 Windows Store 应用程序进行本地化

用 Windows Store 应用程序进行本地化基于前面学习的概念，但带来了一些新理念，如下所述。为了获得最佳的体验，需要安装 Multilingual App Toolkit(http://msdn.microsoft.com/en-us/windows/apps/hh848309.aspx)。

区域性、区域和资源的概念是相同的，但因为 Windows Store 应用程序可以用 C#和 XAML、C++和 XAML、JavaScript 和 HTML 来编写，所以这些概念必须可用于所有的语言。只有 Windows Runtime 能用于所有这些编程语言和 Windows Store 应用程序。因此，用于全球化和资源的新名称空间可通过 Windows Runtime 来使用：Windows.Globalization 和 Windows.AppalicationModel.Resources。在全球化名称空间中包含 Calendar、GeographicRegion(对应于.NET 的 RegionInfo)和 Language 类。在其子名称空间中，还有一些数字和日期格式化类随着语言的不同而改变。在 C#和 Windows Store 应用程序中，仍可以使用.NET 类表示区域性和区域。

下面举一个例子，说明如何用 Windows Store 应用程序进行本地化。使用 Blank APP(XAML) Visual Studio 项目模板创建一个小应用程序，用 Basic Page 模板替代自动生成的页面 MainPage.xaml。在页面上添加两个 TextBlock 和一个 TextBox 控件。

在代码文件的 OnNavigatedTo()方法中，可以把具有当前格式的日期赋予 text1 控件的 Text 属性。DateTime 结构可以用非常类似于以前的方式使用。但注意有几个方法不能用于 Windows Store 应用程序——例如，不能使用 ToLongDateString()方法，但可以使用相同格式的 ToString()方法：

```
private void navigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
  text1.Text = DateTime.Today.ToString("D");
}
```

28.9.1 使用资源

在 Windows Store 应用程序中，可以用文件扩展名 resw 替代 resx，以创建资源文件。在后台，resw 文件使用相同的 XML 格式，可以使用相同的 Visual Studio 资源编辑器创建和修改这些文件。下例使用如图 28-17 所示的结构。子文件夹 Message 包含一个子目录 en-us，在其中创建了两个资源文件 Errors.resw 和 Messages.resw。在 Strings\en-us 文件夹中，创建了资源文件 Resources.resw。

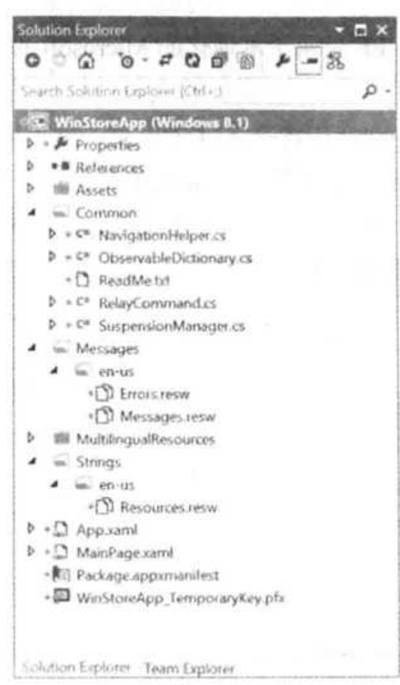


图 28-17

Messages.resw 文件包含一些英语文本资源，Hello 的值是 Hello World，资源的名称是 GoodDay、GoodEvening 和 GoodMorning。文件 Resources.resw 包含资源 Text3.Text 和 Text3.Width，其值分别是“This is a sample message for Text4”和 300。

在代码中，使用 Windows.AppalicationModel.Resources 名称空间中的 ResourcesLoader 类可以访问资源。这里使用字符串“Messages”作为 GetForCurrentView 方法的参数。因此，要使用资源文件 Messages.resw。调用 GetString 方法，会检索键为“Hello”的资源。

```
var resourceLoader = ResourceLoader.GetForCurrentView("Messages");
text2.Text = resourceLoader.GetString("Hello");
```

在 Windows Store 应用程序中，也可以直接在 XAML 代码中使用资源。对于下面的 TextBox，给 x:Uid 特性赋予值 Text3。这样，就会在资源文件 Resources.resw 中搜索名为 Text3 的资源。这个资源文件包含键 Text3.Text 和 Text3.Width 的值。检索这些值，并设置 Text 和 Width 属性：

```
<TextBox x:Uid="Text3" HorizontalAlignment="Left" Margin="50"
TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"/>
```

28.9.2 使用多语言应用程序工具集进行本地化

为了本地化 Windows Store 应用程序，可以下载前面提及的 Multilingual App Toolkit。这个工具包集成在 Visual Studio 2013 中。安装了该工具包后，就可以通过 Tools | Enable Multilingual Toolkit，为 Windows Store 应用程序启用它。这会在项目文件中添加一个生成命令，在 Solution Explorer 的上下文菜单中添加一个菜单项。选择 Add Translation Languages，打开如图 28-18 所示的对话框，在其中可以选择要翻译为哪种语言。该示例选择 Pseudo Language、French、German 和 Spanish。

图 28-18

对于这些语言，可以使用 Microsoft Translator。这个工具现在创建一个 MultilingualResources 子目录，其中包含所选语言的.xlf 文件。.xlf 文件用 XLIFF(XML Localisation Interchange File Format)标准定义，XLIFF 是用于本地化的 Microsoft 独立标准。

下次启动项目的生成过程时，XLIFF 文件就会从所有资源中填充相应的内容。在 Solution Explorer 中选择 XLIFF 文件，就可以把它们直接发送给翻译过程。为此，选择 Send for Translation，打开电子邮件，添加 XLIFF 文件作为附件。

因为系统上有了 Multilingual Toolkit，所以也可以打开如图 28-19 所示的 Multilingual Editor，启动翻译过程。单击 Translate 按钮，就会使用 Microsoft Translation Service 自动翻译所有的资源值。

图 28-19

没有进行人工检查，就不要使用翻译的结果。该工具会为每个已翻译的资源显示状态。自动翻译完成后，状态设置为 Needs Review。自动翻译的结果可能不正确，有时还很可笑。

28.10 小结

本章讨论了.NET 应用程序的全球化和本地化。对于应用程序的全球化，我们讨论了 System.Globalization 名称空间，它用于格式化依赖于区域性的数字和日期。此外，说明了在默认情况下，字符串的排序取决于区域性。我们使用不变的区域性进行独立于区域性的排序。并且，本章讨论了如何使用 CultureAndRegionInfoBuilder 类创建自定义区域性。

应用程序的本地化使用资源来实现。资源可以放在文件、附属程序集或自定义存储器(如数据库)中。本地化所使用的类位于 System.Resources 名称空间中。要从其他地方读取资源，如附属程序集或资源文件，可以创建自定义资源读取器。

我们还学习了如何本地化 Windows Forms 应用程序、WPF 应用程序、ASP.NET 应用程序和 Windows Store 应用程序。除此之外，还了解了不同语言中的一些重要词汇。

下一章介绍 XAML，XAML 用于 WPF、Silverlight、XPS、Windows Workflow Foundation 和 Windows Store 应用程序，它提供了许多技术的基础。

第29章

核心 XAML

本章要点

- XAML 语法
- 依赖属性
- 附加属性
- 标记扩展
- 动态加载 XAML

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 代码简介
- XAML 简介
- XAML 语法
- 依赖对象
- 冒泡演示
- 附加属性
- 标记扩展

29.1 XAML 的作用

编写.NET 应用程序时，需要了解的通常不仅仅是 C#语法。如果编写 WPF 应用程序、使用 WF、创建 XPS 文档、编写 Silverlight 或者 Windows 8 应用程序，就还需要 XAML。XAML(eXtensible Application Markup Language，可扩展应用程序标记语言)是一种声明性的 XML 语法，上述这些应用程序通常需要 XAML。本章详细介绍 XAML 的语法，以及可用于这种标记语言的扩展机制。

29.2 XAML 概述

XAML 代码使用文本 XML 来声明。XAML 代码可以使用设计器创建，也可以手动编写。Visual Studio 包含的设计器可给 WPF、Silverlight、WF 或 Windows Store 应用程序编写 XAML 代码。也可以使用其他工具创建 XAML，如 Microsoft Expression Design 和 Microsoft Expression Blend。

XAML 和几种技术一起使用，但这些技术是有区别的。利用 XML 名称空间 http://schemas.microsoft.com/winfx/2006/xaml/presentation(通过 WPF 和 Windows Store 应用程序映射为默认名称空间)，可以定义 WPF 对 XAML 的扩展。WPF 使用依赖属性、附加属性和几个 WPF 特有的标记扩展。WF 4 使用 XML 名称空间 http://schemas.microsoft.com/netfx/2009/xaml/activities 来定义工作流活动。XML 名称空间 http://schemas.microsoft.com/winfx/2009/xaml 通常映射到 x 前缀上，并定义对所有 XAML 词汇通用的功能。

在 WPF 应用程序中，XAML 元素映射到.NET 类。对于 XAML，这并不是一个严格的要求。在 Silverlight 1.0 中，.NET 不能用于插件，只能使用 JavaScript 解释和通过编程方法访问 XAML 代码。这种情况自 Silverlight 2.0 以来有了改变，.NET Framework 的一个小型版本是 Silverlight 插件的一部分。对于 WPF，每个 XAML 元素都对应一个类。WF 也是这样，例如，XAML 元素 DoWhile 是一个循环活动，名称空间 System.Activities 中的 DoWhile 类支持它。XAML 元素 Button 与名称空间 System.Windows.Controls 中的 Button 类相同。在 Windows 8 应用程序中，每个 XAML 元素都映射到一个.NET 类或 Windows Runtime 类。

把.NET 名称空间映射到 XML 别名上，也可以在 XML 中使用自定义.NET 类，参见后面关于使用自定义.NET 类的内容。在.NET 4 中，XAML 的语法得到了增强，这个版本称为 XAML 2009。XAML 的第一个版本是 XAML 2006，它在 XML 名称空间 http://schemas.microsoft.com/winfx/2006/xaml/presentation 中定义。XAML 的新版本支持增强语法，如 XAML 代码中的泛型。但 Visual Studio 2013 版本中的 WPF、WF 和 Windows Store 应用程序设计器仍基于 XAML 2006。可以在应用程序中直接使用 XAML 2009 加载 XAML。本章将介绍 XAML 2009 的变化。

XAML 代码在生成过程中会发生什么？为了编译 WPF 项目，应在程序集 PresentationBuildTasks 中定义 MSBuild 任务 MarkupCompilePass1 和 MarkupCompilePass2。这些 MSBuild 任务会创建标记代码的二进制表示 BAML(Binary Application Markup Language，二进制应用程序标记语言)，并添加到程序集的.NET 资源中。在运行期间，会使用该二进制表示。

读写 XAML 和 BAML 可以通过读取器和写入器来完成。在 System.Xaml 名称空间中，包含用于核心 XAML 特性的类，如抽象的 XamlReader 类和 XamlWriter 类，以及读写对象和 XAML XML 格式的具体实现方式。System.Windows.Markup 名称空间中的一些特性可用于使用程序集 System.Xaml 中的 XAML 的所有技术。在这个名称空间中，位于 PresentationFramework 程序集的类是 WPF 专用的扩展。例如，为 WPF 特性进行了优化的其他 XamlReader 类和 XamlWriter 类就在该程序集中。

29.2.1 元素如何映射到.NET 对象上

如上一节所述，XAML 元素通常映射到.NET 类上。下面利用 C#控制台项目在 Window 中通过编程方式创建一个 Button 对象。如下面的代码所示，实例化一个 Button 对象，并把其 Content 属性

设置为一个字符串；定义一个 Window，设置其 Title 和 Content 属性。要编译这段代码，需要引用程序集 PresentationFramework、PresentationCore、WindowBase 和 System.Xaml(代码文件 CodeIntro/Program.cs)。

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.XAML
{
  class Program
  {
    [STAThread]
    static void Main()
    {
      var b = new Button
      {
        Content = "Click Me!"
      };
      var w = new Window
      {
        Title = "Code Demo",
        Content = b
      };

      var app = new Application();
      app.Run(w);
    }
  }
}
```

使用 XAML 代码可以创建类似的 UI。与以前一样，这段代码创建了一个包含 Button 元素的 Window 元素。为 Window 元素设置了其内容和 Title 特性(XAML 文件 XAMLIntro/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="XAML Demo" Height="350" Width="525">
  <Button Content="Click Me!" />
</Window>
```

当然，上面的代码没有定义 Application 实例，它也可以用 XAML 定义。在 Application 元素中，设置了 StartupUri 特性，它链接到包含主窗口的 XAML 文件上(XAML 文件 XAMLIntro/App.xaml)。

```
<Application x:Class="Wrox.ProCSharp.XAML.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

29.2.2 使用自定义.NET 类

要在 XAML 代码中使用自定义.NET 类，只需要在 XAML 中声明.NET 名称空间，并定义一个 XML 别名。为了说明这个过程，下面定义了一个简单的 Person 类及其 FirstName 和 LastName 属性(代码文件 DemoLib/Person.cs)。

```
namespace Wrox.ProCSharp.XAML
{
  public class Person
  {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public override string ToString()
    {
      return string.Format("{0} {1}", FirstName, LastName);
    }
  }
}
```

在 XAML 中，定义一个 XML 名称空间别名 local，它映射到.NET 名称空间 Wrox.ProCSharp.XAML 上。现在可以通过该别名使用这个名称空间中的所有类。在 WPF 中，关键字 clr-namespace 映射到.NET 名称空间，而在 Windows 8 应用程序中，使用 using 关键字映射.NET 或 Windows Runtime 名称空间。

在 XAML 代码中，添加了一个列表框，其中包含 Person 类型的项。使用 XAML 特性，设置 FirstName 和 LastName 属性的值。运行该应用程序时，ToString()方法的输出会显示在列表框中(XAML 文件 XAMLIntro/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="XAML Demo" Height="350" Width="525">
  <StackPanel>
    <Button Content="Click Me!" />
    <ListBox>
      <local:Person FirstName="Stephanie" LastName="Nagel" />
      <local:Person FirstName="Matthias" LastName="Nagel" />
    </ListBox>
  </StackPanel>
</Window>
```

如果.NET 名称空间与 XAML 代码不在同一个程序集中，就必须在 XAML 名称空间别名中包含该程序集名，例如 xmlns:local="clr-namespace:Wrox.ProCSharp.XAML; assembly= XAMLIntro"。这里，私有和共享程序集都可以通过该引用来使用。对于共享程序集，需要指定其全名，包括版本号、区域性和程序集的密钥令牌。共享程序集的详细信息可参见第 19 章。

要把.NET 名称空间映射到 XML 名称空间上，可以使用程序集特性 XmlnsDefinition。这个特性的一个参数定义了 XML 名称空间，另一个参数定义了.NET 名称空间。使用这个特性，也可以把多个.NET 名称空间映射到一个 XML 名称空间上(代码文件 DemoLib/AssemblyInfo.cs)。

```
[assembly: XmlnsDefinition("http://www.wrox.com/Schemas/2010", "Wrox.ProCSharp.XAML")]
```

有了这个特性，XAML 代码中的名称空间声明就可以改为映射到 XML 名称空间上(XAML 文件 XAMLIntro/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="http://www.wrox.com/Schemas/2010"
        Title="XAML Demo" Height="350" Width="525">
  <StackPanel>
    <Button Content="Click Me!" />
    <ListBox>
      <local:Person FirstName="Stephanie" LastName="Nagel" />
      <local:Person FirstName="Matthias" LastName="Nagel" />
    </ListBox>
  </StackPanel>
</Window>
```

29.2.3　把属性用作特性

只要属性的类型可以表示为字符串，或者可以把字符串转换为属性类型，就可以把属性设置为特性。下面的代码片段用特性设置了 Button 元素的 Content 和 Background 属性。因为 Content 属性的类型是 object，所以可以接受字符串。Background 属性的类型是 Brush，Brush 类型把 BrushConverter 类定义为一个转换器类型，这个类用 TypeConverter 特性进行注解。BrushConverter 使用一个颜色列表，从 ConvertFromString()方法中返回一个 SolidColorBrush。

```
<Button Content="Click Me!" Background="LightGoldenrodYellow" />
```

类型转换器派生自 System.ComponentModel 名称空间中的基类 TypeConverter。需要转换的类的类型用 TypeConverter 特性定义了类型转换器。WPF 使用许多类型转换器把 XML 特性转换为特定的类型。ColorConverter、FontFamilyConverter、PathFigureCollectionConverter、ThicknessConverter、GeometryConverter 是大量类型转换器中的几个。

29.2.4　把属性用作元素

总是可以使用元素语法给属性提供值。Button 类的 Background 属性可以用子元素 Button.Background 设置。这样，可以把比较复杂的画笔应用于这个属性(如 LinearGradientBrush)，如下面的示例所示。

下面的示例在设置内容时，既没有使用 Content 特性也没有使用 Button.Content 元素来写入内容，而是直接把内容写入为 Button 元素的一个子值。这是可行的，因为 Button 类的一个基类是 Content-

Control，所以应用了 ContentProperty 特性，它把 Content 属性标记为 ContentProperty: [ContentProperty("Content")]。通过这个标记属性，就可以把属性的值写入为子元素(XAML 文件 XAMLSyntax/MainWindow.xaml)。

```
<Button>
  Click Me!
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
      <GradientStop Offset="0" Color="Yellow" />
      <GradientStop Offset="0.3" Color="Orange" />
      <GradientStop Offset="0.7" Color="Red" />
      <GradientStop Offset="1" Color="DarkRed" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

29.2.5 基本的.NET 类型

在 XAML 2006 中，核心.NET 类型需要从 XML 名称空间中引用，这与所有其他的.NET 类相同。例如，String 用 sys 别名来引用，如下所示：

```
<sys:String xmlns:sys="clr-namespace:System;assembly=mscorlib>Simple String</sys:String>
```

XAML 2009 用别名 x 定义了 String、Boolean、Object、Decimal、Double、Int32 等类型。

```
<x:String>Simple String</x:String>
```

29.2.6 使用集合和 XAML

在包含 Person 元素的 ListBox 中，已经使用过 XAML 中的集合。在 ListBox 中，列表项直接定义为子元素。另外，LinearGradientBrush 包含了一个 GradientStop 元素集合。这是可行的，因为基类 ItemsControl 把 ContentProperty 特性设置为该类的 Items 属性，GradientBrush 基类把 ContentProperty 特性设置为 GradientStops。

下面的长版本代码在定义背景时直接设置了 GradientStops 属性，并把 GradientStopCollection 元素设置为它的子元素：

```
<Button Click="OnButtonClick">
   Click Me!
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
      <LinearGradientBrush.GradientStops>
        <GradientStopCollection>
          <GradientStop Offset="0" Color="Yellow" />
          <GradientStop Offset="0.3" Color="Orange" />
          <GradientStop Offset="0.7" Color="Red" />
          <GradientStop Offset="1" Color="DarkRed" />
        </GradientStopCollection>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

要定义数组，可以使用x:Array扩展。x:Array扩展有一个Type属性，可以用于指定数组元素的类型。

```
<x:Array Type="local:Person">
  <local:Person FirstName="Stephanie" LastName="Nagel" />
  <local:Person FirstName="Matthias" LastName="Nagel" />
</x:Array>
```

Windows Store应用程序不支持x:Array扩展。

因为XAML 2006不支持泛型，所以要在XAML中使用泛型集合类，就需要定义一个派生自泛型类的非泛型类，再使用这个非泛型类。XAML 2009直接支持泛型，允许通过x:TypeArguments特性定义泛型类型，如下面的ObservableCollection<T>所示：

```
<ObservableCollection x:TypeArguments="local:Person">
  <local:Person FirstName="Stephanie" LastName="Nagel" />
  <local:Person FirstName="Matthias" LastName="Nagel" />
</ObservableCollection>
```

29.2.7 用XAML代码调用构造函数

如果类没有默认的构造函数，就不能在XAML 2006中使用它。在XAML 2009中，可以使用x:Arguments以通过参数调用构造函数。下面的Person类用构造函数进行实例化，该构造函数需要两个String参数：

```
<local:Person>
  <x:Arguments>
    <x:String>Stephanie</x:String>
    <x:String>Nagel</x:String>
  </x:Arguments>
</local:Person>
```

29.3 依赖属性

WPF使用依赖属性完成数据绑定、动画、属性变更通知、样式化等。对于数据绑定，绑定到.NET属性源上的UI元素的属性必须是依赖属性。

从外部来看，依赖属性像是正常的.NET属性。但是，正常的.NET属性通常还定义了由该属性的get和set访问器访问的数据成员。

```
private int val;
public int Value
{
   get
   {
      return val;
   }
   set
   {
```

```
            val = value;
        }
    }
```

依赖属性不是这样。依赖属性通常也有 get 和 set 访问器。在 get 和 set 访问器的实现代码中，调用了 GetValue()和 SetValue()方法。GetValue()和 SetValue()方法是基类 DependencyObject 的成员，依赖对象需要使用这个类——它们必须在 DependencyObject 的派生类中实现。

有了依赖属性，数据成员就放在由基类管理的内部集合中，仅在值发生变化时分配数据。对于没有变化的值，数据可以在不同的实例或基类之间共享。GetValue()和 SetValue()方法需要一个 DependencyProperty 参数。这个参数由类的一个静态成员定义，该静态成员与属性同名，并在该属性名的后面追加 Property 术语。对于 Value 属性，静态成员的名称是 ValueProperty。DependencyProperty.Register()是一个辅助方法，可在依赖属性系统中注册属性。在下面的代码片段中，使用 Register()方法和 3 个参数定义了属性名、属性的类型和拥有者(即 MyDependencyObject 类)的类型(代码文件 DependencyObjectDemo/MyDependencyObject.cs)。

```
public int Value
{
  get { return (int)GetValue(ValueProperty); }
  set { SetValue(ValueProperty, value); }
}
public static readonly DependencyProperty ValueProperty =
   DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject));
```

29.3.1 创建依赖属性

下面的示例定义的不是一个依赖属性，而是 3 个依赖属性。MyDependencyObject 类定义了依赖属性 Value、Minimum 和 Maximum。所有这些属性都是用 DependencyProperty.Register()方法注册的依赖属性。GetValue()和 SetValue()方法是基类 DependencyObject 的成员。对于 Minimum 和 Maximum 属性，定义了默认值，用 DependencyProperty.Register()方法设置该默认值时，可以把第 4 个参数设置为 ProperMetaData。使用带一个参数 ProperMetaData 的构造函数，把 Minimum 属性设置为 0，把 Maximum 属性设置为 100。

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  class MyDependencyObject : DependencyObject
  {
    public int Value
    {
      get { return (int)GetValue(ValueProperty); }
      set { SetValue(ValueProperty, value); }
    }
    public static readonly DependencyProperty ValueProperty =
       DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject));
    public int Minimum
    {
      get { return (int)GetValue(MinimumProperty); }
      set { SetValue(MinimumProperty, value); }
```

```
    }
    public static readonly DependencyProperty MinimumProperty =
       DependencyProperty.Register("Minimum", typeof(int), typeof(MyDependencyObject),
                                   new PropertyMetadata(0));
    public int Maximum
    {
      get { return (int)GetValue(MaximumProperty); }
      set { SetValue(MaximumProperty, value); }
    }
    public static readonly DependencyProperty MaximumProperty =
       DependencyProperty.Register("Maximum", typeof(int), typeof(MyDependencyObject),
                                   new PropertyMetadata(100));
  }
}
```

在 get 和 set 属性访问器的实现代码中，只能调用 GetValue()和 SetValue()方法。使用依赖属性，可以通过 GetValue()和 SetValue()方法从外部访问属性的值，WPF 也是这样做的；因此，强类型化的属性访问器可能根本就不会被调用，包含它们仅为了方便在自定义代码中使用正常的属性语法。

29.3.2 强制值回调

依赖属性支持强制检查。通过强制检查，可以检查属性的值是否有效，例如该值是否在某个有效范围之内。因此，本例包含了 Minimum 和 Maximum 属性。现在 Value 属性的注册变更为把事件处理方法 CoerceValue()传递给 ProperMetaData 对象的构造函数，再把 ProperMetaData 对象传递为 DependencyProperty.Register()方法的一个参数。现在，在 SetValue()方法的实现代码中，对属性值的每次变更都调用 CoerceValue()方法。在 CoerceValue()方法中，检查 set 值是否在指定的最大值和最小值之间；如果不在，就设置该值(代码文件 DependencyObjectDemo/MyDependencyObject.cs)。

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  class MyDependencyObject : DependencyObject
  {
    public int Value
    {
      get { return (int)GetValue(ValueProperty); }
      set { SetValue(ValueProperty, value); }
    }
    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject));
        new PropertyMetadata(0, null, CoerceValue));
    public int Minimum
    {
      get { return (int)GetValue(MinimumProperty); }
      set { SetValue(MinimumProperty, value); }
    }
    public static readonly DependencyProperty MinimumProperty =
```

```
        DependencyProperty.Register("Minimum", typeof(int), typeof(MyDependencyObject),
                                    new PropertyMetadata(0));
    public int Maximum
    {
      get { return (int)GetValue(MaximumProperty); }
      set { SetValue(MaximumProperty, value); }
    }
    public static readonly DependencyProperty MaximumProperty =
        DependencyProperty.Register("Maximum", typeof(int), typeof(MyDependencyObject),
                                    new PropertyMetadata(100));
    private static object CoerceValue(DependencyObject element, object value)
    {
      int newValue = (int)value;
      MyDependencyObject control = (MyDependencyObject)element;

      newValue = Math.Max(control.Minimum, Math.Min(control.Maximum, newValue));
      return newValue;
    }
  }
}
```

29.3.3 值变更回调和事件

为了获得值变更的信息，依赖属性还支持值变更回调。在属性值发生变化时调用的 DependencyProperty.Register()方法中，可以添加一个 DependencyPropertyChanged 事件处理程序。在示例代码中，把 OnValueChanged()处理方法赋予 ProperMetaData 对象的 PropertyChangedCallback 属性。在 OnValueChanged()方法中，可以用 DependencyPropertyChangedEventArgs()参数访问属性的新旧值。

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  class MyDependencyObject : DependencyObject
  {
    public int Value
    {
      get { return (int)GetValue(ValueProperty); }
      set { SetValue(ValueProperty, value); }
    }
    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject),
        new PropertyMetadata(0, OnValueChanged, CoerceValue));
    //...
    private static void OnValueChanged(DependencyObject obj,
                                       DependencyPropertyChangedEventArgs args)
    {
      int oldValue = (int)args.OldValue;
      int newValue = (int)args.NewValue;
      //...
    }
  }
}
```

29.3.4 事件的冒泡和隧道

元素可以包含在其他元素中。通过 XAML 和 WPF，可以定义 Button 包含一个 Listbox，该 Listbox 又包含 Button 控件项。单击一个内层 Button 控件时，Click 事件就应传递到外部。Click 事件是一个冒泡事件。PreviewMouseMove 事件是一个隧道事件，它从外部向内部移动，外部控件首先接收到内部控件引发的事件。MouseMove 事件跟在 PreviewMouseMove 事件的后面，是一个冒泡事件，从内部向外部移动。WPF 支持事件的冒泡和隧道，这些事件常常成对使用。

.NET 事件的核心信息参见第 8 章。

为了说明冒泡过程，下面的 XAML 代码包含 4 个 Button 控件，其中外部的 StackPanel 给 Button.Click 事件定义了一个事件处理程序 OnOuterButtonClick()。button2 包含一个 Listbox，该 Listbox 又包含两个 Button 控件作为其子项，还包含 Click 事件处理程序 OnButton2()。这两个内部按钮也有与 Click 事件关联的事件处理程序(XAML 文件 BubbleDemo/MainWindow.xaml)。

```
<Window x:Class=" Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
  <StackPanel x:Name="stackPanel1" Button.Click="OnOuterButtonClick">
    <Button x:Name="button1" Content="Button 1" Margin="5" />
    <Button x:Name="button2" Margin="5" Click="OnButton2" >
      <ListBox x:Name="listBox1">
        <Button x:Name="innerButton1" Content="Inner Button 1" Margin="4" Padding="4"
                Click="OnInner1" />
        <Button x:Name="innerButton2" Content="Inner Button 2" Margin="4" Padding="4"
                Click="OnInner2" />
      </ListBox>
    </Button>
    <ListBox ItemsSource="{Binding}" />
  </StackPanel>
</Window>
```

事件处理方法在代码隐藏中实现。该处理方法的第二个参数是 RoutedEventArgs 类型，它提供了事件的 Source 和 OriginalSource 信息。尽管这个按钮并没有直接关联的 Click 事件，但单击 Button1 按钮时，会调用处理方法 OnOuterButtonClick()；该事件会向上冒泡到容器元素。这里，Source 和 OriginalSource 是 Button1。如果第一次单击 Button2，就调用事件处理程序 OnButton2()，之后调用 OnOuterButtonClick()。因为处理程序 OnButton2()改变了 Source 属性，所以通过 OnOuterButtonClick() 会看到与前面的处理程序不同的 Source。事件的 Source 属性可以改变，但 OriginalSource 是只读的。单击 innerButton1 按钮，会调用 OnInner1()事件处理程序，之后调用 OnButton2()和 OnOuterButtonClick()。事件向上冒泡。单击 innerButton2 按钮，仅调用处理程序 OnInner2()，因为其 Handled 属性设置为 true。在这里事件的向上冒泡停止了(代码文件 BubbleDemo/MainWindow.xaml.cs)。

```
using System;
using System.Collections.ObjectModel;
using System.Windows;
```

```
namespace BubbleDemo
{
  public partial class MainWindow : Window
  {
    private ObservableCollection<string> messages = new ObservableCollection<string>();
    public MainWindow()
    {
      InitializeComponent();
      this.DataContext = messages;
    }
    private void AddMessage(string message, object sender, RoutedEventArgs e)
    {
      messages.Add(String.Format("{0}, sender: {1}; source: {2}; original source: {3}",
          message, (sender as FrameworkElement).Name,
          (e.Source as FrameworkElement).Name,
          (e.OriginalSource as FrameworkElement).Name));
    }
    private void OnOuterButtonClick(object sender, RoutedEventArgs e)
    {
     AddMessage("outer event", sender, e);
    }
    private void OnInner1(object sender, RoutedEventArgs e)
    {
     AddMessage("inner1", sender, e);
    }
    private void OnInner2(object sender, RoutedEventArgs e)
    {
      AddMessage("inner2", sender, e);
      e.Handled = true;
    }
    private void OnButton2(object sender, RoutedEventArgs e)
    {
      AddMessage("button2", sender, e);
      e.Source = sender;
    }
  }
}
```

改变源的同时也改变事件类型很常见。例如，Button 类响应鼠标的按下和释放事件，向上冒泡它们，而不是创建按钮单击事件。

如果不同处理程序的实现代码非常类似(如容器中的多个按钮)，则在容器控件中只编写一个事件处理程序来响应冒泡事件非常有益。在实现代码中，只需要区分 sender 或 source。

要在自定义类中定义事件的冒泡和隧道，应把 MyDependencyObject 更改为支持值变更时发生的事件。为了支持事件的冒泡和隧道，该类必须派生自 UIElement，而不是 DependencyObject，因为这个类要为事件定义 AddHandler()和 RemoveHandler()方法。

为了让 MyDependencyObject 的调用者接收到值变更的信息，该类定义了 ValueChanged 事件。这个事件用显式的 add 和 remove 处理程序来声明，其中调用了基类的 AddHandler()和 RemoveHandler()方法。AddHandler()和 RemoveHandler()方法需要一个 RoutedEvent 类型和委托作为参数。路由事件 ValueChangedEvent 的声明与依赖属性非常类似，它也声明为一个静态成员，并调用 EventManagerRegisterRoutedEvent()方法来注册。这个方法需要事件名、路由策略(可以是 Bubble、Tunnel 和 Direct)、处理程序的类型，以及所有者类的类型。EventManager 类还可以注册静态事件，获取所注册事件的信息(代码文件 DependencyObjectDemo/MyDependencyObject.cs)。

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  class MyDependencyObject : UIElement
  {
    public int Value
    {
      get { return (int)GetValue(ValueProperty); }
      set { SetValue(ValueProperty, value); }
    }
    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject),
        new PropertyMetadata(0, OnValueChanged, CoerceValue));
    //...
    private static void OnValueChanged(DependencyObject obj,
                                       DependencyPropertyChangedEventArgs args)
    {
      MyDependencyObject control = (MyDependencyObject)obj;
      var e = new RoutedPropertyChangedEventArgs<int>((int)args.OldValue,
          (int)args.NewValue, ValueChangedEvent);
      control.OnValueChanged(e);
    }
    public static readonly RoutedEvent ValueChangedEvent =
        EventManager.RegisterRoutedEvent("ValueChanged", RoutingStrategy.Bubble,
            typeof(RoutedPropertyChangedEventHandler<int>), typeof(MyDependencyObject));
    public event RoutedPropertyChangedEventHandler<int> ValueChanged
    {
      add
      {
        AddHandler(ValueChangedEvent, value);
      }
      remove
      {
        RemoveHandler(ValueChangedEvent, value);
      }
    }
    protected virtual void OnValueChanged(RoutedPropertyChangedEventArgs<int> args)
    {
      RaiseEvent(args);
    }
  }
}
```

现在就可以使用冒泡功能，其方式与以前使用按钮单击事件一样。

29.4 附加属性

依赖属性是可用于特定类型的属性。而通过附加属性，可以为其他类型定义属性。一些容器控件为其子控件定义了附加属性；例如，如果使用 DockPanel 控件，就可以为其子控件使用 Dock 属性。Grid 控件定义了 Row 和 Column 属性。

下面的代码片段说明了附加属性在 XAML 中的情况。Button 类没有 Dock 属性，但它是从 DockPanel 控件附加的。

```
<DockPanel>
  <Button Content="Top" DockPanel.Dock="Top" Background="Yellow" />
  <Button Content="Left" DockPanel.Dock="Left"Background="Blue" />
</DockPanel>
```

附加属性的定义与依赖属性非常类似，如下面的示例所示。定义附加属性的类必须派生自基类 DependencyObject，并定义一个普通的属性，其中 get 和 set 访问器访问基类的 GetValue()和 SetValue()方法。这些都是类似之处。接着不调用 DependencyProperty 类的 Register()方法，而是调用 RegisterAttached()方法。RegisterAttached()方法注册一个附加属性，现在它可用于每个元素(代码文件 AttachedPropertyDemo/MyAttachedPropertyProvider.cs)。

```
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  class MyAttachedPropertyProvider : DependencyObject
  {
    public int MyProperty
    {
      get { return (int)GetValue(MyPropertyProperty); }
      set { SetValue(MyPropertyProperty, value); }
    }
    public static readonly DependencyProperty MyPropertyProperty =
        DependencyProperty.RegisterAttached("MyProperty", typeof(int),
            typeof(MyAttachedPropertyProvider));
    public static void SetMyProperty(UIElement element, int value)
    {
      element.SetValue(MyPropertyProperty, value);
    }
    public static int GetMyProperty(UIElement element)
    {
      return (int)element.GetValue(MyPropertyProperty);
    }
  }
}
```

似乎 DockPanel.Dock 属性只能添加到 DockPanel 控件中的元素。实际上，附加属性可以添加到任何元素上。但无法使用这个属性值。DockPanel 控件能够识别这个属性，并从其子元素中读取它，以安排其子元素。

在 XAML 代码中，附加属性现在可以附加到任何元素上。第二个 Button 控件 button2 为自身附加了属性 MyAttachedPropertyProvider.MyProperty，其值指定为 5(XAML 文件 AttachedPropertyDemo/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="MainWindow" Height="350" Width="525">
  <Grid x:Name="grid1">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Button Grid.Row="0" x:Name="button1" Content="Button 1" />
    <Button Grid.Row="1" x:Name="button2" Content="Button 2"
            local:MyAttachedPropertyProvider.MyProperty="5" />
    <ListBox Grid.Row="2" x:Name="list1" />
  </Grid>
</Window>
```

在代码隐藏中执行相同的操作时，必须调用 MyAttachedPropertyProvider 类的静态方法 SetMyProperty()。不能扩展 Button 类，使其包含某个属性。SetMyProperty()方法获取一个应由该属性及其值扩展的 UIElement 实例。在如下的代码片段中，把该属性附加到 button1 中，其值设置为 44。

属性设置之后的 foreach 循环从 Grid 元素 grid1 的所有子元素中检索附加属性的值。检索这个值使用了 MyAttachedPropertyProvider 类的 GetProperty()方法。它从 DockPanel 和 Grid 控件中检索其子控件的设置，以安排这些子控件(代码文件 AttachedPropertyDemo/MainWindow.xaml.cs)。

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
      MyAttachedPropertyProvider.SetMyProperty(button1, 44);
      foreach (object item in LogicalTreeHelper.GetChildren(grid1))
      {
        FrameworkElement e = item as FrameworkElement;
```

```
        if (e != null)
          list1.Items.Add(String.Format("{0}: {1}", e.Name,
            MyAttachedPropertyProvider.GetMyProperty(e)));
      }
    }
  }
}
```

有一些机制可用于以后扩展类。扩展方法可以用于扩展带方法的任何类。扩展方法只能扩展带方法但不带属性的类。扩展方法参见第3章。ExpandoObject类允许扩展带方法和属性的类。要使用这个功能，类必须派生自ExpandoObject类。ExpandoObject类和动态类型参见第12章。

第35章和第36章将介绍许多不同的附加属性，例如容器控件Canvas、DockPanel、Grid的附加属性，以及Validation类的ErrorTemplate属性。

29.5 标记扩展

通过标记扩展，可以扩展XAML的元素或特性语法。如果XML特性包含花括号，就表示这是标记扩展的一个符号。特性的标记扩展常常用作简写记号，而不再使用元素。

这种标记扩展的示例是StaticResourceExtension，它可查找资源。下面是带有gradientBrush1键的线性渐变笔刷的资源(XAML文件MarkupExtensionDemo/MainWindow.xaml)：

```
<Window.Resources>
  <LinearGradientBrush x:Key="gradientBrush1" StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
    <GradientStop Offset="0" Color="Yellow" />
    <GradientStop Offset="0.3" Color="Orange" />
    <GradientStop Offset="0.7" Color="Red" />
    <GradientStop Offset="1" Color="DarkRed" />
  </LinearGradientBrush>
</Window.Resources>
```

使用StaticResourceExtension，通过特性语法来设置TextBlock的Background属性，就可以引用这个资源。特性语法通过花括号和没有Extension后缀的扩展类名来定义。

```
<TextBlock Text="Test" Background="{StaticResource gradientBrush1}" />
```

该特性简写记号的较长形式使用元素语法，如下面的代码片段所示。StaticResourceExtension定义为TextBlock.Background元素的一个子元素。通过一个特性把ResourceKey属性设置为gradientBrush1。在上面的示例中，没有用ResourceKey属性设置资源键(这也是可行的)，但使用一个构造函数重载来设置资源键。

```
<TextBlock Text="Test">
```

```
    <TextBlock.Background>
      <StaticResourceExtension ResourceKey="gradientBrush1" />
    </TextBlock.Background>
  </TextBlock>
```

29.6 创建自定义标记扩展

要创建标记扩展，可以定义基类 MarkupExtension 的一个派生类。大多数标记扩展名都有 Extension 后缀(这个命名约定类似于特性的 Attribute 后缀，参见第 15 章)。有了自定义标记扩展后，就只需要重写 ProvideValue()方法，它返回扩展的值。返回的类型用 MarkupExtensionReturnType 特性注解类。对于 ProvideValue()方法，需要传递一个 IServiceProvider 对象。通过这个接口，可以查询不同的服务，如 IProvideValueTarget 或 IXamlTypeResolver。IProvideValueTarget 可以用于访问通过 TargetObject 和 TargetProperty 属性应用标记扩展的控件和属性。IXamlTypeResolver 可用于把 XAML 元素名解析为 CLR 对象。自定义标记扩展类 CalculatorExtension 定义了 double 类型的属性 X 和 Y，并通过枚举定义了一个 Operation 属性。根据 Operation 属性的值，在 X 和 Y 输入属性上执行不同的计算，并返回一个字符串(代码文件 MarkupExtensionDemo/CalculatorExtension.cs)。

```
using System;
using System.Windows;
using System.Windows.Markup;
namespace Wrox.ProCSharp.XAML
{
  public enum Operation
  {
    Add,
    Subtract,
    Multiply,
    Divide
  }
  [MarkupExtensionReturnType(typeof(string))]
  public class CalculatorExtension : MarkupExtension
  {
    public CalculatorExtension()
    {
    }
    public double X { get; set; }
    public double Y { get; set; }
    public Operation Operation { get; set; }
    public override object ProvideValue(IServiceProvider serviceProvider)
    {
      IProvideValueTarget provideValue =
          serviceProvider.GetService(typeof(IProvideValueTarget))
          as IProvideValueTarget;
      if (provideValue != null)
      {
        var host = provideValue.TargetObject as FrameworkElement;
        var prop = provideValue.TargetProperty as DependencyProperty;
      }
```

```
        double result = 0;
        switch (Operation)
        {
          case Operation.Add:
            result = X + Y;
            break;
          case Operation.Subtract:
            result = X - Y;
            break;
          case Operation.Multiply:
            result = X * Y;
            break;
          case Operation.Divide:
            result = X / Y;
            break;
          default:
            throw new ArgumentException("invalid operation");
        }
        return result.ToString();
      }
    }
  }
```

标记扩展现在可以在第一个 TextBlock 上与特性语法一起使用，把值 3 和 4 加在一起，或者在第二个 TextBlock 上与元素语法一起使用(XAML 文件 MarkupExtensionDemo/MainWindow.xaml)。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="MainWindow" Height="350" Width="525">
  <StackPanel>
    <TextBlock Text="{local:Calculator Operation=Add, X=3, Y=4}" />
    <TextBlock>
      <TextBlock.Text>
        <local:CalculatorExtension>
          <local:CalculatorExtension.Operation>
            <local:Operation>Multiply</local:Operation>
          </local:CalculatorExtension.Operation>
          <local:CalculatorExtension.X>7</local:CalculatorExtension.X>
          <local:CalculatorExtension.Y>11</local:CalculatorExtension.Y>
        </local:CalculatorExtension>
      </TextBlock.Text>
    </TextBlock>
  </StackPanel>
</Window>
```

29.7 XAML 定义的标记扩展

标记扩展提供了许多功能，实际上本章已经使用了 XAML 定义的标记扩展。21.6 节中的 x:Array 就定义为标记扩展类 ArrayExtension。有了这个标记扩展，就可以不使用特性语法，因为使用特性语法很难定义元素列表。

用 XAML 定义的其他标记扩展有 TypeExtension(x:Type)，它根据字符串输入返回类型；NullExtension(x:Null)可以用于在 XAML 中把值设置为空；StaticExtension(x:static)可调用类的静态成员。

WPF、WF 和 WCF 都定义了专用于这些技术的标记扩展。WPF 使用标记扩展访问资源，以用于数据绑定和颜色转换。WF 联合使用了标记扩展和活动；WCF 给端点定义指定了标记扩展。

29.8 读写 XAML

有几个 API 可用于读写 XAML。高级 API 易于使用，但功能较少，而低级 API 的功能较多。还存在专用技术的 API，它们使用专门的 WPF 或 WF 特性。XAML 可以从文本 XML 形式、BAML 或对象树中读取，还可以写入 XML 或对象树。

一般的高级 API 位于 System.Xaml 名称空间中。XamlService 类允许加载、分析、保存和转换 XAML。XamlService.Load()方法可以从文件、流或读取器中加载 XAML 代码，或者使用 XamlReader 对象来加载。XamlReader(在 System.Xaml 名称空间中)是一个抽象基类，它有几种具体的实现方式。XamlObjectReader 读取对象树，XamlXmlReader 从 XML 文件中读取 XAML，Baml2006Reader 读取二进制形式的 XAML。System.Activities.Debugger 名称空间中的 XamlDebuggerXmlReader 是一个专用于 WF 的读取器，并具备特殊的调试支持。

把字符串中的 XAML 代码传递给 XamlService 时，可以使用 Parse()方法。XamlService.Save()方法可用于保存 XAML 代码。通过 Save()方法可以使用与 Load()方法类似的数据源。所传递的对象可以保存为字符串、流、TextWriter、XamlWriter 或 XmlWriter。XamlWriter 是一个抽象基类。派生自 XamlWriter 的类是 XamlObjectWriter 和 XamlXmlWriter。

把 XAML 从一种格式转换为另一种格式时，可以使用 XamlService.Transform()方法。给 Transform()方法传递 XamlReader 和 XamlWriter，就可以转换特定读取器和写入器支持的任意格式。

除了使用高级 API XamlService 类之外，还可以直接使用一般的低级 API，也就是说，使用特定的 XamlReader 和 XamlWriter 类。使用读取器，可以通过 Read()方法从 XAML 树中逐个节点地读取。

一般的 XamlService 类不支持特定的 WPF 特性，如依赖属性或可冻结的对象。要读写 WPF XAML，可以使用 System.Windows.Markup 名称空间中的 XamlReader 和 XamlWriter 类，该名称空间在 PresentationFramework 程序集中定义，因此可以访问 WPF 特性。这些类的名称可能令人迷惑，因为不同名称空间的类使用相同的名称。System.Xaml.XamlReader 是读取器的抽象基类，System.Windows.Markup. XamlReader 是读取 XAML 的 WPF 类。在使用 WPF XamlReader 类的 Load()方法时，它接受 System.Xaml.XamlReader 作为参数，所以可能更容易混淆。

给 WF 读取 XAML 的一个优化版本是 System.Activities 名称空间中的 WorkflowXamlService，这个类用于在运行期间创建动态活动。

动态活动和 Windows Workflow Foundation 参见第 45 章。

下面的简单示例从文件中动态加载 XAML，以创建一个对象树，并把该对象树附加到一个容器元素中，如 StackPanel：

```
FileStream stream = File.OpenRead("Demo1.xaml");
object tree = System.Windows.Markup.XamlReader.Load(stream);
container1.Children.Add(tree as UIElement);
```

29.9 小结

本章介绍了 XAML 的核心功能和一些专门的特性，如依赖属性、附加属性、事件的冒泡和隧道，以及标记扩展。通过这些特性，不仅介绍了基于 XAML 技术的基础知识，还讨论了 C#和.NET 特性(如属性和事件)如何适用于已扩展的用例。属性增强为支持变更通知和有效性验证(依赖属性)，把属性添加到控件中不会真正影响这类属性(附加属性)。事件增强为具备冒泡和隧道功能。

所有这些特性都是不同 XAML 技术的基础，如 WPF、WF 和 Windows 8 应用程序。

本书的许多章节都涉及 XAML，尤其是介绍 WPF 的第 35 章和第 36 章、介绍 XPS 的第 37 章、介绍 Windows Store 应用程序的第 38、39 章和介绍 Windows Workflow Foundation 的第 45 章。

第 30 章将讨论 MEF，即 Managed Extensibility Framework。

第30章

Managed Extensibility Framework

本章要点

- Managed Extensibility Framework 的体系结构
- 使用特性的 MEF
- 基于约定的注册
- 协定
- 部件的导出和导入
- 宿主应用程序使用的容器
- 查找部件的分类

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 基于特性的示例
- 基于约定的示例
- WPF 计算器

30.1 概述

插件可以给现有的应用程序添加功能。我们可以创建一个宿主应用程序，它随着时间的推移会获得越来越多的功能——这些功能可能是开发人员团队编写的，不同的供应商也可以创建插件来扩展自己的应用程序。

目前，插件用于许多不同的应用程序，如 Internet Explorer 和 Visual Studio。Internet Explorer 是一个宿主应用程序，它提供了一个插件框架，供许多公司提供浏览网页时的扩展。Shockwave Flash Object 允许浏览包含 Flash 内容的网页。Google 工具栏提供了可以从 Internet Explorer 上快速访问的特定 Google 功能。Visual Studio 也有一个插件模型，它允许用不同级别的扩展程序来扩展 Visual

Studio。

对于自定义应用程序，总是可以创建一个插件模型，动态加载和使用程序集中的功能。但需要解决查找和使用插件的问题。这个任务可以使用 Managed Extensibility Framework 自动完成。MEF 也可以在较小的作用域上使用。为了创建边界，MEF 有助于删除部件和使用这些部件的客户端或调用者之间的依赖性。这种依赖性也可以使用接口或委托来删除。但 MEF 还有助于使用类别找到部件，并依次连接调用者和部件。

本章介绍的主要名称空间是 System.ComponentModel.Composition。

30.2 MEF 的体系结构

.NET 4.5 Framework 提供了两个技术，以编写动态加载插件的灵活的应用程序。一种技术是本章介绍的 Managed Extensibility Framework(MEF)；另一种技术自从.NET 3.5 以来就有，即 Managed Add-in Framework(MAF)。MAF 使用一个管道在插件和宿主应用程序之间通信，使开发过程比较复杂，但通过应用程序域甚或不同的进程使插件彼此分开。在这方面，MEF 是两种技术中比较简单的。MAF 和 MEF 可以合并起来，发挥各自的长处(而且完成了两倍的工作)。

MEF 通过部件和容器来构建，如图 30-1 所示。容器在类别中查找部件，类别在程序集或目录中查找部件。容器把入口连接到出口上，因此使部件可用于宿主应用程序。

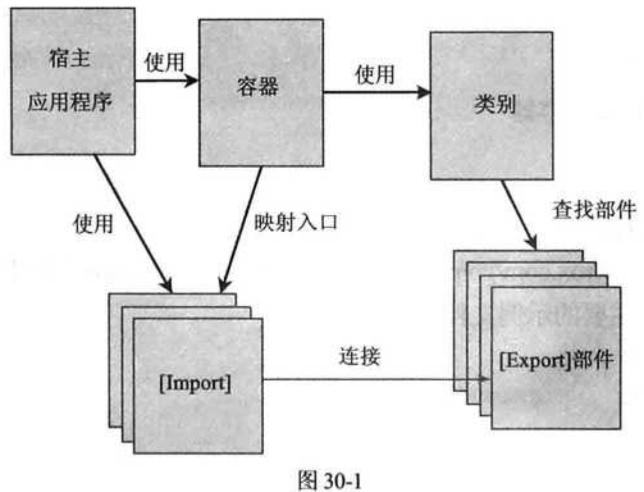


图 30-1

下面是部件加载的完整过程。前面提及，在类别中查找部件。类别使用出口来查找其部件。出口提供程序访问类别，提供类别中的出口。多个出口提供程序可以连接成链，以定制出口，例如使自定义出口提供程序只允许部件用于特定的用户或角色。容器使用出口提供程序把入口连接到出口上，该容器自己就是一个出口提供程序。

MEF 包含 3 大类别：用于宿主的类、基元类和基于特性机制的类。宿主类包含类别和容器。基元类可以用作基类，扩展 MEF 体系结构，以使用其他技术连接出口和入口。当然，通过反射构成基于特性机制的实现方式的类，如 Export 和 Import 特性，以及提供扩展方法、更便于使用基于特性的部件的类，这些也是 MEF 的一部分。

MEF 实现方式基于特性，这些特性指定哪些部件应导出，并把它们映射到入口上。但是，这个技术很灵活，允许使用抽象基类 ComposablePart、扩展方法以及 Reflection-ModelServices 类中基于反射的机制来实现其他技术。

30.2.1 使用属性的 MEF

下面用一个简单的例子来说明 MEF 体系结构。宿主应用程序可以动态地加载插件。通过 MEF，把插件表示为部件。部件定义为出口，并加载到一个导入部件的容器中。容器使用类别来查找部件；类别列出部件。

在这个例子中，创建一个简单的控制台应用程序作为宿主，以包含库中的计算器插件。为了使宿主和计算器插件彼此独立，需要 3 个程序集。其中一个程序集 CalculatorContract 包含了插件程序集和宿主可执行文件都使用的协定。插件程序集 SimpleCalculator 实现了协定程序集定义的协定。宿主使用协定程序集调用插件。

CalculatorContract 程序集中的协定通过两个接口 ICalculator 和 IOperation 来定义。ICalculator 接口定义了 GetOperations()和 Operate()方法。GetOperations()方法返回插件计算器支持的所有操作对应的列表，Operate()方法可调用操作。这个接口很灵活，因为计算器可以支持不同的操作。如果该接口定义了 Add()和 Subtract()方法，而不是灵活的 Operate()方法，就需要一个新版本的接口来支持 Divide()和 Multiply()方法。而使用本例定义的 ICalculator 接口，计算器就可以提供任意多个操作，且有任意多个操作数(代码文件 AttributeBasedSample/CalculatorContract/ICalculator.cs)。

```
using System.Collections.Generic;
namespace Wrox.ProCSharp.MEF
{
  public interface ICalculator
  {
    IList<IOperation>GetOperations();
    double Operate(IOperation operation, double[] operands);
  }
}
```

ICalculator 接口使用 IOperation 接口返回操作列表，并调用一个操作。IOperation 接口定义了只读属性 Name 和 NumberOperands(代码文件 AttributeBasedSample/CalculatorContract/IOperation.cs)。

```
namespace Wrox.ProCSharp.MEF
{
  public interface IOperation
  {
    string Name { get; }
    int NumberOperands { get; }
  }
}
```

CalculatorContract 程序集不需要引用 MEF 程序集，只要.NET 接口包含在其中即可。

插件程序集 SimpleCalculator 包含的类实现了协定定义的接口。Operation 类实现了 IOperation 接口。这个类仅包含接口定义的两个属性。该接口定义了属性的 get 访问器；内部的 set 访问器用于在程序集内部设置属性(代码文件 AttributeBasedSample/SimpleCalculator/Operation.cs)。

```
namespace Wrox.ProCSharp.MEF
{
  public class Operation : IOperation
  {
    public string Name { get; internal set; }
    public int NumberOperands { get; internal set; }
  }
}
```

Calculator 类实现了 ICalculator 接口，从而提供了这个插件的功能。按照 Export 特性的定义，Calculator 类导出为部件。这个特性在 System.ComponentModel.Composition 程序集的 System.ComponentModel.Composition 名称空间中定义(代码文件 AttributeBasedSample/SimpleCalculator/Calculator.cs)。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  [Export(typeof(ICalculator))]
  public class Calculator : ICalculator
  {
    public IList<IOperation> GetOperations()
    {
      return new List<IOperation>()
      {
        new Operation { Name="+", NumberOperands=2},
        new Operation { Name="-", NumberOperands=2},
        new Operation { Name="/", NumberOperands=2},
        new Operation { Name="*", NumberOperands=2}
      };
    }
    public double Operate(IOperation operation, double[] operands)
    {
      double result = 0;
      switch (operation.Name)
      {
        case "+":
          result = operands[0] + operands[1];
          break;
        case "-":
          result = operands[0] - operands[1];
          break;
        case "/":
          result = operands[0] / operands[1];
          break;
        case "*":
          result = operands[0] * operands[1];
          break;
        default:
```

```
            throw new InvalidOperationException(String.Format(
               "invalid operation {0}", operation.Name));
         }
         return result;
      }
   }
}
```

宿主应用程序是一个简单的控制台应用程序。插件使用 Export 特性定义导出的内容，对于宿主应用程序，Import 特性定义了所使用的信息。在本例中，Import 特性注解了 Calculator 属性，以设置和获取实现 ICalculator 接口的对象。因此实现了这个接口的任意计算器插件都可以在这里使用(代码文件 AttributeBasedSample/SimpleHost/Program.cs)。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using Wrox.ProCSharp.MEF.Properties;
namespace Wrox.ProCSharp.MEF
{
  class Program
  {
    [Import]
    public ICalculator Calculator { get; set; }
```

在控制台应用程序的入口方法 Main()中，创建了 Program 类的一个新实例，接着调用 Run()方法。在 Run()方法中，创建了一个 DirectoryCatalog，并用 AddInDirectory 进行初始化，AddInDirectory 在应用程序配置文件中配置。Settings.Default.AddInDirectory 通过项目属性 Settings，使用一个强类型化的类来访问自定义配置。

CompositionContainer 类是一个部件存储库。这个容器用 DirectoryCatalog 初始化，从这个类别指定的目录中获取部件。ComposeParts()是一个扩展了 CompositionContainer 类的扩展方法，它用 AttributeModelServices 类定义。这个方法需要把带 Import 特性的部件和参数一起传递。因为 Program 类有 Import 特性和 Calculator 属性，所以 Program 类的实例可以传递给这个方法。在用于入口的实现代码中，搜索并映射出口。成功调用这个方法后，就可以使用映射到入口的出口了。如果不是所有入口都可以映射到出口上，就抛出一个 ChangeRejectedException 类型的异常，捕获该异常以输出错误消息，并从 Run()方法退出。

```
static void Main()
{
  var p = new Program();
  p.Run();
}
public void Run()
{
  var catalog = new DirectoryCatalog(Settings.Default.AddInDirectory);
  var container = new CompositionContainer(catalog);
  try
```

```
{
  container.ComposeParts(this);
}
catch (ChangeRejectedException ex)
{
  Console.WriteLine(ex.Message);
  return;
}
```

通过 Calculator 属性，可以使用 ICalculator 接口中的方法。GetOperations()方法调用前面创建的插件的方法，它返回 4 个操作。要求用户指定应调用的操作并输入操作数的值后，就调用插件方法 Operate()。

```
      var operations = Calculator.GetOperations();
      var operationsDict = new SortedList<string, IOperation>();
      foreach (var item in operations)
      {
        Console.WriteLine("Name: {0}, number operands: {1}", item.Name,
            item.NumberOperands);
        operationsDict.Add(item.Name, item);
      }
      Console.WriteLine();
      string selectedOp = null;
      do
      {
        try
        {
          Console.Write("Operation? ");
          selectedOp = Console.ReadLine();
          if (selectedOp.ToLower() == "exit" || !operationsDict.ContainsKey(selectedOp))
            continue;
          var operation = operationsDict[selectedOp];
          double[] operands = new double[operation.NumberOperands];
          for (int i = 0; i < operation.NumberOperands; i++)
          {
            Console.Write("\t operand {0}? ", i + 1);
            string selectedOperand = Console.ReadLine();
            operands[i] = double.Parse(selectedOperand);
          }
          Console.WriteLine("calling calculator");
          double result = Calculator.Operate(operation, operands);
          Console.WriteLine("result: {0}", result);
        }
        catch (FormatException ex)
        {
          Console.WriteLine(ex.Message);
          Console.WriteLine();
          continue;
        }
      } while (selectedOp != "exit");
    }
  }
}
```

运行应用程序的一个示例输出如下：

```
Name: +, number operands: 2
Name: -, number operands: 2
Name: /, number operands: 2
Name: *, number operands: 2
Operation? +
        operand 1? 3
        operand 2? 5
calling calculator
result: 8
Operation? -
        operand 1? 7
        operand 2? 2
calling calculator
result: 5
Operation? exit
```

无须重编译宿主应用程序，就可以使用另一个完全不同的插件库。AdvCalculator 程序集定义了 Calculator 类的另一种实现方式，以提供更多的操作。通过把程序集复制到宿主应用程序中 DirectoryCatalog 指定的目录下，就可以用这个计算器替代另一个计算器(代码文件 AttributeBased Sample/SimpleCalculator/Calculator.cs)。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  [Export(typeof(ICalculator))]
  public class Calculator : ICalculator
  {
    public IList<IOperation> GetOperations()
    {
      return new List<IOperation>()
      {
        new Operation { Name="+", NumberOperands=2},
        new Operation { Name="-", NumberOperands=2},
        new Operation { Name="/", NumberOperands=2},
        new Operation { Name="*", NumberOperands=2},
        new Operation { Name="%", NumberOperands=2},
        new Operation { Name="++", NumberOperands=1},
        new Operation { Name="--", NumberOperands=1}
      };
    }
        public double Operate(IOperation operation, double[] operands)
        {
            double result = 0;
            switch (operation.Name)
            {
                case "+":
                    result = operands[0] + operands[1];
                    break;
                case "-":
                    result = operands[0] - operands[1];
```

```
                    break;
                case "/":
                    result = operands[0] / operands[1];
                    break;
                case "*":
                    result = operands[0] * operands[1];
                    break;
                case"%":
                    result = operands[0] % operands[1];
                    break;
                case"++":
                    result = ++operands[0];
                    break;
                case"--":
                    result = --operands[0];
                    break;
                default:
                    throw new InvalidOperationException(
                       String.Format("invalid operation {0}", operation.Name));
            }
            return result;
        }
    }
}
```

前面讨论了 MEF 体系结构中的入口、出口和类别。下一节介绍.NET 4.5 的一个新功能：基于约定的部件注册。

30.2.2 基于约定的部件注册

.NET 4.5 的一个 MEF 新功能是基于约定的部件注册，它不再需要对导出的部件使用特性。它的一个使用场合是不能访问类的源代码，而该类应用作部件来添加特性时，应使用这个功能。另一个使用场合是库的用户希望不再需要处理入口的特性时。ASP.NET MVC4 允许对 MEF 进行基于约定的注册。这个技术基于 Model View Controller(MVC)模式，使用带后缀 Controller 的控制器类，这是查找控制器的一个命名约定。

ASP.NET MVC 在第 41 章讨论。

基于约定的部件注册需要添加对程序集 System.ComponentModel.Composition.Registration 的引用。

引入基于约定的部件注册会构建与前面使用特性时相同的示例代码，但特性不再是必需的，因此这里不重复这些相同的代码。这里也使用类 Calculator 实现了相同的协定接口 ICalculator 和 IOperation，以及几乎相同的部件。与类 Calculator 的区别是它没有 Export 特性。

创建宿主应用程序，所有这些就会变得更有趣。与前面类似，创建一个 ICalculator 类型的属性，如下面的代码片段所示——它只是没有 Import 特性。在 Main 方法中，创建 Program 的一个新实例，这是因为 Calculator 属性是 Program 类的一个实例属性(代码文件 ConventionBasedSample/SimpleHost/Program.cs)：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Registration;

namespace Wrox.ProCSharp.MEF
{
  public class Program
  {
   public ICalculator Calculator { get; set; }
  static void Main()
  {
    var p = new Program();
    p.Run();
  }
```

在 Run 方法中，创建了一个新 RegistrationBuilder 实例。这个类型用于定义出口和入口的约定。在示例代码中，给所有派生于接口类型 ICalculator 的类型定义了一个出口，调用方法 conventions.ForTypesDerivedFrom<ICalculator>.Export<ICalculator>，该出口的类型是 ICalculator。这类似于给所有实现了接口 ICalculator 的类型应用特性 Export[typeof(ICalculator)]。为了把导出的类型映射到 Calculator 属性上，ForType<Program>指定了 Program 类型的约定，ImportProperty<ICalculator>方法把一个入口映射到指定的属性上。与前面的示例一样，创建一个 DirectoryCatalog，在指定的目录中查找部件。DirectoryCatalog 的构造函数允许传递 RegistrationBuilder，以提供约定的信息。调用 CompositionService 的 SatisfyImportOnce 方法，开始在目录中搜索。调用这个方法后，就指定属性 Calculator，现在可以调用 Calculator 的方法了：

```
public void Run()
{
  var conventions = new RegistrationBuilder();
  conventions.ForTypesDerivedFrom<ICalculator>().Export<ICalculator>();
  conventions.ForType<Program>().ImportProperty<ICalculator>(p => p.Calculator);

  var catalog = new DirectoryCatalog(
      Properties.Settings.Default.AddInDirectory, conventions);

  using (CompositionService service = catalog.CreateCompositionService())
  {
   service.SatisfyImportsOnce(this, conventions);
  }

  CalculatorLoop();
}
```

如上所示，RegistrationBuilder是基于约定的部件注册和MEF的核心，它使用一个流畅的API，还提供了特性所带来的所有灵活性。约定通过ForType可以应用于特定的类型，对于派生于基类或实现了接口的类型，ForTypesDerivedFrom.ForTypesMatching允许指定灵活的谓词。例如，ForTypesMatching(t => t.Name.EndsWidth("Controller"))把一个约定应用于名称以Controller结尾的所有类型。

选择类型的方法返回一个PartBuilder。有了PartBuilder，就可以定义出口和入口，并应用元数据。PartBuilder提供了几个方法来定义出口：Export会导出到特定的类型上，ExportInterfaces会导出一系列接口，ExportProperties会导出属性。使用导出方法导出多个接口或属性，就可以应用谓词，进一步定义选择。这也适用于导入属性或构造函数，其方法分别是ImportProperty、ImportProperties和SelectConstructors。

简要介绍了使用MEF、特性和约定的两种方式后，下面使用一个WPF应用程序来托管插件，详细论述MEF的不同选项。

30.3 定义协定

下面的示例应用程序扩展了第一个应用程序。宿主应用程序是一个WPF应用程序，它加载计算器插件以实现计算功能，还加载其他插件，把它们自己的用户界面引入宿主。

编写WPF应用程序的更多信息可参见第35章和第36章。

对于计算，使用前面定义的相同协定：ICalculator和IOperation接口。添加的另一个协定是ICalculatorExtension。这个接口定义了可由宿主应用程序使用的UI属性。该属性的get访问器返回一个FrameworkElement，它允许插件返回任意派生自FrameworkElement的WPF元素，并在宿主应用程序中显示为用户界面(代码文件WPFCalculator/CalculatorContract/ICalculator.cs)。

```
using System.Windows;
namespace Wrox.ProCSharp.MEF
{
  public interface ICalculatorExtension
  {
    FrameworkElement UI { get; }
  }
}
```

.NET接口用于去除在实现该接口的插件和使用该接口的插件之间的依赖性。这样，.NET接口也是用于MEF的一个优秀协定，去除了在宿主应用程序和插件之间的依赖性。如果接口在一个单独的程序集中定义，与CalculatorContract程序集一样，宿主应用程序和插件就没有直接的依赖关系。然而，宿主应用程序和插件仅引用协定程序集。

从MEF的角度来看，根本不需要接口协定。协定可以是一个简单的字符串。为了避免与其他协定冲突，字符串名应包含名称空间名，例如Wrox.ProCSharp.MEF.SampleContract，如下面的代码片段所示。这里的Foo类使用Export特性导出，并把一个字符串传递给该特性，而不是传递给接口。

```
[Export("Wrox.ProCSharp.MEF.SampleContract")]
public class Foo
{
  public string Bar()
  {
    return "Foo.Bar";
  }
}
```

把协定用作字符串的问题是，类型提供的方法、属性和事件都不是强定义的。调用者或者需要引用类型 Foo 才能使用它，或者使用.NET 反射来访问其成员。C# 4 的 dynamic 关键字使反射更便于使用，在这种情况下非常有帮助。

宿主应用程序可以使用 dynamic 类型导入 Wrox.ProCSharp.MEF.SampleContract 协定：

```
[Import("Wrox.ProCSharp.MEF.SampleContract")]
public dynamic Foo { get; set; }
```

有了 dynamic 关键字，Foo 属性就可以用于直接访问 Bar()方法。这个方法的调用会在运行期间解析：

```
string s = Foo.Bar();
```

协定名和接口也可以联合使用，定义只有接口和协定同名时才能使用的协定。这样，就可以对于不同的协定使用同一个接口。

dynamic 类型参见第 12 章。

30.4 导出部件

在前面的例子中，包含了 SimpleCalculator 部件，它导出了 Calculator 类型及其所有的方法和属性。下面的例子也包含 SimpleCalculator，其实现方式与前面相同，但还导出了另外两个部件 TemperatureConversion 和 FuelEconomy。这些部件为宿主应用程序提供了 UI。

30.4.1 创建部件

WPF 用户控件库 TemperatureConversion 定义了如图 30-2 所示的用户界面。这个控件提供了摄氏、华氏和绝对温度之间的转换。在第一和第二个组合框中，可以选择转换的源和目标。Calculate 按钮会启动执行转换的计算过程。

该用户控件为温度转换提供了一个简单的实现方式。TempConversionType 枚举定义了这个控件可能进行的不同转换。在构造函数中设置用户控件的 DataContext 属性，使枚举值显示在两个组合框中。ToCelsiusFrom()方法把参数 t 从其原始值转换为摄氏值。温度的源类型用第二个参数 TempConversionType 定义。FromCelsiusTo()方法把摄氏值转换为所选的温度值。OnCalculate()方法是 Button.Click 事件的处理程序，它调用 ToCelsiusFrom()和 FromCelsiusTo()方法，根据用户选择的转换类型执行转换(代码文件 WPFCalculator/TemperatureConversion/TemperatureConversion.xaml.cs)。

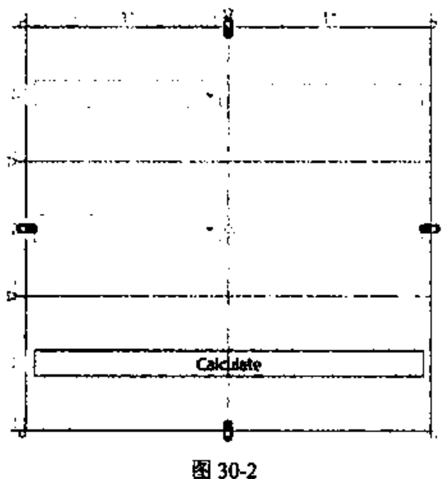


图 30-2

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.MEF
{
  public enum TempConversionType
  {
    Celsius,
    Fahrenheit,
    Kelvin
  }
  public partial class TemperatureConversion : UserControl
  {
    public TemperatureConversion()
    {
      InitializeComponent();
      this.DataContext = Enum.GetNames(typeof(TempConversionType));
    }
    private double ToCelsiusFrom(double t, TempConversionType conv)
    {
      switch (conv)
      {
        case TempConversionType.Celsius:
          return t;
        case TempConversionType.Fahrenheit:
          return (t - 32) / 1.8;
        case TempConversionType.Kelvin:
          return (t - 273.15);
        default:
          throw new ArgumentException("invalid enumeration value");
      }
    }
    private double FromCelsiusTo(double t, TempConversionType conv)
    {
      switch (conv)
```

```
        {
          case TempConversionType.Celsius:
            return t;
          case TempConversionType.Fahrenheit:
            return (t * 1.8) + 32;
          case TempConversionType.Kelvin:
            return t + 273.15;
          default:
            throw new ArgumentException("invalid enumeration value");
        }
      }
      private void OnCalculate(object sender, System.Windows.RoutedEventArgs e)
      {
        try
        {
          TempConversionType from;
          TempConversionType to;
          if (Enum.TryParse<TempConversionType>(
             (string)comboFrom.SelectedValue, out from) &&
                 Enum.TryParse<TempConversionType>(
                       (string)comboTo.SelectedValue, out to))
          {
            double result = FromCelsiusTo(
                ToCelsiusFrom(double.Parse(textInput.Text), from), to);
            textOutput.Text = result.ToString();
          }
        }
        catch (FormatException ex)
        {
          MessageBox.Show(ex.Message);
        }
      }
    }
  }
```

到目前为止，这个控件仅是一个简单的 WPF 用户控件。为了创建 MEF 部件，要使用 Export 特性导出 TemperatureCalculatorExtension 类。这个类实现 ICalculatorExtension 接口，从 UI 属性中返回用户控件 TemperatureConversion(代码文件 WPFCalculator/TemperatureConversion/TemperatureConversionExtension.cs)。

```
using System.ComponentModel.Composition;
using System.Windows;
namespace Wrox.ProCSharp.MEF
{
  [Export(typeof(ICalculatorExtension))]
  public class TemperatureCalculatorExtension : ICalculatorExtension
  {
    private TemperatureConversion control;
    public FrameworkElement UI
    {
      get
```

```
        {
          return control ?? (control = new TemperatureConversion());
        }
      }
    }
  }
```

另一个实现了 ICalculatorExtension 接口的用户控件是 FuelEconomy。通过这个控件，可以计算每加仑行驶的英里数或每行驶 100 千米的升数，该用户界面如图 30-3 所示。

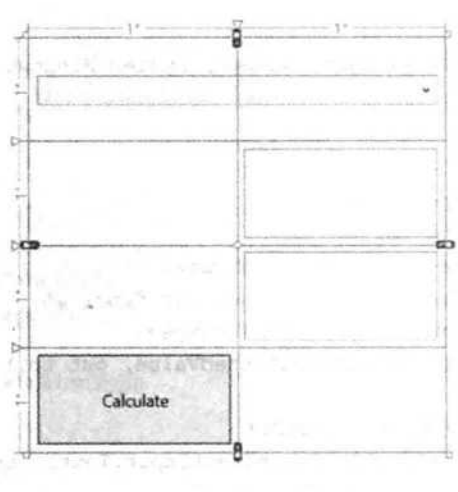


图 30-3

下面的代码片段显示了类 FuelEconomyViewModel，它定义了在用户界面上绑定的几个属性，例如 fuelEcoType 列表，它允许用户在英里数和千米数之间选择；此外还定义了 Fuel 和 Distance 属性，由用户填充这些属性：

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.MEF
{
  public class FuelEconomyViewModel : BindableBase
  {
    public FuelEconomyViewModel()
    {
      InitializeFuelEcoTypes();
    }

    private List<FuelEconomyType> fuelEcoTypes;
    public List<FuelEconomyType> FuelEcoTypes
    {
      get
      {
        return fuelEcoTypes;
      }
    }

    private void InitializeFuelEcoTypes()
    {
```

```
      var t1 = new FuelEconomyType
      {
        Id = "lpk",
        Text = "L/100 km",
        DistanceText = "Distance (kilometers)",
        FuelText = "Fuel used (liters)"
      };
      var t2 = new FuelEconomyType
      {
        Id = "mpg",
        Text = "Miles per gallon",
        DistanceText = "Distance (miles)",
        FuelText = "Fuel used (gallons)"
      };
      fuelEcoTypes = new List<FuelEconomyType>() { t1, t2 };
    }

    private FuelEconomyType selectedFuelEcoType;

    public FuelEconomyType SelectedFuelEcoType
    {
      get { return selectedFuelEcoType; }
      set { SetProperty(ref selectedFuelEcoType, value); }
    }

    private string fuel;
    public string Fuel
    {
      get { return fuel; }
      set { SetProperty(ref fuel, value); }
    }

    private string distance;
    public string Distance
    {
      get { return distance; }
      set { SetProperty(ref distance, value); }
    }

    private string result;
    public string Result
    {
      get { return result; }
      set { SetProperty(ref result, value); }
    }
  }
}
```

示例代码中使用的基类 BindableBase 仅提供了接口 INotifyPropertyChanged 的实现代码。这个类位于 CalculatorUtil 程序集中。

计算在OnCalculate()方法中完成。OnCalculate()是Calculate按钮的单击事件处理程序(代码文件WPFCalculator/FuelEconomy/FuelEconomyUC.xaml.cs)。

```
private void OnCalculate(object sender, RoutedEventArgs e)
{
  double fuel = double.Parse(viewModel.Fuel);
  double distance = double.Parse(viewModel.Distance);
  FuelEconomyType ecoType = viewModel.SelectedFuelEcoType;
  double result = 0;
  switch (ecoType.Id)
  {
    case "lpk":
      result = fuel / (distance / 100);
      break;
    case "mpg":
      result = distance / fuel;
      break;
    default:
     break;
  }
  viewModel.Result = result.ToString();
}
```

再次使用Export特性实现ICalculatorExtension接口并导出(代码文件WPFCalculator/FuelEconomy/FuelCalculatorExtension.cs)。

```
using System.ComponentModel.Composition;
using System.Windows;

namespace Wrox.ProCSharp.MEF
{
  [Export(typeof(ICalculatorExtension))]
  public class FuelCalculatorExtension : ICalculatorExtension
  {
    private FrameworkElement control;
    public FrameworkElement UI
    {
      get
      {
        return control ?? (control = new FuelEconomyUC());
      }
    }
  }
}
```

在继续WPF计算器例子以导入用户控件之前，先看看导出的其他选项。不仅可以导出整个类型，还可以导出属性和方法，也可以给出口添加元数据信息。

30.4.2 导出属性和方法

除了导出整个类，包括属性、方法和事件之外，还可以仅导出属性或方法。给属性添加Export特性就可以导出属性，以使用无法修改源代码的类(如.NET Framework或第三方库中的类)。为此，只需要定义这个特定类型的属性，并导出该属性。

导出方法允许进行比使用类型更精细的控制。调用者不需要了解类型。方法通过委托来导出。下面的代码片段用 Export 特性定义了 Add()和 Subtract()方法。导出的类型是委托 Func<double, double, double>，这个委托接受两个 double 参数，返回类型是 double。对于没有返回类型的方法，可以使用委托 Action<T>(代码文件 WPFCalculator/Operations/Operations.cs)。

```
using System;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  public class Operations
  {
    [Export("Add", typeof(Func<double, double, double>))]
    public double Add(double x, double y)
    {
      return x + y;
    }
    [Export("Subtract", typeof(Func<double, double, double>))]
    public double Subtract(double x, double y)
    {
      return x - y;
    }
  }
}
```

Func<T>和 Action<T>委托参见第 8 章。

导出的方法从 SimpleCalculator 插件中导入。部件本身可以使用其他部件。要使用导出的方法，需要用 Import 特性声明委托。这个特性与用 Export 特性声明的委托类型同名(代码文件 WPFCalculator/SimpleCalculator/Calculator.cs)。

SimpleCalculator 本身是一个导出了 ICalculator 接口的部件，它由导入的部件组成。

```
[Export(typeof(ICalculator))]
public class Calculator : ICalculator
{
  [Import("Add", typeof(Func<double, double, double>))]
  public Func<double, double, double> Add { get; set; }

  [Import("Subtract", typeof(Func<double, double, double>))]
  public Func<double, double, double> Subtract { get; set; }
```

导入的方法用 Add 和 Substract 委托表示，它们在 Operate()方法中通过这两个委托来调用。

```
public double Operate(IOperation operation, double[] operands)
{
  double result = 0;
  switch (operation.Name)
  {
```

```
        case "+":
          result = Add(operands[0], operands[1]);
          break;
        case "-":
          result = Subtract(operands[0], operands[1]);
          break;
        case "/":
          result = operands[0] / operands[1];
          break;
        case "*":
          result = operands[0] * operands[1];
          break;
        default:
          throw new InvalidOperationException(
             String.Format("invalid operation {0}", operation.Name));
      }
      return result;
    }
```

30.4.3 导出元数据

利用导出功能，还可以附加元数据信息。元数据允许添加除了名称和类型之外的其他信息。这些信息可用于添加功能信息，确定在入口端应使用哪些出口。

导出的 Add()方法现在改为用 ExportMetadata 特性添加速度功能(代码文件 WPFCalculator /Operations/Operation.cs)：

```
[Export("Add", typeof(Func<double, double, double>))]
[ExportMetadata("speed", "fast")]
public double Add(double x, double y)
{
  return x + y;
}
```

为了包含从 Add()方法的另一种实现方式中选择的选项，实现了另一个方法，它具有不同的速度性能，但委托的类型和名称相同(代码文件 WPFCalculator /Operations/Operation2.cs)：

```
public class Operations2
{
  [Export("Add", typeof(Func<double, double, double>))]
  [ExportMetadata("speed", "slow")]
  public double Add(double x, double y)
  {
    Thread.Sleep(3000);
    return x + y;
  }
}
```

因为有多个导出的 Add()方法，所以必须改变入口的定义。如果多个出口有相同的名称和类型，就可以使用 ImportMany 特性。这个特性应用于数组或 IEnumeration<T>接口。ImportMany 在 30.4 节中详细解释。为了访问元数据，可以使用一个 Lazy<T, TMetadata>数组。Lazy<T>类用于支持类型在第一次使用时的惰性初始化。Lazy<T, TMetadata>派生自 Lazy<T>，除了基类支持的成员之外，它还支持通过 Metadata 特性访问元数据信息。在例子中，方法通过 Func<double, double, double>委托

来引用，它是 Lazy<T, TMetadata>的第一个泛型参数。第二个泛型参数是用于收集元数据的 IDictionary<string, object>。ExportMetadata 特性可以多次使用，以添加多个功能，且总是包含字符串类型的键和 Object 类型的值(代码文件 WPFCalculator/SimpleCalculator/Calculator.cs)。

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, IDictionary<string, object>>[]
   AddMethods { get; set; }
//[Import("Add", typeof(Func<double, double, double>))]
//public Func<double, double, double> Add { get; set; }
```

对 Add()方法的调用现在改为遍历 Lazy<Func<double, double, double>, IDictionary<string, object>>元素集合。通过 Metadata 属性，检查功能的键；如果速度功能的值是 fast，就使用 Lazy<T>的 Value 属性调用操作，以获取委托。

```
case "+":
  // result = Add(operands[0], operands[1]);
  foreach (var addMethod in AddMethods)
{
  if (addMethod.Metadata.ContainsKey("speed") &&
       (string)addMethod.Metadata["speed"] == "fast")
    result = addMethod.Value(operands[0], operands[1]);
}
// result = operands[0] + operands[1];
break;
```

除了使用 ExportMetadata 特性之外，还可以创建一个派生自 ExportAttribute 的自定义导出特性类。SpeedExportAttribute 类定义了一个 Speed 类型的额外 Speed 属性(代码文件 WPFCalculator/CalculatorUtils/SpeedExportAttribute.cs)：

```
using System;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  public enum Speed
  {
    Fast,
    Slow
  }
  [MetadataAttribute]
  [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]
  public class SpeedExportAttribute : ExportAttribute
  {
    public SpeedExportAttribute(string contractName, Type contractType)
        : base(contractName, contractType) { }
    public Speed Speed { get; set; }
  }
}
```

如何创建自定义特性的更多内容可参见第 15 章。

有了导出的 Add()方法，现在可以使用 SpeedExport 特性替代 Export 和 ExportMetadata 特性(代码文件 WPFCalculator/Operations/Operations.cs)：

```
[SpeedExport("Add", typeof(Func<double, double, double>), Speed=Speed.Fast)]
public double Add(double x, double y)
{
  return x + y;
}
```

对于入口，需要一个包含所有元数据的接口，这样才能访问强类型化的功能。通过 SpeedExport 特性定义的功能仅是速度。ISpeedCapabilities 接口使用定义 SpeedExport 特性时使用的枚举类型 Speed 来定义 Speed 属性(代码文件 WPFCalculator/CalculatorUtils/ISpeedCapabilities.cs)：

```
namespace Wrox.ProCSharp.MEF
{
  public interface ISpeedCapabilities
  {
    Speed Speed { get; }
  }
}
```

现在可以使用 ISpeedCapabilities 接口替代前面定义的字典，修改入口的定义(代码文件 WPFCalculator/SimpleCalculator/Calculator.cs)：

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, ISpeedCapabilities>[]
   AddMethods { get; set; }
```

通过入口，现在可以直接使用 ISpeedCapabilities 接口的 Speed 属性：

```
foreach (var addMethod in AddMethods)
{
  if (addMethod.Metadata.Speed == Speed.Fast)
    result = addMethod.Value(operands[0], operands[1]);
}
```

30.4.4 使用元数据进行惰性加载

MEF 元数据不仅可用于根据元数据信息选择部件，还可以在实例化部件之前给宿主应用程序提供部件的信息。

下面的示例为计算器扩展 FuelEconomy 和 TemperatureConversion 提供图像的标题、描述和链接(代码文件 WPFCalculator/CalculatorContract/ICalculatorExtensionMetadata.cs)：

```
public interface ICalculatorExtensionMetadata
{
  string Title { get; }
  string Description { get; }

  string ImageUri { get; }
}
```

为了便于使用，创建了一个出口属性 CalculatorExtensionExport，如下面的代码片段所示。其实现代码非常类似于前面的 SpeedExport 特性(代码文件 WPFCalculator/CalculatorUtils/ICalculator-

ExtensionAttribute.cs)：

```
using System;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
  [MetadataAttribute]
  [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]
  public class CalculatorExtensionExportAttribute : ExportAttribute
  {
    public CalculatorExtensionExportAttribute(Type contractType)
      : base(contractType) { }

    public string Title { get; set; }
    public string Description { get; set; }

    public string ImageUri { get; set; }
  }
}
```

现在可以修改部件的导出。在下面的两个代码片段中，CalculatorExtensionExport 属性应用于部件 FuelEconomy 和 TemperatureConversion。这两个部件也使用两幅图像 Fuel.png 和 Temperature.png，这两幅图像会在生成过程中复制到插件目录中。这些图像也可以在宿主应用程序中使用，在实例化部件之前显示信息(代码文件 WPFCalculator/FuelEconomy/FuelCalculatorExtension.cs 和 TemperatureConversionExtension.cs)：

```
[CalculatorExtensionExport(typeof(ICalculatorExtension),
  Title = "Fuel Economy",
  Description = "Calculate fuel economy",
  ImageUri = "Fuel.png")]
public class FuelCalculatorExtension : ICalculatorExtension
[CalculatorExtensionExport(typeof(ICalculatorExtension),
  Title = "Temperature Conversion",
  Description="Convert Celsius to Fahrenheit and Fahrenheit to Celsius",
  ImageUri = "Temperature.png")]
public class TemperatureCalculatorExtension : ICalculatorExtension
```

30.5 导入部件

下面使用 WPF 用户控件和 WPF 宿主应用程序。宿主应用程序的设计视图如图 30-4 所示。Calculator 应用程序是一个 WPF 应用程序，它会加载功能完备的计算器插件(该插件实现 ICalculator 和 IOperation 接口)，以及带用户界面的插件(它实现 ICalculatorExtension 接口)。为了连接到部件的出口上，需要使用入口。

计算器宿主应用程序使用类 CalculatorViewModel 把输入和结果数据绑定到用户界面上。属性 CalcExtensions 包含所有可用的扩展插件，属性 ActiveatedExtensions 包含已加载的扩展列表(代码文件 WPFCalculator/Calculator/CalculatorViewModel.cs)：

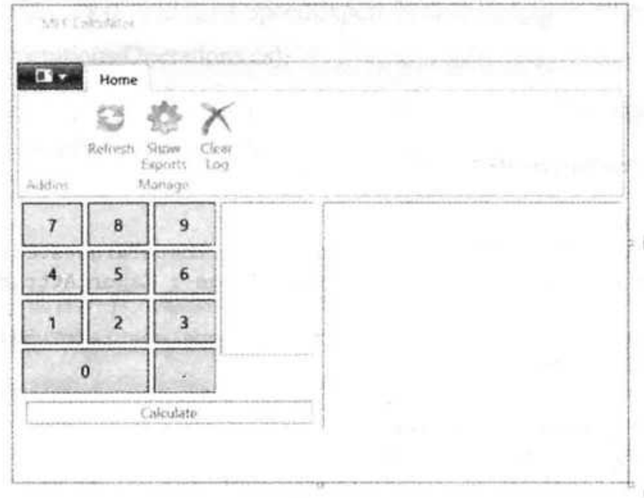


图 30-4

```
using System;
using System.Collections.ObjectModel;

namespace Wrox.ProCSharp.MEF
{
  public class CalculatorViewModel : BindableBase
  {
    private string status;

    public string Status
    {
      get { return status; }
      set { SetProperty(ref status, value); }
    }

    private string input;
    public string Input
    {
      get { return input; }
      set { SetProperty(ref input, value); }
    }

    private string result;
    public string Result
    {
      get { return result; }
      set { SetProperty(ref result, value); }
    }

    private string fullInputText;
    public string FullInputText
    {
      get { return fullInputText; }
      set { fullInputText = value; }
    }

    private readonly ObservableCollection<IOperation> calcAddInOperators =
```

```
        new ObservableCollection<IOperation>();
    public object syncCalcAddInOperators = new object();
    public ObservableCollection<IOperation> CalcAddInOperators
    {
      get
      {
        return calcAddInOperators;
      }
    }

    private readonly ObservableCollection<Lazy<ICalculatorExtension>>
        calcExtensions = new ObservableCollection<Lazy<ICalculatorExtension>>();
    public ObservableCollection<Lazy<ICalculatorExtension>> CalcExtensions
    {
      get
      {
        return calcExtensions;
      }
    }

    private readonly ObservableCollection<Lazy<ICalculatorExtension>>
      activatedExtensions = new ObservableCollection<Lazy<ICalculatorExtension>>();
    public object syncActivatedExtensions = new object();
    public ObservableCollection<Lazy<ICalculatorExtension>> ActivatedExtensions
    {
      get
      {
        return activatedExtensions;
      }
    }
  }
}
```

30.5.1 导入连接

入口连接到出口上。使用导出的部件时，需要一个入口来建立连接。通过 Import 特性，可以连接到一个出口上。如果应加载多个插件，就需要 ImportMany 特性，并需要把它定义为一个数组类型或 IEnumerable<T>。因为宿主计算器应用程序允许加载实现了 ICalculatorExtension 接口的多个计算器扩展，所以 CalculatorExtensionImport 类定义了 IEnumerable<ICalculatorExtension>类型的 CalculatorExtensions 属性，以访问所有的计算器扩展部件(代码文件 WPFCalculator/Calculator/CalculatorExtensionImport.cs)。

```
using System.Collections.Generic;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  public class CalculatorExtensionImport
  {
    [ImportMany(AllowRecomposition=true)]
    public IEnumerable<ICalculatorExtension> CalculatorExtensions { get; set; }
  }
}
```

Import 和 ImportMany 特性允许使用 ContractName 和 ContractType 把入口映射到出口上。可以用这两个特性设置的其他属性有 AllowRecomposition 和 RequiredCreationPolicy。AllowRecomposition 属性允许在应用程序运行期间动态映射到新出口上，还允许卸载出口。RequiredCreationPolicy 属性允许选择在请求器之间共享部件(CreationPolicy.Shared)、不共享部件(CreationPolicy.NonShared)，或者是否应由容器定义策略(CreationPolicy.Any)。

为了查看所有的导入是否成功，可以实现 IPartImportsSatisfiedNotification 接口。这个接口只定义了一个方法 OnImportsSatisfied()，在类的所有导入都成功时调用它。在 CalculatorImport 类中，该方法触发 ImportsSatisfied 事件(代码文件 WPFCalculator/Calculator/CalculatorImport.cs)：

```
using System;
using System.ComponentModel.Composition;
using System.Windows.Controls;
namespace Wrox.ProCSharp.MEF
{
  public class CalculatorImport : IPartImportsSatisfiedNotification
  {
    public event EventHandler<ImportEventArgs> ImportsSatisfied;
    [Import(typeof(ICalculator))]
    public ICalculator Calculator { get; set; }

    public void OnImportsSatisfied()
    {
      if (ImportsSatisfied != null)
        ImportsSatisfied(this, new ImportEventArgs {
            StatusMessage = "ICalculator import successful" });
    }
  }
}
```

在创建 CalculatorImport 类时，CalculatorImport 类的事件连接到一个事件处理程序上，把一条消息写入在 UI 上绑定的 Status 属性中，以显示状态信息(代码文件 WPFCalculator/CalculatorManager/MainWindow.xaml.cs)：

```
public sealed class CalculatorManager : IDisposable
{
  private DirectoryCatalog catalog;
  private CompositionContainer container;
  private CalculatorImport calcImport;
  private CalculatorExtensionImport calcExtensionImport;
  private CalculatorViewModel vm;

  public CalculatorManager(CalculatorViewModel vm)
  {
    this.vm = vm;
  }

  public async void InitializeContainer()
  {
    catalog = new DirectoryCatalog(Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
```

```
    calcImport = new CalculatorImport();

    calcImport.ImportsSatisfied += (sender, e) =>
    {
      vm.Status += string.Format("{0}\n", e.StatusMessage);
    };

    await Task.Run(() =>
      {
        container.ComposeParts(calcImport);
      });

    await InitializeOperationsAsync();
  }
```

30.5.2 部件的惰性加载

部件默认从容器中加载，例如调用 CompositionContainer 上的扩展方法 ComposeParts()来加载。利用 Lazy<T>类，部件可以在第一次访问时加载。Lazy<T>类型允许后期实例化任意类型 T，并定义 IsValueCreated 和 Value 属性。IsValueCreated 属性是一个布尔值，它返回包含的类型 T 是否已经实例化的信息。Value 属性在第一次访问包含的类型 T 时初始化它，并返回 T 的实例。

插件的入口可以声明为 Lazy<T>类型，如示例 Lazy<ICalculator>所示(代码文件 WPFCalculator/Calculator/CalculatorImport.cs)：

```
[Import(typeof(ICalculator))]
publicLazy<ICalculator>Calculator { get; set; }
```

调用导入的属性也需要对访问 Lazy<T>类型的 Value 属性进行一些修改。calcImport 是一个 CalculatorImport 类型的变量。Calculator 属性返回 Lazy<ICalculator>。Value 属性以惰性的方式实例化导入的类型，并返回 ICalculator 接口，在该接口中现在可以调用 GetOperations()方法，从计算器插件中获得所有支持的操作(代码文件 WPFCalculator/Calculator/CalculatorManager.cs)。

```
public Task InitializeOperationsAsync()
{
  Contract.Requires(calcImport != null);
  Contract.Requires(calcImport.Calculator != null);
  return Task.Run(() =>
    {
      var operators = calcImport.Calculator.Value.GetOperations();
      lock (vm.syncCalcAddInOperators)
      {
        vm.CalcAddInOperators.Clear();

        foreach (var op in operators)
        {
          vm.CalcAddInOperators.Add(op);
        }
      }
    });
}
```

30.5.3 用惰性实例化的部件读取元数据

部件 FuelEconomy 和 TemperatureConversion 是实现接口 ICalculatorExtension 的所有部件，也是惰性加载的。如前所述，集合可以用 IEnumerable<T>类型的属性来导入。部件采用惰性方式来实例化，属性就可以是 IEnumerable<Lazy<T>>类型。这些部件在实例化前需要提供它们的信息，才能给用户显示使用这些部件可以导出的内容信息。这些部件还为使用元数据提供了额外的信息，如前所述。元数据信息可以使用 Lazy 类型和两个泛型参数来访问。使用 Lazy<ICalculatorExtension, ICalculatorExtensionMetadata>，其中第一个泛型参数 ICalculatorExtension 用于访问实例化类型的成员；第二个泛型参数 ICalculatorExtensionMetadata 用于访问元数据信息(代码文件 WPFCalculator/Calculator/CalculatorExtensionImport.cs)：

```
public class CalculatorExtensionImport : IPartImportsSatisfiedNotification
{
  public event EventHandler<ImportEventArgs> ImportsSatisfied;

  [ImportMany(AllowRecomposition = true)]
  public IEnumerable<Lazy<ICalculatorExtension, ICalculatorExtensionMetadata>>
     CalculatorExtensions { get; set; }

  public void OnImportsSatisfied()
  {
    if (ImportsSatisfied != null)
      ImportsSatisfied(this, new ImportEventArgs
      { StatusMessage = "ICalculatorExtension imports successful" });
  }
}
```

方法 RefreshExtensions 根据 Lazy 类型导入计算器扩展部件，并将 Lazy 类型添加到前面的集合 CalcExtensions 中(代码文件 WPFCalculator/Calculator/CalculatorManager.cs)：

```
public void RefreshExensions()
{
  catalog.Refresh();
  calcExtensionImport = new CalculatorExtensionImport();
  calcExtensionImport.ImportsSatisfied += (sender, e) =>
  {
    vm.Status += String.Format("{0}\n", e.StatusMessage);
  };

  container.ComposeParts(calcExtensionImport);
  vm.CalcExtensions.Clear();
  foreach (var extension in calcExtensionImport.CalculatorExtensions)
  {
    vm.CalcExtensions.Add(extension);
  }
}
```

在 XAML 代码中，绑定了元数据信息。Lazy 类型的 Metadata 属性返回 ICalculatorExtensionMetadata。这样就可以访问 Description、Title 和 ImageUri，以进行数据绑定，而无须实例化插件(代码文件 WPFCalculator/Calculator/MainWindow.xaml)：

```
<RibbonGroup Header="Addins" ItemsSource="{Binding CalcExtensions}">
  <RibbonGroup.ItemTemplate>
    <DataTemplate>
      <RibbonButton ToolTip="{Binding Metadata.Description}"
        Label="{Binding Metadata.Title}" Tag="{Binding}"
        LargeImageSource="{Binding Metadata.ImageUri,
            Converter={StaticResource bitmapConverter}}"
        Command="local:CalculatorCommands.ActivateExtension" />
    </DataTemplate>
  </RibbonGroup.ItemTemplate>
</RibbonGroup>
```

为了从 ImageUri 属性返回的链接中获得图像，实现一个返回 BitmapImage 的值转换器(代码文件 WPFCalculator/Calculator/UriToBitmapConverter.cs)：

```
public class UriToBitmapConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
  {
    BitmapImage image = null;
    string uri = value.ToString();
    if (!string.IsNullOrEmpty(uri))
    {
      var stream = File.OpenRead(Path.Combine(
          Properties.Settings.Default.AddInDirectory, uri));
      image = new BitmapImage();
      image.BeginInit();
      image.StreamSource = stream;
      image.EndInit();
      return image;
    }
    else
    {
      return null;
    }
  }

  public object ConvertBack(object value, Type targetType, object parameter,
      CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

WPF 值转换器参见第 36 章。

图 30-5 显示了正在运行的应用程序，其中读取了计算器扩展中的元数据，它包含图像、标题和描述。

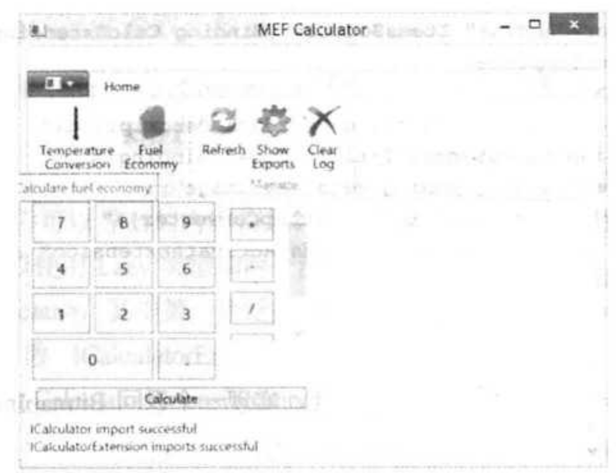


图 30-5

30.6 容器和出口提供程序

部件通过容器来导入。宿主部件的类型在 System.ComponentModel.Composition.Hosting 名称空间中定义。CompositionContainer 类是部件的容器。在这个类的构造函数中，可以指定多个 ExportProvider 对象，以及一个 ComposablePartCatalog。类别是部件的源，参见 30.6 节。出口提供程序允许用重载的 GetExport<T>()方法以编程方式访问所有出口。出口提供程序用于访问类别，CompositionContainer 类本身就是一个出口提供程序，因此可以在其他容器中嵌套容器。

调用 Compose()方法时，会加载部件(假定它们不是惰性加载)。前面一直在使用 ComposePart()方法，如下面的 InitializeContainer()方法所示(代码文件 WPFCalculator/Calculator/CalculatorManager.cs)：

```
public async void InitializeContainer()
{
    catalog = new DirectoryCatalog(
                    Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
    calcImport = new CalculatorImport();
    calcImport.ImportsSatisfied += (sender, e) =>
        {
            textStatus.Text += String.Format("{0}\n", e.StatusMessage);
        };
    await Task.Run(() =>
      {
        container.ComposeParts(calcImport);
      });
    await InitializeOperationsAsync();
}
```

ComposePart()是一个用 AttributeModelServices 类定义的扩展方法。这个类提供的方法使用特性和.NET 反射访问部件信息，并把部件添加到容器中。除了使用这个扩展方法之外，还可以使用 CompositionContainer 类的 Compose()方法。Compose()方法与 CompositionBatch 类一起工作。CompositionBatch 类可以用于定义应在容器中添加或删除哪些部件。AddPart()和 RemovePart()方法有重载版本，可以用于添加有 Import 属性的部件(calcImport 是 CalculatorImport 类的一个实例，包含

Import 特性)，或者派生自 ComposablePart 基类的部件。

```
var batch = new CompositionBatch();
batch.AddPart(calcImport);
container.Compose(batch);
```

宿主应用程序 Calculator 使用的两种部件用相同的方式搜索。实现了 ICalculator 接口的部件在启动应用程序时立即实例化。实现了 ICalculatorExtension 接口的部件仅在用户单击功能区控件中的部件信息时才实例化。单击功能区控件中的按钮会调用 OnActivateExtension 处理方法。在这个方法的实现代码中，使用 Lazy<T>类型的 Value 属性实例化选中的 ICalculatorExtension 部件，接着把一个引用添加到 ActivateExtension 集合中(代码文件 WPFCalculator/Calculator/ MainWindow.xaml.cs)：

```
private void OnActivateExtension(object sender, ExecutedRoutedEventArgs e)
{
  var button = e.OriginalSource as RibbonButton;
  if (button != null)
  {
    Lazy<ICalculatorExtension> control = button.Tag as
      Lazy<ICalculatorExtension>;
    FrameworkElement el = control.Value.UI;
    viewModel.ActivatedExtensions.Add(control);
  }
}
```

所有激活的 ICalculatorExtension 部件都在 TabControl 元素中显示为一个 TabItem，因为 TabControl 绑定到 ActivateExtensions 属性上。对于 TabItem 控件，定义 ItemTemplate 和 ContentTemplate。ItemTemplate 定义了一个标题，以显示标题和关闭部件的按钮；ContentTemplate 使用 UI 属性访问部件的用户界面(XAML 文件 WPFCalculator/Calculator/ MainWindow.xaml)：

```
<TabControl Grid.Row="1" Grid.Column="1" Margin="2"
    ItemsSource="{Binding ActivatedExtensions}">
  <TabControl.ContentTemplate>
    <DataTemplate>
      <ContentPresenter Content="{Binding Value.UI}" />
    </DataTemplate>
  </TabControl.ContentTemplate>
  <TabControl.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" Margin="0">
        <TextBlock Text="{Binding Metadata.Title}" Margin="0" />
        <Button Content="X" Margin="5,1"
          Command="local:CalculatorCommands.CloseExtension"
          Tag="{Binding}" />
      </StackPanel>
    </DataTemplate>
  </TabControl.ItemTemplate>
</TabControl>
```

ItemTemplate 中的关闭按钮会激活 CloseExtension 命令。这会调用 OnCloseExtension 处理方法，在该方法中，会从 ActivateExtension 集合中删除部件(代码文件 WPFCalculator/Calculator/ MainWindow.xaml.cs)：

```
private void OnCloseExtension(object sender, ExecutedRoutedEventArgs e)
{
  Button b = e.OriginalSource as Button;
  if (b != null)
  {
    Lazy<ICalculatorExtension> ext = b.Tag as Lazy<ICalculatorExtension>;
    if (ext != null)
    {
      viewModel.ActivatedExtensions.Remove(ext);
    }
  }
}
```

通过出口提供程序，可以实现 ExportsChanged 事件的处理程序，以获得所添加和删除的出口的相关信息。ExportsChangedEventArgs 类型的参数 e 包含所添加和删除的出口列表，这个列表会写入 Status 属性(代码文件 WPFCalculator/Calculator/ CalculatorManager.cs)：

```
container = new CompositionContainer(catalog);
container.ExportsChanged += (sender, e) =>
{
  var sb = new StringBuilder();

  foreach (var item in e.AddedExports)
  {
    sb.AppendFormat("added export {0}\n", item.ContractName);
  }
  foreach (var item in e.RemovedExports)
  {
    sb.AppendFormat("removed export {0}\n", item.ContractName);
  }
  vm.Status += sb.ToString();
};
```

30.7　类别

类别定义了 MEF 搜索所请求部件的位置。示例应用程序使用 DirectoryCatalog 从指定的目录中加载包含部件的程序集。利用 DirectoryCatalog，可以通过 Changed 事件获得变更信息，并遍历所有已添加和删除的定义。DirectoryCatalog 本身没有注册文件系统的变更。相反，必须调用 DirectoryCatalog 的 Refresh()方法，如果自从上一次读取目录以来发生了变化，就会触发 Changing 和 Changed 事件(代码文件 WPFCalculator/Calculator/ CalculatorManager.cs)。

```
public async void InitializeContainer()
{
  catalog = new DirectoryCatalog(Properties.Settings.Default.AddInDirectory);

  catalog.Changed += (sender, e) =>
  {
    var sb = new StringBuilder();

    foreach (var definition in e.AddedDefinitions)
    {
      foreach (var metadata in definition.Metadata)
```

```
            {
              sb.AppendFormat("added definition with metadata - key: {0}, " +
                 "value: {1}\n", metadata.Key, metadata.Value);
            }
          }

          foreach (var definition in e.RemovedDefinitions)
          {
            foreach (var metadata in definition.Metadata)
            {
              sb.AppendFormat("removed definition with metadata - key: {0}, " +
                 "value: {1}\n", metadata.Key, metadata.Value);
            }
          }

          vm.Status += sb.ToString();
        };

        container = new CompositionContainer(catalog);

        //...
      }
```

要获取加载到目录中的新插件的即刻通知，可以使用 System.IO.FileSystemWatcher，注册插件目录上的变化，再通过 FileSystemWatcher 的 Changed 事件调用 DirectoryCatalog 的 Refresh()方法。

CompositionContainer 类只需要一个 ComposablePartCatalog 就可以查找部件。DirectoryCatalog 派生自 ComposablePartCatalog。其他类别有 AssemblyCatalog、TypeCatalog 和 AggregateCatalog。下面比较这些类别：

- DirectoryCatalog 在目录中搜索部件。
- AssemblyCatalog 在引用的程序集中直接搜索部件。与 DirectoryCatalog 相反，目录中的程序集可能在运行期间发生变化，而 AssemblyCatalog 是不变的，部件不能改变。
- TypeCatalog 在类型列表中搜索入口。Enumerable<Type>可以传递给这个类别的构造函数。
- AggregateCatalog 是类别的类别。这个类别可以从多个 ComposablePartCatalog 对象中创建，并搜索上述所有类别。例如，可以创建一个 AssemblyCatalog，在程序集中搜索入口；再创建两个 DirectoryCatalog 对象，以搜索两个不同的目录；最后创建一个 AssemblyCatalog，合并入口搜索的 3 个类别。

运行示例应用程序时(如图 30-6 所示)，会加载 SimpleCalculator 插件，此时可以执行插件支持的一些计算操作。从 AddIns 菜单中，可以启动实现了 ICalculatorExtender 接口的插件，在选项卡控件中查看这些插件的用户界面。出口信息、目录类别中的变化以及状态信息显示在底部，还可以从插件目录中删除 ICalculatorExtender 插件(此时应用程序没有运行)，在应用程序运行期间把插件复制到目录中，在运行期间刷新插件，以查看新插件。

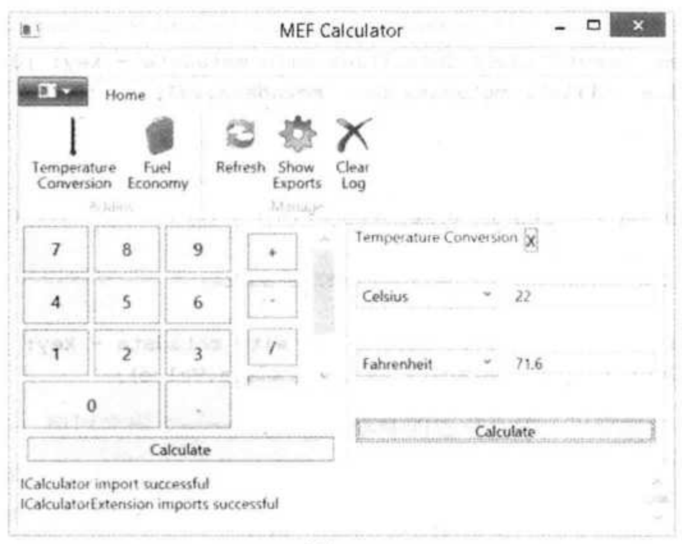


图 30-6

30.8　小结

本章介绍了 MEF 的部件、容器和类别，讨论了应用程序在构建时如何完全独立于其部件，如何动态加载来自不同类别的部件，例如程序集或目录列表。

MEF 实现方式使用特性或约定来查找和连接插件，我们介绍了基于约定的新部件注册技术，它允许在不使用特性的情况下导出部件，这样就可以在无法修改源代码的地方使用部件来添加特性，还可以创建基于 MEF 的框架，它不需要框架的用户添加特性就能导入部件。

我们还学习了部件如何通过惰性方式加载进来，仅在需要时实例化。部件可以提供元数据，为客户端提供足够的信息，以确定部件是否应实例化。

第 31 章介绍 Windows Runtime 的基础知识，以创建 Windows Store 应用程序。

第31章

Windows 运行库

本章要点

- Windows 运行库概述
- 理解语言投射(language projection)
- 理解 Windows 运行库组件
- 处理应用程序的生命周期
- 存储应用程序设置
- 定义和使用功能

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 语言投射
- 生命周期管理
- 应用程序设置

31.1 概述

从 Windows 8 开始，Microsoft 提供了一种新的运行库，为编写 Windows 应用程序带来了一种新的风格。这个运行库就是 Windows 运行库(Windows Runtime，WinRT)。该运行库中的类包含属性、方法和事件，并且使用委托，所以看上去它与.NET 很相似，但是实际上它是一个原生库。本章将介绍 Windows 运行库的基础知识，演示它与.NET 的区别，并讨论如何集成 Windows 运行库和.NET 来编写 Windows Store 应用程序。

Windows 8 包含这个运行库的 1.0 版本，Windows 8.1 包含这个运行库的 2.0 版本。

在 C#、C++和 JavaScript 中使用 Windows 运行库很简单。虽然.NET 允许其他语言使用这个框架，但是必须对这些语言进行修改。如果你熟悉 JScript.NET，就会知道这是用于.NET 编程的

JavaScript 语言。使用 JScript.NET 时，JavaScript 代码能够直接访问.NET 类的方法和属性。

与.NET 不同，Windows Runtime 则通过调整自身来适应不同的语言，使得开发人员能够在自己熟悉的环境中进行工作，无论这个环境是 C#、JavaScript 还是 C++。在.NET 语言(如 C#和 Visual Basic)中使用 Windows 运行库时，Windows 运行库看起来与.NET 一样。在 C++中使用时，Windows 运行库包含的方法与 C++标准库中的方法类似。在 JavaScript 中使用时，Windows 运行库的方法和大小写与 JavaScript 库相同。

Windows 运行库主要作为 Windows API 的替代。.NET Framework 在很大程度上是 Windows API 的包装，它提供的托管类可被 C#用来访问操作系统的功能。Windows 运行库是一个原生 API，使用起来与.NET Framework 类一样方便。.NET Framework 独立于操作系统，可以在 Windows Vista、Windows 7、Windows 8 和 Windows 8.1 上使用.NET 4.5。它会识别操作系统的具体版本。

Windows 运行库 2.0 版本只能在 Windows 8.1 中使用。这一点与 Windows API 很类似。Windows 运行库能够与未来的版本兼容。使用 Windows 8 的 Windows 运行库编写的应用程序将能够在 Windows 8.1 上运行，反之则不一定。这是因为，如果应用程序是使用 Windows 8.1 的 Windows 运行库编写的，就不能运行在 Windows 8 上。Windows 8 的一些 API 与 Windows 8.1 隔离，但仍可用。这些 API 可能不能用于 Windows 的未来版本。

31.1.1 .NET 与 Windows 运行库的比较

使用 C#编写 Windows Store 应用程序时，既可以使用 Windows 运行库，又可以使用.NET Framework。但是，Windows Store 应用程序并不能使用.NET Framework 的全部类和名称空间，有时只能使用这些类的部分方法。Windows 运行库是一个沙盒 API，包装了新的 UI 类和部分 Windows API。

对于 Windows Store 应用程序不能使用的.NET 名称空间，可以使用对应的 WinRT 名称空间，如表 31-1 所示：

表 31-1

.NET 名称空间	WinRT 名称空间
System.Net.Sockets	Windows.Networking.Sockets
System.Net.WebClient	Windows.Networking.BackgroundTransfer and System.Net.HttpClient
System.Resources	Windows.ApplicationModel.Resources
System.Security.IsolatedStorage	Windows.Storage
System.Windows	Windows.UI.Xaml

31.1.2 名称空间

要了解 Windows 运行库提供什么功能，最好的方法是查看其名称空间。Windows 运行库中的类包含在名称空间中，这一点与.NET Framework 类似。不同点在于，.NET Framework 以 System 名称空间开始，而 Windows 运行库以 Windows 名称空间开始。另外，.NET Framework 中有一部分是公共标准，可以针对其他平台实现(参见 Mono，地址为：http://www.go-mono.org)，但是 Windows 运行库则只能在 Windows 上使用。

图 31-1 显示了 Windows 运行库中的主要名称空间。表 31-2 对这些名称空间进行了解释：

表 31-2

Windows 运行库名称空间	描 述
Windows.System	Windows.System 名称空间及其子名称空间包含用于启动应用程序的 Launcher 类、带有远程桌面信息的类以及用于执行后台任务的 ThreadPool 和 ThreadPoolTimer 等
Windows.Foundation	Windows.Foundation 名称空间的子名称空间可用于集合、诊断和元数据
Windows.ApplicationModel	Windows.ApplicationModel 的子名称空间可用于管理 Windows Store 许可、在 Charms bar 中打开搜索、与其他应用程序共享数据、与联系人协作以及创建后台任务
Windows.Globalization	Windows.Globalization 名称空间定义了日历、区域、语言、日期以及数字格式
Windows.Graphics	Windows.Graphics 名称空间用于处理图像以及打印
Windows.Media	Windows.Media 名称空间可用于访问音频和视频、使用摄像头、以及使用 PlayTo 将标准视频传输到其他设备上播放
Windows.Data	Windows.Data 名称空间包含用于 XML、HTML 以及 JSON 的类
Windows.Devices	Windows.Devices 名称空间用于与设备和传感器交互，例如，使用 Accelerometer、Compass、Gyrometer、Inclinometer、LightSensor、OrientationSensor，发送 SMS，使用地理位置，访问 Point of Service(POS)设备、智能卡，连接到 Wi-Fi Direct 设备上等
Windows.Storage	Windows.Storage 名称空间包含用于读写文件的类，例如 StorageFile 和 StorageFolder，还包含流和文件选取器类，以及使用 Mszip、Lzms、Xpress 和 XpressHuff 算法进行压缩和解压缩的类
Windows.Security	Windows.Security 名称空间的子名称空间可用于通过身份验证、凭据和加密确保应用程序的安全性
Windows.Networking	Windows.Networking 名称空间用于客户端套接字编程。它还可以用于启动后台传输，当用户切换到其他任务后，该传输仍可继续进行

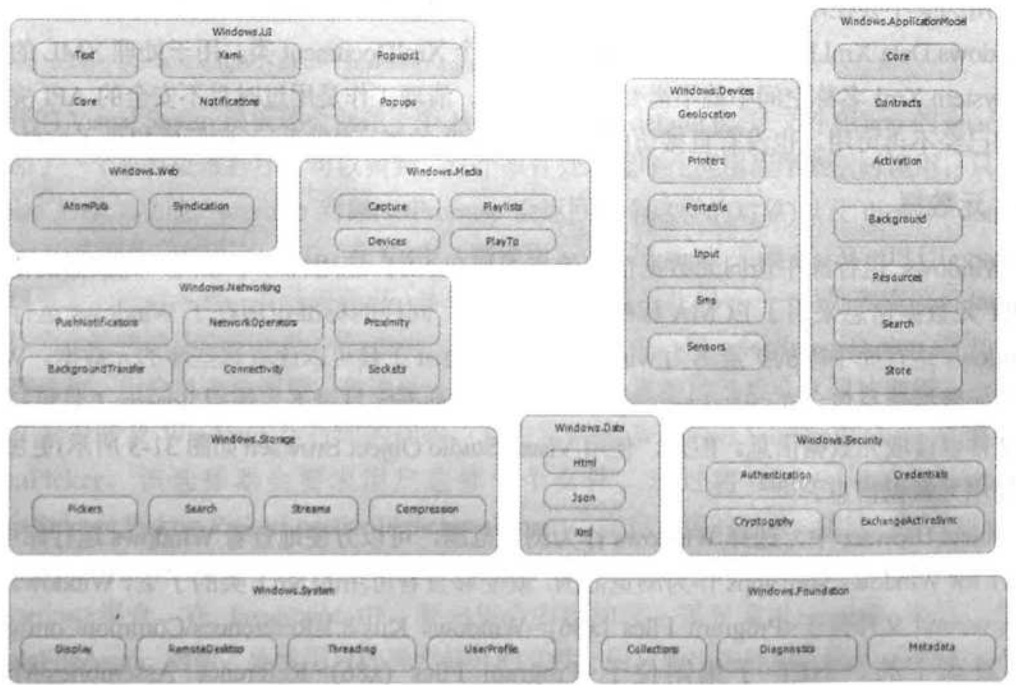


图 31-1

对于 XAML，有一个子名称空间 Windows.UI.Xaml，它又包含另外几个子名称空间，如图 31-2 所示。这些子名称空间分别用于形状(Shapes 名称空间)；XAML 文档(Documents 名称空间)；数据绑定(Data 名称空间)；位图、画刷、路径、变换(Media 名称空间)；触摸、鼠标和键盘输入(Input 名称空间)；XAML 读取器(Markup 名称空间)；XAML 元素的打印(Printing 名称空间)等。

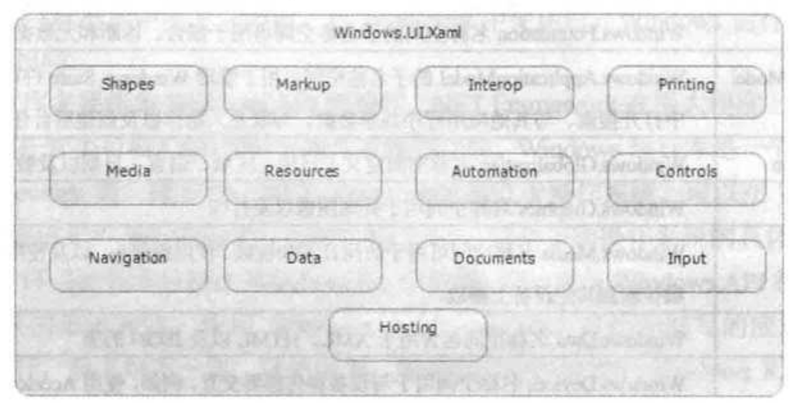


图 31-2

显然，Windows 运行库的名称空间提供了实现各种功能的可能性。当然，相比 Windows 运行库，.NET 运行库显得十分庞大，但是 Windows 运行库主要用于 Windows Store 应用程序，所以并不需要.NET Framework 中的某些部分。

可以把.NET Framework 的一个子集和 Windows 运行库结合起来使用。但是，没有给 Windows Store 应用程序提供某些类别的类。例如，Windows Store 应用程序不需要服务器端代码，没有 ASP.NET 以及 WCF 的服务端部分，但是可以使用 WCF 的客户端部分。使用 WCF 客户端类，可以与运行在服务器上的 WCF 服务端进行通信。

Windows Store 应用程序不能使用的.NET 类还包括用于删除重复和进行清理的类。Windows 运行库中可用的类不会在用于 Windows Store 应用程序的.NET Framework 中重复出现。Windows 运行库在 Windows.Data.Xml.Dom 名称空间中已经包含一个 XmlDocument 类，用于处理 XML 的 DOM，所以在 System.Xml 名称空间中就不需要有这个类了。清理工作是用过时且不安全的 API 完成的，现在它们已经不再可用。也没有直接访问 Windows API 的类，Windows 运行库提供了这些功能。

31.1.3 元数据

在 Windows 运行库中访问元数据信息的方式与在.NET 应用程序中相同，元数据的格式也一样。.NET 元数据信息采用了 ECMA 标准(ECMA 335)，同样的标准也用在了 Windows 运行库中。

Windows 运行库中库的扩展名为.winmd，使用 ildasm 工具可以读取这些库的元数据。Windows 运行库的元数据信息包含在<windows>\system32\WinMetadata 目录下。使用 ildasm 工具可打开该目录中的文件以读取元数据信息。但是，使用 Visual Studio Object Browser(如图 31-3 所示)更加方便，它也会使用元数据信息。

在 Object Browser 中，选择 Windows 作为浏览范围，可以方便地查看 Windows 运行库中的类；选择.NET for Windows Sore apps 作为浏览范围，就能够查看可用的.NET 类的子集。Windows 使用的 Windows.winmd 文件位于<Program Files (x86)>\Windows Kits\8.1\References\CommonConfiguration\Neutral 目录下的，.NET 子集则位于<Program Files (x86)>\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5.1 目录下。这些文件只包含定义了什么功能可用的元数据信息。

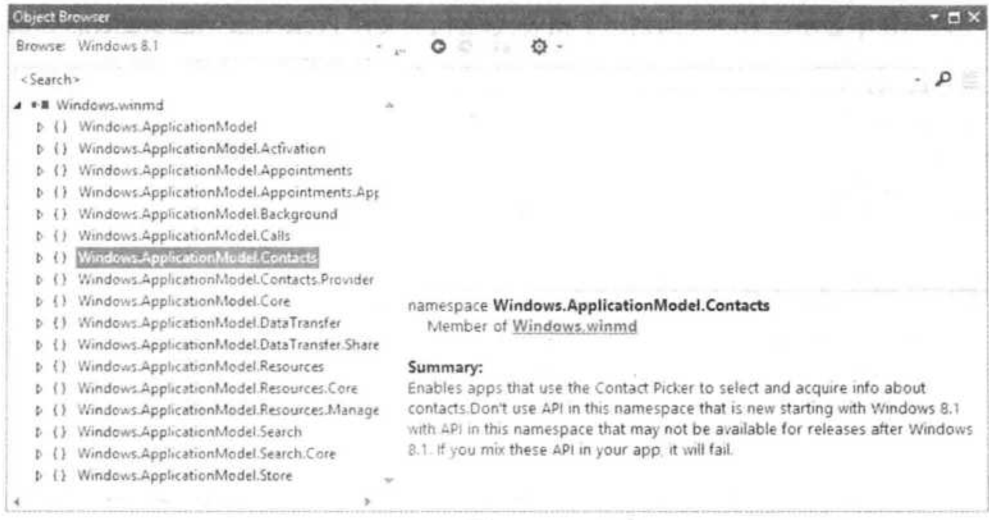


图 31-3

31.1.4 语言投射

元数据用于不同的语言创建不同的投射。在 C++、C#和 JavaScript 中使用时，Windows 运行库看上去是不同的。本节将创建两个 Windows Store 示例应用程序，以演示 JavaScript 和 C#在使用 Windows 运行库时的区别。示例应用程序包含一个按钮和一张图片。单击按钮将启动一个文件选择器。它允许用户选择一张图片，之后显示该图片。

第一个应用程序使用 JavaScript 构建。首先，选择 JavaScript Windows Store Blank App 模板。然后修改生成的 HTML 文件 default.html，使其包含 button 和 img 元素，如下面的代码片段所示(代码文件 ProjectionWithJavaScript/default.html)：

```
<button id="selectImageButton">Select an Image</button>
<p />
<img alt="Some Image" src="Images/logo.png" id="image1" />
```

选择的 Visual Studio 模板还会创建一个 JavaScript 文件 default.js。该文件在 app 的 onactivated 事件中添加了一个事件处理程序。可以猜到，这个事件处理程序在应用程序激活时调用。只需在调用 setPromise 后添加功能。promise 对象(setPromise 会返回一个这样的对象)以及 then 和 done 函数是 JavaScript 中实现异步编程的方式。在 C#中，则使用 async 和 await 关键字。WinJS.UI.processAll 函数会处理页面的所有元素，并管理绑定。本例关注在此之后的代码——一旦处理完成，done 函数就会调用作为参数传递给它的函数。在 done 函数的实现代码中，向 selectImageButton 添加一个 click 事件处理函数。用户单击按钮后，传递给 addEventListener 函数的函数会立即被调用。

现在看看涉及 Windows 运行库的代码。首先实例化 Windows.Storage.Pickers 名称空间中的 FileOpenPicker。该选择器会要求用户选择一个文件。通过将 suggestedStartLocation 设为 Windows.Storage.Pickers.PickerLocationId.picturesLibrary，指定首选的目录。然后，使用 fileTypeFilter 向文件类型筛选器列表中添加文件扩展名。fileTypeFilter 是 FileOpenPicker 的一个属性，它返回一个 IVector<string>集合。在 JavaScript 中，要向集合中添加项，需要调用 append 函数。通过调用 pickSingleFileAsync 函数，要求用户选择要打开的文件。用户做出选择后，then 函数定义了接下来要执行的操作。file 参数用于创建一个 BLOB 对象，该对象赋给 image 元素的 src 特性，使选中的图

片文件能够在 UI 中显示出来。下面列出了相关代码(代码文件 ProjectionWithJavaScript/js/default.js):

```
(function () {
  "use strict";

  var app = WinJS.Application;
  var activation = Windows.ApplicationModel.Activation;

  app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
      if (args.detail.previousExecutionState !==
        activation.ApplicationExecutionState.terminated) {
        // TODO: This application has been newly launched. Initialize
        // your application here.
      } else {
        // TODO: This application has been reactivated from suspension.
        // Restore application state here.
      }
      args.setPromise(WinJS.UI.processAll().done(function () {
        document.getElementById("selectImageButton").addEventListener(
          "click", function () {
          var picker = new Windows.Storage.Pickers.FileOpenPicker();
          picker.suggestedStartLocation =
            Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
          picker.fil eTypeFilter.append(".jpg");
          picker.fileTypeFilter.append(".png");
          picker.pickSingleFileAsync().then(function (file) {
            if (file) {
              var imageBlob = URL.createObjectURL(file);
              document.getElementById("image1").src = imageBlob;
            }
          });
        });
      }));
    }
  };
```

现在使用 XAML 和 C#完成相同的工作。这个项目将使用 Windows Store Blank App(XAML)模板。与上一个应用程序一样，这里也添加一个 Button 元素和一个 Image 元素，不过这次使用 XAML 代码而不是 HTML(代码文件 ProjectionWithCSharp/MainPage.xaml):

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Button Grid.Row="0" Click="OnOpenImage">Select an Image</Button>
  <Image Grid.Row="1" x:Name="image1" />
</Grid>
```

在把 OnOpenImage 处理程序赋值给按钮的 Click 事件后，代码变得有趣起来。OnOpenImage 中也创建了一个 FileOpenPicker 对象。但是，在之后的一行可以注意到一点区别：这里，Suggested-StartLocation 是 PickerLocationId 枚举类型。枚举值是全部大写形式，这与 C#相同。FileTypeFilter

没有返回 IVector<string>，而是返回 IList<string>。调用 FileTypeFilter 的 Add 方法可以添加文件扩展名。在 JavaScript 中，更常见的做法是使用 append 函数而不是 Add 方法。使用 await 关键字，可以等待返回一个 StorageFile 后再执行其他操作。最后，将一个 BitmapImage 对象赋给 XAML Image 元素。BitmapImage 本身会收到一个随机访问流来加载图片(代码文件 ProjectionWithCSharp/MainPage.xaml.cs)：

```
private async void OnOpenImage(object sender, RoutedEventArgs e)
{
  var picker = new FileOpenPicker();
  picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
  picker.FileTypeFilter.Add(".jpg");
  picker.FileTypeFilter.Add(".png");
  StorageFile file = await picker.PickSingleFileAsync();
  var image = new BitmapImage();
  image.SetSource(await file.OpenReadAsync());
  image1.Source = image;
}
```

这两个示例使用了相同的类，但是由于使用的编程语言不同，方法看上去也不同。在.NET 中，对类型、方法、属性和事件采用 PascalCase 约定。在 JavaScript 中，为类型使用 PascalCase 约定，为方法和属性使用 camelCase，为事件使用小写字母。语言投射会处理语言的这种转换，它不仅会更改大小写，还会更改数据类型和方法名。后面的小节将展开介绍这方面的内容。

31.1.5 Windows 运行库中的类型

本节将讨论 Windows 运行库定义的类型，说明它们是如何划分的以及它们与.NET 类型的区别。与各个语言的交互是 Windows 运行库定义的类型的一个重要方面。例如，向一个 Windows 运行库方法传递.NET 字符串时，不应该仅仅因为存在两种不同的技术，就必须在每个方法调用中创建一个新字符串。在这里，传递数据的操作称为封送(marshalling)，第 23 章曾详细讨论过这个主题。无论是使用运行库中的类和方法，还是使用 C#创建在其他语言(如 C++和 JavaScript)中使用的类型，理解 Windows 运行库中的类型都十分重要。

下面看看 Windows 运行库中的类型：

- 字符串——在 Windows 运行库中，字符串定义为句柄：HSTRING。句柄是一个字符串引用，它引用该字符串的第一个字符。HSTRING 添加到.NET 的 String 类型的缓冲区中，以便访问该字符串的其余部分。在.NET 中，字符串是不可变的，所以不能修改。WinRT 总是假定字符串缓冲区是不可变的，且以 NULL 值结尾，这与.NET 字符串相同。.NET 中的 String 类型会映射到 HSTRING。
- 基本数据类型——这些类型包括 Int16、Int32、Int64、Single、Double 等，它们直接映射到对应的.NET 类型。WinRT 中的 DataTime 映射到.NET 中的 DataTimeOffset。
- 数组——简单的数组(如 Int32[])可用于基本集合。
- 枚举——它们会映射到 C# enum。也可以使用标志和非标志枚举。
- 结构——WinRT 中也提供了结构，但是它们与.NET 结构类型不同。WinRT 中的结构只能包含基本数据类型和字符串。.NET 结构可以实现接口，但是 WinRT 结构不可以。WinRT 中的结构就是一个简单的数据容器。

- 接口——接口是 Windows 运行库的核心。第 23 章讨论了 COM 接口，如 IUnknown。Windows 运行库提供的类是新一代的 COM 对象。除了 IUnknown，这些类还实现了 IInspectible 接口。与 IUnknown 接口不同的是，IInspectible 接口提供了关于被实现的接口以及类名的信息。
- 泛型接口——Windows 运行库也支持泛型接口。IIterable<T>可用于枚举集合，IVector<T>表示可随机访问的元素集合。这两个接口分别映射到 IEnumerable<T>和 IList<T>。
- 运行库类——运行库类包括 Windows.Storage.StorageFile 等的类。运行库类实现了接口，例如，StorageFile 实现了 IStorageFile、IStorageFileItem、IRandomAccessStreamReference、IInputStreamReference 和 IStorageItemProperties。当然，它还实现了 IUnknown 和 IInspectible 接口。在.NET 应用程序中，通常只需要使用运行库类，而不必直接使用接口。

31.2 Windows 运行库组件

虽然 Windows 运行库和.NET 在数据类型上存在一些区别，但是很多数据类型很容易映射，例如，HSTRING 直接映射到 String。如果要用.NET 创建 Windows 运行库组件，这些区别就十分重要。Visual Studio 提供了一个 Windows Runtime Component 模板，使用该模板创建的组件可用于其他使用 Windows 运行库的语言，如 C++和 JavaScript。这与 Portable 类库不同。使用 Portable 类库创建的类可用于其他.NET 变体，例如使用完整的.NET Framework(如 WPF)、Silverlight 或 Windows Phone 的应用程序。一个第三方库是 Class Library(Windows Store 应用程序)。这个库只能用于 Windows Store 应用程序，但允许使用 Windows 运行库中的类。

创建 Windows 运行库组件时，其公有类必须被密封(sealed)。除了对 UI 组件的密封没有太严格的要求外，Windows 运行库组件只支持密封类型。下面的小节将仔细讨论自动映射和有时需要自己处理的映射。

31.2.1 集合

Windows 运行库定义的集合接口会自动映射到.NET 集合接口，如表 31-3 所示：

表 31-3

Windows 运行库	.NET
IIterable<T>	IEnumerable<T>
IIterator<T>	IEnumerator<T>
IVector<T>	IList<T>
IVectorView<T>	IReadOnlyList<T>
IMap<K, V>	IDictionary<K, V>
IMapView<K, V>	IReadOnlyDictionary<K, V>

创建 Windows 运行库组件时，需要返回一个接口，如下面代码段中的 IList<string>。不能直接返回 List<string>。IList<string>会自动映射到 IVector<T>。同样，可以使用 IList<T>作为方法的参数。

```
public IList<string> GetStringList()
{
  return new List<string> { "one", "two" };
}
```

31.2.2 流

流与集合不同。使用 Windows 运行库流的基础当然是接口。Windows.Storage.Streams 名称空间提供了一些具体的流类，如 FileInputStream、FileOutputStream 和 RandomAccessStream，以及使用流的读写器类，如 DataReader 和 DataWriter。所有这些类都基于接口。.NET 则没有为流使用接口。Windows 运行库组件的公有签名总是要求使用流接口，如 IIputStream、IOutputStream 和 IRandomAccessStream。这些接口与.NET 类型之间的相互映射通过 WindowsRuntimeStreamExtensions 中定义的扩展方法完成。如下面的代码片段所示，AsStreamForRead 从 IInputStream 创建了一个 Stream 对象：

```
public void StreamSample(IInputStream inputStream)
{
  var reader = new StreamReader(inputStream.AsStreamForRead());
  //...
}
```

其他扩展方法包括从 IOutputStream 创建 Stream 对象的 AsStreamForWrite 方法，以及从 IRandomAccessStream 创建 Stream 对象的 AsStream 方法。从.NET 流创建 Windows 运行库流时，可以使用 AsInputStream 方法从 Stream 创建 IInputStream，使用 AsOutputStream 方法从 Stream 创建 IOutputStream，使用 AsRandomAccessStream 方法从 Stream 创建 IRandomAccessStream。

下面的代码示范了一个仅使用 Windows 运行库类的例子。首先创建一个 StorageFolder 对象，使其引用允许应用程序写入的本地文件夹。Application.Current 返回了 ApplicationData 类的当前实例，该类定义了读写数据的本地、roaming 和临时文件夹的属性。roaming 文件夹的内容会复制到用户登录的不同系统中。该文件夹的大小有限制(配额)，在写作本书时，这个大小是每个应用程序 100kB。示例代码在本地文件夹中创建了一个新文件。方法 CreateFileAsync 返回了一个 StorageFile 对象。StorageFile 是一个 Windows 运行库对象，所以会在调用 OpenAsync 时返回一个实现了 IRandomAccessStream 接口的对象。可以把 IRandomAccessStream 传递给 Windows 运行库类 DataWriter 的构造函数，将一个字符串写入文件中：

```
StorageFolder folder = ApplicationData.Current.LocalFolder;
StorageFile file = await folder.CreateFileAsync("demo1.txt");
IRandomAccessStream stream = await file.OpenAsync(FileAccessMode.ReadWrite);
using (var writer = new DataWriter(stream))
{
  writer.WriteString("Hello, WinRT");
  await writer.FlushAsync();
}
```

这个示例完全是 Windows 运行库代码。接下来的代码则混合了 Windows 运行库和.NET。与前面一样，创建一个 StorageFolder，对其使用扩展方法 OpenStreamForWriteAsync。这是在 WindowsRuntimeStorageExtensions 类中定义的一个扩展方法，它返回一个.NET Stream 对象。使用该

方法后，所有使用流的.NET类(如StreamWriter类)就可以写入文件。WindowsRuntimeStorageExtensions类定义在System.IO名称空间下的.NET程序集System.Runtime.WindowsRuntime中：

```
StorageFolder folder = ApplicationData.Current.LocalFolder;
Stream stream = await folder.OpenStreamForWriteAsync("demo2.text",
    CreationCollisionOption.ReplaceExisting);
using (var writer = new StreamWriter(stream))
{
  await writer.WriteLineAsync("Hello, .NET");
  await writer.FlushAsync();
}
```

使用流时，还经常需要用到byte[]。Windows运行库为byte[]定义了接口IBuffer。字节数组有类似于流的混合匹配操作。WindowsRuntimeBuffer类(包含在System.Runtime.InteropServices.WindowsRuntime名称空间中)的Create方法会从byte[]返回一个实现了IBuffer的对象。WindowsRuntimeBufferExten- sions类提供了从byte[]获得IBuffer的方法(AsBuffer)，将byte[]复制到IBuffer或者将IBuffer复制到byte[]的方法(CopyTo)，以及从IBuffer创建byte[]的方法(ToArray)。

31.2.3 委托与事件

委托与事件编程在Windows运行库中与在.NET中看上去十分类似，但是底层的实现差别很大。Windows运行库事件必须是Windows运行库委托类型(或者与Windows运行库委托类型匹配的类型)。事件的类型不能是EventHandler，但是使用EventHandler<object>没有问题。

Windows运行库在实现事件时，使用了WindowsRuntimeMarshal类。该类的AddEventHandler和RemoveEventHandler分别用于为事件添加和删除处理程序。

31.2.4 异步操作

在.NET中进行async操作时，需要使用Task对象。没有返回值的async方法会返回一个Task，有返回值的async方法会返回一个Task<T>。Windows运行库中的async方法则基于IAsyncAction和IAsyncOperation<T>接口。

Windows运行库中的async方法的用法与.NET中的async方法相同，也要用到关键字async和await。下面的代码片段使用了Windows运行库的Windows.Data.Xml.Dom名称空间中的XmlDocument类。LoadFromUriAsync方法会返回IAsyncOperation<XmlDocument>。它的用法与返回Task<XmlDocument>的方法相同：

```
private async Task<string> XmlDemo()
{
  Uri uri = new Uri("http://www.cninnovation.com/downloads/Racers.xml");
  XmlDocument doc = await XmlDocument.LoadFromUriAsync(uri);
  return doc.GetXml();
}
private async void OnXml(object sender, RoutedEventArgs e)
{
  text1.Text = await XmlDemo();
}
```

如果创建的是 Windows 运行库组件，那么公有的 async 方法不能返回 Task。下面的两个代码片段展示了该方法的编写方法。如果方法不返回值，就需要返回 IAsyncAction。使用扩展方法 AsAsyncAction 可将 Task 转换为 IAsyncAction。用于任务的扩展方法定义在 WindowsRuntimeSystem-Extensions 类中：

```
public IAsyncAction TaskSample()
{
  return Task.Run(async () =>
  {
    await Task.Delay(3000);
  }).AsAsyncAction();
}
```

返回值的 async 方法需要返回 IAsyncOperation<T>。扩展方法 AsAsyncOperation 可将 Task<T> 转换为 IAsyncOperation<T>：

```
public IAsyncOperation<int> TaskWithReturn(int x, int y)
{
  return Task<int>.Run<int>(async () =>
  {
    await Task.Delay(3000);
    return x + y;
  }).AsAsyncOperation();
}
```

31.3　Windows Store 应用程序

Visual Studio 为创建 Windows Store 应用程序提供了几个模板，如图 31-4 所示。Blank App 模板是最基本的模板，只包含一个空页面。Grid App 模板包含 3 个显示分组网格信息的页面。Hub App 包含三个类似于 Grid App 模板的页面，但 Hub App 使用新的 Hub 控件。Split App 模板包含两个页面，允许使用一个项列表进行导航。Class Library 模板是一个.NET 库，可以在使用 C#或 Visual Basic 创建的不同 Windows Store 应用程序中重用。Windows Runtime Component Library 模板创建的库可在使用其他语言(如 C++和 JavaScript)创建的不同 Windows 8 项目中重用。使用 Windows Runtime Component Library 模板时，方法的公有签名必须匹配前一节介绍的 Windows 运行库类型。

本例将选择 Blank App 模板。在 Windows Store 项目中，除了通常的项目设置，还有一个针对部署包的配置。为了配置部署包，在 Solution Explorer 中单击 Package.appxmanifest 文件，打开如图 31-5 所示的 Manifest Designer。使用此设计器可配置应用程序设置，如徽标图片、应用程序名称、旋转信息以及颜色。设计器的第二个选项卡是 Capabilities，它用于定义应用程序包含的功能，以及哪些功能需要有权限才能访问(如文档库或音乐库等目录，话筒和摄像头等设备)。Declarations 选项卡则定义了应用程序需要访问的内容，例如摄像头设置、注册文件类型关联、可否作为共享目标)。Packaging 选项卡用于定义应用程序徽标、证书、版本号和其他一些描述所部署应用程序的选项。

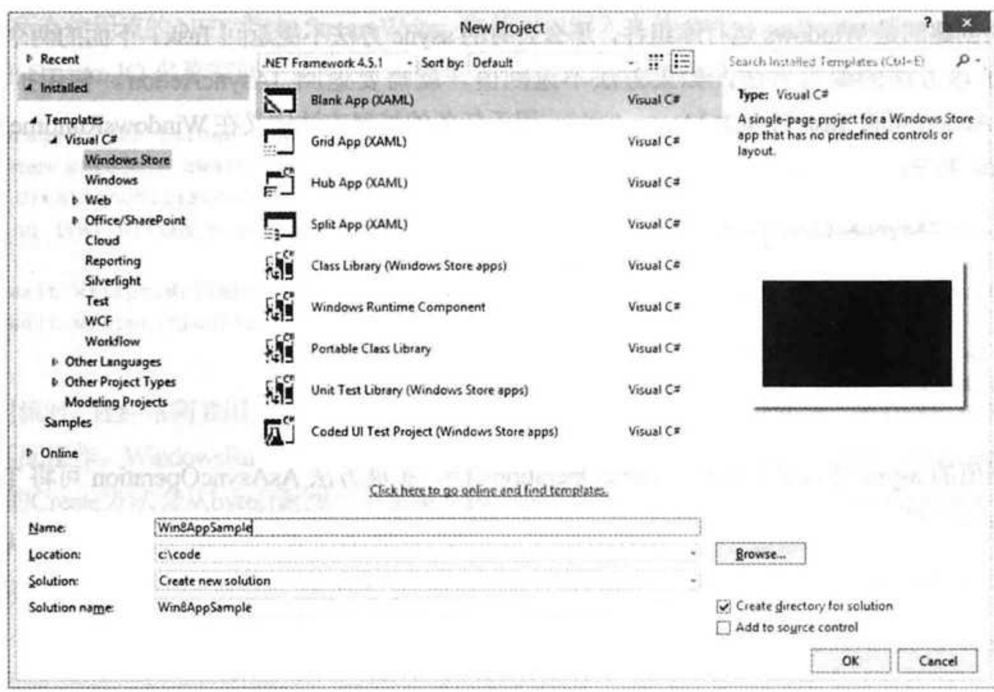


图 31-4

图 31-5

Manifest Designer 创建的 XML 文件如下所示，其中包含了所有配置的值：

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest"
  xmlns:m2="http://schemas.microsoft.com/appx/2013/manifest">
  <Identity Name="920e63ef-9344-4925-a408-86f056314f31" Publisher="CN=Christian"
    Version="1.0.0.0" />
  <Properties>
    <DisplayName>Win8AppSample</DisplayName>
```

```
    <PublisherDisplayName>CN innovation</PublisherDisplayName>
    <Logo>Assets\StoreLogo.png</Logo>
  </Properties>
  <Prerequisites>
    <OSMinVersion>6.3.0</OSMinVersion>
    <OSMaxVersionTested>6.3.0</OSMaxVersionTested>
  </Prerequisites>
  <Resources>
    <Resource Language="x-generate" />
  </Resources>
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe"
      EntryPoint="Win8AppSample.App">
      <m2:VisualElements
        DisplayName="Win8AppSample"
        Square150x150Logo="Assets\Logo.png"
        Square30x30Logo="Assets\SmallLogo.png"
        Description="Win8AppSample"
        ForegroundText="light"
        BackgroundColor="#464646">
        <m2:SplashScreen Image="Assets\SplashScreen.png" />
      </m2:VisualElements>
    </Application>
  </Applications>
  <Capabilities>
    <Capability Name="internetClient" />
  </Capabilities>
</Package>
```

使用 Visual Studio 生成并启动项目，会自动把该项目部署到系统上。也可以创建一个 app 包，以便将应用程序部署到 Windows Store 上，或者用于商业应用程序的旁加载。

第 18 章讨论了 Windows Store 应用程序包的部署。

31.4　应用程序的生命周期

Windows Store 应用程序的生命周期与桌面应用程序不同。在桌面应用程序中，用户控制着应用程序的启动和终止。有时需要使用任务管理器来终止一个挂起的应用程序，但是通常用户决定了并发运行多少应用程序。例如，同时运行 Visual Studio 的多个实例、Outlook、几个 Internet Explorer 窗口和一些工具并不罕见。另一方面，有的用户在关闭一个应用程序后，才打开下一个应用程序。Windows Store 应用程序的工作方式与此完全不同。

在 Windows Store 应用程序中，用户通常不会终止应用程序。启动应用程序的通常方式是单击其磁贴。当然，这与桌面应用程序很类似。但是在此之后，二者就有了显著的区别。当用户启动 Windows Store 应用程序后，它就处于运行模式(running mode)。一旦启动另一个应用程序，使前一个应用程序不可见，该应用程序就进入暂停模式(suspended mode)。在暂停模式下，电池消耗就会降

低。应用程序会驻留在内存中，但是不再能使用 CPU、磁盘或网络资源。该应用程序的全部线程都会暂停。在进入暂停模式前，应用程序有机会做一些处理工作，例如此时应用程序应该保存状态。

当用户切换到前一个应用程序时，该应用程序就会立即从暂停状态恢复(因为它仍然在内存中)，并进入前台。其模式再次成为运行模式。从暂停模式切换到运行模式并不需要用户执行特殊的操作，应用程序数据仍然保留在内存中，所以只需要重新激活该应用程序。

当内存资源短缺时，Windows 可以终止暂停应用程序的进程，从而终止该应用程序。应用程序不会收到任何消息，所以不能对此事件做出反应。因此，应用程序应该在进入暂停模式前做一些处理工作，保存其状态。等到应用程序终止时进行处理就晚了。

31.4.1 应用程序的执行状态

应用程序的状态使用 ApplicationExecutionState 枚举定义。该枚举定义了 NotRunning、Running、Suspended、Terminated 和 ClosedByUser 状态。应用程序需要知道并存储自己的状态，因为用户在返回应用程序时希望继续原来的操作。

在 App 类的 OnLaunched 方法中，可以使用 LauchActivatedEventArgs 参数的 PreviousExecutionState 属性获取应用程序的前一个执行状态。如果应用程序是在安装后第一次启动，在重启计算机后启动，或者用户上一次在任务管理器中终止了其进程，那么该应用程序的前一个状态是 NotRunning。如果用户单击应用程序的图标时应用程序已经激活，或者应用程序通过某个激活契约激活，则其前一个执行状态为 Running。如果应用程序被暂停，那么激活它时 PreviousExecutionState 属性会返回 Suspended。一般来说，在这种情况中不需要执行什么特殊操作。因为状态仍在内存中可用。在暂停状态下，应用程序不使用 CPU 循环，也没有磁盘访问。

用户在停止应用程序时，Windows 8 和 Windows 8.1 有一个重要区别。用户停止运行在 Windows 8 上的应用程序时，应用程序会在 10 秒后停止，它利用这 10 秒时间存储其状态，以便在下一次启动时重用该状态。而在 Windows 8.1 上，用户关闭应用程序时，应用程序会从屏幕和应用程序切换列表中删除，但它仍留在内存中。设置 Windows.UI.ViewManagement.ApplicationView.TerminateAppOnFinalViewClose 属性，可以切换到以前的方式。这个设置只能用于从 Windows 8 迁移来的程序，这种程序需要先关闭，才能再次启动(例如清理不一致的状态)。

应用程序可以实现一个或者多个激活契约，然后由其中的某一个激活契约激活。搜索和分享就是这样的契约。用户不必首先启动应用程序，就可以在应用程序中搜索某些词。此时应用程序才会启动。另外，用户可以从另一个应用程序共享数据，并通过将某个 Windows 8 应用程序用作分享目标来启动该应用程序。第 38 章将讨论应用程序契约。

为了演示 Windows Store 应用程序的生命周期，使用 Blank App 模板创建一个应用程序 LifecycleSample。然后，添加一个新的 Basic Page，命名为 MainPage，并用它替换项目中原来创建的 MainPage。Visual Studio 的项模板 Basic Page 会在 Common 目录中添加几个文件，其中的 SuspensionManager.cs 文件包含类 SuspensionManager，这个类对状态管理有很大的帮助。

为了演示导航状态，除了 MainPage 外，应用程序还包含两个基本页面：Page 1 和 Page2。这两个页面都包含一个用于导航的按钮。MainPage 中按钮的 Click 事件处理程序导航到 Page 2，如下面

的代码片段所示(代码文件 MainPage.xaml.cs)：

```
private void OnGotoPage2(object sender, RoutedEventArgs e)
{
  Frame.Navigate(typeof(Page2));
}
```

其他处理程序与此类似，目的就是当应用程序在切换页面的过程中终止时，用户可以返回原来打开的页面。

31.4.2 Suspension Manager

SuspensionManager 类(代码文件 LifecycleSample/Common/SuspensionManager.c 中)包含的 SaveAsync方法可以保存框架导航状态和会话状态。该方法使用SaveFrameNavigationState方法中所有注册的框架来设置导航状态。在下例中，框架状态添加到会话状态中，以便同时保存它们。首先使用DataContractSerializer将会话状态写入MemoryStream，然后写入XML文件_sessionState.xml：

```
private static Dictionary<string, object> _sessionState =
  new Dictionary<string, object>();
private static List<Type> _knownTypes = new List<Type>();
private const string sessionStateFilename = "_sessionState.xml";

public static async Task SaveAsync()
{
  try
  {
    // Save the navigation state for all registered frames
    foreach (var weakFrameReference in _registeredFrames)
    {
      Frame frame;
      if (weakFrameReference.TryGetTarget(out frame))
      {
        SaveFrameNavigationState(frame);
      }
    }
    // Serialize the session state synchronously to avoid
    // asynchronous access to shared state
    MemoryStream sessionData = new MemoryStream();
    DataContractSerializer serializer =
      new DataContractSerializer(typeof(Dictionary<string, object>),
      _knownTypes);
    serializer.WriteObject(sessionData, _sessionState);
    // Get an output stream for the SessionState file and write the
    // state asynchronously
    StorageFile file = await
      ApplicationData.Current.LocalFolder.CreateFileAsync(
        sessionStateFilename, CreationCollisionOption.ReplaceExisting);
    using (Stream fileStream = await file.OpenStreamForWriteAsync())
    {
      sessionData.Seek(0, SeekOrigin.Begin);
      await sessionData.CopyToAsync(fileStream);
    }
  }
```

```
    catch (Exception e)
    {
      throw new SuspensionManagerException(e);
    }
  }

  private static void SaveFrameNavigationState(Frame frame)
  {
    var frameState = SessionStateForFrame(frame);
    frameState["Navigation"] = frame.GetNavigationState();
  }
```

为了还原数据，可以使用 RestoreAsync 方法。该方法打开一个 StorageFile，并使用 DataContract-Serializer 读取数据：

```
  public static async Task RestoreAsync()
    {
    _sessionState = new Dictionary<String, Object>();

    try
    {
      // Get the input stream for the SessionState file
      StorageFile file =
        await ApplicationData.Current.LocalFolder.GetFileAsync(
        sessionStateFilename);
      using (IInputStream inStream =
        await file.OpenSequentialReadAsync())
      {
        // Deserialize the Session State
        DataContractSerializer serializer = new DataContractSerializer(
          typeof(Dictionary<string, object>), _knownTypes);
      _sessionState =
        (Dictionary<string, object>)serializer.ReadObject(
          inStream.AsStreamForRead());
      }
      // Restore any registered frames to their saved state
      foreach (var weakFrameReference in _registeredFrames)
      {
        Frame frame;
        if (weakFrameReference.TryGetTarget(out frame))
        {
          frame.ClearValue(FrameSessionStateProperty);
          RestoreFrameNavigationState(frame);
        }
      }
    }
    catch (Exception e)
    {
      throw new SuspensionManagerException(e);
    }
  }
```

下一节将添加使用 SuspensionManager 保存导航状态的代码。

31.4.3 导航状态

为了在暂停应用程序时保存状态，将 App 类的 Suspending 事件传递给 OnSuspending 事件处理程序。当应用程序进入暂停模式时，该事件就会触发。使用参数 SuspendingEventArgs 的 SuspendingOperation 属性可以访问 SuspendingOperation 对象。GetDeferral 属性可以延迟应用程序的暂停，从而在暂停前有时间做一些处理。GetDeferral 返回一个 SuspendingDeferral，调用其 Complete 方法可通知运行库暂停状态已完成。SuspendingOperation 的 Deadline 属性定义了最长可以延迟多久才完成暂停，使用此属性可以知道距离应用程序暂停还有多久。在获取 SuspendingDeferral 后，调用 SuspensionManager 的 SaveAsync 方法来保存应用程序状态：

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();
  await SuspensionManager.SaveAsync();
  deferral.Complete();
}
```

为了记忆导航信息，需要在 SuspensionManager 中注册框架。重写 Application 类的 OnLaunched 方法，在创建 Frame 对象后将其注册到 SuspensionManager 中。这样，在暂停应用程序时就会保存框架导航信息，当应用程序状态恢复时相应的变量也会恢复。如果在终止应用程序后再次启动应用程序(通过检查 PreviousExecutionState 属性可判断)，就会调用 SuspensionManager 的 RestoreAsync 方法，从框架中检索导航数据。在导航到 MainPage 之前，检查框架的 Content 属性，确认该属性不为 NULL。如果不为空，说明状态恢复时已填充内容。如果没有完成恢复，则导航到 MainPage(代码文件 LifecycleSample/App.xaml.cs)：

```
protected async override void OnLaunched(LaunchActivatedEventArgs args)
{
  Frame rootFrame = Window.Current.Content as Frame;

  // Do not repeat app initialization when the Window already has content,
  // just ensure that the window is active
  if (rootFrame == null)
  {
    // Create a Frame to act as the navigation context and navigate to the first page
    rootFrame = new Frame();
    // Set the default language
    rootFrame.Language = Windows.Globalization.ApplicationLanguages.Languages[0];

    SuspensionManager.RegisterFrame(rootFrame, "appFrame");

    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      await SuspensionManager.RestoreAsync();
    }
    Window.Current.Content = rootFrame;
  }

  if (rootFrame.Content == null)
  {
    if (!rootFrame.Navigate(typeof(MainPage), e.Arguments))
```

```
    {
      throw new Exception("Failed to create initial page");
    }
  }
```

31.4.4 测试暂停

现在启动该应用程序，导航到另一个页面，然后打开另一个应用程序，并等待前一个应用程序终止。如果将 Status Values 选项设为“Show suspended status”，可以在任务管理器的 More details 视图中看到暂停的应用程序。但是，在测试暂停时，这不是一个简单的方法(因为应用程序可能在很久之后才暂停)，而且不能调试不同的状态。

使用调试器则不同。如果应用程序一旦失去焦点就会暂停，那么每到达一个焦点就会暂停，因此在调试器中运行时，暂停是被禁用的，正常的暂停机制不会起作用。但是，模拟暂停很容易。打开 Debug Location 工具栏，可以看到 3 个按钮：Suspend、Resume 和 Suspend And Shutdown。如果选择 Suspend And Shutdown，然后再次启动应用程序，那么应用程序将从前一个状态 ApplicationExecutionState.Terminated 继续运行，所以会打开用户之前打开的页面。

31.4.5 页面状态

用户输入的任何数据也应该恢复。为了进行演示，在第二个页面上创建一个输入字段。使用一个简单的 Page2Data 类表示该输入字段，类中包含一个 Data 属性，如下面的代码片段所示(代码文件 LifecycleSample/DataModel/Page2Data.cs)：

```
[DataContract]
public class Page2Data : BindableBase
  {
    private string data;

    [DataMember]
    public string Data
    {
      get { return data; }
      set { SetProperty(ref data, value); }
    }
  }
```

基类BindableBase实现了接口INotifyPropertyChanged，以通知属性值何时改变(代码文件Lifecycle Sample/Utilities/BindableBase.cs)：

```
[DataContract]
public class BindableBase : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(
      [CallerMemberName] string propertyName = null)
    {
      PropertyChangedEventHandler handler = PropertyChanged;
      if (handler != null)
      {
        handler(this, new PropertyChangedEventArgs(propertyName));
```

```
      }
    }

    protected void SetProperty<T>(ref T item, T value,
      [CallerMemberName] string propertyName = null)
    {
      if (!EqualityComparer<T>.Default.Equals(item, value))
      {
        item = value;
        OnPropertyChanged(propertyName);
      }
    }
  }
```

输入的数据绑定到一个 TextBox 元素(代码文件 Page2.xaml):

```
<StackPanel Grid.Row="1" DataContext="{Binding Page2Data}">
  <TextBox Text="{Binding Data, Mode=TwoWay}" />
  <Button Content="Goto Page 3" Click="OnGotoPage3" />
</StackPanel>
```

在 Page2 类中，定义一个 Page2Data 变量，用于绑定到 UI 元素：

```
public sealed partial class Page2 : Page
{
  private Page2Data data;
```

为变量赋值并绑定到 UI 的操作可在 navigationHelper_LoadState 方法中完成。这是 navigationHelper 类上 LoadState 事件的处理程序。在该方法的实现代码中，使用 pageState 来确认字典中是否已经包含 Page2 的数据的键。如果是，则访问这个值并赋值给数据变量。否则，创建一个新的 Page2Data 对象：

```
protected void navigationHelper_LoadState(object sender,
  LoadStateEventArgs e)
{
  if (pageState != null && pageState.ContainsKey("Page2"))
  {
    data = pageState["Page2"] as Page2Data;
  }
  else
  {
    data = new Page2Data() { Data = "initital data" };
  }
  this.DefaultViewModel["Page2Data"] = data;
}
```

离开页面时，调用navigationHelper_SaveState方法。和前面一样，SaveState方法由navigationHelper类的SaveState事件调用。其实现代码只需要添加页面的状态：

```
protected void navigationHelper_SaveState(object sender,
  SaveStateEventArgs e)
{
  e.PageState.Add("Page2", data);
}
```

示例不仅写入简单的数据类型，还写入了自定义类型 Page2Data，所以这个类型需要添加到已知的类型中，以进行序列化(代码文件 LifecycleSample/DataModel/App.xaml.cs)：

```
SuspensionManager.KnownTypes.Add(typeof(Page2Data));
```

再次运行应用程序，测试暂停和终止。输入字段的状态设为前一次会话的值。

31.5 应用程序的设置

处理 Windows Store 应用程序的设置时，不应该直接在页面上添加控件，然后允许用户修改它们的值。实际上，Windows Store 应用程序有一个预定义的区域，用于应用程序设置：Charms bar。打开 Charms bar 的方式是：使用触摸方式时，从右向左滑动；使用鼠标时，将鼠标移动到右上角。在 Charms bar 中，选择 Settings 命令(如图 31-6 所示)可打开针对当前应用程序的设置。

为了通过这些设置注册应用程序，只需要在 SettingsPane 类的 CommandsRequested 事件中添加一个命令处理程序，如下面的代码片段所示(代码文件 MainPage.xaml.cs)。InitSettings 通过 OnLaunched 方法调用：

```
private void InitSettings()
{
  SettingsPane.GetForCurrentView().CommandsRequested +=
  AppCommandsRequested;
}
```

SettingsPane 还提供了一个静态的 Show 方法，可用于打开由应用程序直接控制的设置。如果需要让用户在第一次启动应用程序时对应用程序做一些初始配置，那么打开 Settings 窗格是一个不错的选择。

一旦用户打开设置，就会调用命令处理程序 MainPage_CommandsRequests。这个处理程序只需要在 SettingsPaneCommandRequest 对象中添加命令。通过 SettingsPaneCommandsRequestEventArgs 参数的 Request 属性，可访问该对象。ApplicationCommands 返回 SettingsCommand 的一个 IList，用于添加命令，包括定义 ID、在 UI 中显示的标签以及一个事件处理程序。当用户单击命令时，就会调用这个事件处理程序。下面的代码创建了两个命令，并把它们添加到 Settings 窗格中：

```
void AppCommandsRequested(SettingsPane sender,
    SettingsPaneCommandsRequestedEventArgs args)
  {
    var command1 = new SettingsCommand("command1", "Command 1",
      new UICommandInvokedHandler(Command1));
    args.Request.ApplicationCommands.Add(command1);
    var command2 = new SettingsCommand("command2", "Command 2",
      new UICommandInvokedHandler(Command2));
    args.Request.ApplicationCommands.Add(command2);
  }
```

当用户单击 Settings 时，会显示如图 31-7 所示的命令。除了这些命令以外，所有的 Windows Store 应用程序还可以使用 Permissions 命令。该命令可显示应用程序在使用功能时请求的所有权限。如果应用程序在商店中可用，就会显示另一个命令：Rate and Review。

```
private async void Command1(IUICommand command)
{
  await Launcher.LaunchUriAsync(
    new Uri("http://www.cninnovation.com"));
}
```

单击一个命令会调用应用程序的事件处理程序，允许根据需要响应和调整设置。Command1 事件处理程序使用 Launcher 类打开 Web 浏览器：

图 31-6

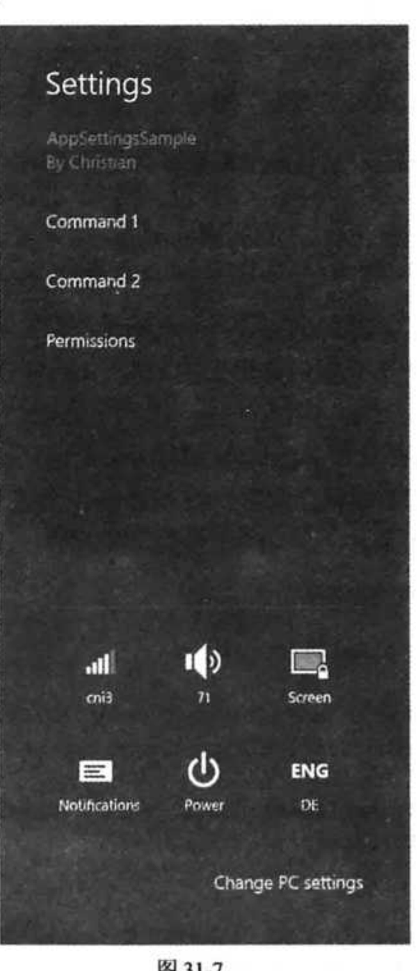


图 31-7

如果需要用户提供更多的信息，可以在事件处理程序中显示一个弹出窗口。Visual Studio 2013 提供了 Settings Flyout 模板，用于给设置创建 UI。下面的代码片段显示了定义输入控件的 XAML 代码(代码文件 SettingsDemo/SampleSettingsPane.xaml)：

```
<SettingsFlyout
  x:Class="AppSettingsSample.SettingsFlyout1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:AppSettingsSample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  IconSource="Assets/SmallLogo.png"
  Title="SettingsFlyout1"
```

```
  d:DesignWidth="346">

  <StackPanel VerticalAlignment="Stretch" HorizontalAlignment="Stretch" >
    <StackPanel Style="{StaticResource SettingsFlyoutSectionStyle}">
      <TextBlock Style="{StaticResource TitleTextBlockStyle}"
        Text="Sample Header" />
      <TextBox Margin="0,0,0,25" Header="Text1"
        Text="{Binding Text1, Mode=TwoWay}" />

    </StackPanel>
  </StackPanel>
</SettingsFlyout>
```

调用Show 方法，从Command2事件处理程序中打开弹出窗口(代码文件AppSettingsSample/App.xaml.cs)：

```
private void Command2(IUICommand command)
{
  var flyout = new SettingsFlyout1();
  flyout.Show();
}
```

用户数据很容易使用 ApplicationData 类来保存。LocalSettings 属性用于在 PC 上本地保存，而 RoamingSettings 把数据保存到用户的 Microsoft 账户附带的漫游存储器上。辅助类 RoamingSettings 提供了 GetValue 和 SetValue 方法，它们可以使用 RoamingSettings 属性(代码文件 AppSettingsSample/Utilities/RoamingSettings.cs)：

```
static class RoamingSettings
{
  public static T GetValue<T>(T defaultValue = default(T),
    [CallerMemberName] string propertyName = null)
  {
    return (T)(ApplicationData.Current.RoamingSettings.Values[propertyName]
    ?? defaultValue);
  }

  public static void SetValue<T>(T value,
  [CallerMemberName] string propertyName = null)
  {
    ApplicationData.Current.RoamingSettings.Values[propertyName] = value;
  }
}
```

弹出窗口的 Text1 属性使用 RoamingSettings 类型检索并存储设置(代码文件 AppSettingsSample/SettingsFlyout1.xaml.cs)：

```
public string Text1
{
  get { return RoamingSettings.GetValue(String.Empty); }
  set { RoamingSettings.SetValue(value); }
}
```

31.6　小结

本章介绍了 Windows 运行库的核心功能及其与.NET 应用程序的区别。通过语言投射，应用程序能够以与所用编程语言兼容的方式使用 Windows 运行库。从本章可以看到，使用语言投射时，原生 API 可以很方便地用来编写.NET 代码。

本章还介绍了 Windows 运行库组件使用的接口，这些接口如何自动映射到.NET 接口(集合)，以及何时需要使用不同的类型以及使用扩展方法转换它们。

本章还讨论了编写 Windows Store 应用程序的核心概念，包括生命周期和应用程序设置，以及如何定义和使用功能。

关于编写 Windows Store 应用程序的详细信息，请参考第 38 章。第 38 章介绍了如何使用 XAML 编写 Windows Store 应用程序的用户界面，包括 App bar 和 Windows Store 专用布局控件等功能。第 39 章介绍了如何使用传感器和设备。

前面介绍了.NET 的核心功能和 Windows 运行库，从下一章将开始介绍数据访问。在 Windows Store 应用程序中，不能直接使用 ADO.NET，但是 ADO.NET 对于服务很重要——Windows Store 应用程序当然是需要和服务交互的。第 32 章对于 Windows 桌面应用程序也很重要。

第Ⅳ部分

数　　据

- 第 32 章　核心 ADO.NET
- 第 33 章　ADO.NET Entity Framework
- 第 34 章　处理 XML

第32章

核心 ADO.NET

本章要点

- 连接数据库
- 执行命令
- 调用存储过程
- ADO.NET 对象模型
- 使用 XML 和 XML 架构

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- ExecutingCommands
- StoredProcs
- DataReader
- IndexerTesting
- AsyncDataReaders
- SimpleDataset
- ManufacturedDataset
- DataAdapter
- DataAdapterUpdate
- Auditing

32.1 ADO.NET 概述

本章讨论如何使用 ADO.NET 访问 C#程序中的数据，主要介绍如何使用 SqlConnection 类和 OleDbConnection 类连接数据库，以及断开与数据库的连接。深入讨论命令对象上的各种选项，并说明如何为 Sql 类和 OleDB 类的每个选项使用命令。如何使用命令对象来调用存储过程，这些存储

过程的结果如何集成到缓存在客户端上的数据中。

ADO.NET对象模型与ADO中可用的对象完全不同，本章将讨论DataSet、DataTable、DataRow和DataColumn类。DataSet类也可以包含表之间、约束之间的关系。类层次结构自从.NET Framework 2.0版本之后有许多变化，本章也将介绍这些变化。最后将讨论XML架构，它是构建ADO.NET的基础。

下面首先简要介绍 ADO.NET。

ADO.NET 比现有 API 在技术上高出很多。它与 ADO 仅仅是名称类似，类和访问数据的方法则完全不同。

ADO(ActiveX Data Objects)是一个 COM 组件库，在过去的几年中，这些组件有许多版本。ADO 主要包含 Connection、Command、Recordset 和 Field 对象。使用 ADO 时，要打开与数据库的连接，选择一些数据，并把它们放在记录集中，这些记录集由字段组成，接着处理这些数据，并在服务器上更新它们，最后关闭连接。ADO 还引入了一个概念：断开连接的记录集，当不适合使连接打开相当长的时间时，就可以使用该概念。

ADO 还有几个没有完全解决的问题，其中最著名的就是笨拙的断开连接的记录集。对这个支持的需求要比以往“以 Web 为中心”的计算更高，所以需要一种新方式。ADO.NET 和 ADO(不仅仅是名称)有许多相似之处，所以从 ADO 升级到 ADO.NET 不会很困难。而且，如果使用的是 SQL Server，它有一组很好的托管类，调整这些类可以很好地发挥出数据库的最佳性能。单是这一个理由，就足以迁移到 ADO.NET 了。

ADO.NET 附带了 3 个数据库客户端名称空间，第 1 个用于 SQL Server，第 2 个用于 ODBC 数据源，第 3 个用于通过 OLE DB 实现的数据库。如果数据库不是 SQL Server，就应在线搜索一个专门的.NET 提供程序，如果找不到这样的.NET 提供程序，就应使用 OLE DB 路由，除非还能使用 ODBC。如果使用 Oracle 作为数据库，就可以访问 Oracle .NET Developer 站点，从 www.oracle.com/technology/tech/windows/odpnet/index.html 上获取其.NET 提供程序 ODP.NET。

32.1.1 名称空间

本章所有的示例都以某种方式访问数据。表 32-1 所示的名称空间提供了在.NET 数据访问中使用的类和接口。

表 32-1

名称空间	说明
System.Data	所有数据访问泛型类
System.Data.Common	各个数据提供程序共享(或重写)的类
System.Data.EntityClient	Entity Framework 类
System.Data.Linq.SqlClient	LINQ to SQL 提供程序类
System.Data.Odbc	ODBC 提供程序的类
System.Data.OleDb	OLE DB 提供程序的类
System.Data.ProviderBase	新的基类和连接工厂类
System.Data.Sql	用于 SQL Server 数据访问的新泛型接口和类
System.Data.SqlClient	SqlServer 提供程序的类
System.Data.SqlTypes	SqlServer 数据类型

下面两节列出 ADO.NET 中主要的类。

32.1.2 共享类

ADO.NET 包含许多类，无论是使用 SQL Server 类，还是使用 OLE DB 类，都可以使用它们。表 32-2 列出的类包含在 System.Data 名称空间中。

表 32-2

类	说 明
DataSet	这个对象主要用于断开连接，它可以包含一组 DataTable，以及这些表之间的关系
DataTable	数据的一个容器，DataTable 由一个或多个 DataColumn 组成，每个 DataColumn 由一个或多个包含数据的 DataRow 组成
DataRow	许多数值，类似于数据库表的一行，或电子表格中的一行
DataColumn	该对象包含列的定义，如名称和数据类型
DataRelation	DataSet 类中两个 DataTable 类之间的链接，用于外键和主/从关系
Constraint	为 DataColumn 类(或一组数据列)定义规则，如唯一值
DataColumnMapping	将数据库中的列名映射到 DataTable 中的列名
DataTableMapping	将数据库中的表名映射到 DateSet 中的 DataTable

32.1.3 数据库专用类

除了上一节介绍的共享类外，ADO.NET 还包含许多数据库专用类。这些类实现一组在 System.Data 名称空间中定义的标准接口，根据需要允许类按照一般形式来使用。例如，SqlConnection 类和 OleDbConnection 类派生于实现了 IDbConnection 接口的 DbConnection 类。表 32-3 列出了数据库专用类。

表 32-3

类	说 明
SqlCommand、OleDbCommand 和 ODBCCommand	用作 SQL 语句或存储过程调用的包装器，SqlCommand 类的示例详见本章后面的内容
SqlCommandBuilder、OleDbCommandBuilder 和 ODBCCommandBuilder	用于从一条 SELECT 语句中生成 SQL 命令(如 INSERT、UPDATE 和 DELETE 语句)
SqlConnection、OleDbConnection 和 ODBCConnection	用于连接数据库。类似于 ADO 连接，示例详见本章后面的内容
SqlDataAdapter、OleDbDataAdapter 和 ODBCDataAdapter	用于存储 select、insert、update 和 delete 命令的类，因此可以用于填充 DataSet 和更新数据库，SqlDataAdapter 的示例详见本章后面的内容
SqlDataReader、OleDbDataReader 和 ODBCDataReader	用作只向前的连接数据读取器，SqlDataReader 的示例详见本章后面的内容
SqlParameter、OleDbParameter 和 ODBCParameter	用于为存储过程定义一个参数，如何使用 SqlParameter 的示例详见本章后面的内容
SqlTransaction、OleDbTransaction 和 ODBCTransaction	用于数据库事务，包装在一个对象中

ADO.NET 类最重要的功能是：它们是以断开连接的方式工作，这在目前以 Web 为中心的环境中非常重要。我们常常构建一个服务(例如在线书店)，以连接到一个服务器，检索一些数据，再在客户端上处理这些数据，之后重新连接服务器，并把数据传递回去，进行处理。ADO.NET 的断开连接的本质就可以启用这种操作。

传统的 ADO 2.1 引入了断开连接的记录集，它允许从数据库中检索数据，把它们传递给客户端，进行处理，再重新连接服务器。但它们使用起来常常很繁琐，因为断开连接的工作方式不是一开始就设计好的。ADO.NET 类则不同，除了一种情况([provider]DataReader)外，它们都用于脱机处理数据库。

本章将介绍.NET Framework 中用于数据访问的类和接口，主要论述连接数据库时使用的 SQL Server 类，因为 Framework SDK 示例安装了一个 SQL Server Express 数据库(SQL Server)。在大多数情况下，OLE DB 类和 ODBC 类类似于 SQLServer 代码。

32.2 使用数据库连接

为了访问数据库，需要提供某种连接参数，如运行数据库的计算机和登录证书。使用 ADO 的用户会很快熟悉.NET 连接类 OleDbConnection 和 SqlConnection，图 32-1 显示了两个连接类及类的层次结构。

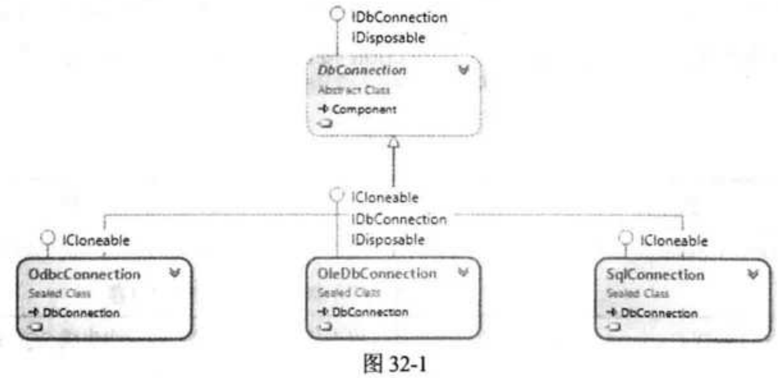


图 32-1

本章的示例使用 Northwind 数据库，可以在网络上搜索 Northwind and pubs Sample Databases for SQL Server，找到它。下面的代码段说明了如何创建、打开和关闭 Northwind 数据库的连接。

```
using System.Data.SqlClient;

string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();

// Do something useful

conn.Close();
```

如果以前使用过 ADO 或 OLE DB，就会很熟悉连接字符串。如果使用的是 OleDb 提供程序，就应该能剪切和粘贴旧代码。在该示例的连接字符串中，使用的参数如下所示。连接字符串中的参数用分号分隔开。

- server=(local)：表示要连接到的数据库服务器。SQL Server 允许在同一台计算机上运行多个不同的数据库服务器实例，这里连接到默认的 SQL Server 实例。如果使用 SQL Express，就把服务器部分改为 server=./sqlexpress。
- integrated security=SSPI：这个参数使用 Windows Authentication 连接到数据库，最好在源代码中使用这个参数，而不是用户名和密码。
- database=Northwind：这描述了要连接到的数据库实例。每个 SQL Server 进程都可以提供几个数据库实例。

如果忘记数据库连接字符串的格式，就可以使用下面的 URL：www.connectionstrings.com。

这个 Northwind 示例使用定义好的连接字符串打开数据库连接，再关闭该连接。一旦打开连接后，就可以对数据源执行命令，完成后，就可以关闭连接。

SQL Server 有另一种模式的身份验证。它可以使用 Windows 集成的安全性，这样在登录时提供的证书就会传递给 SQL Server。为此，应删除连接字符串的 uid 和 pwd 部分，并添加 Integrated Security=SSPI。

在本章的下载代码中，有一个文件 login.cs 简化了本章的示例。它链接到所有的示例代码，包括用于这些示例的数据库连接信息；可以修改该文件，使用自己的服务器名称、用户名和密码。在默认情况下，该文件使用 Windows 集成的安全性，但是可以根据需要修改用户名和密码。

32.2.1 管理连接字符串

在以前的.NET 版本中，由开发人员管理数据库的连接字符串，其方法常常是把连接字符串存储在应用程序配置文件中，或者更常见的是，在应用程序的某个地方硬编码连接字符串。

从.NET 2.0 开始，有一种预定义的方式来存储连接字符串，甚至是以类型未知的方式使用数据库连接。例如，现在可以编写应用程序，然后插入各个数据库提供程序，而这些都无须修改主应用程序。

要定义数据库连接字符串，应使用配置文件中的<connectionStrings>部分。在这里可以指定连接的名称、数据库连接字符串的实际参数，还需要指定这个连接类型的提供程序。下面是一个例子：

```
<configuration>
  ...
  <connectionStrings>
    <add name="Northwind"
         providerName="System.Data.SqlClient"
         connectionString="server=(local);integrated security=SSPI;database=Northwind" />
  </connectionStrings>
</configuration>
```

本章将在其他例子中使用这个连接字符串。

一旦在配置文件中定义了数据库连接信息，就需要在应用程序中利用该信息。我们常常要创建

如下方法，根据连接的名称检索数据库连接：

```
private DbConnection GetDatabaseConnection ( string name )
{
  ConnectionStringSettings settings =
    ConfigurationManager.ConnectionStrings[name];

  DbProviderFactory factory = DbProviderFactories.GetFactory
    ( settings.ProviderName );

  DbConnection conn = factory.CreateConnection ( );
  conn.ConnectionString = settings.ConnectionString;

  return conn;
}
```

这段代码首先读取指定的连接字符串段(使用 ConfigurationStringSettings 类)，再从基类 DbProviderFactories 中申请一个提供程序工厂，这要使用 ProviderName 属性，它在应用程序配置文件中设置为"System.Data.SqlClient"。它会映射为实际的工厂类，用于为 SQL Server 生成数据库连接，在本例中应使用 System.Data.SqlClient 中的 SqlClientFactory 类。必须添加对 System. Configuration 程序集的引用，才能解析上述代码使用的 ConfigurationManager 类。

这对于获得数据库连接似乎是不必要的工作，如果应用程序从来不运行在其他数据库(除了为它设计的数据库之外)上，这些工作的确没有必要。但如果使用前面的工厂方法和泛型 Db*类(如 DbConnection、DbCommand 和 DbDataReader)，就会发现，以后将该应用程序迁移到另一个数据库系统上非常简单。

32.2.2　高效地使用连接

一般情况下，当在.NET 中使用“稀缺”的资源时，如数据库连接、窗口或图形对象，最好确保每个资源在使用完后立即关闭。尽管.NET 的设计人员实现了自动的垃圾收集，垃圾最终都会被回收，但仍需要尽可能早地释放资源，以避免出现资源匮乏的情况。

当编写访问数据库的代码时，这都非常明显，因为使连接打开的时间略长于需要的时间，就可能影响其他会话。在极端的情况下，不关闭连接会使其他用户无法进入一整组数据表，极大地降低了应用程序的性能。因为关闭数据库连接应是强制的，所以本节讨论如何构建代码，把一直打开资源的风险降到最低。主要有两种方式可以确保数据库连接等类似的“稀缺”资源在使用完后立即释放。下面就介绍这两种方式。

1. 第一种方式——利用 try...catch...finally 语句块

确保释放资源的第一种方式是利用 try...catch...finally 块，确保在 finally 块中关闭任何已打开的连接。下面是一个小示例：

```
try
{
   // Open the connection
   conn.Open();
   // Do something useful
}
```

```
catch ( SqlException ex )
{
   // Log the exception
}
finally
{
   // Ensure that the connection is freed
   conn.Close ( );
}
```

在 finally 块中，可以释放已经使用的任何资源。这种方式的唯一麻烦是必须确保关闭连接。很容易忘记在 finally 块中添加关闭连接的命令，所以应在编码风格上添加一些不容易出现反常情况的内容。

另外，在给定的方法中可能会打开许多资源(如两个数据库连接和一个文件)，这样 try…catch…finally 块的层次有时可能不容易看懂。但还有另一种方式可以确保资源的关闭——使用 using 语句。

2. 第二种方式——使用 using 语句块

在开发 C#的过程中，.NET 在对象不再引用之后清理它们的方法是使用非决定性的析构方式，这已经引起了非常热烈的讨论。在 C++中，对象只要使用完毕，就会自动调用其析构函数。这对于基于资源的类的设计人员是非常好的消息，因为如果用户忘记关闭资源，则最好使用析构函数。只要对象使用完毕，就会调用 C++析构函数。所以，例如，如果出现了异常，但没有捕获，有析构函数的对象就会调用它们的析构函数。

在 C#和其他托管语言中，没有自动的、决定性的析构概念，而是有一个垃圾收集器，它会在未来的某个时刻释放资源。它是非决定性的，因为我们不能确定这个过程在什么时候发生。忘记关闭数据库连接可能会导致.NET 可执行程序的各种问题。幸运的是，我们还有解决的方法。下面的代码说明了如何使用 using 子句确保在退出块后立即释放实现 IDisposable 接口(详见第 14 章)的对象。

```
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";

using ( SqlConnection conn = new SqlConnection ( source ) )
{
   // Open the connection
   conn.Open ( );

   // Do something useful
}
```

在这个实例中，无论块是如何退出的，using 子句都会确保关闭数据库连接。

查看一下连接类的 Dispose()方法的 IL 代码，它们都检查连接对象的当前状态，如果其状态为打开，就调用 Close()方法。浏览.NET 程序集的一个强大工具是 Reflector(可以从 www.reflector.com 上获得)。这个工具允许查看任何.NET 方法的 IL 代码，还可以把 IL 代码反编译为 C#源代码，让我们轻松地确定给定方法的作用。

在编程时，应至少使用这两种方法中的一种，或者两种方法都使用。无论在哪里获得资源，最好都使用 using()语句，因为尽管我们都打算编写 Close()语句，但有时会忘记，并且出现异常时 using

子句就会发挥作用。因为这两种方式都没有好的异常处理方式来替代，所以在大多数情况下，最好组合使用这两种方法，如下面的示例所示。

```
try
{
    using (SqlConnection conn = new SqlConnection ( source ))
    {
        // Open the connection
        conn.Open ( );

        // Do something useful
        // Close it myself
        conn.Close ( );
    }
}
catch (SqlException e)
{
    // Log the exception & rethrow
    throw;
}
```

这里调用了 Close()方法，但严格来说这是不必要的，因为 using 子句将确保在任何情况下都执行关闭操作。但是，应确保像这样的任何资源尽可能早地释放。因为在块的其余部分可能有更多的代码，而没有必要锁定资源。

另外，如果在 using 块中出现了异常，using 子句就会确保在资源上调用 IDisposable.Dispose()方法，在本例中将确保总是关闭数据库连接。这样，与必须确保在异常子句中关闭连接相比，代码的可读性更高。还要注意，异常定义为 SqlException，而不是捕获所有异常的 Exception 类型——应总是捕获特定的异常，把不显式处理的所有其他异常放在执行栈中。如果专用的数据类可以处理错误，并执行一些操作，就应仅捕获这个异常。

最后，如果编写一个封装资源的类，那么无论该资源是什么，都应实现 IDisposable 接口，以关闭资源。这样，任何使用该类的代码都可以利用 using()语句，并确保资源被释放。

32.2.3 事务

通常，对数据库要进行多次更新，这些更新必须在事务的范围内进行。我们常常要在代码中查找一个事务对象，它传递给许多方法，以更新数据库，但在.NET Framework 2.0 及其更高版本中，在 System.Transactions 程序集中添加了 TransactionScope 类，它极大地简化了事务代码的编写，因为可以把几个事务方法合并到一个事务范围中，事务流会根据需要执行每个方法。

下面的代码是在 SQL Server 连接上开始事务处理：

```
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";

using (TransactionScope scope = new
    TransactionScope(TransactionScopeOption.Required))
{
    using (SqlConnection conn = new SqlConnection(source))
    {
```

```
            // Do something in SQL
            .

            // Then mark complete
            scope.Complete();
        }
    }
```

这段代码使用 scope.Complete()方法把事务显式地标记为完成。如果不调用这个方法，事务就会回滚，以便不对数据库进行任何修改。

在使用事务作用域时，可以选择在该事务中执行的命令的独立级别。该级别确定了如何在一个数据库会话中查看在另一个数据库会话中所进行的修改，并不是所有数据库引擎都支持表 32-4 所示的 4 个级别。

表 32-4

独立级别	说　明
ReadCommitted	SQL Server 的默认级别。这个级别可以确保只有第一个事务提交后，在第二个事务中才能访问第一个事务写入的数据
ReadUncommitted	即使一个事务还没有提交数据，也允许另一个事务从数据库中读取数据。例如，如果两个用户在访问同一个数据库，第一个用户插入一些数据，但没有完成事务(通过 Commit 或 Rollback 方法)，第二个用户把自己的独立级别设置为 ReadUncommitted，因此可以访问数据
RepeatableRead	这个级别扩展了 ReadCommitted 级别，确保如果在事务中使用了相同的语句，无论是否有其他潜在的数据库更新，总是可以返回相同的数据。这个级别要求对数据进行额外的锁定，这会降低性能。这个级别可以保证，对于初始查询的每一行，都不会修改数据，但允许显示“幻象(phantom)”行——这些行是在事务运行时，由另一个事务插入的全新数据行
Serializable	这是最“高级”的事务级别，对数据库中的数据进行串行化访问。利用这种独立级别，不会显示幻象行，所以在可串行化的事务中使用的 SQL 语句总是检索相同的数据。可串行化的事务对性能的负面影响不应低估，如果不肯定是否需要使用这个独立级别，最好不要使用它

SQL Server 的默认独立级别 ReadCommitted 是数据一致性和数据可用性之间的一种很好的折中，因为它比 RepeatableRead 或 Serializable 模式中需要的数据锁定都少。但是，有时应提高独立级别，这样在.NET 中，才能从一种非默认的级别开始事务处理。使用哪个级别没有硬性规则，全凭经验。

如果当前使用的是不支持事务的数据库，就应转而使用支持它的数据库。一旦我们成为可以完全信任的雇员，且拥有错误数据库的全部访问权限，就可能试输入 delete from bug where id=99999 以删除对应的错误，但实际上输入的是“<”而不是“=”，此时会删除整个错误数据库，这可不是我们希望的。幸好 IS 小组每天晚上都会备份该数据库，可以还原它，但使用回滚命令会更简单。

32.3 命令

32.2 节简要介绍了针对数据库执行的命令。简言之，命令就是一个要在数据库上执行的包含 SQL 语句的文本字符串。命令也可以是一个存储过程，或者返回表中所有列和所有行的表的名称(换言之，SELECT *样式的子句)。

把 SQL 子句作为一个参数传递给 Command 类的构造函数，就可以构造一条命令，如下例所示：

```
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(select, conn);
```

<provider>Command 类有一个 CommandType 属性，它用来定义某条命令是 SQL 子句、存储过程的调用，还是完整的表语句(仅从给定的表中选择所有列和行)。表 32-5 总结了 CommandType 枚举。

表 32-5

命令类型	样 例
Text(默认)	String select = "SELECT ContactName FROM Customers" SqlCommand cmd = new SqlCommand(select, conn);

StoredProcedure	SqlCommandcmd=newSqlCommand("CustOrderHist", conn);
	cmd.CommandType = CommandType.StoredProcedure;
	cmd.Parameters.Add("@CustomerID", "QUICK");

TableDirect	OleDbCommand cmd = new OleDbCommand("Categories", conn); cmd.CommandType = CommandType.TableDirect;

在执行存储过程时，需要把参数传送给过程。上面的示例直接设置了参数@CustomerID，尽管设置参数的值还可以使用其他方式。注意自从.NET 2.0 开始，给命令参数集合添加了 AddWithValue()方法，而废弃了 Add(name, value)成员。如果习惯于使用这个构建参数的原始方法来调用存储过程，在重新编译代码时，就会得到一个编译警告。最好现在就修改代码，因为 Microsoft 在.NET 的后续版本中将删除旧方法。

TableDirect 命令类型只对 OleDb 提供程序有效——如果试图把这个命令类型用于其他提供程序，就会抛出异常。

32.3.1 执行命令

定义好命令后，就需要执行它。执行语句有许多方式，这取决于要从命令中返回什么数据。<provider>Command 类提供了下述可执行的命令：

- ExecuteNonQuery()——执行命令，但不返回任何结果。
- ExecuteReader()——执行命令，返回一个类型化的 IDataReader。

- ExecuteScalar()——执行命令，返回结果集中第一行第一列的值。

除了上述命令外，SqlCommand 类也提供了下面的方法：

- ExecuteXmlReader()——执行命令，返回一个 XmlReader 对象，它可以遍历从数据库中返回的 XML 片段。

1. ExecuteNonQuery()方法

这个方法一般用于 UPDATE、INSERT 或 DELETE 语句，其中唯一的返回值是受影响的记录个数。但如果调用带输出参数的存储过程，该方法就有返回值：

```
static void ExecuteNonQuery()
{
    string select = "UPDATE Customers " +
                    "SET ContactName = 'Bob' " +
                    "WHERE ContactName = 'Bill'";
    SqlConnection conn = new SqlConnection(GetDatabaseConnection());
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    int rowsReturned = cmd.ExecuteNonQuery();
    Console.WriteLine("{0} rows returned.", rowsReturned);
    conn.Close();
}
```

ExecuteNonQuery()方法返回命令所影响的行数，它是一个整数。

2. ExecuteReader()方法

这个方法执行命令，并根据使用的提供程序返回一个类型化的 DataReader 对象，返回的对象可以用于遍历返回的记录，如下面的代码所示。

```
static void ExecuteReader()
{
    string select = "SELECT ContactName,CompanyName FROM Customers";
    SqlConnection conn = new SqlConnection(GetDatabaseConnection());
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    SqlDataReader reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        Console.WriteLine("Contact: {0,-20} Company: {1}",
                          reader[0], reader[1]);
    }
}
```

图 32-2 显示了这段代码的结果。

本章的后面将讨论<provider>DataReader 对象。

3. ExecuteScalar()方法

在许多情况下，需要从 SQL 语句返回一个结果，如给定表中的记录个数，或者服务器上的当前日期/时间。ExecuteScalar()方法就可以用于这些场合：

```
C:\WINDOWS\system32\cmd.exe
Contact: Michael Holz          Company: Richter Supermarkt
Contact: Alejandra Camino      Company: Romero y tomillo
Contact: Jonas Bergulfsen      Company: Santé Gourmet
Contact: Jose Pavarotti        Company: Save-a-lot Markets
Contact: Hari Kumar            Company: Seven Seas Imports
Contact: Jytte Petersen        Company: Simons bistro
Contact: Dominique Perrier     Company: Spécialités du monde
Contact: Art Braunschweiger    Company: Split Rail Beer & Ale
Contact: Pascale Cartrain      Company: Suprêmes délices
Contact: Liz Nixon             Company: The Big Cheese
Contact: Liu Wong              Company: The Cracker Box
Contact: Karin Josephs         Company: Toms Spezialitäten
Contact: Miguel Angel Paolino  Company: Tortuga Restaurante
Contact: Anabela Domingues     Company: Tradição Hipermercados
Contact: Helvetius Nagy        Company: Trail's Head Gourmet Provisioners
Contact: Palle Ibsen           Company: Vaffeljernet
Contact: Mary Saveley          Company: Victuailles en stock
Contact: Paul Henriot          Company: Vins et alcools Chevalier
Contact: Rita Müller           Company: Die Wandernde Kuh
Contact: Pirkko Koskitalo      Company: Wartian Herkku
Contact: Paula Parente         Company: Wellington Importadora
Contact: Karl Jablonski        Company: White Clover Markets
Contact: Matti Karttunen       Company: Wilman Kala
Contact: Zbyszek Piestrzeniewicz Company: Wolski  Zajazd
Press any key to continue . . .
```

图 32-2

```
static void ExecuteScalar()
{
    string select = "SELECT COUNT(*) FROM Customers";
    SqlConnection conn = new SqlConnection(GetDatabaseConnection());
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    object o = cmd.ExecuteScalar();
    Console.WriteLine(o);
}
```

该方法返回一个对象，根据需要，可以把该对象强制转换为合适的类型。如果所调用的 SQL 只返回一列，则最好使用 ExecuteScalar()方法来检索这一列。这也适合于只返回一个值的存储过程。

4. ExecuteXmlReader()方法(只用于 SqlClient 提供程序)

顾名思义，这个方法执行 SQL 语句，给调用者返回一个 XmlReader 对象。SQL Server 允许使用 FOR XML 子句来扩展 SQL 的 SELECT 子句。这个子句可以带有下述 3 个选项中的一个：

- FOR XML AUTO——根据 FROM 子句中的表构建一棵树
- FOR XML RAW——把结果集中的行映射到元素，其中的列映射到属性
- FOR XML EXPLICIT——必须指定要返回的 XML 树的形状

对于本示例，使用 AUTO：

```
static void ExecuteXmlReader()
{
    string select = "SELECT ContactName,CompanyName " +
                    "FROM Customers FOR XML AUTO";
    SqlConnection conn = new SqlConnection(GetDatabaseConnection());
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    XmlReader xr = cmd.ExecuteXmlReader();
    xr.Read();
    string data;
    do
    {
        data = xr.ReadOuterXml();
        if (!string.IsNullOrEmpty(data))
           Console.WriteLine(data);
```

```
    } while (!string.IsNullOrEmpty(data));
    conn.Close();

}
```

注意，必须导入 System.Xml 名称空间，才能输出返回的 XML。这个名称空间和.NET Framework 其他的 XML 功能将在第 34 章中详细论述。本例在 SQL 语句中包含了 FOR XML AUTO 子句，然后调用 ExecuteXmlReader()方法。代码的结果如图 32-3 所示。

SQL 子句指定了 FROMCustomers，这样 Customers 类型的元素就显示在输出中。为它添加特性，每个特性对应于从数据库中选择出来的每一列。这就为从数据库中选择的每一行构建了 XML 片段。

```
C:\WINDOWS\system32\cmd.exe
<Customers ContactName="Art Braunschweiger" CompanyName="Split Rail Beer &amp; Ale" />
<Customers ContactName="Pascale Cartrain" CompanyName="Suprêmes délices" />
<Customers ContactName="Liz Nixon" CompanyName="The Big Cheese" />
<Customers ContactName="Liu Wong" CompanyName="The Cracker Box" />
<Customers ContactName="Karin Josephs" CompanyName="Toms Spezialitäten" />
<Customers ContactName="Miguel Angel Paolino" CompanyName="Tortuga Restaurante" />
<Customers ContactName="Anabela Domingues" CompanyName="Tradição Hipermercados" />
<Customers ContactName="Helvetius Nagy" CompanyName="Trail's Head Gourmet Provisioners" />
<Customers ContactName="Palle Ibsen" CompanyName="Vaffeljernet" />
<Customers ContactName="Mary Saveley" CompanyName="Victuailles en stock" />
<Customers ContactName="Paul Henriot" CompanyName="Vins et alcools Chevalier" />
<Customers ContactName="Rita Müller" CompanyName="Die Wandernde Kuh" />
<Customers ContactName="Pirkko Koskitalo" CompanyName="Wartian Herkku" />
<Customers ContactName="Paula Parente" CompanyName="Wellington Importadora" />
<Customers ContactName="Karl Jablonski" CompanyName="White Clover Markets" />
<Customers ContactName="Matti Karttunen" CompanyName="Wilman Kala" />
<Customers ContactName="Zbyszek Piestrzeniewicz" CompanyName="Wolski  Zajazd" />
Press any key to continue . . .
```

图 32-3

32.3.2　调用存储过程

用命令对象调用存储过程，就是定义存储过程的名称，给过程的每个参数添加参数定义，然后用上一节中给出的其中一种方法执行命令。

为了使本节的示例更有说服力，下面定义一组可用于在 Northwind 样本数据库的 Region 表中插入、更新和删除记录的存储过程。尽管 Region 表很小，但它可以用于给每种常见的存储过程编写示例，它是一个很好的示例。

1. 调用没有返回值的存储过程

调用存储过程最简单的示例是不给调用者返回任何值。下面定义了两个这样的存储过程，一个用于更新先前已有的 Region 记录，另一个用于删除指定的 Region 记录。

(1) 记录的更新

更新 Region 记录很简单，因为(假定主键不能更新)只有一列可以更新。本例使用的存储过程利用代码插入数据库，这就是如下所示的 RegionUpdate 过程。该存储过程定义为一个字符串资源，在 02_StoredProcs 项目的 Strings.resx 文件中：

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
                               @RegionDescription NCHAR(50)) AS
    SET NOCOUNT OFF
    UPDATE Region
        SET RegionDescription = @RegionDescription
        WHERE RegionID = @RegionID
GO
```

给实际表执行更新命令，需要重复选择和返回全部已更新的记录。这个存储过程接受两个输入参数(@RegionID 和@RegionDescription)，对数据库执行 UPDATE 语句。

要在.NET 代码中运行这个存储过程，需要定义一条 SQL 命令，并执行它：

```
SqlCommand cmd = new SqlCommand("RegionUpdate", conn);

cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue ( "@RegionID", 23 );
cmd.Parameters.AddWithValue ( "@RegionDescription", "Something" );
```

这段代码新建一个 SqlCommand 对象 aCommand，并把它定义为一个存储过程。然后，使用 AddWithValue()方法依次添加每个参数，这会构建一个参数，并设置其值，也可以手工构建 SqlParameter 实例，并根据需要把它们添加到 Parameters 集合中。

该存储过程接受两个参数：正在更新的 Region 记录的唯一主键；给这个记录的新描述。创建命令后，就可以用下面的命令执行它：

```
cmd.ExecuteNonQuery();
```

由于该过程没有返回值，因此使用 ExecuteNonQuery()方法就足够了。命令参数可以使用 AddWithValue()方法直接设置，也可以通过构建 SqlParameter 实例来设置。注意参数集合可以按位置或参数名来索引。

(2) 记录的删除

下一个存储过程可用于从数据库中删除一个 Region 记录：

```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
    SET NOCOUNT OFF
    DELETE FROM Region
    WHERE RegionID = @RegionID
GO
```

这个过程只需要该记录的主键值。下面的代码使用 SqlCommand 对象调用这个存储过程：

```
SqlCommand cmd = new SqlCommand("RegionDelete", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@RegionID", SqlDbType.Int, 0,
                                    "RegionID"));
cmd.UpdatedRowSource = UpdateRowSource.None;
```

这条命令只接收一个参数，如下面的代码所示，它执行 RegionDelete 存储过程，这是一个按照名称设置参数的示例。如果有许多对同一个存储过程的类似调用，就应构建 SqlParameter 实例，并设置其值，如下面的代码所示，其性能要比为每个调用重新构建整个 SqlCommand 更好：

```
cmd.Parameters["@RegionID"].Value= 999;
cmd.ExecuteNonQuery();
```

2. 调用返回输出参数的存储过程

前面两个执行存储过程的示例都没有返回值。如果存储过程包含输出参数，它们就需要在.NET 客户端中定义，以便在过程返回时填充其输出参数。下面的示例说明了如何在数据库中插入记录，

并把该记录的主键返回给调用者。

Region 表仅由主键(RegionID)和描述字段(RegionDescription)组成。要插入一个记录，必须生成该数字主键，再把新行插入到数据库中。在这个示例中，通过在存储过程中创建一个主键，简化了主键的生成。使用的方法未经过任何加工，这就是本章的后面用一节的篇幅介绍键的生成的原因。现在使用这个原始示例就足够了：

```
CREATE PROCEDURE RegionInsert(@RegionDescription NCHAR(50),
                              @RegionID INTEGER OUTPUT)AS
    SET NOCOUNT OFF
    SELECT @RegionID = MAX(RegionID)+ 1
    FROM Region
    INSERT INTO Region(RegionID, RegionDescription)
    VALUES(@RegionID, @RegionDescription)
GO
```

插入过程新建一个 Region 记录，在数据库本身生成主键值时，这个值作为输出参数从过程中返回(@RegionID)。这对于这个简单示例足够了，但对于比较复杂的表(特别是有默认值的表)，通常不使用输出参数，而选择插入的整行，把该行返回给调用者。.NET 类可以处理这两种情况。下面的代码说明了如何调用 RegionInsert 存储过程：

```
SqlCommand cmd = new SqlCommand("RegionInsert", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@RegionDescription",
                                    SqlDbType.NChar,
                                    50,
                                    "RegionDescription"));
cmd.Parameters.Add(new SqlParameter("@RegionID",
                                    SqlDbType.Int,
                                    0,
                                    ParameterDirection.Output,
                                    false,
                                    0,
                                    0,
                                    "RegionID",
                                    DataRowVersion.Default,
                                    null));
cmd.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

其中参数的定义比较复杂。第二个参数@RegionID 定义为包含其参数方向，在这个示例中是 Output。除这个标志之外，该示例还在最后一行使用 UpdateRowSource 枚举表示数据通过输出参数从这个存储过程中返回。当从一个 DataTable 中执行存储过程调用时，主要使用这个标志。

调用这个存储过程类似于前面的示例，但在这个实例中，需要在执行该过程后读取输出参数：

```
cmd.Parameters["@RegionDescription"].Value = "South West";
cmd.ExecuteNonQuery();
int newRegionID = (int) cmd.Parameters["@RegionID"].Value;
```

在执行该命令后，读取@RegionID 参数的值，并把它强制转换为整数。上述命令的一个缩写版本是 ExecuteScalar()方法，它返回存储过程返回的第一个值(返回为对象)。

如果调用的存储过程返回输出参数和一组记录行，该怎么办？此时，应定义合适的参数，不是

调用 ExecuteNonQuery()方法，而应调用另一个方法(如 ExecuteReader())，以遍历所有返回的记录。

32.4 快速数据访问：数据读取器

虽然数据读取器(data reader)是从数据源中选择某些数据的最简单快捷的方法，但这也是功能最弱的一个方法。不能直接实例化数据阅读器，即调用 ExecuteReader()方法后从相应数据库的命令对象(如 SqlCommand)中返回的实例。

下面的代码说明了如何从 Northwind 数据库的 Customer 表中选择数据。这个示例连接到数据库，选择许多记录，循环所选的记录，并把它们输出到控制台上。

这个示例使用 OLE DB 提供程序作为 SQL 提供程序中一个简化的数据暂存器。在大多数情况下，OleDbClient 类与 SqlClient 类是一对一的关系，例如，OleDbConnection 对象就类似于前面示例使用的 SqlConnection 对象。

要对 OLE DB 数据源执行命令，应使用 OleDbCommand 类。下面的代码执行一条简单的 SQL 语句，并读取记录，返回一个 OleDbDataReader 对象。注意下面的第二条 using 指令使 OleDb 类可用。

```
using System;
using System.Data.OleDb;
```

因为目前所利用的大部分数据提供程序都在同一个程序集中发布，所以只需要引用 System.Data.dll 程序集，就可以导入本节使用的所有类。

下面的代码包含本章讨论过的许多熟悉的 C#功能：

```
public class DataReaderExample
{
  public static void Main(string[] args)
  {
    string source = "Provider=SQLOLEDB;" +
                    "server=(local);" +
                    "integrated security=SSPI;" +
                    "database=northwind";
    string select = "SELECT ContactName,CompanyName FROM Customers";
    OleDbConnection conn = new OleDbConnection(source);
    conn.Open();
    OleDbCommand cmd = new OleDbCommand(select, conn);
    OleDbDataReader aReader = cmd.ExecuteReader();
    while(aReader.Read())
      Console.WriteLine("'{0}' from {1}",
                        aReader.GetString(0), aReader.GetString(1));
    aReader.Close();
    conn.Close();
  }
}
```

在前面的示例中，下面的代码根据源连接字符串，新建一个 OLE DB .NET 数据库连接：

```
OleDbConnection conn = new OleDbConnection(source);
conn.Open();
OleDbCommand cmd = new OleDbCommand(select, conn);
```

第 3 行根据特定的 SELECT 语句新建一个 OleDbCommand 对象，和执行命令时所使用的数据库连接。当有一条有效的命令时，就需要执行它，它返回一个初始化后的 OleDbDataReader：

```
OleDbDataReader aReader = cmd.ExecuteReader();
```

OleDbDataReader 是一个只向前的连接读取器，即只能沿着一个方向遍历记录，而使用的数据库连接一直打开，直到关闭该数据读取器为止。

OleDbDataReader 会使数据库连接一直处于打开状态，直到显式地关闭它为止。

OleDbDataReader 类不能直接实例化，它总是通过调用 OleDbCommand 类的 ExecuteReader()方法来返回。一旦打开一个数据读取器，就可以用各种方式访问包含在该读取器中的数据。

关闭 OleDbDataReader 对象(显式调用 Close()方法或通过垃圾收集器收集对象)时，底层的连接也会关闭，这取决于调用了哪个 ExecuteReader()方法。如果调用了 ExecuteReader()方法，并传递了 CommandBehavior.CloseConnection，就可以在关闭读取器时强制关闭连接。

OleDbDataReader 类有一个索引器，它可以使用常见的数组风格的语法访问任何字段(尽管不是类型安全的访问)：

```
object o = aReader[0];
```

或者

```
object o = aReader["CategoryID"];
```

假定 CategoryID 字段是 SELECT 语句中用于填充读取器的第一个字段，那么这两行语句在功能上等价，尽管后者比前者慢一些。为了验证这一点，编写一个简单的测试程序，从打开的数据读取器中对同一列进行 100 万次的迭代访问，仅为了获取一些足够大的数字来读取。虽然在一个死循环中可能并不会对同一列读取 100 万次，但按每(微)秒来计算，就可能编写出最佳的代码。

访问 DataReader 中的数据还有一种更好的方式：使用类型安全的 GetInt32、GetDouble 和其他类似的方法。在编写本书的第 1 版时，GetInt32 是在打开的数据读取器对象上读取整数的最快方法。在 6 核 AMD 盒中，其速度如表 32-6 所示：

表 32-6

访 问 方 法	100 万次迭代所需的时间
数字索引器-reader[0]	23ms
字符串索引器-reader["field"]	109ms
方法调用-reader. GetInt32	177ms

这些数字令人惊讶。在这段代码的以前版本(旧 Framework 版本)中，GetInt32 总是胜过了其他方法(约高出 10 倍)。

但在目前的版本中，Microsoft 肯定做了一些优化，现在 GetInt32 是最慢的方法了——主要是由于 x64 处理器带来的更好的 JIT 编译、功能内联和更佳优化。即使花大量的时间查看每种情况的 IL

代码，来确定出现这种情形的原因，也可能找不出原因。

下面的示例与上一示例基本相同，但在这个实例中分别用 SQL 提供程序和 SQL 类的引用替换了 OLE DB 提供程序和对 OLE DB 类的所有引用。该示例在本书网站的 04_DataReaderSql 项目下：

```
using System;
using System.Data.SqlClient;

public class DataReaderSql
{
  public static int Main(string[] args)
  {
    string source = "server=(local);" +
                    "integrated security=SSPI;" +
                    "database=northwind";
    string select = "SELECT ContactName,CompanyName FROM Customers";
    SqlConnection conn = new SqlConnection(source);
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    SqlDataReader aReader = cmd.ExecuteReader();
    while(aReader.Read())
      Console.WriteLine("'{0}' from {1}", aReader.GetString(0),
                        aReader.GetString(1));
    aReader.Close();
    conn.Close();
    return 0;
  }
}
```

注意一下区别是什么？如果正在输入这些代码，就用 sql 替换所有 OleDb，改变数据源字符串，并重新编译。这很容易。

对 SQL 提供程序的索引器进行相同的性能测试，这次 GetInt32 是最快的方法，结果如表 32-7 所示：

表 32-7

访 问 方 法	100 万次迭代所需的时间
数字索引器-reader[0]	59ms
字符串索引器-reader["field"]	153ms
方法调用-reader. GetInt32)	38ms

这说明，在使用 SqlDataReader 时，应使用类型安全的 GetXXX 方法，而使用 OleDbDataReader 时，应使用数字索引器。

32.5 异步数据访问：使用 Task 和 await

前面学习了访问数据读取器中各个部件的最高效方法，现在应讨论数据访问的另一个方面——异步访问。

访问数据时，我们几乎总是没有处理该数据，大多数情况下，我们都在访问另一台机器上的数

据，所以应限制在这些机器之间传递的数据量。另外，为了提供系统正在响应的假象，也可能需要异步访问数据。

异步请求数据的主要方式是使用 SqlCommand(或 OleDbCommand)类，因为这些类包含了使用 APM(Asynchronous Programming Model)的方法，APM 提供了 BeginExecuteReader 和 EndExecuteReader 方法，并使用 IAsyncResult 接口。这些方法自从.NET 1.0 版本以来就可以使用，但在.NET 4.0 中，Microsoft 添加了 Task 类，更新了许多内置类的 API。使用 Task 类异步访问数据比以前要容易得多。

要使用 Task 类访问数据，一般应编写下面的示例函数：

```
public static Task<int> GetEmployeeCount()
{
  using(SqlConnection conn = new SqlConnection(GetDatabaseConnection()))
  {
    SqlCommand cmd = new SqlCommand("WAITFOR DELAY '0:0:02';select count(*) from
      employees", conn);
    conn.Open();

    return cmd.ExecuteScalarAsync().ContinueWith(t => Convert.ToInt32(t.Result));
  }
}
```

这些代码创建了一个 Task 对象，它可以等待调用者调用，例如，可以构建从不同的表中读取数据的任务，并把它们执行为独立的任务。这里的语法初看起来有点奇怪，但它展示了 Task 类的一些功能。创建运行很慢的 SqlCommand 后(在 SQL 代码中延迟 2 秒)，就使用 ExecuteScalarAsync 调用这个命令。这会返回一个对象，所以使用 ContinueWith 把第一个任务的返回值转换为整数。于是，代码现在包含两个任务：第一个任务选择对象，第二个任务把该对象转换为整数。

这种模式初看起来有点古怪，但在使用 ExecuteScalarAsync 方法返回 SqlDataReader 时，它就显出其威力了。因为在继续执行的任务中，可以把它转换为一个对象实例列表，这些对象实例从数据读取器返回的数据中构建。

异步任务的一个常见用法是分叉和连接，即先把流分叉为一组异步任务，再在所有任务的末尾把它们连接起来。在.NET 中，这是使用 Task 类实现的。可以把调用分叉为几个方法，如前面的示例所示，这些方法返回任务，然后调用 Task.WaitAll，并将任务集传递给该方法，把结果连接起来。本节的示例代码在本书网站的 06_AsyncDataReaders 项目中。

```
var t1 = GetEmployeeCount();
var t2 = GetOrderCount();

Task.WaitAll(t1, t2);
```

在.NET 4.5 版本的 C#中添加了 async 和 await 关键字，它们可以用于简化任务的异步执行。在函数声明中添加 async 修饰符，给代码添加 await，就可以更新前面的示例，如下所示：

```
public async static Task<int> GetEmployeeCount()
{
```

```
    using (SqlConnection conn = new SqlConnection(GetDatabaseConnection()))
    {
        SqlCommand cmd = new SqlCommand("WAITFOR DELAY '0:0:02';select count(*) from
          employees", conn);
        conn.Open();

        return await cmd.ExecuteScalarAsync().ContinueWith(t => Convert.ToInt32(t.Result));
    }
}
```

在调用代码中，现在可以编写如下代码，来调用 async 方法了：

```
public async static Task GetEmployeesAndOrders()
{
    int employees = await GetEmployeeCount();
    int orders = await GetOrderCount();

    Console.WriteLine("Number of employes: {0}, Number of orders: {1}", employees, orders);
}
```

必须在方法声明中添加 async 关键字，指出这是一个异步方法(它使用了 await)。接着，就可以使用 await 调用其他异步方法了，代码类似于调用简单的方法，但编译器会自动构建所有异步协调的代码。

注意在前面的示例中，两个 await 调用会有效地交替运行两个任务，所以如果希望真正异步调用这些方法，就需要降低一级，直接使用 Task 类。

32.6 管理数据和关系：DataSet 类

DataSet 类是数据的脱机容器。它不包含数据库连接的概念，实际上存储在 DataSet 类中的数据不一定来源于数据库，它可以是来自 CSV 文件、XML 文件的记录，或是从测量设备中读取的点。

DataSet 类由一组数据表组成，每个表都有一组数据列和数据行，如图 32-4 所示。除了定义数据外，还可以在 DataSet 类中定义表之间的链接。例如，我们常常要定义父/子关系(通常也称为主/从关系)。表中的一个记录(即 Order)链接到另一个表的许多记录上(即 Order_Details)，这种关系可以在 DataSet 类中定义和导航。

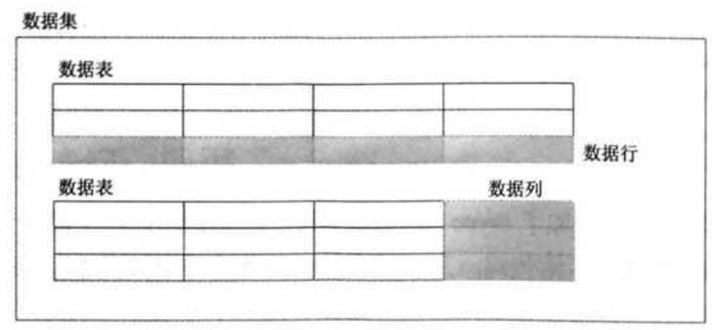


图 32-4

重点是记住，DataSet 类基本上是内存中的数据库，其中包含了所有表、关系和约束。下一节描述和 DataSet 一起使用的类。

DataSet 和相关类基本上被 Entity Framework 代替，这里介绍它们仅作为背景知识。

32.6.1 数据表

数据表非常类似于物理数据库表，它由一组包含特定属性的列组成，可能包含 0 行或多行数据。数据表也可以定义主键(它可以是一列或多列)，列上也可以包含约束。这些信息对应的通用术语在本章的其他部分称为“架构”。

为数据表定义架构有几种方式(实际上把 DataSet 类当作一个整体)，这些在介绍了数据列和数据行后讨论。图 32-5 显示了一些可通过数据表访问的对象。

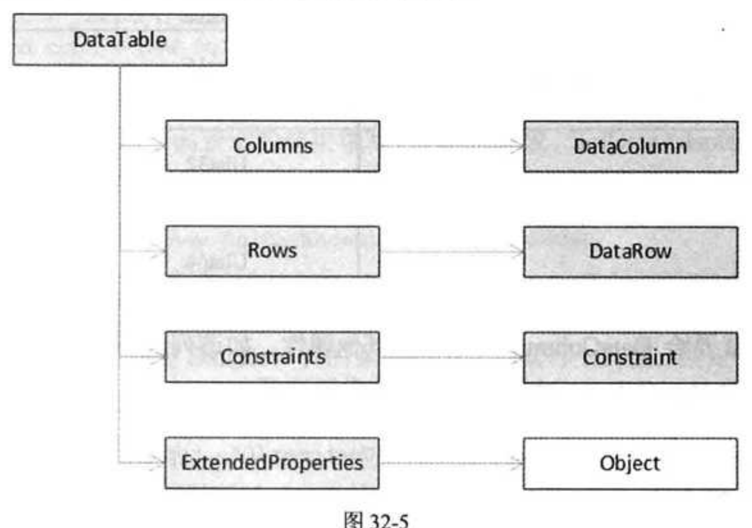


图 32-5

DataTable 对象(和 DataColumn)可以附带任意多个扩展属性。这个集合可以用与对象相关的用户自定义信息来填充。例如，某一列有一个输入掩码，用于验证列的内容是否有效，比较常见的示例是美国的社会安全号。当数据在中间层中构造，要返回给客户端，进行某些处理时，最适合使用扩展的属性。例如，可以在扩展的属性中存储数字列的有效性标准(如 min 和 max)，在验证用户输入时在 UI 层使用它。

填充数据表时，可以从数据库中选择数据，从文件中读取数据，或在代码中手工填充，Rows 集合会包含这些检索出来的数据。

Columns 集合包含已经添加到表中的 DataColumn 实例，它们定义了数据的架构，如数据类型、是否可为空和默认值等。Constraints 集合可以用唯一约束或主键约束来填充。

数据表使用架构信息的一个示例是在 DataGrid 中显示数据时。DataGrid 控件使用属性(如列的数据类型)来确定该列应使用什么控件。数据库中的 bit 字段在 DataGrid 中显示为一个复选框。如果列在数据库架构中定义为 NOT NULL，该信息就存储在 DataColumn 中，以便在用户试图移出数据行时测试该列。

32.6.2 数据列

DataColumn 对象定义了 DataTable 中某列的属性，如该列的数据类型，该列是否为只读，以及其他属性。可以在代码中创建列，或者在运行期间自动生成列。

在创建列时，给它指定名称也很有用；否则运行库就会为该列生成一个名称，其格式是 Column*n*，其中 *n* 是一个递增的数字。

列的数据类型可以在构造函数中提供，也可以通过设置 DataType 属性来指定。一旦把数据加载到数据表中，就不能改变列的数据类型，否则会抛出一个 ArgumentException 异常。

创建的数据列可以包含表 32-8 所示的.NET Framework 数据类型。

表 32-8

Boolean	Decimal
Int64	TimeSpan
Byte	Double
Sbyte	UInt16
Char	Int16
Single	UInt32
DateTime	Int32
String	UInt64

一旦创建它，就要给 DataColumn 对象设置其他属性，如该列是否可为空或者设置默认值。下面的代码段显示了给 DataColumn 对象设置的一些常见选项：

```
DataColumn customerID = new DataColumn("CustomerID", typeof(int));
customerID.AllowDBNull = false;
customerID.ReadOnly = false;
customerID.AutoIncrement = true;
customerID.AutoIncrementSeed = 1000;
DataColumn name = new DataColumn("Name", typeof(string));
name.AllowDBNull = false;
name.Unique = true;
```

可以给 DataColumn 对象设置如表 32-9 所示的属性。

表 32-9

属　性	说　明
AllowDBNull	如果为 true，该列就可以设置为 DBNull
AutoIncrement	指定该列的值自动生成为一个递增的数字
AutoIncrementSeed	定义 AutoIncrement 列最初的种子值
AutoIncrementStep	定义自动生成的列值之间的递增量，默认值为 1
Caption	可以用于在屏幕上显示列名
ColumnMapping	指定当 DataSet 类通过调用 DataSet.WriteXml 来保存时，列如何映射到 XML 上

(续表)

属　性	说　明
ColumnName	列名，如果没有在构造函数中设置，就由运行库自动生成列名
DataType	定义列的 System.Type 值
DefaultValue	可以定义列的默认值
Expression	定义在所计算的列中使用的表达式

1. 数据行

这个类构成了 DataTable 类的另一部分。数据表中的列根据 DataTable 类来定义，表中的实际数据用 DataRow 对象来访问。下面的示例说明了如何访问数据表中的行。首先是连接的详细信息：

```
string source = "server=(local);" +
                " integrated security=SSPI;" +
                "database=northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
```

下面的代码引入了 SqlDataAdapter 类，它用于把数据置入 DataSet 中。SqlDataAdapter 类使用 SQL 子句，在 DataSet 类中用下面查询的结果填写 Customers 表。SqlDataAdapter 类将在 32.7 节中进一步讨论。

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

在下面的代码中，注意使用 DataRow 类的索引器访问数据行上的值。给定列的值可以用几个重载的索引器来检索，这样就可以通过已知的列号、列名或 DataColumn 来检索数据的值：

```
foreach(DataRow row in ds.Tables["Customers"].Rows)
   Console.WriteLine("'{0}' from {1}", row[0],row[1]);
```

DataRow 类最吸引人的一个方面就是它的版本功能。DataRow 类可以接收某一特定行上指定列的各个值，其版本见表 32-10。

表 32-10

DataRow 类的 Version 值	说　明
Current	列中目前存在的值，如果没有进行编辑，该值与初值相同。如果进行了编辑，该值就是最后输入的有效值
Default	默认值(换言之，列的任何默认设置)
Original	最初从数据库中选择出来的列值。如果调用 DataRow 类的 AcceptChanges 方法，该值就更新为 Current 值
Proposed	对列进行修改时，可以检索到这个已改变的值。如果在行上调用 BeginEdit 方法，并进行修改，每一列都会有一个推荐值，直到调用 EndEdit 或 CancelEdit 方法为止

可以以许多方式使用给定列的版本。例如，在数据库中更新数据行时，常常使用如下 SQL 语句：

```
UPDATE Products
SET    Name = Column.Current
WHERE  ProductID = xxx
AND    Name = Column.Original;
```

显然，这段代码永远不会编译，但它说明了某一行中某一列的初值和当前值的一种用法。

要从 DataRow 类的索引器中检索某个版本的值，应使用其中一个索引器方法，它把 DataRowVersion 值作为一个参数。下面的代码段说明了如何获得 DataTable 对象中每一列的所有值：

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
  foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
  {
    Console.WriteLine ("{0} Current = {1}", dc.ColumnName,
                                         row[dc,DataRowVersion.Current]);
    Console.WriteLine (" Default = {0}", row[dc,DataRowVersion.Default]);
    Console.WriteLine (" Original = {0}",
                     row[dc,DataRowVersion.Original]);
  }
}
```

整行有一个状态标志 RowState，它可以用于确定在持久化到数据库时需要对该行进行什么操作。把 RowState 属性设置为跟踪对 DataTable 所做的所有改变，如添加新行、删除现有行，以及改变表中的列。当数据与数据库同步时，行的状态标志用于确定应执行什么 SQL 操作。这些标志由 DataRowState 枚举定义，如表 32-11 所示。

表 32-11

DataRowState 值	说 明
Added	指出把新数据行添加到 DataTable 的 Rows 集合中。在客户端上创建的所有行都设置为这个值，在与数据库同步时，最终会使用 SQL 的 INSERT 语句
Deleted	指出通过 DataRow.Delete()方法把 DataTable 中的数据行标记为删除。该行仍存在于 DataTable 中，但在屏幕上通常看不到它(除非显式设置 DataView)。在 DataTable 中标记为已删除的数据行将在与数据库同步时从数据库中删除
Detached	指出某一行在创建后立即显示为这个状态，调用 DataRow.Remove()方法也可以返回这个状态。分离的行不是任何数据表的一部分，因此处于这种状态的行不能使用任何 SQL 语句
Modified	如果任何列中的值发生了改变，数据行就处于这个状态
Unchanged	指出自从最后一次调用 AcceptChanges()方法以来，该行都没有发生改变

行的状态也取决于在其上调用的方法。一般在成功更新数据源(即把改变持久化到数据库后)之后调用 AcceptChanges()方法。

修改 DataRow 中的数据最常见的方式是使用索引器，但如果对数据进行了许多修改，就需要考虑使用 BeginEdit()和 EndEdit()方法。

在对 DataRow 中的列进行了修改后，就会在该行的 DataTable 上引发 ColumnChanging 事件。该事件可以重写 DataColumnChangeEventArgs 类的 ProposedValue 属性，按照需要修改它。这是在列值上进行某些数据有效性验证的一种方式。如果在进行修改前调用 BeginEdit()方法，就不会引发 ColumnChanging 事件，于是可以进行多次修改，再调用 EndEdit()方法，持久化这些修改。如果要回退到初值，就应调用 CancelEdit()方法。

DataRow 可以以某种方式链接到其他数据行上，在数据行之间能够建立可导航的链接，这在主/从数据表中非常常见。DataRow 包含 GetChildRows()方法，该方法可以从同一个 DataSet 的另一个表中把一组相关行返回为当前行。这些将在 32.5.2 节中介绍。

2. 架构的生成

为 DataTable 创建架构有 3 种方式：

- 让运行库来完成
- 编写代码来创建表
- 使用 XML 架构生成器

下面介绍这 3 种方式。

(1) 运行库生成的架构

前面的 DataRow 示例用下面的代码从数据库中选择数据，并填充一个 DataSet 类：

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

这很容易使用，但它也有几个缺点。例如，必须利用默认的列名来处理——这是可行的，但在某些情况下，还要把物理数据库的列(如 PKID)重命名为一个对用户更友好的名称。为此，可以在 SQL 子句中给列指定别名，如在 SELECT PID AS PersonID FROM Person Table 中。但最好不要在 SQL 中重命名列，因为列实际上只需要在屏幕上显示一个“好”的名称即可。

自动生成 DataTable/DataColumn 的另一个潜在问题是不能控制运行库为列选择的数据类型。运行库可以确定正确的数据类型，但有时需要对此有更多的控制。例如，为给定的列定义枚举类型，以简化类的用户代码。如果接受运行库生成的默认列类型，该列就可能是一个 32 位的整数，而不是有预定义选项的枚举。

最后，也是最有可能出的问题是，在使用自动生成的表时，不能对 DataTable 中的数据进行类型安全的访问——索引器就会返回 object 的实例，而不是派生的数据类型。如果要用类型强制转换表达式对代码进行修改，就可以跳过下面的章节。

(2) 手工编码的架构

用生成的代码来创建 DataTable，再用相关联的 DataColumns 来填充相当简单。本节的示例将访问 Northwind 数据库中的 Product 表，如图 32-6 所示。

Products

Column Name	Data Type	Allow Nulls
ProductID	int	☐
ProductName	nvarchar(40)	☐
SupplierID	int	☑
CategoryID	int	☑
QuantityPerUnit	nvarchar(20)	☑
UnitPrice	money	☑
UnitsInStock	smallint	☑
UnitsOnOrder	smallint	☑
ReorderLevel	smallint	☑
Discontinued	bit	☐
		☐

图 32-6

下面的代码生成一个 DataTable，它对应于图 32-6 的架构(但没有包含可空的列)：

```
public static void ManufactureProductDataTable(DataSet ds)
{
   DataTable products = new DataTable("Products");
   products.Columns.Add(new DataColumn("ProductID", typeof(int)));
   products.Columns.Add(new DataColumn("ProductName", typeof(string)));
   products.Columns.Add(new DataColumn("SupplierID", typeof(int)));
   products.Columns.Add(new DataColumn("CategoryID", typeof(int)));
   products.Columns.Add(new DataColumn("QuantityPerUnit", typeof(string)));
   products.Columns.Add(new DataColumn("UnitPrice", typeof(decimal)));
   products.Columns.Add(new DataColumn("UnitsInStock", typeof(short)));
   products.Columns.Add(new DataColumn("UnitsOnOrder", typeof(short)));
   products.Columns.Add(new DataColumn("ReorderLevel", typeof(short)));
   products.Columns.Add(new DataColumn("Discontinued", typeof(bool)));
   ds.Tables.Add(products);
}
```

可以改变 DataRow 示例中的代码，使用如下新生成的表定义：

```
string source = "server=(local);" +
                "integrated security=sspi;" +
                "database=Northwind";
string select = "SELECT * FROM Products";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter cmd = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
ManufactureProductDataTable(ds);
cmd.Fill(ds, "Products");
foreach(DataRow row in ds.Tables["Products"].Rows)
   Console.WriteLine("'{0}' from {1}", row[0], row[1]);
```

ManufactureProductDataTable()方法新建一个 DataTable，依次添加每一列，最后把这个表追加到 DataSet 中表的清单上。DataSet 有一个索引器，它的参数是表名，给调用者返回该 DataTable。

上面的示例仍不是类型安全的，因为在列上使用了索引器来检索数据。最好是有一个类(或一组类)派生自 DataSet、DataTable 和 DataRow，为表、行和列定义类型安全的存取器。可以自己生成这段代码，这并不是特别乏味，最终将得到可以进行数据类型安全访问的类。

如果不愿意自己生成这些类型安全的类，就可以使用帮助。.NET Framework 对本节开头列出的第 3 种方法提供了支持：允许使用 XML 架构来定义 DataSet、DataTables 和本节介绍的其他类。这种方法详见 32.7 节。

32.6.3　数据关系

在编写应用程序时，常常需要获取和缓存各种信息表。DataSet 类是这些信息的容器，使用一般的 OLE DB，需要提供一种奇怪的 SQL 方言来强制分层的数据关系，提供程序本身不能没有它自己的“怪癖”。

另一方面，DataSet 类从一开始就用于建立数据表之间的关系。本节的代码说明了如何手工生成并填充两个数据表。如果不能访问 SQL Server 或 NorthWind 数据库，就可以自由地运行这个示例。

```
DataSet ds = new DataSet("Relationships");
ds.Tables.Add(CreateBuildingTable());
ds.Tables.Add(CreateRoomTable());
ds.Relations.Add("Rooms",
                 ds.Tables["Building"].Columns["BuildingID"],
                 ds.Tables["Room"].Columns["BuildingID"]);
```

本示例使用的表如图 32-7 所示。这两个表仅包含一个主键和名称字段，Room 表有一个 BuildingID 外键。

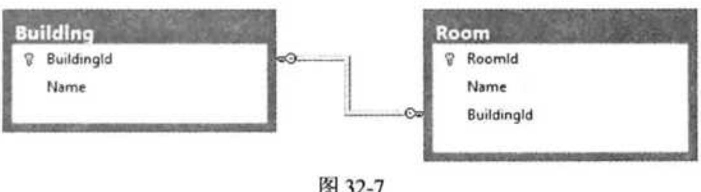


图 32-7

这些表都非常简单，下面的代码说明了如何迭代 Buildings 表中的行，并遍历对应的关系，以列出 Rooms 表中的所有子行。

```
foreach(DataRow theBuilding in ds.Tables["Building"].Rows)
{
  DataRow[] children = theBuilding.GetChildRows("Rooms");
  int roomCount = children.Length;
  Console.WriteLine("Building {0} contains {1} room{2}",
                    theBuilding["Name"],
                    roomCount,
                    roomCount> 1 ? "s": "");
  // Loop through the rooms
  foreach(DataRow theRoom in children)
     Console.WriteLine("Room: {0}", theRoom["Name"]);
}
```

DataSet 类和其他分层的旧 recordset 对象之间的主要区别是关系显示的方式。在分层的 Recordset 对象中，关系显示为行中的一个伪列，该列本身是一个可以迭代的 Recordset 对象。但在 ADO.NET 中，通过调用 GetChildRows()方法就可以遍历关系。

```
DataRow[] children = theBuilding.GetChildRows("Rooms");
```

该方法有许多形式，但上面的示例只使用关系的名称在父子行之间来回遍历。它返回一个行数组，使用前面示例中的索引器就可以更新这些行。

数据关系更有趣的地方是可以用两种方式遍历这些数据。在 DataTable 类上使用 ParentRelations 属性，不仅可以从父数据行中找到子数据行，还可以从子记录中找到父数据行。这个属性返回一个 DataRelationCollection，该集合可以使用[]数组语法来索引(例如，DataRelations["Rooms"])，另外，GetChildRows()方法也可以如下所示进行调用：

```
foreach(DataRow theRoom in ds.Tables["Room"].Rows)
{
  DataRow[] parents = theRoom.GetParentRows("Rooms");
  foreach(DataRow theBuilding in parents)
    Console.WriteLine("Room {0} is contained in building {1}",
                      theRoom["Name"],
                      theBuilding["Name"]);
}
```

GetParentRows()方法(返回 0 行或多行数据对应的一个数组)或 GetParentRow()方法(根据给定的某种关系检索一个父行)都有许多重写版本，可以检索出父行。

32.6.4 数据约束

DataTable 类不仅仅擅长于改变在客户端上创建的列的数据类型。ADO.NET 允许在列上创建一组约束，对数据应用一些规则。

运行库目前支持表 32-12 所示的约束类型，它们包含在 System.Data 名称空间的类中。

表 32-12

约　束	说　明
ForeignKeyConstraint	在 DataSet 的两个 DataTable 之间强制链接
UniqueConstraint	确保给定列中的项是唯一的

1. 设置主键

在关系数据库的表中，可以提供一个主键，该主键可以基于 DataTable 中的一列或多列。

下面的代码为 Product 表创建了一个主键，其架构是前面手动构建的。表上的主键只是约束的一种形式。当把主键添加到 DataTable 中时，运行库也会对键列生成一个唯一约束。这是因为实际上并没有 PrimaryKey 约束类型，主键是一列或多列上的唯一约束。

```
public static void ManufacturePrimaryKey(DataTable dt)
{
  DataColumn[] pk = new DataColumn[1];
  pk[0] = dt.Columns["ProductID"];
  dt.PrimaryKey = pk;
}
```

因为主键可以包含几列，所以它可以作为一个 DataColumn 数组输入。通过给表的一个列数组指定属性，就可以给这些列设置主键。

要检查表中的约束，可以迭代 ConstraintsCollection。上述代码自动生成的约束是 Constraint1，这个名称没有什么用，因此应避免这种情况，最好先在代码中创建约束，然后定义组成主键的列。

创建主键之前，下面的代码给约束命名：

```
DataColumn[] pk = new DataColumn[1];
pk[0] = dt.Columns["ProductID"];
dt.Constraints.Add(new UniqueConstraint("PK_Products", pk[0]));
dt.PrimaryKey = pk;
```

唯一约束可以应用到任意多列上。

2. 设置外键

除了唯一约束外，DataTable 类还可以包含外键约束，它们主要用于强制主/从关系，如果正确地建立了约束，外键约束还可用于在表之间复制列。在主/从关系的表中，常常有一个父记录(订单)和许多子记录(订单行)，它们通过父记录的主键链接起来。

因为外键约束只能作用于同一个 DataSet 中的表，所以下面的示例使用 Northwind 数据库中的 Categories 表，给该表和 Products 表之间指定约束，如图 32-8 所示。

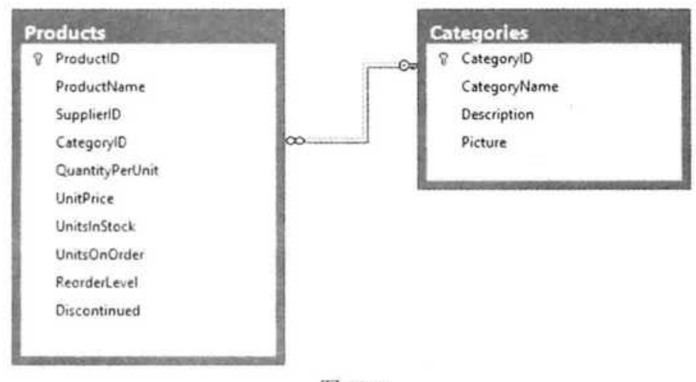


图 32-8

第一步是为 Categories 表生成一个新的数据表：

```
DataTable categories = new DataTable("Categories");
categories.Columns.Add(new DataColumn("CategoryID", typeof(int)));
categories.Columns.Add(new DataColumn("CategoryName", typeof(string)));
categories.Columns.Add(new DataColumn("Description", typeof(string)));
categories.Constraints.Add(new UniqueConstraint("PK_Categories",
                               categories.Columns["CategoryID"]));
categories.PrimaryKey = new DataColumn[1]
                              {categories.Columns["CategoryID"]};
```

上述代码的最后一行给 Categories 表创建主键。在本例中，主键是一个单列，但可以使用数组语法在多个列上生成一个键。

然后，需要在两个表之间创建约束：

```
DataColumn parent = ds.Tables["Categories"].Columns["CategoryID"];
DataColumn child = ds.Tables["Products"].Columns["CategoryID"];
ForeignKeyConstraint fk =
   new ForeignKeyConstraint("FK_Product_CategoryID", parent, child);
```

```
fk.UpdateRule = Rule.Cascade;
fk.DeleteRule = Rule.SetNull;
ds.Tables["Products"].Constraints.Add(fk);
```

这个约束应用到 Categories.CategoryID 和 Products.CategoryID 之间的链接上。有 4 个不同的 ForeignKeyConstraint，但应使用可以给约束命名的 ForeignKeyConstraint。

3. 设置更新和删除约束

除了在父表和子表之间定义约束之外，还可以在更新约束中的一列时定义应执行的操作。

上面的示例设置了更新规则和删除规则，在对父表中的列(或行)执行某种操作时，使用这些规则，这些规则用来确定应对影响到的子表中的行进行什么操作。通过 Rule 枚举可以应用 4 种不同的规则：

- Cascade——如果更新了父键，就应把新的键值复制到所有子记录中。如果删除了父记录，则也将删除子记录，这是默认选项。
- None——不执行任何操作，这个选项会留下子数据表中的孤立行。
- SetDefault——如果定义了一个外键列，那么每个受影响的子记录都把外键列设置为其默认值。
- SetNull——所有子行都把键列设置为 DBNull(按照 Microsoft 使用的命名约定，键列实际上应是 SetDBNull)。

只有 DataSet 类的 EnforceConstraints 属性为 true，才能在 DataSet 类中强制约束。

本节介绍了组成 DataSet 类中约束部分的主类，揭示了如何在代码中手工生成这些类。还可以使用.NET 附带的 XML 架构文件和 XSD 工具定义 DataTable、DataRow、DataColumn、DataRelation 和 Constraint。下一节将说明如何建立一个简单的架构，以及如何生成类型安全的类，以访问数据。

32.7 XML 架构：用 XSD 生成代码

XML 已经在 ADO.NET 中确立了牢固的地位——实际上，现在在对象之间远程传递数据的格式是 XML。有了.NET 运行库，就可以在 XML 架构定义文件(XSD)中描述 DataTable。而且，可以定义整个 DataSet 类，其中有许多 DataTable 类，这些表之间有一组关系，并可以包括全面描述数据的各种其他信息。

在定义 XSD 文件时，运行库中有一个工具可以把这个架构转换为对应的数据访问类，如类型安全的 DataTable 类，如上所述。本节先介绍一个简单的 XSD 文件(Products.xsd)，该文件描述与前面 Products 示例相同的信息，再扩展它，使其包括一些额外的功能。

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="Products" targetNamespace="http://tempuri.org/XMLSchema1.xsd"
  xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
   <xs:element name="Product">
```

```
        <xs:complexType>
          <xs:sequence>
            <xs:element name="ProductID" msdata:ReadOnly="true"
              msdata:AutoIncrement="true" type="xs:int" />
            <xs:element name="ProductName" type="xs:string" />
            <xs:element name="SupplierID" type="xs:int" minOccurs="0" />
            <xs:element name="CategoryID" type="xs:int" minOccurs="0" />
            <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0" />
            <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
            <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" />
            <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" />
            <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" />
            <xs:element name="Discontinued" type="xs:boolean" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
```

第 34 章将详细论述其中的一些选项。现在应知道，这个文件基本上定义一个架构，其中把 id 属性设置为 Products。还定义了一个比较复杂的 Product 类，其中包含了许多元素，每个元素对应于 Products 表中的一个字段。

这些元素映射到如下数据类上：

- Products 架构映射到派生自 DataSet 类的一个类上。
- Product 复杂类型映射到派生自 DataTable 类的一个类上。
- 每个子元素映射到派生自 DataColumn 的一个类上。
- 所有列的集合映射到派生自 DataRow 类的一个类上。

.NET Framework 中有一个工具，只要输入 XSD 文件，该工具就可以生成这些类的代码。因为该工具唯一的工作是执行 XSD 文件上的各种功能，所以它称为 XSD.EXE。

假定把上面的文件另存为 Product.xsd，在命令提示符上输入下述命令，把该文件转换为代码：

```
xsd Product.xsd /d
```

这会创建文件 Product.cs。

有许多选项可以和 XSD 一起使用，以改变生成的输出结果，其中一些比较常用的选项如表 32-13 所示。

表 32-13

选　项	说　明
/dataset (/d)	启用派生自 DataSet、DataTable 和 DataRow 的类
/language:<language>	允许选择编写输出文件的语言。C#是默认语言，也可以选择用 VB 编写 Visual Basic .NET 文件
/namespace:<namespace>	允许定义生成代码应驻留其中的名称空间，默认为没有名称空间

下面节选了 Products 架构在 XSD 中的输出结果，并做了略微的修改，以满足本书的相应格式。要查看完整的输出结果，可以在 Products 架构(或者自己建立的某个架构)上运行 XSD.EXE，查看生成的.cs 文件。该示例包含所有源代码和 Products.xsd 文件。

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.5456
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

//
// This source code was auto-generated by xsd, Version=2.0.50727.3238.
//


/// <summary>
///Represents a strongly typed in-memory cache of data.
///</summary>
[global::System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.
TypedDataSetGenerator", "2.0.0.0")]
[global::System.Serializable()]
[global::System.ComponentModel.DesignerCategoryAttribute("code")]
[global::System.ComponentModel.ToolboxItem(true)]
[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema")]
[global::System.Xml.Serialization.XmlRootAttribute("Products")]
[global::System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet")]
public partial class Products : global::System.Data.DataSet {

    private ProductDataTable tableProduct;

    private global::System.Data.SchemaSerializationMode _schemaSerializationMode =
global::System.Data.SchemaSerializationMode.IncludeSchema;

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public Products() {
        this.BeginInit();
        this.InitClass();
        global::System.ComponentModel.CollectionChangeEventHandler
schemaChangedHandler = new
global::System.ComponentModel.CollectionChangeEventHandler(this.SchemaChanged);
        base.Tables.CollectionChanged += schemaChangedHandler;
        base.Relations.CollectionChanged += schemaChangedHandler;
        this.EndInit();
    }
```

为了集中论述公共接口，这段代码删除了所有受保护成员和私有成员。ProductDataTable 和 ProductRow 定义显示了两个嵌套类的位置，后面会实现它们。下面简单地解释一下派生自 DataSet 的类，之后讨论这些类的代码。

Products()构造函数调用一个私有方法 InitClass()，该方法构造派生自 DataTable 类的 ProductDataTable 类的一个实例，并把该表添加到 DataSet 类的 Tables 集合中。Products 数据表可以通过下面的代码来访问：

```
DataSet ds = new Products();
```

```
DataTable products = ds.Tables["Products"];
```

或者，更简单的方式是使用 Product 属性来访问，该属性在派生的 DataSet 对象上可用：

```
DataTable products = ds.Product;
```

因为 Product 属性是强类型化的，所以可以使用 ProductDataTable，而不是上述代码中的 DataTable 引用。

ProductDataTable 类包含更多的代码(注意这是一段节选的代码)：

```
[global::System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.
TypedDataSetGenerator", "2.0.0.0")]
[global::System.Serializable()]
[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")]
public partial class ProductDataTable : global::System.Data.DataTable, global::System.
Collections.IEnumerable {

    private global::System.Data.DataColumn columnProductID;

    private global::System.Data.DataColumn columnProductName;

    private global::System.Data.DataColumn columnSupplierID;

    private global::System.Data.DataColumn columnCategoryID;

    private global::System.Data.DataColumn columnQuantityPerUnit;

    private global::System.Data.DataColumn columnUnitPrice;

    private global::System.Data.DataColumn columnUnitsInStock;

    private global::System.Data.DataColumn columnUnitsOnOrder;

    private global::System.Data.DataColumn columnReorderLevel;

    private global::System.Data.DataColumn columnDiscontinued;

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public ProductDataTable() {
        this.TableName = "Product";
        this.BeginInit();
        this.InitClass();
        this.EndInit();
    }
```

ProductDataTable 类派生自 DataTable 类，并实现 IEnumerable 接口，为表中的每一列定义了一个私有的 DataColumn 实例，通过调用私有的 InitClass 成员，再次从构造函数中初始化这些实例。DataRow 类将使用每一列(后面介绍)。

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.ComponentModel.Browsable(false)]
public int Count {
    get {
        return this.Rows.Count;
```

```
        }
    }

    // Other row accessors removed for clarity — there is one for each column
```

给表添加数据行由AddProductRow()方法的两个重载版本实现(虽然它们的内容完全不同，但名称相同)。第一个重载方法接受一个已经构造出来的DataRow，且没有返回值。另一个重载方法则接受一组参数值，每个参数对应于DataTable中的一列，该重载方法新构造一行，设置该新行中的值，把该行添加到DataTable对象中，并给调用者返回该行。这些迥然不同的函数不应有相同的名称。

```
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public ProductRow AddProductRow(string ProductName, int SupplierID, int CategoryID, string
    QuantityPerUnit, decimal UnitPrice, short UnitsInStock, short UnitsOnOrder, short ReorderLevel,
    bool Discontinued) {
        ProductRow rowProductRow = ((ProductRow)(this.NewRow()));
        object[] columnValuesArray = new object[] {
                null,
                ProductName,
                SupplierID,
                CategoryID,
                QuantityPerUnit,
                UnitPrice,
                UnitsInStock,
                UnitsOnOrder,
                ReorderLevel,
                Discontinued};
        rowProductRow.ItemArray = columnValuesArray;
        this.Rows.Add(rowProductRow);
        return rowProductRow;
    }
```

派生自DataSet的类中的InitClass成员把表添加到DataSet类中，与此相同，ProductDataTable类中的InitClass成员把列添加到DataTable类中。每一列的属性都按需要设置，再把该列追加到列集合的尾部。

```
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    private void InitClass() {
       this.columnProductID = new global::System.Data.DataColumn("ProductID", typeof(int),
null,
    global::System.Data.MappingType.Element);
       base.Columns.Add(this.columnProductID);
       this.columnProductName = new global::System.Data.DataColumn("ProductName",
typeof(string),
    null, global::System.Data.MappingType.Element);
       base.Columns.Add(this.columnProductName);
       this.columnSupplierID = new global::System.Data.DataColumn("SupplierID", typeof(int),
null,
    global::System.Data.MappingType.Element);
       base.Columns.Add(this.columnSupplierID);
       this.columnCategoryID = new global::System.Data.DataColumn("CategoryID", typeof(int),
null,
    global::System.Data.MappingType.Element);
       base.Columns.Add(this.columnCategoryID);
```

```
        this.columnQuantityPerUnit = new global::System.Data.DataColumn("QuantityPerUnit",
    typeof(string), null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnQuantityPerUnit);
        this.columnUnitPrice = new global::System.Data.DataColumn("UnitPrice",
typeof(decimal),
    null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnUnitPrice);
        this.columnUnitsInStock = new global::System.Data.DataColumn("UnitsInStock",
typeof(short),
    null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnUnitsInStock);
        this.columnUnitsOnOrder = new global::System.Data.DataColumn("UnitsOnOrder",
typeof(short),
    null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnUnitsOnOrder);
        this.columnReorderLevel = new global::System.Data.DataColumn("ReorderLevel",
typeof(short),
    null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnReorderLevel);
        this.columnDiscontinued = new global::System.Data.DataColumn("Discontinued",
typeof(bool),
    null, global::System.Data.MappingType.Element);
        base.Columns.Add(this.columnDiscontinued);
        this.columnProductID.AutoIncrement = true;
        this.columnProductID.AllowDBNull = false;
        this.columnProductID.ReadOnly = true;
        this.columnProductName.AllowDBNull = false;
        this.columnDiscontinued.AllowDBNull = false;
    }
```

NewRowFromBuilder()方法在 DataTable 类的 NewRow()方法内部调用。这里新建了强类型化的一行。DataRowBuilder 实例由 DataTable 类创建，其成员仅能在 System.Data 程序集中访问。

```
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    protected override global::System.Data.DataRow NewRowFromBuilder(global::System.Data.
    DataRowBuilder builder) {
        return new ProductRow(builder);
    }
```

最后一个要讨论的类是 ProductRow 类，它派生自 DataRow 类。这个类用于提供对数据表中所有字段的类型安全的访问。它封装了特定行的存储器，并提供成员来读取(和写入)表中的每个字段。

另外，对于每个可空字段，有函数可以把该字段设置为 null，并检查该字段是否为 null。下面的示例列出了 SupplierID 列的函数：

```
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.
    TypedDataSetGenerator", "2.0.0.0")]
    public partial class ProductRow : global::System.Data.DataRow {

         private ProductDataTable tableProduct;

         [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
         internal ProductRow(global::System.Data.DataRowBuilder rb) :
                 base(rb) {
            this.tableProduct = ((ProductDataTable)(this.Table));
```

```
    }

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public int ProductID {
        get {
            return ((int)(this[this.tableProduct.ProductIDColumn]));
        }
        set{
            this[this.tableProduct.ProductIDColumn] = value;
        }
    }
```

下面的代码利用XSD工具中的类的输出从Product表中检索数据，并在控制台上显示这些数据：

```
using System;
using System.Data.SqlClient;

namespace _10_XSDDataset
{
    class Program
    {
        static void Main(string[] args)
        {
            string select = "SELECT * FROM Products";

            using (SqlConnection conn = new SqlConnection(GetDatabaseConnection()))
            {
                SqlDataAdapter da = new SqlDataAdapter(select, conn);

                Products ds = new Products();

                da.Fill(ds, "Product");

                foreach (Products.ProductRow row in ds.Product)
                    Console.WriteLine("'{0}' from {1}",
                        row.ProductID,
                        row.ProductName);

                conn.Close();
            }
        }

        private static string GetDatabaseConnection()
        {
            return "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
        }
    }
}
```

XSD文件的输出结果包含一个派生自DataSet类的Products类，它使用数据适配器来创建和填充。foreach语句使用强类型化的ProductRow和Product属性，Product属性返回Product数据表。

要编译这个示例使用的.XSD，在Visual Studio命令提示行下执行下面的命令：

```
xsd product.xsd /d
```

这条命令把.XSD 转换为代码，以便在 Visual Studio 中更容易访问它。

32.8 填充 DataSet 类

定义了数据集的架构，并准备好 DataTables、DataColumns、Constrain 类和一些必需内容后，就需要用这些信息填充 DataSet 类。从外部源中读取数据，并把数据插入到 DataSet 类中，有两种方式：

- 使用数据适配器
- 把 XML 读入 DataSet 类

下面讨论这两种方式。

32.8.1 用数据适配器填充 DataSet

32.5.2 节简要介绍了 SqlDataAdapter 类，使用该类的代码如下所示：

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

加粗代码行显示了所用的 SqlDataAdapter 类；其他数据适配器类在功能上与 SqlDataAdapter 类完全相同。

为了把数据插入到 DataSet 类中，需要执行某种形式的命令以选择该数据。该命令可以是 SQL SELECT 语句，一个存储过程的调用，或者是 TableDirect 命令(用于 OLE DB 提供程序)。上面的示例使用了 SqlDataAdapter 类的一个构造函数，把传递过来的 SQL SELECT 子句转换为一个 SqlCommand，在适配器上调用 Fill()方法时执行这条命令。

在本章前面的存储过程示例中，定义了 INSERT、UPDATE 和 DELETE 过程，但没有给出 SELECT 过程，本节介绍该过程，并说明如何从 SqlDataAdapter 类上调用存储过程，从而把数据填充到 DataSet 类中。

在数据适配器上使用存储过程

首先需要定义一个存储过程，SELECT 数据的存储过程如下所示：

```
CREATE PROCEDURE RegionSelect AS
  SET NOCOUNT OFF
  SELECT * FROM Region
GO
```

这个存储过程由代码中的 InitialiseDatabase 方法在数据库中创建。

接着，需要定义一个执行该存储过程的 SqlCommand，同样，这段代码非常简单，并且大部分已经在前一节中出现过：

```
private static SqlCommand GenerateSelectCommand(SqlConnection conn)
{
  SqlCommand aCommand = new SqlCommand("RegionSelect", conn);
```

```
        aCommand.CommandType = CommandType.StoredProcedure;
        aCommand.UpdatedRowSource = UpdateRowSource.None;
        return aCommand;
    }
```

这个方法生成了一个 SqlCommand，该 SqlCommand 在执行时会调用 RegionSelect 过程。最后是把这条命令和 SqlDataAdapter 类关联起来，并调用 Fill()方法：

```
DataSet ds = new DataSet();
// Create a data adapter to fill the DataSet
SqlDataAdapter da = new SqlDataAdapter();
// Set the data adapter's select command
da.SelectCommand = GenerateSelectCommand (conn);
da.Fill(ds, "Region");
```

其中新建了一个 SqlDataAdapter 类，把生成的 SqlCommand 赋予数据适配器的 SelectCommand 属性，然后调用执行存储过程的 Fill()方法，把返回的所有行插入到 Region 表的 DataSet 类中(在本例中，它由运行库生成)。

数据适配器不仅能通过执行命令来选择数据，32.8 节会介绍数据适配器的其他功能。

32.8.2 从 XML 中填充 DataSet 类

除了为给定的 DataSet 类和相关表生成架构外，DataSet 类还可以读写本地 XML 中的数据，如磁盘上的文件、数据流或文本读取器。

要把 XML 加载到 DataSet 类中，只需要调用一个 ReadXML()方法，如下面的示例所示，从磁盘文件中读取数据：

```
DataSet ds = new DataSet();
ds.ReadXml(".\\MyData.xml");
```

ReadXml()方法试图从输入的 XML 中加载任何内联架构信息，如果找到了某个架构，就使用这个架构验证从该文件中加载的数据的有效性。如果没有找到内联架构，DataSet 类就会在加载数据时扩展其内部的结构，这类似于前面示例中的 Fill()方法的作用，该方法检索数据，并根据选择的数据构造 DataTable 类。

32.9 持久化 DataSet 类的修改

在 DataSet 类中编辑完数据后，通常需要持久化这些改变。最常见的示例是从数据库中选择数据，并把它显示给用户，把这些更新返回给数据库。

在没有“连接数据库的”应用程序中，这些改变可以持久化到 XML 文件中，并传输到中间层的应用程序服务器上，然后进行处理，以更新几个数据源。

DataSet 类可以用于这两个示例，而且这很容易完成。

32.9.1 通过数据适配器进行更新

除了 SqlDataAdapter 类最有可能包含的 SelectCommand 之外，还可以定义 InsertCommand、UpdateCommand 和 DeleteCommand。顾名思义，这些对象都是适用于相应提供程序的命令对象(如

SqlCommand 或 OleDbCommand)实例。

有了这种灵活性后，就可以自由调整应用程序，具体方法是对频繁使用的命令(如 select 和 insert)采用合适的存储过程来执行，对不常使用的命令(如 delete)直接采用 SQL 命令来执行。一般建议为所有数据库交互操作提供存储过程，因为这会更快，更容易调整。

本节的示例使用 32.3.2 节中的存储过程，插入、更新和删除 Region 记录，再把这些与上面编写的 RegionSelect 过程结合起来，生成一个新示例，这个示例使用这些命令来检索和更新 DataSet 类中的数据。代码的主体在下一节介绍。

1.新插入一行

把新行添加到 DataTable 中有两种方式。第一种方式是调用 NewRow()方法，返回一空白行，然后向其填充数据，最后把它添加到 Rows 集合中，如下所示：

```
DataRow r = ds.Tables["Region"].NewRow();
r["RegionID"]=999;
r["RegionDescription"]="North West";
ds.Tables["Region"].Rows.Add(r);
```

第二种方式是把一个数据数组传递给 Rows.Add()方法，如下面的代码所示：

```
DataRow r = ds.Tables["Region"].Rows.Add
            (new object [] { 999, "North West" });
```

DataTable 中的所有新行都把自己的 RowState 设置为 Added。在对数据库进行修改前，这个示例先转储记录，以便把下面的行添加到 DataTable 中(以任何一种方式)。注意右边一列显示行的状态：

```
New row pending inserting into database
   1 Eastern                                                   Unchanged
   2 Western                                                   Unchanged
   3 Northern                                                  Unchanged
   4 Southern                                                  Unchanged
 999 North West                                                Added
```

要从 DataAdapter 更新数据库，可以调用其中一个 Update()方法：

```
da.Update(ds, "Region");
```

对于 DataTable 中的新行，这将执行存储过程(在本例中是 RegionInsert)，本示例然后转储数据的状态，因此可以查看对数据库进行的修改。

```
New row updated and new RegionID assigned by database
 1 Eastern                                                     Unchanged
 2 Western                                                     Unchanged
 3 Northern                                                    Unchanged
 4 Southern                                                    Unchanged
 5 North West                                                  Unchanged
```

查看 DataTable 中的最后一行。在代码中把 RegionID 设置为 999，但在执行 RegionInsert 存储过程后，该值改为 5。这是有意的——数据库通常会生成主键，并且更新 DataTable 中的数据。DataTable 中数据的更新是因为源代码中的 SqlCommand 定义会把 UpdatedRowSource 属性设置为 UpdateRowSource.OutputParameters：

```
SqlCommand aCommand = new SqlCommand("RegionInsert", conn);

aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
                        SqlDbType.NChar,
                        50,
                        "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID",
                        SqlDbType.Int,
                        0,
                        ParameterDirection.Output,
                        false,
                        0,
                        0,
                        "RegionID", // Defines the SOURCE column
                        DataRowVersion.Default,
                        null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

这段代码的作用是：无论何时数据适配器执行这条命令，输出参数都应映射到该数据行的源，在本例中它是 DataTable 中的一行。该标志说明了应更新什么数据——存储过程有一个输出参数映射到 DataRow，它应用的列是 RegionID，因为这是在命令的定义中定义的。

UpdateRowSource 的值如表 32-14 所示。

表 32-14

UpdateRowSource 值	说　明
Both	存储过程可以返回输出参数和一个完整的数据库记录。这两个数据源都用于更新源数据行
FirstReturnedRecord	该命令返回一个记录，该记录的内容应合并到最初的源 DataRow 中，当给定的表有许多默认(或计算)列时，使用这个值很有用，因为在执行 INSERT 语句之后，这些行需要与客户端上的 DataRow 同步。例如'INSERT(列)INTO(表)WITH(主键)', 'SELECT (列) FROM (表) WHERE (主键) '。返回的记录应合并到源数据行上
None	丢弃从该命令返回的所有数据
OutputParameters	命令的任何输出参数都映射到 DataRow 的对应列上

2.更新现有的行

要更新 DataTable 中的已有行，只需要使用带有一个列名或列号的 DataRow 类的索引器即可，如下面的代码所示：

```
r["RegionDescription"]="North West England";
r[1] = "North West England";
```

这两条语句等价(在本例中)：

```
Changed RegionID 5 description
  1 Eastern                                                     Unchanged
  2 Western                                                     Unchanged
  3 Northern                                                    Unchanged
  4 Southern                                                    Unchanged
  5 North West England                                          Modified
```

在更新数据库前，被更新的行应把其状态设置为 Modified，如上所示。把改变永久化到数据库中下，这个状态会变回 Unchanged。

3.删除行

删除行需要调用 Delete()方法：

```
r.Delete();
```

虽然被删除的行把其行状态设置为 Deleted，但不能从被删除的 DataRow 中读取列，因为它们不再有效。当调用适配器的 Update()方法时，所有被删除的行都会使用 DeleteCommand，在本例中是执行 RegionDelete 存储过程。

32.9.2　写入 XML 输出结果

如上所述，DataSet 类支持在 XML 中定义其架构，如同可以从 XML 文档中读取数据，也可以把数据写入 XML 文档。

DataSet.WriteXml()方法可以输出存储在 DataSet 类中的各部分数据，可以选择只输出数据，也可以输出数据和架构。下面是为两个 Region 示例编写的代码：

```
ds.WriteXml(".\\WithoutSchema.xml");
ds.WriteXml(".\\WithSchema.xml", XmlWriteMode.WriteSchema);
```

第一个文件 WithoutSchema.xml 如下所示：

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
    <Region>
        <RegionID>1</RegionID>
        <RegionDescription>Eastern</RegionDescription>
    </Region>
    <Region>
        <RegionID>2</RegionID>
        <RegionDescription>Western</RegionDescription>
    </Region>
    <Region>
        <RegionID>3</RegionID>
        <RegionDescription>Northern</RegionDescription>
    </Region>
    <Region>
        <RegionID>4</RegionID>
        <RegionDescription>Southern</RegionDescription>
    </Region>
</NewDataSet>
```

RegionDescription 上的闭合标记在页面的右边，因为数据库列定义为 NCHAR(50)，这是一个包含 50 个字符的字符串，其中用空格填充。

WithSchema.xml 文件产生的输出结果包含 DataSet 类的 XML 架构和数据本身：

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true"
      msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Region">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="RegionID" msdata:AutoIncrement="true"
                  msdata:AutoIncrementSeed="1" type="xs:int" />
                <xs:element name="RegionDescription" type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  <Region>
    <RegionID>1</RegionID>
    <RegionDescription>Eastern</RegionDescription>
  </Region>
  <Region>
    <RegionID>2</RegionID>
    <RegionDescription>Western</RegionDescription>
  </Region>
  <Region>
    <RegionID>3</RegionID>
    <RegionDescription>Northern</RegionDescription>
  </Region>
  <Region>
    <RegionID>4</RegionID>
    <RegionDescription>Southern</RegionDescription>
  </Region>
</NewDataSet>
```

注意，使用 msdata 架构中的文件，它定义 DataSet 类中列的附加属性，如 AutoIncrement 和 AutoIncrementSeed，这些属性直接对应于 DataColumn 类上的可定义属性。

32.10 使用 ADO.NET

本节介绍使用 ADO.NET 开发数据访问应用程序时的一些常见情况，例如如何在使用多层方式实现的应用程序中使用 ADO.NET，如何有效地生成 SQL 键。本节介绍的主题并不匹配本章的其他章节。

32.10.1 分层开发

开发与数据交互的应用程序时，常常要把应用程序分层，常见的模型是一个应用层(前端)、一个数据服务层和数据库本身(后端)。

但使用这个模型的难题是，确定在层之间传输什么数据，以及应采用什么格式来传输数据。有

了 ADO.NET，就不必担心这个问题了，这种设计一开始就支持这种体系结构。

在 ADO.NET 中，对复制完整的记录集有比 OLE DB 更好的支持。在.NET 中，复制 DataSet 只需要使用下面的代码：

```
DataSet source = {some dataset};
DataSet dest = source.Copy();
```

这将创建源 DataSet 的一个完全相同的副本——将会复制所有 DataTable、DataColumn、DataRow 和 Relation，所有数据都与它在源 DataSet 中所处的状态完全相同。如果只需要复制 DataSet 的架构，就可以试试下面的代码：

```
DataSet source = {some dataset};
DataSet dest = source.Clone();
```

这也会复制所有表、关系等，但每个复制的 DataTable 都是空的。这个过程非常简单。

在编写一个分层的系统(无论该系统是基于 Windows 客户端应用程序还是基于 Web)时，常见的要求是在层之间附带尽可能少的数据。这在传输数据集时减少了资源的消耗。

要达到这个要求，DataSet 类提供了 GetChanges()方法。这个简单的方法执行许多任务，并返回一个 DataSet，其中只包含从源数据集中修改过的行。这是在层之间传递数据时最理想的情况，因为只传递一组少量的数据。

下面的示例说明了如何生成一个修改后的 DataSet：

```
DataSet source = {some dataset};
DataSet dest = source.GetChanges();
```

这也很乏味，下面介绍的内容会比较有趣。GetChanges()方法有两个重载版本，一个重载版本接受 DataRowState 枚举的一个值，并返回对应于该状态的行。GetChanges()方法只调用 GetChanges(Deleted | Modified | Added)，该方法首先检查，以确保调用 HasChanges()方法进行了一些修改。如果没有修改，就立即给调用者返回空值。

下一个操作是复制当前的 DataSet。一旦完成后，就会创建新的 DataSet，以忽略违反约束的情况(EnforceConstraints = false)，然后，把每个表中所有修改的行都复制到新的 DataSet 中。

在得到一个只包含修改内容的 DataSet 后，就可以把它们移动到数据服务层上进行处理。数据在数据库中更新后，修改后的数据集就返回给调用者(例如，来自存储过程的一些输出参数在某些列上有已更新的值)，然后使用 Merge()方法，把这些修改的内容合并到源 DataSet 中。操作的顺序如图 32-9 所示。

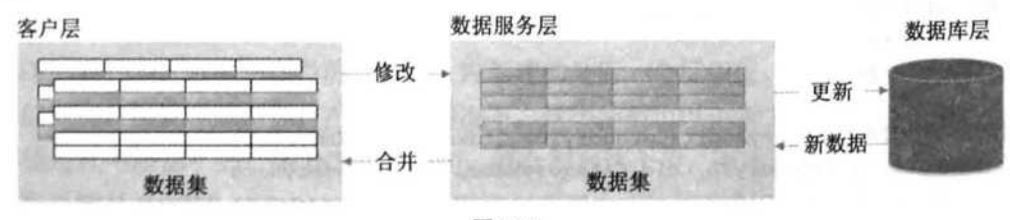


图 32-9

32.10.2 生成 SQL Server 的键

本章前面给出的 RegionInsert 存储过程在给数据库插入数据时生成了一个主键值。该示例中生成主键的方法相当原始，不能很好地扩展，实际的应用程序应利用生成键的其他策略。

首先要定义一个 Identity 列，并从存储过程中返回@@IDENTITY 值。下面的存储过程显示了如何为 Northwind 样本数据库中的 Categories 表定义该值。在 SQL 查询分析器中输入这个存储过程，或者运行代码下载中的 StoredProcs.sql 文件：

```
CREATE PROCEDURE CategoryInsert(@CategoryName NVARCHAR(15),
                                @Description NTEXT,
                                @CategoryID INTEGER OUTPUT) AS
  SET NOCOUNT OFF
  INSERT INTO Categories (CategoryName, Description)
     VALUES(@CategoryName, @Description)
  SELECT @CategoryID = @@IDENTITY
GO
```

这将把新的一行插入到 Category 表中，并给调用者返回生成的主键(CategoryID 列的值)。在 SQL 查询分析器中输入下述 SQL 命令，可以测试该过程：

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

当把该过程作为批处理命令来执行时，会把新的一行插入到 Category 表中，并返回新记录的标识，然后把该记录显示给用户。

假定过了几个月后，有人要添加一个简单的审计跟踪，它记录所有插入的记录，以及对 Category 名的修改。此时，定义如图 32-10 所示的一个表，记录 Category 的新名和旧名。

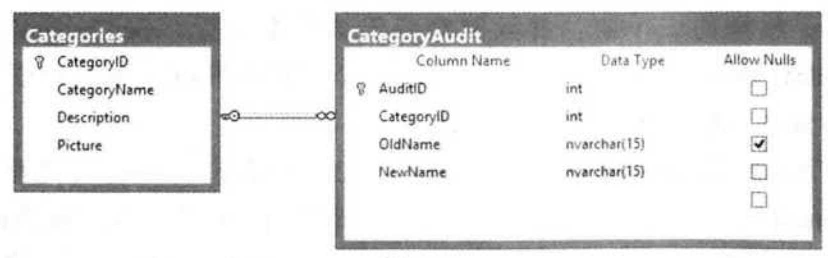


图 32-10

这个表的脚本包含在本书网站的 StoredProcs.sql 文件中。AuditID 列定义为一个 IDENTITY 列，然后构造两个数据库触发器，数据库触发器使用如下代码记录对 CategoryName 字段的修改：

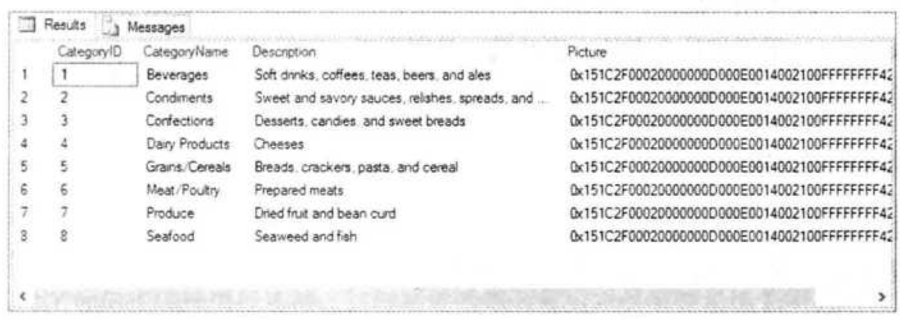
```
CREATE TRIGGER CategoryInsertTrigger
  ON Categories
  AFTER UPDATE
AS
  INSERT INTO CategoryAudit(CategoryID, OldName, NewName )
    SELECT old.CategoryID, old.CategoryName, new.CategoryName
    FROM Deleted AS old,
       Categories AS new
    WHERE old.CategoryID = new.CategoryID;
GO
```

对于习惯使用 Oracle 存储过程的用户，SQL Server 其实没有 OLD 行和 NEW 行的概念，而是使用一个插入触发器，在内存中有一个 Inserted 表可用于插入记录，在 Deleted 表中删除和更新旧记录。

插入触发器是一个内存中的 Inserted 表，旧记录的删除和更新放在 Deleted 表中。CategoryInsert-Trigger 触发器检索已处理的记录的 CategoryID，把它与 CategoryName 列的新旧值一起存储。

现在，调用原始存储过程插入一个新 CategoryID 时，会得到一个标识值，但它不再是插入到 Categories 表中的行的标识值，而是 CategoryAudit 表中为该行生成的新值。

要查看这些值，可打开 SQL Server Enterprise 管理器的一个副本，查看 Categories 表的内容，如图 32-11 所示。图 32-11 列出了 Northwind 数据库中的所有类别。



Results | Messages

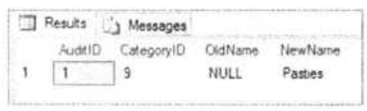
	CategoryID	CategoryName	Description	Picture
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales	0x151C2F00020000000D000E0014002100FFFFFFFF42
2	2	Condiments	Sweet and savory sauces, relishes, spreads, and ...	0x151C2F00020000000D000E0014002100FFFFFFFF42
3	3	Confections	Desserts, candies, and sweet breads	0x151C2F00020000000D000E0014002100FFFFFFFF42
4	4	Dairy Products	Cheeses	0x151C2F00020000000D000E0014002100FFFFFFFF42
5	5	Grains/Cereals	Breads, crackers, pasta, and cereal	0x151C2F00020000000D000E0014002100FFFFFFFF42
6	6	Meat/Poultry	Prepared meats	0x151C2F00020000000D000E0014002100FFFFFFFF42
7	7	Produce	Dried fruit and bean curd	0x151C2F00020000000D000E0014002100FFFFFFFF42
8	8	Seafood	Seaweed and fish	0x151C2F00020000000D000E0014002100FFFFFFFF42

图 32-11

因为 Categories 表中的下一个 identity 值是 9，所以执行下面的代码新插入一行，以查看返回了什么 ID。

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

在测试 PC 上其输出值是 1。如果查看图 32-12 中的 CategoryAudit 表，则会发现这是新插入的审核记录的标识，不是新建的 category 记录的标识。



Results | Messages

	AuditID	CategoryID	OldName	NewName
1	1	9	NULL	Pasties

图 32-12

问题是@@IDENTITY 实际上在起作用，它返回由会话创建的最后一个标识值，如图 32-12 所示，所以它不是 100%的可靠。

除了@@IDENTITY 外，还可以使用另外两个标识函数，它们都不会出任何问题。第一个函数是 SCOPE_IDENTITY()，它返回在当前“范围”内创建的最后一个标识值。SQL Server 把该范围定义为一个存储过程、触发器或函数。在大多数情况下，这个函数可以正常工作。但如果因某种原因，有人在存储过程中添加了另一条 INSERT 语句，就会得到这个值，而不是期望的值。

另一个函数是 IDENT_CURRENT()，它返回任何范围中为给定表生成的最后一个标识值。例如，如果两个用户同时访问 SQL Server，其中一个用户就有可能得到另一个用户生成的标识值。

找出这个问题的原因不太容易。在 SQL Server 中使用 IDENTITY 列时要多加小心。

32.10.3 命名约定

下面的提示和约定与.NET 并不直接相关，但应共享和遵循它们，特别是在给约束命名时。如果你对此主题已经有自己的观点，就可以跳过本节。

1. 数据库表的约定

- **总是使用单数名称**——例如，Product 而不是 Products，这是一个普遍适用的约定，因为我们必须给客户解释某种数据库架构，从语法上看，“Product 表包含产品”要比“Products 表包含产品”好得多。但 Northwind 数据库并没有遵循这一约定。
- **给表中的字段采用某种形式的命名约定**——我们采用的是表的主键<Table>_ID(假定主键是一列)，字段采用 Name，这是考虑到记录的用户友好性，记录本身的任何文本信息采用 Description。采用好的命名约定意味着，只要查看一下数据库中的几乎任何表，从直觉上就知道其中的字段主要用于什么目的。

2. 数据库列的约定

- 使用单数名称，而不是复数名称。
- 链接到另一个表中的列名应与该表的主键名相同。例如，链接到 Product 表的列名为 Product_ID。链接到 Sample 表的列名为 Sample_ID。这并不总是可行的，特别是如果一个表有另一个表的多个引用，这个命名约定就无效。此时应使用其他方式命名。
- 日期字段应有一个_On 后缀，如 Modified_On、Created_On。按照这种命名约定，如果读取一些 SQL 输出，就很容易从列的名称中知道该列的含义。
- 记录用户操作的字段名应有一个_By 后缀，如 Modified_By 和 Created_By，这将有助于理解。

3. 约束的约定

- 如果可能，在约束名中包含表名和列名，如 CK_<Table>_<Field>。例如，对于 Person 表中的 Sex 列，其检查约束可以是 CK_Person_Sex，而对于 product 和 supplier 之间的外键关系，对应的外键约束是 FK_Product_Supplier_ID。
- 约束类型的前面加一个前缀，如 CK 表示检查约束，FK 表示外键约束。也可以指定更为特殊的名称，如 Age 列上的 CK_Person_Age_GT0 表示该年龄应大于 0。
- 如果必须限制约束名的长度，则可以在其中包含表名，而不包含列名。在发现有违反约束的情况时，通常很容易推断哪个表出现错误，但有时不容易检查出是哪一列出了问题。Oracle 允许名称的长度最长为 30 个字符，而很容易超过这种限制。

4. 存储过程

在过去几年中，把 C 放在每个声明的类前面令人困惑。同样，许多 SQL Server 开发人员也困惑于在每个存储过程的前面加上 sp_或类似的东西，这不是一个好方法。

SQL Server 在所有的系统存储过程前面使用 sp_前缀。所以用户会对 sp_widget 是否为 SQL Server 标准存储过程产生疑问。另外，当查找存储过程时，SQL Server 会把带有 sp_前缀的过程与没有该

前缀的过程区别对待。

如果使用这个前缀，但没有用该存储过程的数据库/拥有者来限定，SQL Server 就会在当前范围内查找，然后跳到主数据库中查找存储过程。没有 sp_前缀，用户就会早一些得到错误。更糟糕的是本地存储过程(在自己的数据库中创建)与系统存储过程有相同的名称和参数。应尽可能避免这种情况，如果有疑问，就不要使用前缀。

在调用存储过程时，它们应以过程的拥有者作为前缀，如 dbo.selectWidgets。这将比不使用该前缀略快一些，因为 SQL Server 查找该存储过程所做的工作较少。有时这不会对应用程序的执行速度有很大的影响，但它是一个可以自由使用的调整技巧。

总之，在数据库或代码中命名实体时，应保持一致。

32.11 小结

数据访问是一个很大的主题，特别是在.NET 中，因为有非常丰富的内容要涵盖。本章概述了 ADO.NET 名称空间中的主要类，揭示了在处理数据源中的数据时如何使用这些类。

首先，使用 SqlConnection (SQL Server 专用)和 OleDbConnection(用于任何 OLE DB 数据源)，探讨了 Connection 对象的用法。这两个类的编程模型非常类似，以至于一般其中一个类可以替代另一个类，代码仍能继续运行。

接着阐述了如何正确地进行连接，这样稀缺的资源就可以尽可能早地关闭。所有连接类都实现 IDisposable 接口，在对象放在 using 子句中时调用该接口。如果本章只有一件值得注意的事，那就是尽早关闭数据库连接的重要性。

此外，通过执行没有返回数据的示例，和利用输入和输出参数调用存储过程的示例，讨论了数据库命令。本章描述了各种执行方法，包括只能在 SQL Server 提供程序上使用的 ExecuteXmlReader 方法。这大大简化了基于 XML 的数据的选择和处理。

这里详细介绍了 System.Data 名称空间中的泛型类，包括 DataSet、DataTable、DataColumn、DataRow 以及关系和约束。DataSet 类是数据的最佳容器，各种方法使之成为跨层数据流的理想容器。DataSet 中的数据可以用 XML 来表示，以利于传输，另外，还有一些方法可以在层之间传输最少量的数据。把许多数据表放在一个 DataSet 中可以大大提高其可用性。

除了把架构存储在 DataSet 中外，.NET 还包括数据适配器，它与各种 Command 对象组合使用，可以把数据选择出来，放在 DataSet 中，以后还可以更新数据存储器中的数据。数据适配器的一个优点是可以为 4 种操作(SELECT、INSERT、UPDATE 和 DELETE)定义不同的命令。系统可以根据数据库架构信息和一条 SELECT 语句创建一组默认的命令，但为了得到最佳性能，可以使用一组存储过程，并相应地定义 DataAdapter 的命令，仅把需要的信息传送给这些存储过程。

我们还介绍了 XSD 工具(xsd.exe)，使用一个示例来说明如何在.NET 中基于 XML 架构使用类。产生的类可以用于应用程序，它们的自动生成减少了大量的输入工作。

最后论述了用于数据库开发的一些最佳实践和命名约定。访问 SQL Server 数据库的更多知识，详见第 33 章。

第33章

ADO.NET Entity Framework

本章要点

- 编程模型
- 映射
- 实体类
- 对象上下文
- 关系
- 查询数据
- 更新
- Code First

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 图书示例
- 付款示例
- 方程式 1 赛车示例
- Code First 示例
- Code First 迁移

33.1 用 Entity Framework 编程

ADO.NET Entity Framework 是一个对象-关系的映射架构，它提供了 ADO.NET 的一个抽象，可基于引用的数据库获取对象模型。可以通过 Entity Framework 使用不同的编程模型：Model First、Database First 和 Code First。Model First 和 Database First 都通过一个映射文件来提供映射信息，而使用 Code First，则映射信息全部通过 C#代码来处理。本章介绍这三个编程模型的信息。

本章使用 CSDL(Conceptual Schema Definition Language，概念架构定义语言)、SSDL(Storage Schema Definition Language，存储架构定义语言)和 MSL(Mapping Schema Language，映射架构语言)给出数据库和实体类之间的映射信息。讨论实体之间的不同关系，如对象的一个层次结构一个表关系、一个类型一个表关系和 n 对 n 关系。

本章还将描述从代码中直接通过 EntityClient 提供程序访问数据库的不同方式，如何使用 Entity SQL 或帮助方法创建 Entity SQL，如何使用 LINQ to Entities，也会讨论对象跟踪，以及数据上下文如何包含变化的信息，以更新数据。最后学习如何通过 Entity Framework 使用 POCO(Plain Old CLR Objects)，以及如何使用 Code First 编程模型。

本章使用 Books 和 Formula1 数据库。这些数据库包含在 http://www.wrox.com 代码示例的下载包中。

ADO.NET Entity Framework 提供了从关系数据库架构到对象的映射。关系数据库和面向对象的语言用不同的方式定义了关联。例如，样本数据库 Formula1 包含 Racers 表和 RaceResults 表。要访问某个赛手的所有 RaceResults 行，需要执行一条 SQL join 语句。在面向对象的语言中，更常见的是定义一个 Racer 类和一个 RaceResult 类，使用 Racer 类的 RaceResults 属性访问赛手的比赛结果。

在 Entity Framework 推出之前，对于对象-关系映射，就可以使用 DataSet 类和类型化的数据集。DataSet 非常类似于数据库的结构，它包含 DataTable、DataRow、DataColumn 和 DataRelation 类，而不提供对象支持。ADO.NET Entity Framework 支持直接定义完全独立于数据库结构的实体类，并把它们映射到数据库的表和关系上。通过应用程序使用对象，应用程序就可以免受数据库修改的影响。

ADO.NET Entity Framework 使用 Entity SQL 为存储器定义基于实体的数据库查询(T-SQL 的一个扩展)。LINQ to Entities 允许使用 LINQ 语法来查询数据。对象上下文保存了变化的实体信息，从而在把实体写回存储器时，提供这些信息。

Microsoft 把核心框架中越来越多的部件移动到 NuGet 包中，这表示，不需要等待整个 .NET Framework 的更新，就可以发布新功能。在 Entity Framework 的最新版本中，越来越多的部件移动到 NuGet 包中。在 Entity Framework 6 中(本书讨论这个版本)，框架就完全包含在一个 NuGet 包中。为了不与以前的版本冲突，一些部件现在位于新名称空间，但其中的类和成员没有改变。

包含 ADO.NET Entity Framework 中的类的名称空间如表 33-1 所示。

表 33-1

名称空间	说　明
System.Data	这是用于 ADO.NET 的主要名称空间。在 ADO.NET Entity Framework 中，这个名称空间包含了与实体相关的异常类，如 MappingException 和 QueryException 异常类
System.Data.Core.Common	这个名称空间包含由.NET 数据提供程序共享的类。DbProviderServices 类是一个抽象基类，它必须由 ADO.NET Entity Framework 提供程序实现
System.Data.Core.Common .CommandTrees	这个名称空间包含构建表达式树的类
System.Data.Entity	这个名称空间包含用于 Code First 开发模型的类

(续表)

名 称 空 间	说 明
System.Data.Entity.Design	这个名称空间包含由设计器用于创建 EDM (Entity Data Model，实体数据模型)文件的类
System.Data.Entity.Core.EntityClient	这个名称空间指定由.NET Framework 数据提供程序访问 ADO.NET Entity Framework 的类。EntityConnection、EntityCommand 和 EntityReader 可用于访问 Entity Framework

33.2 Entity Framework 映射

ADO.NET Entity Framework 通过 Model First 和 Database First，提供了几个把数据库表映射到对象上的层。通过 Database First，可以从一个数据库架构开始，使用 Visual Studio 项模板创建完整的映射。还可以先用设计器(Model First)设计实体类，再把它映射到数据库上，在该数据库中，表和表之间的关系可以有完全不同的结构。

需要定义的层如下：

- 逻辑层——该层定义关系数据
- 概念层——该层定义.NET 类
- 映射层——该层定义从.NET 类到关系表和关联的映射。

下面是一个简单的数据库架构，如图 33-1 所示，其中包含 Books 表和 Authors 表，以及关联表 BookAuthors，它把作者映射到图书上。

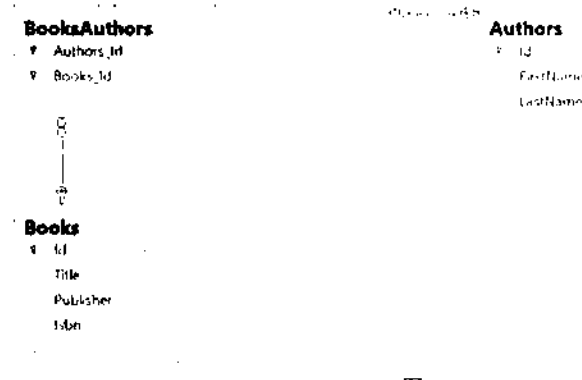


图 33-1

Code First 编程使用映射，如 33.9 节所述。

33.2.1 逻辑层

逻辑层由 SSDL(Store Schema Definition Language，存储架构定义语言)定义，描述了数据库表及其关系的结构。

下面的代码使用 SSDL 来描述 3 个表：Books、Authors 和 BookAuthors。EntityContainer 元素描

述所有包含 EntitySet 元素的表和包含 AssociationSet 元素的表。表的部分用 EntityType 元素定义。在 EntityType Books 中，Id、Title、Publisher 和 ISBN 列用 Property 元素定义。Property 元素包含了定义数据类型的 XML 属性。Key 元素定义表的键。下述代码在 BooksDemo/BooksModel.edmx 代码文件中。

```
<edmx:StorageModels>
  <Schema Namespace="BooksModel.Store" Alias="Self"
      Provider="System.Data.SqlClient"
      ProviderManifestToken="2008"
      xmlns:store=
      "http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
      xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
    <EntityContainer Name="BooksModelStoreContainer">
      <EntitySet Name="Authors" EntityType="BooksModel.Store.Authors"
          store:Type="Tables" Schema="dbo" />
      <EntitySet Name="Books" EntityType="BooksModel.Store.Books"
          store:Type="Tables" Schema="dbo" />
      <EntitySet Name="BooksAuthors" EntityType="BooksModel.Store.BooksAuthors"
          store:Type="Tables" Schema="dbo" />
      <AssociationSet Name="FK_BooksAuthors_Authors"
          Association="BooksModel.Store.FK_BooksAuthors_Authors">
        <End Role="Authors" EntitySet="Authors" />
        <End Role="BooksAuthors" EntitySet="BooksAuthors" />
      </AssociationSet>
      <AssociationSet Name="FK_BooksAuthors_Books"
          Association="BooksModel.Store.FK_BooksAuthors_Books">
        <End Role="Books" EntitySet="Books" />
        <End Role="BooksAuthors" EntitySet="BooksAuthors" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Authors">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="int" Nullable="false"
          StoreGeneratedPattern="Identity" />
      <Property Name="FirstName" Type="nvarchar" Nullable="false"
          MaxLength="50" />
      <Property Name="LastName" Type="nvarchar" Nullable="false"
          MaxLength="50" />
    </EntityType>
    <EntityType Name="Books">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="int" Nullable="false"
          StoreGeneratedPattern="Identity" />
      <Property Name="Title" Type="nvarchar" Nullable="false" MaxLength="50" />
      <Property Name="Publisher" Type="nvarchar" Nullable="false"
          MaxLength="50" />
      <Property Name="Isbn" Type="nchar" MaxLength="18" />
```

```
    </EntityType>
    <EntityType Name="BooksAuthors">
      <Key>
        <PropertyRef Name="BookId" />
        <PropertyRef Name="AuthorId" />
      </Key>
      <Property Name="BookId" Type="int" Nullable="false" />
    <Property Name="AuthorId" Type="int" Nullable="false" />
    </EntityType>
    <Association Name="FK_BooksAuthors_Authors">
      <End Role="Authors" Type="BooksModel.Store.Authors" Multiplicity="1" />
      <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors"
          Multiplicity="*" />
      <ReferentialConstraint>
        <Principal Role="Authors">
          <PropertyRef Name="Id" />
        </Principal>
        <Dependent Role="BooksAuthors">
          <PropertyRef Name="AuthorId" />
        </Dependent>
      </ReferentialConstraint>
    </Association>
    <Association Name="FK_BooksAuthors_Books">
      <End Role="Books" Type="BooksModel.Store.Books" Multiplicity="1" />
      <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors"
          Multiplicity="*" />
      <ReferentialConstraint>
        <Principal Role="Books">
          <PropertyRef Name="Id" />
        </Principal>
        <Dependent Role="BooksAuthors">
          <PropertyRef Name="BookId" />
        </Dependent>
      </ReferentialConstraint>
    </Association>
  </Schema>
</edmx:StorageModels>
```

BooksModel.edmx 文件包含 SSDL、CSDL 和 MSL。使用 XML 编辑器可以打开这个文件，查看其内容。

33.2.2 概念层

概念层定义了.NET 类。该层用 CSDL(Conceptual Schema Definition Language，概念架构定义语言)定义。

图 33-2 显示了用 ADO.NET 实体数据模型设计器定义的实体 Author 和 Book。

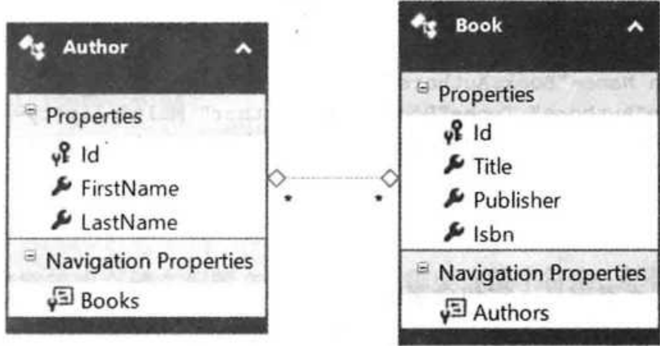


图 33-2

下面是定义实体类型 Author 和 Book 的 CSDL 内容(在 BooksDemo/BooksModel.edmx 代码文件中)。从 Books 数据库中创建它们。

```
<edmx:ConceptualModels>
  <Schema Namespace="BooksModel" Alias="Self" xmlns:annotation=
      "http://schemas.microsoft.com/ado/2009/02/edm/annotation"
      xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
    <EntityContainer Name="BooksEntities" annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Authors" EntityType="BooksModel.Author" />
      <EntitySet Name="Books" EntityType="BooksModel.Book" />
      <AssociationSet Name="BooksAuthors" Association="BooksModel.BooksAuthors">
        <End Role="Authors" EntitySet="Authors" />
        <End Role="Books" EntitySet="Books" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Author">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="Int32" Nullable="false"
          annotation:StoreGeneratedPattern="Identity" />
      <Property Name="FirstName" Type="String" Nullable="false" MaxLength="50"
          Unicode="true" FixedLength="false" />
      <Property Name="LastName" Type="String" Nullable="false" MaxLength="50"
          Unicode="true" FixedLength="false" />
      <NavigationProperty Name="Books" Relationship="BooksModel.BooksAuthors"
          FromRole="Authors" ToRole="Books" />
    </EntityType>
    <EntityType Name="Book">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="Int32" Nullable="false"
          annotation:StoreGeneratedPattern="Identity" />
      <Property Name="Title" Type="String" Nullable="false" MaxLength="50"
          Unicode="true" FixedLength="false" />
      <Property Name="Publisher" Type="String" Nullable="false" MaxLength="50"
          Unicode="true" FixedLength="false" />
      <Property Name="Isbn" Type="String" MaxLength="18" Unicode="true"
          FixedLength="true" />
      <NavigationProperty Name="Authors" Relationship="BooksModel.BooksAuthors"
```

```
          FromRole="Books" ToRole="Authors" />
    </EntityType>
    <Association Name="BooksAuthors">
      <End Role="Authors" Type="BooksModel.Author" Multiplicity="*" />
      <End Role="Books" Type="BooksModel.Book" Multiplicity="*" />
    </Association>
  </Schema>
</edmx:ConceptualModels>
```

实体用 EntityType 元素定义，它包含 Key、Property 和 NavigationProperty 元素，以描述所创建的类的属性。Property 元素包含的属性描述设计器生成的类的.NET 特性的名称和类型。Association 元素连接 Author 和 Book 类型。Multiplicity="*"表示一个作者可以编写多本图书，一本图书可以由多个作者编写。

33.2.3 映射层

映射层使用 MSL(Mapping Specification Language，映射规范语言)把 CSDL 中的实体类型定义映射到 SSDL 上。下面的规范(代码文件 BooksDemo/BooksModel.edmx)包含一个 Mapping 元素，该元素包含 EntityTypeMapping 元素来引用 CSDL 的 Book 类型，并定义 MappingFragment 来引用 SSDL 中的 Authors 表。ScalarProperty 把包含 Name 特性的.NET 类的属性映射到数据库表中包含 ColumnName 特性的列上。

```
<edmx:Mappings>
  <Mapping Space="C-S"
      xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
    <EntityContainerMapping StorageEntityContainer="BooksModelStoreContainer"
        CdmEntityContainer="BooksEntities">
    <EntitySetMapping Name="Authors">
        <EntityTypeMapping TypeName="BooksModel.Author">
          <MappingFragment StoreEntitySet="Authors">
            <ScalarProperty Name="Id" ColumnName="Id" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <EntitySetMapping Name="Books">
        <EntityTypeMapping TypeName="BooksModel.Book">
          <MappingFragment StoreEntitySet="Books">
            <ScalarProperty Name="Id" ColumnName="Id" />
            <ScalarProperty Name="Title" ColumnName="Title" />
            <ScalarProperty Name="Publisher" ColumnName="Publisher" />
            <ScalarProperty Name="Isbn" ColumnName="Isbn" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <AssociationSetMapping Name="BooksAuthors" TypeName=
          "BooksModel.BooksAuthors" StoreEntitySet="BooksAuthors">
        <EndProperty Name="Authors">
          <ScalarProperty Name="Id" ColumnName="AuthorId" />
        </EndProperty>
        <EndProperty Name="Books">
```

```
        <ScalarProperty Name="Id" ColumnName="BookId" />
      </EndProperty>
    </AssociationSetMapping>
  </EntityContainerMapping>
 </Mapping>
</edmx:Mappings>
```

33.2.4 连接字符串

在设计器中，连接字符串存储在配置文件中。EDM 需要连接字符串，它不同于一般的 ADO.NET 连接字符串，因为需要映射信息。映射使用关键字 metadata 来定义，metadata 需要 3 个对象：

- metadata 关键字，带分隔符的映射文件列表
- 不变的提供程序名 Provider(该提供程序用于访问数据源)
- Provider connection string(用于指定依赖于提供程序的连接字符串)

下面的代码段显示了一个示例连接字符串。通过 metadata 关键字，带分隔符的映射文件列表引用 BooksModel.csdl、BooksModel.ssdl 和 BooksModel.msl 文件，它们包含在程序集中用 res:前缀定义的资源里。在 Visual Studio 中，设计人员只使用了一个文件 BooksModel.edmx，它包含 CSDL、SSDL 和 MSL。把 Custom Tool 属性设置为 EntityModelCodeGenerator，会创建包含在资源中的 3 个文件。

在 Provider connection String 设置中，可以使用连接字符串设置找到数据库的连接字符串。这部分与第 32 章讨论的简单的 ADO.NET 连接字符串相同，且取决于用 provider 设置的提供程序。

```
<connectionStrings>
  <add name="BooksEntities"
    connectionString="metadata=res://*/BooksModel.csdl|res://*/BooksModel.ssdl|
      res://*/BooksModel.msl;provider=System.Data.SqlClient;
      provider connection string=&quot;Data Source=(local);
      Initial Catalog=Books;Integrated Security=True;Pooling=False;
      MultipleActiveResultSets=True&quot;"
    providerName="System.Data.EntityClient" />
</connectionStrings>
```

利用连接字符串还可以指定没有在程序集中包含为资源的 CSDL、SSDL 和 MSL 文件。如果希望在部署项目后改变这些文件的内容，采用这种方式就很有用。

33.3 实体

实体类用设计器附带的 T4 模板创建。这些是简单的 POCO 类型，如下面代码中的 Book 类所示(代码文件 BooksDemo/BooksModel/BooksModel.tt/Book.cs)。

T4 是 Text Template Transformation Toolkit(文本模板转换工具集)的缩写，这个工具集允许从文本文件中创建源代码。在 Entity Framework 中，用于创建代码文件的文本文件是定义了映射的 XML 文件。

POCO 是 Plain Old CLR Objects 的缩写，POCO 对象只是带有属性的.NET 类，不需要派生自特定的基类。术语 plain old 来源于 POTS (Plain Old Telephone service)，后来演变为 POX (Plain Old XML)。

Authors 属性的类型是 ICollection<Author>，允许访问 Book 类的作者集合：

```
public partial class Book
{
  public Book()
  {
    this.Authors = new HashSet<Author>();
  }
  public int Id { get; set; }
  public string Title { get; set; }
  public string Publisher { get; set; }
  public string Isbn { get; set; }

  public virtual ICollection<Author> Authors { get; set; }
}
```

Book 实体类很容易使用对象上下文类 BookEntities 来访问。Books 属性返回一个可以迭代的 Book 对象的集合(代码文件 BooksDemo/Program.cs)：

```
using (var data = new BooksEntities())
{
  foreach (var book in data.Books)
  {
    Console.WriteLine("{0}, {1}", book.Title, book.Publisher);
  }
}
```

33.4 对象上下文

要从数据库中检索数据，需要使用 DbContext 类。这个类定义了从实体对象到数据库的映射。在核心 ADO.NET 中，这个类可以和填充 DataSet 的数据适配器相媲美。

设计器创建的 BookEntities 类派生自基类 DbContext。这个类添加了构造函数，来传递连接字符串。在默认的构造函数中，从配置文件中读取连接字符串。可以在配置文件中找到的连接字符串传递给基类的构造函数。

所创建的类定义 Books 和 Authors 属性，它们返回一个 ObjectSet<TEntity>。ObjectSet<TEntity> 派生自 ObjectQuery<TEntity>(代码文件 BooksDemo/BooksModel.Designer.cs)。

```
public partial class BooksEntities : DbContext
{
```

```
    public BooksEntities() : base("name=BooksEntities")
    {
    }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
      throw new UnintentionalCodeFirstException();
    }

    public DbSet<Author> Authors { get; set; }
    public DbSet<Book> Books { get; set; }
  }
```

DbContext 类给调用者提供了几个服务：

- 跟踪已经检索到的实体对象。如果再次查询该对象，就从对象上下文中提取它。
- 保存实体的状态信息。可以获得已添加、修改和删除对象的信息。
- 更新对象上下文中的实体，把改变的内容写入底层存储器中。

DbContext 类的方法和属性如表 33-2 所示。

表 33-2

DbContext 类的方法和属性	说 明
Database	这个属性返回一个与数据上下文关联的 Database 对象。有了这个对象，SQL 查询就可以直接在数据库上执行，创建事务，也可以动态创建数据库
Configuration	这个属性返回一个可用于配置数据上下文的 DbContextConfiguration 对象
ChangeTracker	这个属性返回一个 DbChangeTracker，该 DbChangeTracker 会跟踪检索到的实体对象和该对象在对象上下文中的变化
Set()	这个方法返回一个 DbSet 对象。前面的 Books 和 Authors 强类型化属性就使用这个方法返回 DbSet
Entry()	这个方法把实体对象作为参数，检索该参数，并返回一个 DbEntityEntry。有了 DbEntityEntry，就可以检索通过数据上下文为实体保存的信息，还可以再次从数据库中检索实际数据
SaveChanges() SaveChangesAsync()	从对象上下文中添加、修改和删除对象，不会改变底层存储器中的对象。使用 SaveChanges()方法可以把这些修改持久化到存储器中。SaveChangesAsync()是 Entity Framework6 中新增的，允许启动保存方法，而不会阻塞线程

DbSet 类的方法和属性如表 33-3 所示。

表 33-3

DbSet 类的方法和属性	说 明
Add() AddRange()	在集合中添加一个对象或一组对象
Attach()	Attach()方法把对象附加到集合中。附加的对象通过变更信息来跟踪
Create()	创建一个可以以后附加的新对象。创建新对象时，最好使用这个方法，而不是使用 new 操作符。该方法在后台创建一个派生自实体类的对象

(续表)

DbSet 类的方法和属性	说 明
Find() FindAsync()	在数据上下文中查找对象。如果对象不在上下文中，就从数据库中检索它
Remove()	从上下文中删除对象
AsNoTracking()	这个方法可以添加到查询中，不在上下文中跟踪被检索的对象

SaveAsync 和 FindAsync 是 Entity Framework 6 新增的方法。这个库添加了更多的异步选项，但不能在同一个数据上下文中同时调用多个异步方法(使用多个线程或任务)。数据上下文不是线程安全的，无论使用什么计算机，迟早会出现异常。在调用下一个异步方法之前，需要使用多个数据上下文对象，或者使用 await 。异步编程参见第 13 章。

33.5 关系

实体类型 Book 和 Author 相互关联。一本书可以由一个或多个作者编写，一个作者也可以编写一本或多本书。对应关系基于关联类型的个数和多样性。ADO.NET Entity Framework 支持几种关系，这里介绍其中一些关系，包括 TPT(Table per Type，一种类型一个表)和 TPH(Table per Hierarchy，一个层次结构一个表)。多样性可以是一对一、一对多和多对多。

33.5.1 一个层次结构一个表

在 TPH 中，数据库中的一个表对应实体类的一个层次结构。数据库表 Payments(如图 33-3 所示)包含的列对应于实体类型的一个层次结构。一些列在该层次结构中由所有实体共享，如 Id 和 Amount，Number 列仅用于信用卡和支票付费。

全部映射到同一个 Payments 表的实体类如图 33-4 所示。Payment 是一个抽象基类，它包含的属性用于该层次结构中的所有类型。派生自 Payment 基类的具体类有 CreditCardPayment、CashPayment 和 ChequePayment。除了基类的属性之外，CreditCardPayment 类还有一个 CreditCard 属性，ChequePayment 类还有 BankName 和 BankAccount 属性。

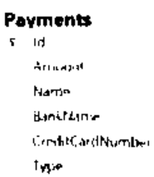


图 33-3

所有这些映射都可以使用设计器来定义。映射细节可以使用 Mapping Details 窗口配置，如图 33-5 所示。具体类的类型选择基于一个 Condition 元素，Condition 元素用 Maps to Payments When Type = CREDITCARD 定义。基于 Type 列的值选择对应的类型。还可以使用其他用于选择类型的选项，例如，可以验证某一列是否不为空。

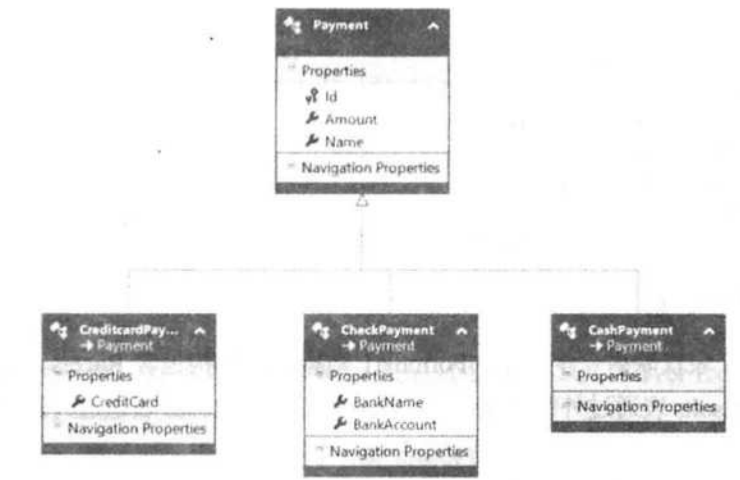


图 33-4

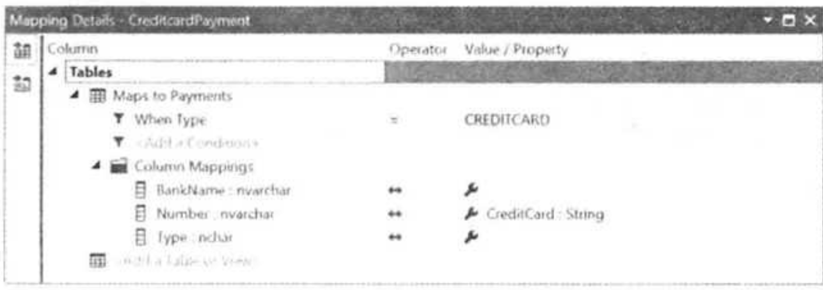


图 33-5

现在，可以迭代 Payments 表中的数据，根据映射返回不同的类型(代码文件 PaymentsDemo/Program.cs)：

```
using (var data = new PaymentsEntities())
{
  foreach (var p in data.Payments)
  {
    Console.WriteLine("{0}, {1} - {2:C}", p.GetType().Name, p.Name,
    p.Amount);
  }
}
```

运行应用程序，会从数据库中返回两个 CashPayment 对象和一个 CreditCardPayment 对象：

```
CreditCardPayment, Gladstone - $22.00
CashPayment, Donald - $0.50
CashPayment, Scrooge - $80,000.00
```

使用 OfType 方法，可以很容易地从指定的类型中获取结果：

```
foreach (var p in data.Payments.OfType<CreditcardPayment>())
{
  Console.WriteLine("{0} {1} {2}", p.Name, p.Amount, p.CreditCard);
}
```

从这个查询中生成的 T-SQL 语句非常高效地用 WHERE 子句过滤类型，因为它在模型中定义：

```
SELECT
'0X0X' AS [C1],
[Extent1].[Id] AS [Id],
[Extent1].[Amount] AS [Amount],
[Extent1].[Name] AS [Name],
[Extent1].[Number] AS [Number]
FROM [dbo].[Payments] AS [Extent1]
WHERE [Extent1].[Type] = N'CREDITCARD'
```

33.5.2 一种类型一个表

在 TPT 中，一个表仅映射一个类型。Formula1 数据库的架构包含 Racers、RaceResults、Races 和 Circuits 表。RaceResults 表通过外键 RacerId 与 Racers 表关联，Races 表通过外键 CircuitId 与 Circuits 表关联。

图 33-6 显示了实体类型 Racers、RaceResults、Races 和 Circuits。其中有几个一对多关系。

图 33-6

下面的代码块用两个迭代访问赛手及其比赛结果(Formula1Demo/Program.cs)。首先访问 Racer 对象，把 FirstName 和 LastName 属性的值写入控制台中。接着使用 Racer 类的 RaceResults 属性访问所有比赛结果。相关结果通过懒惰加载方式来访问属性，因为在 DbContext 中，把 data.ConfigurationOptions.LazyLoadingEnabled 属性设置为 true。

```
using (var data = new Formula1Entities())
{
  foreach (var racer in data.Racers)
  {
    Console.WriteLine("{0} {1}", racer.FirstName, racer.LastName);
    foreach (var raceResult in racer.RaceResults)
    {
```

```
        Console.WriteLine("\t{0} {1:d} {2}",
        raceResult.Race.Circuit.Name,
        raceResult.Race.Date, raceResult.Position);
      }
    }
  }
```

发送给数据库的 SQL 语句很容易通过不同的方法来验证：

- IntelliTrace：通过 IntelliTrace，ADO.NET 会记录每个发送来的 SQL 语句。这需要 Visual Studio Ultimate。
- SQL Server Profiler：它位于 SQL Server 中，不包含在 SQL Server Express 中。
- 用 Entity Framework 记录。

Entity Framework 6 将 Action<string>赋予 DbContext.Database.Log 属性，就提供了简单的日志功能。例如：

```
data.Database.Log = Console.Write;
```

利用这个功能，可以把 SQL 语句写入任何位置。把 Console.Write 方法赋予 Log 属性，会把日志信息写入控制台。同样，可以把日志信息写入跟踪源、或其他设备。

懒惰加载在后台是如何工作的？RaceResults 属性声明为 ICollection<RaceResult>类型：

```
public virtual ICollection<RaceResult> RaceResults { get; set; }
```

RaceResults 类型的实例在 Racer 类型的构造函数内部创建，是一个简单的 HashSet<T>：

```
public Racer()
{
  this.RaceResults = new HashSet<RaceResult>();
}
```

HashSet<T>包含未排序的一组不同元素。HashSet<T>类参见第 10 章。

懒惰加载的关键是，RaceResults 属性声明为 virtual。在后台会创建一个派生自 Racer 基类的代理类，它重写了 RaceResults 属性。有了这些，第一次迭代 RaceResults 属性的集合时，就从数据库中查询比赛结果。

33.5.3　懒惰加载、延迟加载和预先加载

在设计器的默认设置中，设计器把 ContextOptions 的 LazyLoadingEnabled 属性设置为 true，根据请求懒惰加载(lazy loaded)关系。关系的加载还有其他选项。关系也可以预先加载(eager loaded)或延迟加载(delayed loaded)。

预先加载是指，在加载父对象的同时加载关系。在添加对 Include()方法的调用后，立即加载比赛结果、与比赛结果相关的比赛以及与比赛相关的赛道。Include()方法可以通过 ObjectSet<TEntity>类型来使用(Racers 属性的类型是 ObjectSet<Racer>)，接收关系名称。在 foreach 循环中访问

RaceResults 属性，就可以查看所有信息，直到赛道信息为止：

```
foreach (var racer in data.Racers.Include("RaceResults.Race.Circuit"))
{
  Console.WriteLine("{0} {1}", racer.FirstName, racer.LastName);
  foreach (var raceResult in racer.RaceResults)
  {
    Console.WriteLine("\t{0} {1:d} {2}",
      raceResult.Race.Circuit.Name,
      raceResult.Race.Date, raceResult.Position);
  }
}
```

预先加载的优点是如果需要所有相关的对象，则对数据库的请求会比较少。当然，如果并不需要所有相关的对象，懒惰加载或延迟加载会比较适合。

DbSet 类定义了 Include 方法，它的字符串参数定义了应预先加载的关系。Entity Framework 6 提供了一个扩展方法 Include，它扩展了 IQueryable<T>。这个扩展方法接受一个 Func<T, TProperty>类型的参数，以传递 lambda 表达式，而不是字符串，用于处理只访问一个属性关系的情形。其优点是类型安全性——编译器可以检查关系。例如 Include(r =>r.RaceResults)可以使用 Include(“RaceResults”)替代。这个扩展方法在 System.Data.Entity 名称空间中定义。

显式加载需要对 Load()方法的显式调用。下面的代码调用 DbSet<RaceResult>的 Load()方法，从数据库中获取所有比赛结果。加载是立即开始的：

```
data.RaceResults.Load();
```

为了加载一个赛手的相关比赛结果，可以使用 DbCollectionEntry 的 Load()方法。为了获得 DbEntityEntry，可以调用 DbSet 的 Entry 方法。DbEntityEntry 提供了 Collection 方法，可以给它传递带有 ICollection 属性类型的关联名称。通过这个方法，可以使用 IsLoaded 属性验证集合是否已加载。如果集合未加载，就调用 Load 方法，显式加载相关的对象：

```
DbCollectionEntry entry =
  data.Entry(racer).Collection("RaceResults");
if (!entry.IsLoaded)
{
  entry.Load();
}
```

除了传递容易出错的字符串之外，Collection 方法还提供了一个重载版本，以传递一个 lambda 表达式：

```
DbCollectionEntry<Racer, RaceResult> entry =
  data.Entry(racer).Collection(r => r.RaceResults);
if (!entry.IsLoaded)
{
  entry.Load();
```

```
}
```

33.6 查询数据

Entity Framework 提供了几种查询数据的方式：Entity SQL(是 T-SQL 的扩展)；使用辅助方法创建 Entity SQL 以及 LINQ。下面就讨论这些方式。

33.6.1 Entity SQL

Entity SQL 通过添加类型，增强了 T-SQL。这种语法不需要连接语句，因为可以使用实体的关联来替代。使用一个低级的 API——EntityClient，就可以访问 Entity Framework。这个 API 实现为一个 ADO.NET 提供程序，EntityClient 提供了 EntityConnection、EntityCommand、EntityParameter 和 EntityDataReader 类，它们分别派生自基类 DbConnection、DbCommand、DbParameter 和 DbDataReader。

使用这些类的方式与第 32 章使用 ADO.NET 类的方式相同，但需要一个特殊的连接字符串，且使用 Entity SQL 代替 T - SQL，来访问 EDM。

与数据库的连接通过 EntityConnection 类来实现，它需要一个实体连接字符串。这个字符串通过 System.Configuration 名称空间中的 ConfigurationManager 类从配置文件中读取。EntityConnection 类的 CreateCommand()方法返回一个 EntityCommand。EntityCommand 的命令文本使用 CommandText 属性来指定，且需要 Entity SQL 语法。Formula1Entities.Racers 从 Formula1Entities CSDL 定义的 EntityContainer 元素中生成，Racers EntitySet 从 Racers 表中获取所有赛手。command.ExecuteReader 返回一个数据读取器，以逐行读取数据(代码文件 QueryDemo/Program.cs)：

```
string connectionString =
   ConfigurationManager.ConnectionStrings["Formula1Entities"]
   .ConnectionString;
var connection = new EntityConnection(connectionString);
await connection.OpenAsync();
EntityCommand command = connection.CreateCommand();
command.CommandText = "[Formula1Entities].[Racers]";
DbDataReader reader = await command.ExecuteReaderAsync(
   CommandBehavior.SequentialAccess | CommandBehavior.CloseConnection);
while (await reader.ReadAsync())
{
 Console.WriteLine("{0} {1}", reader["FirstName"], reader["LastName"]);
}
reader.Close();
```

前面的示例使用 ConfigurationManager 类从配置文件中读取连接字符串。要使用这个类，不仅需要导入 System.Configuration 名称空间，还必须引用 System.Configuration 程序集。

下面讨论更多的 Entity SQL 语法选项。这里仅列出了几个语法选项，它们有助于开始使用 Entity SQL。在 MSDN 文档中有完整的参考资料。

前面的示例说明了 Entity SQL 如何在 EntityContainer 和 EntitySet 中使用 CSDL 中的定义，例如，

使用 Formula1Entities.Racers 可以从 Racers 表中获取所有赛手。

除了检索所有列之外，还可以使用 EntityType 的 Property 元素。这看起来非常类似于上一章使用的 T-SQL 查询：

```
EntityCommand command = connection.CreateCommand();
command.CommandText =
   "SELECT Racers.FirstName, Racers.LastName FROM Formula1Entities.Racers";
DbDataReader reader = await command.ExecuteReaderAsync(
   CommandBehavior.SequentialAccess | CommandBehavior.CloseConnection);
while (await reader.ReadAsync())
{
  Console.WriteLine("{0} {1}", reader.GetString(0), reader.GetString(1));
}
reader.Close();
```

Entity SQL 中没有 SELECT *。前面通过请求 EntitySet 来检索所有列。使用 SELECT VALUE 还可以获得所有列，如下一段代码所示。这条语句还使用一个包含 WHERE 的筛选器，通过查询获取特定的出版商。注意 CommandText 用@字符指定参数，但添加到 Parameters 集合中的参数不使用@字符把某个值写入相同的参数中：

```
EntityCommand command = connection.CreateCommand();
command.CommandText =
  "SELECT VALUE it FROM [Formula1Entities].[Racers] AS it " +
  "WHERE it.Nationality = @Country";
command.Parameters.AddWithValue("Country", "Austria");
```

下面修改对象上下文和映射功能。

33.6.2 使用 DbSqlQuery

使用 DbSqlQuery<T>类可以定义 SQL 查询语句。SqlQuery 方法允许通过第一个参数传递 SQL 语句。第二个参数的类型是 params object[]，它允许把任意多个参数传递给 SQL 语句。这些参数的类型是 SqlParameter，如下所示：

```
DbSqlQuery<Racer> racers = data.Racers.SqlQuery(
  "SELECT * FROM Racers WHERE nationality = @country",
  new SqlParameter("country", country));
```

除了把 SqlParameter 对象传递为参数之外，还可以指定简单变量——假定遵循参数命名约定@p#：

```
DbSqlQuery<Racer> racers = data.Racers.SqlQuery(
  "SELECT * FROM Racers WHERE nationality = @p0", country);
```

从 SqlQuery 方法返回的对象利用数据上下文来跟踪。如果不应跟踪对象，就可以使用 AsNoTracking 方法：

```
DbSqlQuery<Racer> racers = data.Racers.SqlQuery(
  "SELECT * FROM Racers WHERE nationality = @p0", country)
  .AsNoTracking();
```

33.6.3　LINQ to Entities

本书的几章介绍了 LINQ to Query 对象、数据库和 XML。当然，LINQ 也能用于查询实体。在 LINQ to Entities 中，LINQ 查询的数据源是 DbQuery<T>类。因为 DbQuery<T>类实现了 IQueryable 接口，所以用于查询的扩展方法用 System.Linq 名称空间中的 Queryable 类定义。用这个类定义的扩展方法有一个参数 Expression<T>，这就是编译器把表达式树写入程序集的原因。第 11 章介绍了表达式树，表达式树从 DbQuery<T>类中解析到 SQL 查询中。

可以使用如下简单的 LINQ 查询返回赢得超过 40 场比赛的赛手(代码文件 QueryDemo/Program.cs)：

```
using (var data = new Formula1Entities())
{
  var racers = from r in data.Racers
               where r.Wins > 40
               orderby r.Wins descending
               select r;
  foreach (Racer r in racers)
  {
    Console.WriteLine("{0} {1}", r.FirstName, r.LastName);
  }
}
```

访问 Formula1 数据库的结果如下：

```
Michael Schumacher
Alain Prost
Ayrton Senna
```

还可以定义一个 LINQ 查询来访问关系，如下所示。变量 r 表示赛手，变量 rr 表示所有比赛结果。筛选器用 where 子句定义，只检索榜上有名的瑞士赛手。要完成这个排行榜，应组合结果，并计算排行榜的个数。根据排行榜的完成情况进行排序：

```
using (var data = new Formula1Entities())
{
  var query = from r in data.Racers
              from rr in r.RaceResults
              where rr.Position <= 3 && rr.Position >= 1 &&
                 r.Nationality == "Switzerland"
              group r by r.Id into g
              let podium = g.Count()
              orderby podium descending
              select new
              {
                Racer = g.FirstOrDefault(),
                Podiums = podium
              };
  foreach (var r in query)
  {
    Console.WriteLine("{0} {1} {2}", r.Racer.FirstName, r.Racer.LastName,
    r.Podiums);
  }
}
```

运行这个应用程序，会返回瑞士的3个赛手姓名：

```
Clay Regazzoni 28
Jo Siffert 6
Rudi Fischer 2
```

33.7 把数据写入数据库

读取、搜索和筛选数据库中的数据仅是数据密集型应用程序通常需要执行的一部分操作。把改变的数据写回存储器是另一个要执行的操作。下面几节介绍如下主题：对象跟踪、对象上下文的服务和基础、对象上下文如何得知对象的变化，如何在对象上下文中附加和分离对象，对象上下文如何使用对象的状态来存储实体对象。

33.7.1 对象跟踪

为了修改和保存从存储器中读取的数据，必须在加载实体后跟踪它们。这也要求对象上下文注意实体是否已从存储器中加载了。如果多个查询同时访问同一个记录，对象上下文就需要返回已经加载的实体。对象上下文用 DbChangeTracker 来跟踪加载到上下文中的实体。

下面的示例说明了，如果两个不同的查询从数据库中返回相同的记录，状态管理器就会注意到这一点，因此它不新建实体，而是返回相同的实体。

两个不同的查询用于返回一个实体对象。第一个查询获得来自奥地利、姓氏为 Lauda 的第一个赛手。第二个查询请求来自奥地利的赛手，按照赢得比赛的次数排列赛手，并获取第一个结果。事实上，这是同一个赛手。为了验证返回了同一个实体对象，使用 Object.ReferenceEquals()方法验证两个对象引用是否确实引用同一个实例(代码文件 Formula1Demo/Program.cs)。

```
private static void TrackingDemo()
{
  using (var data = new Formula1Entities())
  {
    Racer niki1 = (from r in data.Racers
                   where r.Nationality == "Austria" && r.LastName == "Lauda"
                   select r).First();
    Racer niki2 = (from r in data.Racers
                   where r.Nationality == "Austria"
                   orderby r.Wins descending
                   select r).First();
    if (Object.ReferenceEquals(niki1, niki2))
    {
      Console.WriteLine("the same object");
    }
  }
}

private static void ObjectStateManager_ObjectStateManagerChanged(
    object sender, CollectionChangeEventArgs e)
{
  Console.WriteLine("Object State change — action: {0}", e.Action);
  Racer r = e.Element as Racer;
```

```
    if (r != null)
      Console.WriteLine("Racer {0}", r.LastName);
  }
```

运行这个应用程序，会看到 niki1 和 niki2 引用的确相同：

```
The same object
```

33.7.2 改变信息

数据上下文也会注意到实体的改变。下面的示例添加并修改数据上下文中的一个赛手，并获得修改的信息。首先，使用 DbSet<T>类的 Add()方法添加一个新赛手，这个方法用 EntityState.Added 信息添加一个新实体。接着查询 Lastname 为 Alonso 的赛手。在这个实体类中，递增 Starts 和 Wins 属性，从而用 EntityState.Modified 信息标记实体。

为了获得所有添加或修改的实体对象，可以调用 DbChangeTracker 的 Entries()方法。这个方法返回一组被跟踪的 ObjectStateEntry 对象，使用 State 属性可以检查这些对象的状态。

在示例代码中，对于状态为 EntityState.Modified 的对象，使用 OriginalValues 和 CurrentValues 属性检索初始值和当前值(代码文件 Formula1Demo/Program.cs)：

```
private static void ChangeInformation()
{
  using (var data = new Formula1Entities())
  {
    var esteban = data.Racers.Create();
    esteban.FirstName = "Esteban";
    esteban.LastName = "Gutierrez";
    esteban.Nationality = "Mexico";
    esteban.Starts = 0;
    data.Racers.Add(esteban);

    Racer fernando = data.Racers.Where(
      r => r.LastName == "Alonso").First();
    fernando.Wins++;
    fernando.Starts++;

    foreach (DbEntityEntry<Racer> entry in
      data.ChangeTracker.Entries<Racer>())
    {
      Console.WriteLine("{0}, state: {1}", entry.Entity, entry.State);
      if (entry.State == EntityState.Modified)
      {
        Console.WriteLine("Original values");
        DbPropertyValues values = entry.OriginalValues;
        foreach (string propName in values.PropertyNames)
        {
          Console.WriteLine("{0} {1}", propName, values[propName]);
        }
        Console.WriteLine();
        Console.WriteLine("Current values");

        values = entry.CurrentValues;
        foreach (string propName in values.PropertyNames)
        {
```

```
            Console.WriteLine("{0} {1}", propName, values[propName]);
        }
    }
}
```

运行这个应用程序，会显示已添加和修改的赛手，已改变的属性在它们的原始值和当前值中显示，如下所示：

```
Esteban Gutierrez, state: Added
Fernando Alonso, state: Modified
Original values
Id 28
FirstName Fernando
LastName Alonso
Nationality Spain
Starts 208
Wins 32
Points
DateOfBirth 81/29/7
DateOfDeath

Current values
Id 28
FirstName Fernando
LastName Alonso
Nationality Spain
Starts 209
Wins 33
Points
DateOfBirth 81/29/7
DateOfDeath
current: 182
```

33.7.3 附加和分离实体

把实体数据返回给调用者，对于从对象上下文中分离对象很重要。例如，如果实体对象从 Web 服务中返回，这就是必须的。这里，如果在客户端上改变实体对象，对象上下文并不知道对应的改变。

在示例代码中，从数据库中检索一个赛手，但在检索对象时没有通过上下文使用 AsNoTracking 方法跟踪它。修改对象的属性，数据上下文不知道这个对象属性进行了修改，通过数据库的上下文写入实体的更改就没有效果：

```
using (var data = new Formula1Entities())
{
  IQueryable<Racer> racers = data.Racers.Where(
    r => r.LastName == "Alonso").AsNoTracking();
  Racer fernando = racers.First();

  // Racer is detached and can be changed independent of the
  // data context
  fernando.Starts++;
```

以后，可以使用 Attach 方法把对象附加到上下文上。但是，仅附加它，状态跟踪器对状态的更改仍一无所知。使用 DbEntityEntry 设置状态，就可以把状态手动设置为需要的状态：

```
data.Racers.Attach(fernando);
data.Entry<Racer>(fernando).State = EntityState.Modified;
```

33.7.4 使用最后一个更改操作写入实体的更改

在状态跟踪器的帮助下，根据所有的变化信息，可以使用 DbContext 类的 SaveChanges()方法，把添加、删除和修改的实体对象写到存储器中。SaveChanges()方法返回写入的实体对象的个数。

如果数据库中用实体类表示的记录在读取之后发生了变化，该怎么办？为了模拟这个场景，创建一个方法，它在读写数据之间包含一个延迟时间。数据访问代码封装在一个处理 DbUpdateConcurrency 异常的 try/catch 语句中：

```
private async static Task UserSaveObjectLastWins(string user, int delay, int starts)
{
  using (var data = new Formula1Entities())
  {
    int changes = 0;
    try
    {
      Racer r1 = data.Racers.Where(r => r.LastName == "Alonso").First();
      r1.Starts = starts;
      await Task.Delay(delay);
      changes = data.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
      Console.WriteLine("{0} error: {1}", user, ex.Message);
    }
    Console.WriteLine("{0} changed {1} record(s)", user, changes);
  }
}
```

启动两个有不同延迟时间的任务，就可以同时运行同一个更新方法。这两个任务完成后，再次从数据库中读取记录：

```
private async static Task SaveObjectsLastWins()
{
  Task t1 = Task.Run(() => UserSaveObjectLastWins("A", 1000, 200));
  Task t2 = Task.Run(() => UserSaveObjectLastWins("B", 2000, 220));
  await Task.WhenAll(t1, t2);
  ReadRecord();
}
```

结果如下所示。无论第一个用户是否修改了同一个记录，第二次更新都会发生。无论数据库中的当前信息是什么，第二次更新都会发生。这个行为称为“应用最后一个更改操作”：

```
A changed 1 record(s)
B changed 1 record(s)
After the update, the new value is 220
```

33.7.5 使用第一个更改操作写入实体的更改

接着修改并发操作的结果，不应用最后一个更改操作。第二次更新应导致一个错误。为此，应

存储所读取的初始值，用数据库中的实际数据验证这个数据。如果数据库中的数据不再相同，更改操作就失败。

在设计器中很容易通过修改 ConcurrencyMode 属性来完成这个操作。对于实体对象的每个属性，可以把 ConcurrencyMode 配置为 Fixed 或 None。Fixed 值表示属性在写入时进行验证，确保该值没有写入时改变。默认值 None 表示忽略任何更改。如果一些属性配置为 Fixed 模式，数据在读取实体对象和写入实体对象的中间阶段发生改变，就抛出一个 DbUpdateConcurrencyException 异常。

与修改 Racer 对象的 Starts 属性一样，新的结果如下：

```
A changed 1 record(s)
B error: Store update, insert, or delete statement affected an unexpected
Number of rows (0). Entities may have been modified or deleted since
entities were loaded. Refresh ObjectStateManager entries.
B changed 0 record(s)
after the update, the new value is 190
```

33.7.6　写入实体的更改并处理冲突

除了仅应用第一个或最后一个更改操作之外，如果更改操作相互冲突，就可以从数据库中读取实际的值，让用户决定是使用用户输入的数据，还是数据库中的当前数据。当然，在实现这个操作之前，应确定这种情形的发生次数，以及是否值得花时间实现这个操作。该实现可能没有这里介绍的这么简单，因为在更新操作中也可能需要处理关系。另外，用户可能不清楚如何回答这个问题。也许用户仅单击 OK，退出对话框，而根本不看其内容。

下面看看如何解决这个问题。也可以使用自动解决器，例如，如果修改了不同的属性，或者对更新的字段使用要点演绎系统的机制。

下面的代码像前面那样进行更新，但异常的处理有所不同。如果抛出了 DbUpdateConcurrencyException 异常，ex.Entries 属性就包含所有失败项。对于每个 DbEntityEntry，对象的当前值(包含用户更改的对象)用 CurrentValues 属性访问。如果需要通过当前数据上下文检索出来的值，就使用 OriginalValues 属性。要获得数据库中的当前值，就可以调用 GetDatabaseValues 方法。利用这个信息可以解决冲突。如前所述，一个选项是询问用户，并提供这个消息，获取用户的解析值。这里作为默认的解析器，复制数据库中的值。解析器的结果是，需要设置初始值和当前值，所以 SaveChanges 下次会成功：

```
using (var data = new Formula1Entities())
{
  int changes = 0;
  bool saveFailed = false;

  do
  {
  try
  {
    saveFailed = false;
    Racer r1 = data.Racers.Where(r => r.LastName == "Alonso").First();
    r1.Starts = starts;
    await Task.Delay(delay);
    changes = data.SaveChanges();
  }
```

```
    catch (DbUpdateConcurrencyException ex)
    {
      saveFailed = true;

      Console.WriteLine("{0} error {1}", user, ex.Message);
      foreach (var entry in ex.Entries)
      {
        DbPropertyValues currentValues = entry.CurrentValues;
        DbPropertyValues databaseValues = entry.GetDatabaseValues();
        DbPropertyValues resolvedValues = databaseValues.Clone();

        AskUser(currentValues, databaseValues, resolvedValues);


        entry.OriginalValues.SetValues(databaseValues);
        entry.CurrentValues.SetValues(resolvedValues);
      }
    }
  }
  while (saveFailed);

  Console.WriteLine("{0} changed {1} record(s)", user, changes);
```

33.8 使用 Code First 编程模型

除了使用模型设计器之外，还可以创建实体类型，用代码定义完整的映射。在 Code First 中，根本就没有包含 CSDL、SSDL 和 MSL 的映射定义，而可以使用基于约定的映射。Code First 使用基于约定的编程方式，类似于 ASP.NET MVC。在基于约定的编程方式中，约定在配置之前使用。例如，不是使用属性或配置文件来定义主键，而只需要用 Id 命名属性，或者名称需要以 Id 结尾，例如 BooksId。这种属性会自动映射到主键上。

本节讨论如何为 Code First 定义实体类型，创建对象上下文，定制所创建的数据库映射。

33.8.1 定义实体类型

对于这个示例，要定义两个实体类型 Menu 和 MenuCard，如图 33-7 所示。一个 Menu 关联一个 MenuCard，MenuCard 包含对卡片中所有 Menu 的引用。

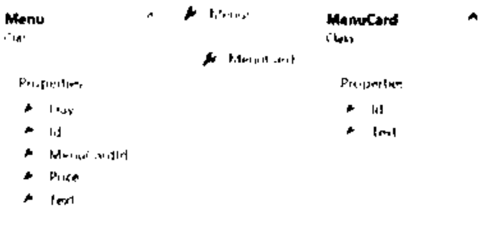


图 33-7

Menu 类的定义如下所示(代码文件 CodeFirst/Menu.cs)。其中没有到数据库键的特定映射，也没有任何其他数据库专用的定义。它只是一个约定。因为一个属性的名称是 Id，所以从这个属性中创建一个主键。把该属性命名为 MenuId，也是有效的。MenuCard 属性的类型是 MenuCard。这将是

一个关系：

```
public class Menu
{
  public int Id { get; set; }
  public string Text { get; set; }
  public decimal Price { get; set; }
  public DateTime? Day { get; set; }
  public MenuCard MenuCard { get; set; }
  public int MenuCardId { get; set; }
}
```

MenuCard类看起来很相似，它有一个Menus属性，允许访问与MenuCard关联的Menu对象(代码文件CodeFirst/MenuCard.cs)：

```
public class MenuCard
{
  public int Id { get; set; }
  public string Text { get; set; }
  public virtual ICollection<Menu> Menus { get; set; }
}
```

33.8.2 创建数据上下文

现在需要一个数据上下文。MenuContext派生于基类DbContext，为表定义了属性，其方式与前面的POCO对象相同(代码文件CodeFirst/MenuContext.cs)：

```
public class MenuContext : DbContext
{
  public MenuContext()
  {
  }
  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuCard> MenuCards { get; set; }
}
```

33.8.3 创建数据库，存储实体

现在可以使用数据上下文了。下面的示例代码(代码文件 CodeFirst/Program.cs)添加了一个MenuCard对象和两个菜单项。接着调用DbContext的SaveChanges方法，把这些项写入数据库：

```
using (var data = new MenuContext())
{
  MenuCard card = data.MenuCards.Create();
  card.Text = "Soups";
  data.MenuCards.Add(card);

  Menu m = data.Menus.Create();
  m.Text = "Baked Potato Soup";
  m.Price = 4.80M;
  m.Day = new DateTime(2012, 9, 20);
  m.MenuCard = card;
  data.Menus.Add(m);
  Menu m2 = data.Menus.Create();
```

```
        m2.Text = "Cheddar Broccoli Soup";
        m2.Price = 4.50M;
        m2.Day = new DateTime(2012, 9, 21);
        m2.MenuCard = card;
        data.Menus.Add(m2);
        try
        {
          data.SaveChanges();
        }
        catch (Exception ex)
        {
          Console.WriteLine(ex.Message);
        }
      }
    }
```

对于前面的代码，从来都没有指定连接字符串，也没有创建数据库。如果数据库不存在，就会自动创建。

33.8.4 数据库

如果数据库不存在，就会自动创建。数据库默认使用服务器名称来创建——例如在 SQL Express LocalDB 中，其名称是(localdb)\v11.0。数据库的名称指定为数据上下文的名称，包括名称空间。在示例应用程序中，数据库的名称是 CodeFirstDemo.MenuContext。数据库中创建的表及其属性和关系如图 33-8 所示。对于 Menus 和 MenuCard 之间的多对一关系，在 Menus 表中创建了一个外键 MenuCard_Id。Menus 表中的 Day 列定义为允许使用空值，因为实体类型 Menu 把这个属性定义为可空。Text 列用 nvarchar(max)定义，允许为空，因为字符串是一个引用类型。Price 是一个数据库浮点数类型，不允许为空。值类型是必需的，但如果使用了可空的值类型，它们就是可选的。

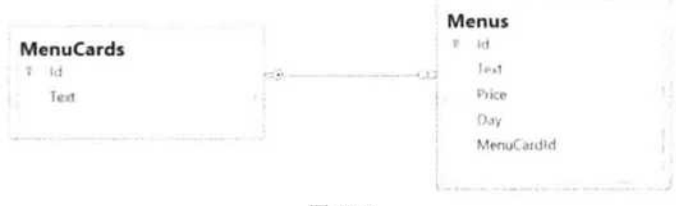


图 33-8

33.8.5 查询数据

下面的示例(代码文件 CodeFirst/Program.cs)演示了如何从数据库中读取数据。创建上下文后，就在外部的 foreach 循环中访问菜单卡，而内部的 foreach 循环查询菜单。当然，也可以使用 LINQ 查询来访问数据，LINQ 查询由上下文转换为 T-SQL：

```
using (var data = new MenuContext())
{
  foreach (var card in data.MenuCards)
  {
```

```
    Console.WriteLine("{0}", card.Text);
    foreach (var menu in card.Menus)
    {
      Console.WriteLine("\t{0} {1:d}", menu.Text, menu.Day);
    }
  }
}
```

DbContext 默认启用了懒惰加载功能。因此先查询菜单卡；为了在每个请求中查询菜单卡中的菜单，创建了检索菜单的 SELECT 语句。与前面的情况类似，也可以进行早期加载：

```
data.Configuration.LazyLoadingEnabled = false;
foreach (var card in data.MenuCards.Include("Menus"))
{
  Console.WriteLine("{0}", card.Text);
  foreach (var menu in card.Menus)
  {
    Console.WriteLine("\t{0} {1:d}", menu.Text, menu.Day);
  }
}
```

33.8.6 定制数据库的生成

可以进行一些简单的定制来生成数据库，例如定义数据库连接字符串。下面的代码(代码文件 CodeFirst/MenuContext.cs)在 MenuContext 类中把数据库连接字符串赋予基类 DbContext 的构造函数。要定义连接字符串，可以创建有指定名称的数据库：

```
public class MenuContext : DbContext
{
  private const string connectionString =
      @"server=(localdb)\v11.0;database=WroxMenus;" +
      "trusted_connection=true";
  public MenuContext()
      : base(connectionString) { }
```

1. 数据注释

为了定制已生成的表，可以使用 System.ComponentModel.DataAnnotations 名称空间中的一些属性。下面的示例代码(代码文件 CodeFirst/ Menu.cs)使用了 StringLengthAttribute 类型，这样，生成的列类型是 nvarchar(50)，而不是 nvarchar(max)：

```
public class Menu
{
  public int Id { get; set; }
  [StringLength(50)] public string Text { get; set; }
  public double Price { get; set; }
  public DateTime? Day { get; set; }
  public MenuCard MenuCard { get; set; }
}
```

可用于定制实体的其他属性有：

- Key 定义名称中没有 Id 的其他列；

- TimeStamp 把属性定义为行版本的列；
- ConcurrencyCheck 使用属性进行乐观并行；
- Association 给关系标记属性。

2. 模型构建器

仅使用属性来定制生成的表和列，会很快达到其极限。这里有一个更灵活的选项：使用模型构建器。模型构建器提供了一个流畅的 API，来定制表、列和关系。

使用 Code First 开发模型，可以使用 DbModelBuilder 类访问模型构建器的功能。有了一个从基类 DbContext 中派生的上下文类，就可以重写方法 OnModelCreating。这个方法在创建数据库模型时调用。使用这个方法，会返回一个 DbModelBuilder 类型的模型构建器。有了这个模型构建器，可以重新命名属性、修改属性的类型，定义约束，构建关系，进行许多其他的定制。

下面的代码段(代码文件 CodeFirst/MenuContext.cs)把 Menus 表中的 Price 列改为 money 类型，Day 列改为 date 类型。Text 列的最大字符串长度改为 40 个字符，且把 Text 列设置为必选列。模型构建器使用了一个流畅的 API。方法 Entity 返回一个 EntityTypeConfiguration。有了方法 Entity 的结果，就可以使用 EntityTypeConfiguration 的其他方法。在包含 Property 方法的 lambda 表达式中，把 Price 属性用作结果，这个方法就返回一个 DecimalPropertyConfiguration。有了它，就可以调用方法 HasColumnType，它把数据库列的类型设置为 money。HasColumnType 方法再次返回一个 DecimalPropertyConfiguration，可以继续下去，调用配置列的其他方法。

如前面 Menu 类中的 MenuCard 属性所示，默认创建了一个名为 MenuCard_Id 的外键。现在扩展 Menu 类，给它添加 int 类型的属性 MenuCardId。要把这个属性用作外键，应使用 HasForeignKey 方法，把外键赋予属性 MenuCardId。方法 OnModelCreating 中的最后一个语句在 MenuCards 表中设置了级联删除功能。如果删除了 MenuCards 表中的一行，具有该 ID 的所有菜单都应删除。

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Menu>().Property(m => m.Price).HasColumnType("money");
  modelBuilder.Entity<Menu>().Property(m => m.Day).HasColumnType("date");
  modelBuilder.Entity<Menu>().Property(m => m.Text).HasMaxLength(40)
    .IsRequired();
  modelBuilder.Entity<Menu>().HasRequired(m => m.MenuCard)
    .WithMany(c => c.Menus).HasForeignKey(m => m.MenuCardId);
  modelBuilder.Entity<MenuCard>().Property(c => c.Text).HasMaxLength(30)
    .IsRequired();
  modelBuilder.Entity<MenuCard>().HasMany(c => c.Menus).WithRequired()
    .WillCascadeOnDelete();
}
```

33.8.7 数据库的自动填充

使用 Code First，可以自动创建数据库，也可以给数据库自动填充数据。为此，只需一个给数据库填充数据的初始化器。数据库初始化器实现了 IDatabaseInitializer<TContext>接口。这个接口由类 CreateDatabaseIfNotExists、DropCreateDatabaseAlways 和 DropCreateDatabaseIfModelChanges 实现。顾名思义，这些数据库初始化器在删除和创建数据库的方式上有所不同。要给数据库定义数据，需要重写这些基类的 Seed 方法。

在下面的代码中，MenuCardsInitalizer 派生自基类 DropCreateDatabaseAlways，所以每次调用初始化器时，都会新建数据库并填充数据。无论用户如何修改数据，这都是最初的状态：

```
public class MenuCardsInitializer : DropCreateDatabaseAlways<MenuContext>
{
  protected override void Seed(MenuContext context)
  {
    var menuCards = new List<MenuCard>()
    {
      new MenuCard{
        Text = "Soups",
        Menus = new List<Menu> {
      new Menu {
        Text = "Baked Potato Soup",
        Price = 4.8M,
        Day = DateTime.Parse("8/14/2013", CultureInfo.InvariantCulture)
      },
      new Menu {
        Text = "Cheddar Broccoli Soup",
        Price = 4.5M,
        Day = DateTime.Parse("8/15/2013", CultureInfo.InvariantCulture)
      }
    }
  },
  new MenuCard{
    Text = "Steaks",
    Menus = new List<Menu> {
      new Menu {
        Text = "New York Sirloin Steak",
        Price = 18.8M,
        Day = DateTime.Parse("8/16/2013", CultureInfo.InvariantCulture)
      },
      new Menu {
        Text = "Rib Eye Steak",
        Price = 14.8M,
        Day = DateTime.Parse("8/17/2013", CultureInfo.InvariantCulture)
      }
    }
  }
};

 menuCards.ForEach(c => context.MenuCards.Add(c));
}
}
```

要激活初始化器，Database 类的静态成员 SetInitializer 定义了要使用的初始化器：

```
Database.SetInitializer<MenuContext>(new MenuCardsInitializer());
```

33.8.8 连接的弹性

如果使用另一个网络的数据库(例如使用 SQL Azure)，就可能出现网络问题，最好根据所抛出的异常，重新尝试执行操作。

使用 Entity Framework 6，有一种简单的方式：不需要修改数据访问代码，只需要修改配置。在配置文件中，最好不要写入太多信息。基于代码的配置是 Entity Framework 6 的另一个优秀功能。

要修改连接的弹性，可以使用派生自 DbConfiguration 的类，调用 SetExecutionStrategy 方法来改变执行策略。第一个参数定义了提供程序名称。在示例代码中(代码文件 CodeFirstSample/MyDatabaseConfiguration.cs)，第一个参数是用于访问 SQL Server 的 System.Data.SqlClient，第二个参数是 Func<IDbExecutionStrategy>类型，它指定一个 lambda 表达式，返回一个实现 IDbExecutionStrategy 接口的对象：

```
class MyDatabaseConfiguration : DbConfiguration
{
  public MyDatabaseConfiguration()
  {
    SetExecutionStrategy("System.Data.SqlClient",
      () => new SqlAzureExecutionStrategy());
  }
}
```

MyDatabaseConfiguration 类是自动实例化的。Entity Framework 运行库在数据上下文所在的程序集中寻找派生自 DbConfiguration 的类。这个类实例化后，在构造函数中，完成了实现方式的配置。除了设置执行策略之外，还可以添加或删除数据库初始化器、日志格式化器、复数化服务和解释器。

Entity Framework 带有一些优秀的执行策略：DefaultExecutionStrategy 用于除 SQL Server 之外的其他数据库。这个执行策略没有任何重试机会。DefaultSqlExecutionStrategy 用于 SQL Server，也没有任何重试机会。但是，它封装了异常，提供了何时启用连接的弹性的有效信息。SqlAzureExecutionStrategy 可以用于 SQL Azure，可以在出现异常时重试，这些异常对 SQL Azure 是临时的。

为了自定义连接的弹性，SqlAzureExecutionStrategy 的构造函数允许指定重试次数和几个重试之间的最大延迟时间。要进行更多的自定义，可以用派生自 DbExecutionStrategy 的类定义一个定制的执行策略(也可以使用 SqlAzureExecutionStrategy 类的功能)。基类定义了虚拟方法 ShouldRetryOn，在该方法中，如果应进行重试，可以根据所发生的异常确定在何处重试。第二个可以重写的方法是 GetNextDelay，通过它可以定义延迟时间。例如，在重试不成功时，可以增加延迟时间。

33.8.9　架构的迁移

每次数据库的架构改变时，可能不希望删除数据库，再重建它。使用 Code First，可以通过一种简单的方式动态改变数据库的架构：迁移。

迁移数据库架构的大部分代码可以通过 Entity Framework 中的工具创建。所有这些都可以在包管理器控制台中完成。

要给数据库定义配置，可以使用包管理器控制台(确保在控制台上通过默认项目设置选择正确的项目)输入命令 Enable-Migrations。而要访问包管理器控制台，可以使用 Tools | Library Package Manager | Package Manager Console。启用迁移功能会通过代码文件 Configuration.cs 给项目添加目录 Migrations。这个文件包含如下所示的 Configuration 类。这个类派生自 DbMigrationsConfiguration 基类，DbMigrationsConfiguration 在 System.Data.Entity.Migrations 名称空间中定义。该名称空间用于所

有迁移类型。使用 Seed 方法可以自动填充数据，如上一节的 Seed 方法所示：

```
internal sealed class Configuration :
  DbMigrationsConfiguration<Wrox.ProCSharp.Entities.MenuContext>
{
  public Configuration()
  {
    AutomaticMigrationsEnabled = false;
  }
  protected override void Seed(MenuContext context)
  {
  }
}
```

现在需要定义初始架构，为此，可以把 AutomaticMigrationsEnabled 设置为 true ，第一次调用应用程序，或者使用 Package Manager Console。Add-Migration <ClassName>会创建定义数据库的代码。指定 Add-Migration InitialSchema 会创建派生自 DbMigration 的 InitialSchema 类，如下所示。Up 方法定义了在创建数据库的过程中需要执行的操作，Down 方法删除数据库：

```
public partial class InitialSchema : DbMigration
{
  public override void Up()
  {
    CreateTable(
        "dbo.MenuCards",
        c => new
            {
              Id = c.Int(nullable: false, identity: true),
              Text = c.String(nullable: false, maxLength: 30),
            })
        .PrimaryKey(t => t.Id);

    CreateTable(
        "dbo.Menus",
        c => new
            {
              Id = c.Int(nullable: false, identity: true),
              Text = c.String(nullable: false, maxLength: 40),
              Price = c.Decimal(nullable: false, storeType: "money"),
              Day = c.DateTime(storeType: "date"),
              MenuCardId = c.Int(nullable: false),
            })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.MenuCards", t => t.MenuCardId, cascadeDelete: true)
        .Index(t => t.MenuCardId);
  }

  public override void Down()
  {
    DropForeignKey("dbo.Menus", "MenuCardId", "dbo.MenuCards");
    DropIndex("dbo.Menus", new[] { "MenuCardId" });
    DropTable("dbo.Menus");
    DropTable("dbo.MenuCards");
  }
}
```

调用 Update-Database 命令，就调用了 Up 方法，接着就可以使用代码读写数据了。

现在，如果用实体类型进行了一些修改，例如 Menu 类型获得了一个额外的 Subtitle 属性，如下所示，数据库架构就需要更新。

```
[StringLength(120)]
public string Subtitle { get; set; }
```

Add-Migration MenuSubtitle 会创建迁移类型，把 Subtitle 列添加到 Menus 表中：

```
public partial class MenuSubtitle : DbMigration
{
  public override void Up()
  {
    AddColumn("dbo.Menus", "Subtitle", c => c.String(maxLength: 120));
  }
  public override void Down()
  {
    DropColumn("dbo.Menus", "Subtitle");
  }
}
```

现在只需要调用 Update-Database 命令，用最新的更改迁移数据库(或启用自动迁移)。

33.9 小结

本章介绍了 ADO.NET Entity Framework 的特性，ADO.NET Entity Framework 基于 CSDL、MSL 和 SSDL 定义的映射，这些 XML 信息描述实体、映射和数据库架构。使用这种映射技术，可以创建不同的关系类型，把实体类映射到数据库表上。

本章还介绍了对象上下文如何保存所检索和更新的实体信息，如何把改变的内容写入存储器中。如何通过 POCO 对象，使用已有的对象库，把对象映射到数据库上，Code First 如何随时创建数据库，并带有基于约定的映射信息。

LINQ to Entities 仅是 ADO.NET Entity Framework 的一部分，它允许使用新的查询语法访问实体。

Entity Framework Code First 使用数据库时，不需要为实体创建模型。本章提到，Code First 不仅可根据实体的代码约定动态地创建数据库，利用一个流畅的 API 更多地控制数据库的创建过程，还可以创建类，以填充初始数据，启动迁移功能，根据应用程序的不同版本改变数据库的架构。

下一章将把 XML 用作数据源，通过 LINQ to XML 创建和查询 XML。

第34章

处 理 XML

本章要点

- XML 标准
- XmlReader 和 XmlWriter
- XmlDocument
- XPathDocument
- XmlNavigator
- LINQ to XML
- 使用 System.Xml.Linq 名称空间中的对象
- 使用 LINQ 查询 XML 文档
- 使用 LINQ to SQL 和 LINQ to XML

本章源代码下载地址(wrox.com)：

打开网页 www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- XmlReaderSample
- ConsoleApplication1
- XmlSample
- XmlSample01

34.1 XML

XML 在.NET Framework 中有重要作用。.NET Framework 不仅允许在应用程序中使用 XML，.NET Framework 本身也在配置文件和源代码文档中使用 XML。另外，SOAP、Web 服务和 ADO.NET 也使用 XML。

为了涵盖 XML 的扩展用法，.NET Framework 包含了 System.Xml 名称空间。这个名称空间包

含许多用于处理 XML 的类。本章将讨论这些类。

本章介绍如何使用 XmlDocument 类，这是 DOM(Document Object Model，文档对象模型)的实现方式，以及.NET 为 SAX 提供的一种替代品(XmlReader 和 XmlWriter 类)。本章还要讨论 XPath 和 XSLT 的类实现方式，接着介绍 XML 和 ADO.NET 如何一起工作，如何把其中一种格式转换为另一种格式。还将介绍如何把对象序列化为 XML，使用 System.Xml.Serialization 名称空间中的类从 XML 文档中创建一个对象(或者反序列化)。更重要的是，要介绍如何把 XML 合并到 C#应用程序中。

注意 XML 名称空间可以用许多不同的方式得到类似的结果。因为我们不可能把所有这些不同方式都放在一章中介绍，所以这里仅探讨其中一种方式，并尽量提及完成同一任务的其他方式。

因为篇幅有限，不能从头开始介绍 XML，所以本章假定读者已经熟悉 XML 技术。因此，你应知道元素、属性和节点，还应知道格式良好的文档的含义，你也应熟悉 SAX 和 DOM。

如果要更多地了解 XML，可以参阅 Wrox 出版社的 *Prefessional XML*(Wiley 出版社，2007 年，ISBN：978-0-471-77777-9)。

除了一般的 XML 用法之外，.NET Framework 还可以通过 LINQ to XML 使用 XML。使用 XPath 搜索 XML 文档也是一种很好的替代方式。

首先简要介绍目前使用的 XML 标准。

34.2 .NET 支持的 XML 标准

W3C(World Wide Web Consortium，万维网联合会)开发了一组标准，它给 XML 提供了强大的功能和潜力。如果没有这些标准，XML 就不会对开发领域有它应有的影响。W3C 网站(www.w3.org)包含 XML 的所有有用信息。

.NET Framework 支持下述 W3C 标准：

- XML 1.0(www.w3.org/TR/1998/REC-xml-19980210)，包括 DTD 支持
- XML 名称空间(www.w3.org/TR/REC-xml-names)，包括流级和 DOM
- XML 架构(www.w3.org/2001/XMLSchema)
- XPath 表达式(www.w3.org/TR/xpath)
- XSLT 转换(www.w3.org/TR/xslt)
- DOM Level 1 核心(www.w3.org/TR/REC-DOM-Level-1)
- DOM Level 2 核心(www.w3.org/TR/DOM-Level-2-Core)
- Soap 1.1(www.w3.org/TR/SOAP)

随着 Framework 走向成熟和 W3C 更新所推荐的标准，标准支持的级别也会改变，因此，必须确保标准和 Microsoft 提供的支持级别都是最新的。

34.3 System.Xml 名称空间

对 XML 处理的支持由.NET 中 System.Xml 名称空间中的类提供。本节介绍(没有特定的顺序)System.Xml

名称空间中一些比较重要的类。表 34-1 列出了主要的 XML 读取器类和写入器类。

表 34-1

类　名	说　明
XmlReader	抽象的读取器类，提供快速、没有缓存的 XML 数据。XmlReader 是只向前的，类似于 SAX 分析器
XmlWriter	抽象的写入器类，以流或文件的格式提供快速、没有缓存的 XML 数据
XmlTextReader	扩展 XmlReader，提供访问 XML 数据的快速只向前流
XmlTextWriter	扩展 XmlWriter，快速生成只向前的 XML 流

表 34-2 列出了用于处理 XML 的其他一些重要的类。

表 34-2

类　名	说　明
XmlNode	抽象类，表示 XML 文档中的一个节点。它是 XML 名称空间中几个类的基类
XmlDocument	扩展 XmlNode，这是 W3C DOM 的实现方式，它给出 XML 文档在内存中的树型表示，可以浏览和编辑它们
XmlDataDocument	扩展 XmlDocument，即从 XML 数据中加载的文档，或从 ADO.NET DataSet 的关系数据中加载的文档，允许把 XML 和关系数据混合在同一个视图中
XmlResolver	抽象类，分析基于 XML 的外部资源，如 DTD 和架构引用，也可以用于处理<xsl:include>和<xsl:import>元素
XmlNodeList	可以迭代的一个 XmlNode 列表
XmlUrlResolver	扩展 XmlResolver，用 URI(Uniform Resource Identifier，统一资源标识符)解析外部资源

System.Xml 名称空间中的许多类都提供了管理 XML 文档和流的方式，而其他类(例如 XmlDataDocument 类)则提供了 XML 数据存储器和存储在 DataSet 中的关系数据之间的桥梁。

XML 名称空间可用于.NET 系列的任何语言，这表示，本章中所有的示例也可以用 VB.NET、托管 C++等来编写。

34.4　使用 System.Xml 类

下面几个示例将使用 books.xml 作为数据源。books.xml 和本章的其他代码示例可以从 Wrox 网站(www.wrox.com)中找到，books.xml 也包含在.NET SDK 的几个示例中。books.xml 文件是假想书店的书目清单，它包含类型、作者姓名、价格和 ISBN 号等信息。

下面是 books.xml 文件：

```
<?xml version='1.0'?>
<!--This file represents a fragment of a book store inventory database-->
```

```
<bookstore>
  <book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
  </book>
</bookstore>
```

34.5 读写流格式的 XML

如果你曾经使用过 SAX，就应很熟悉 XmlReader 类和 XmlWriter 类。基于 XmlReader 的类提供了一种非常迅速、只向前的只读光标来处理 XML 数据。因为它是一个流模型，所以内存要求不是很高。但是，它没有提供基于 DOM 模型的导航功能和读写功能。基于 XmlWriter 的类可以生成遵循 W3C 的 XML 1.0 Namespace Recommendations 的 XML 文档。

XmlReader 和 XmlWriter 都是抽象类。下面的类派生自 XmlReader：

- XmlNodeReader
- XmlTextReader
- XmlValidatingReader

下面的类派生自 XmlWriter 的类：

- XmlTextWriter
- XmlQueryWriter

XmlTextReader 类和 XmlTextWriter 类处理 System.IO 名称空间中一个基于流的对象，或者处理 TextReader/TextWriter 对象。XmlNodeReader 类把 XmlNode 作为其源，而不是一个流。XmlValidatingReader 类添加了 DTD 和架构验证，因此提供了数据的有效性验证。本章后面会详细介绍这些类。

34.5.1 使用 XmlReader 类

XmlReader 类非常类似于 MSXML SDK 中的 SAX。它们最大的一个区别是 SAX 是一种推模型

(push model)，它把数据推入应用程序中，开发人员必须准备接受它，而 XmlReader 是一种拉模型(pull model)，它把应用程序请求的数据拉入该应用程序。这样就有一种更简单、更直观的编程模型。另一个优点是拉模型可以选择把什么数据发送给应用程序。如果不需要所有数据，就不需要处理它。而在推模型中，所有 XML 数据都必须由应用程序处理，无论是否需要这些数据。

下面介绍一个非常简单的示例，以读取 XML 数据，后面将详细介绍 XmlReader 类，这些代码在 XmlReaderSample 文件夹中。下面的代码将读取 book.xml 文档中的数据。在读取每个节点时，都要检查 NodeType 属性。如果节点是一个文本节点，就把其值追加到文本框中(代码文件 XMLReaderSample.sln)：

```
using System.Xml;

private void button3_Click(object sender, EventArgs e)
{
  richTextBox1.Clear();
  XmlReader rdr = XmlReader.Create("books.xml");
  while (rdr.Read())
  {
    if (rdr.NodeType == XmlNodeType.Text)
      richTextBox1.AppendText(rdr.Value + "\r\n");
  }
}
```

如前所述，XmlReader 是一个抽象类。所以，要直接使用 XmlReader 类，必须添加静态方法 Create，该方法返回一个 XmlReader 对象。Create 方法有 9 个重载版本。在上面的例子中，该方法有一个字符串参数，表示 XmlDocument 的文件名。还可以给该方法传送基于流的对象和基于 TextReader 的对象。

另一个可以使用的对象是 XmlReaderSettings，它指定读取器的功能。例如，可以使用架构来验证数据流。把 Schemas 属性设置为一个有效的 XMLSchemaSet 对象来缓存 XSD 架构。接着把 XmlReaderSettings 对象的 XsdValidate 属性设置为 true。

有几个 Ignore 属性可用于控制读取器处理某些节点和值的方式。这些属性包括 IgnoreComments、IgnoreIdentityConstraints、IgnoreInlineSchema、IgnoreProcessingInstructions、IgnoreSchemaLocation 和 IgnoreWhitespace，它们可以从文档中提取某些项。

1. Read()方法

遍历文档有几种方式，如前面的示例所示，Read()方法可以进入下一个节点。然后验证该节点是否有一个值(HasValue())，或者该节点是否有特性(HasAttributes()，稍后介绍)。也可以使用 ReadStartElement()方法，该方法验证当前节点是否是起始元素，如果是起始元素，就可以定位到下一个节点上。如果不是起始元素，就引发一个 XmlException 异常。调用这个方法与调用 Read()方法后再调用 IsStartElement()方法是一样的。

ReadElementString()类似于 ReadString()，但它可以选择以元素名作为参数。如果下一个内容节点不是起始标记，或者如果 Name 参数不匹配当前的节点 Name，就会引发异常。

下面的示例说明了如何使用 ReadElementString()方法。注意因为这个示例使用 FileStream，所以需要利用 using 语句来包括 System.IO 名称空间(代码文件 XMLReaderSample.sln)：

```
private void button6_Click(object sender, EventArgs e)
```

```
{
  richTextBox1.Clear();
        XmlReader rdr = XmlReader.Create("books.xml");
  while (!rdr.EOF)
  {
    //if we hit an element type, try and load it in the listbox
    if (rdr.MoveToContent() == XmlNodeType.Element && rdr.Name == "title")
    {
     richTextBox1.AppendText(rdr.ReadElementString() + "\r\n");
    }
    else
    {
      //otherwise move on
      rdr.Read();
    }
  }
}
```

在 while 循环中，使用 MoveToContent()方法查找类型为 XmlNodeType.Element、名称为 title 的节点。我们使用 XmlTextReader 类的 EOF 属性作为循环条件。如果节点的类型不是 Element，或者名称不是 title，else 子句就会调用 Read()方法进入下一个节点。当查找到一个满足条件的节点时，就把 ReadElementString()方法的结果添加到列表框中。这样就在列表框中添加一个书名。注意，在成功执行 ReadElementString()方法后，不需要调用 Read()方法，因为 ReadElementString()方法已经使用了整个 Element，并定位到下一个节点上。

如果删除了 if 子句中的&&rdr.Name=="title"，在抛出 XmlException 异常时，就必须捕获它。在数据文件中，MoveToContent()方法查找到的第一个元素是<bookstore>，因为它是一个元素，所以通过了 if 语句中的检查。但是，由于它不包含简单的文本类型，因此它会导致 ReadElementString()方法引发一个 XmlException 异常。解决这个问题的一种方式是把 ReadElementString 调用放在它自己的函数中。现在，如果在这个函数中 ReadElementString 调用失败，就可以处理错误，并返回主调函数。

下面就调用这个新方法 LoadTextBox()，把 XmltextReader 类作为参数。进行这些修改后，LoadTextBox()方法如下所示：

```
private void LoadTextBox(XmlReader reader)
{
  try
  {
    richTextBox1.AppendText (reader.ReadElementString() + "\r\n");
  }
  // if an XmlException is raised, ignore it.
  catch(XmlException er){}
}
```

上面示例中的下述代码

```
if (tr.MoveToContent() == XmlNodeType.Element && tr.Name == "title")
{
  richTextBox1.AppendText(tr.ReadElementString() + "\r\n");
}
else
{
```

```
    //otherwise move on
    tr.Read();
  }
```

就会变成：

```
if (tr.MoveToContent() == XmlNodeType.Element)
{
  LoadTextBox(tr);
}
else
{
  //otherwise move on
  tr.Read();
}
```

运行这段代码，结果应与前面示例的结果一样。因此，完成同一个任务有多种不同的方式。这体现了 System.Xml 名称空间中类的灵活性。

XmlReader 类还可以读取强类型化的数据，它有几个 ReadElementContentAs()方法，如 ReadElementContentAsDouble()、ReadElementContentAsBoolean()等。下面的示例说明了如何把对应值读取为小数，并对该值进行数学处理。在本例中，要给价格元素中的值增加 25%：

```
private void button5_Click(object sender, EventArgs e)
{
  richTextBox1.Clear();
  XmlReader rdr = XmlReader.Create("books.xml");
  while (rdr.Read())
  {
    if (rdr.NodeType == XmlNodeType.Element)
    {
      if (rdr.Name == "price")
      {
        decimal price = rdr.ReadElementContentAsDecimal();
        richTextBox1.AppendText("Current Price = " + price + "\r\n");
        price += price * (decimal).25;
        richTextBox1.AppendText("New Price = " + price + "\r\n\r\n");
      }
      else if(rdr.Name== "title")
        richTextBox1.AppendText(rdr.ReadElementContentAsString() + "\r\n");
    }
  }
}
```

如果不能把该值转换为小数，就引发一个 FormatException 异常。与把该值读取为一个字符串，再把它转换为合适的数据类型相比，这个方法的效率较高。

2. 检索特性数据

在运行示例代码时，可能注意到在读取节点时，没有看到特性。这是因为特性不是文档的结构的一部分。针对元素节点，可以检查特性是否存在，并可选择性地检索特性值。

例如，如果有特性，HasAttributes 就返回 true；否则返回 false。AttributeCount 属性确定特性的个数。GetAttribute()方法按照名称或索引来获取特性。如果要一次迭代一个特性，就可以使用

MoveToFirstAttribute()和 MoveToNextAttribute()方法。

下面的示例迭代 book.xml 文档中的特性：

```
private void button7_Click(object sender, EventArgs e)
{
  richTextBox1.Clear();
  XmlReader tr = XmlReader.Create("books.xml");
  //Read in node at a time
  while (tr.Read())
  {
    //check to see if it's a NodeType element
    if (tr.NodeType == XmlNodeType.Element)
    {
      //if it's an element, then let's look at the attributes.
      for (int i = 0; i < tr.AttributeCount; i++)
      {
        richTextBox1.AppendText(tr.GetAttribute(i) + "\r\n");
      }
    }
  }
}
```

这次查找元素节点。找到一个节点后，就迭代其所有的特性，使用 GetAttribute()方法把特性值加载到列表框中。在本例中，这些特性是 genre、publicationdate 和 ISBN。

34.5.2 使用 XmlReader 类进行验证

有时不但要知道文档的格式是良好的，还要确定文档是有效的。XmlReader 类可以使用 XmlReaderSettings类，根据XSD架构验证XML。把XSD架构添加到XMLSchemaSet中，通过Schemas属性可以访问 XMLSchemaSet。XsdValidate 属性还必须设置为 true，这个属性默认为 false。

下面的示例说明了 XmlReaderSettings 类的用法。这个 XSD 架构用于验证 book.xml 文档(代码文件 book.xsd)：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
        elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string" />
              <xs:element name="author">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element minOccurs="0" name="name"
                                              type="xs:string" />
                    <xs:element minOccurs="0" name="first-name"
                                              type="xs:string" />
                    <xs:element minOccurs="0" name="last-name"
                                              type="xs:string" />
                  </xs:sequence>
```

```
                </xs:complexType>
              </xs:element>
              <xs:element name="price" type="xs:decimal" />
            </xs:sequence>
            <xs:attribute name="genre" type="xs:string" use="required" />
            <!--<xs:attribute name="publicationdate"
                          type="xs:unsignedShort" use="required" />-->
            <xs:attribute name="ISBN" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

在 Visual Studio 中这个架构从 book.xml 中生成。注意 publicationdate 属性被注释掉了，这样在验证到该属性时会失败。

下面的代码使用该架构验证 books.xml 文档(代码文件 XMLReaderSample.sln)：

```
private void button8_Click(object sender, EventArgs e)
{

 richTextBox1.Clear();
  XmlReaderSettings settings = new XmlReaderSettings();
  settings.Schemas.Add(null, "books.xsd");
  settings.ValidationType = ValidationType.Schema;
  settings.ValidationEventHandler +=
    new System.Xml.Schema.ValidationEventHandler(settings_ValidationEventHandler);
  XmlReader rdr = XmlReader.Create("books.xml", settings);
  while (rdr.Read())
  {
    if (rdr.NodeType == XmlNodeType.Text)
      richTextBox1.AppendText(rdr.Value + "\r\n");
  }
}
```

创建 XmlReaderSettings 对象设置后，就把 books.xsd 架构添加到 XmlSchemaSet 对象中。XmlSchemaSet 对象的 Add()方法有 4 个重载版本，第一个重载版本把 XmlSchema 对象作为参数，XmlSchema 对象可以用于快速创建架构，而无须在磁盘上创建架构文件。另一个重载版本把另一个 XmlSchemaSet 对象作为参数。第 3 个重载版本接受两个字符串参数。第一个字符串是目标名称空间，第二个字符串是 XSD 文档的 URL。如果目标名称空间参数为空，就使用架构的 targetNamespace。最后一个重载版本也把 targetNamespace 作为第一个参数，但它使用基于 XmlReader 的对象读取架构。XmlSchemaSet 对象在处理要验证的文档之前预处理架构。

引用该架构后，XsdValidate 属性就设置为 ValidationType 枚举的一个值，该枚举的有效值是 DTD、Schema 和 None。如果把选中的值设置为 None，就不进行验证。

因为我们使用的是 XmlReader 对象，所以，如果文档有验证问题，在读取器读取属性或元素之前，就不会发现该问题。验证失败时，会引发一个 XmlSchemaValidationException 异常。这个异常可以在 catch 块中处理，但处理异常会很难控制数据流。为了解决这个问题，可以使用 XmlReaderSettings 类中的 ValidationEvent。这样，就可以处理验证失败，且无须使用异常处理。该事件还可以由验证

警告引发，验证警告不会引发异常。ValidationEvent 传递一个 ValidationEventArgs 对象，该对象包含 Severity 属性。这个属性确定事件是由错误还是警告引发。如果该事件由错误引发，则还会传递引发该事件的异常。还有一个消息属性。在本例中，消息显示在 MessageBox 中。

34.5.3 使用 XmlWriter 类

XmlWriter 类可以把 XML 写入一个流、文件、StringBuilder、TextWriter 或另一个 XmlWriter 对象中。与 XmlTextReader 类一样，XmlWriter 类以只向前、未缓存的方式进行写入。XmlWriter 类的可配置性很高，可以指定是否缩进内容、缩进量、在属性值中使用什么引号，以及是否支持名称空间等信息。与 XmlReader 类一样，这个配置使用 XmlWriterSettings 对象进行。

下面是一个简单的示例，它说明了如何使用 XmlTextWriter 类：

```
private void button9_Click(object sender, EventArgs e)
{
  XmlWriterSettings settings = new XmlWriterSettings();
  settings.Indent = true;
  settings.NewLineOnAttributes = true;
  XmlWriter writer = XmlWriter.Create("newbook.xml", settings);
  writer.WriteStartDocument();
  //Start creating elements and attributes
  writer.WriteStartElement("book");
  writer.WriteAttributeString("genre", "Mystery");
  writer.WriteAttributeString("publicationdate", "2001");
  writer.WriteAttributeString("ISBN", "123456789");
  writer.WriteElementString("title", "Case of the Missing Cookie");
  writer.WriteStartElement("author");
  writer.WriteElementString("name", "Cookie Monster");
  writer.WriteEndElement();
  writer.WriteElementString("price", "9.99");
  writer.WriteEndElement();
  writer.WriteEndDocument();
  //clean up
  writer.Flush();
  writer.Close();
}
```

这里编写一个新的 XML 文件 newbook.xml，并给一本新书添加数据。注意 XmlWriter 类会用新文件覆盖已有文件。本章的后面会把一个新元素或新节点插入到已有文档中，使用 Create()静态方法实例化 XmlWriter 对象。在本例中，把一个表示文件名的字符串和 XmlWriterSettings 类的一个实例传递为参数。

XmlWriterSettings 类的属性控制生成 XML 的方式。CheckedCharacters 属性是一个布尔值，如果 XML 中的字符不遵循 W3C XML 1.0 建议，该属性就会引发一个异常。Encoding 类设置生成 XML 所使用的编码，默认为 Encoding.UTF8。Indent 属性是一个布尔值，它确定元素是否应缩进。把 IndentChars 属性设置为用于缩进的字符串，默认为两个空格。NewLine 属性用于确定换行符。在上面的示例中，把 NewLineOnAttribute 属性设置为 true，所以把每个属性单独放在一行上，更便于读取生成的 XML。

WriteStartDocument()方法添加文档声明。现在开始写入数据。首先是 book 元素，接下来添加 genre、publicationdate 和 ISBN 属性。然后写入 title、author 和 price 元素。注意 author 元素有一个子元素 name。

单击对应按钮，生成 booknew.xml 文件，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<book
  genre="Mystery"
  publicationdate="2001"
  ISBN="123456789">
  <title>Case of the Missing Cookie</title>
  <author>
    <name>Cookie Monster</name>
  </author>
  <price>9.99</price>
</book>
```

在开始和结束写入元素和属性时，要注意控制元素的嵌套。在给 authors 元素添加 name 子元素时，就可以看到这种嵌套。注意 WriteStartElement()和 WriteEndElement()方法调用是如何安排的，以及它们如何在输出文件中生成嵌套的元素。

除了 WriteElementString()和 WriteAttributeString()方法外，还有其他几个专用的写入方法。WriteCData 方法可以输出一个 CData 部分(<!CDATA[...]>)，输出它接受的文本参数。WriteComment()方法以正确的 XML 格式输出注释。WriteChars()方法输出字符缓冲区的内容，其工作方式类似于前面的ReadChars 方法，它们都使用相同类型的参数。WriteChars()方法需要一个缓冲区(一个字符数组)、写入的起始位置(一个整数)和要写入的字符个数(一个整数)。

使用基于 XmlReader 和 XmlWriter 的类读写 XML 非常灵活，使用起来也很简单。下面介绍如何使用 System.Xml 名称空间中的 XmlDocument 类和 XmlNode 类实现 DOM。

34.6 在.NET 中使用 DOM

.NET 中的文档对象模型(Document Object Model，DOM)支持 W3C DOM Level 1 和 Core DOM Level 2 规范。DOM 通过 XmlNode 类来实现。XmlNode 是一个抽象类，它表示 XML 文档的一个节点。

还有一个 XmlNodeList 类，它是节点的一个有序列表。这是一个实时的节点列表，对节点的任何修改都会立即反映在列表中。XmlNodeList 类支持索引访问或迭代访问。

XmlNode 类和 XmlNodeList 类组成了.NET Framework 中 DOM 实现的核心，表 34-3 列出了基于 XmlNode 的一些类。

表 34-3

类 名	说 明
XmlLinkedNode	返回当前节点之前或之后的节点。给 XmlNode 类添加 NextSibling 和 PreviousSibling 属性
XmlDocument	表示整个文档，实现 DOM Level 1 和 Level 2 规范
XmlDocumentFragment	表示文档树的一个片段
XmlAttribute	表示 XmlElement 对象的一个属性对象
XmlEntity	表示一个已分析或未分析的实体节点
XmlNotation	包含在 DTD 或架构中声明的记号

表 34-4 列出了扩展 XmlCharacterData 的类。

表 34-4

类　名	说　明
XmlCDataSection	表示文档中的一个 CData 部分
XmlComment	表示一个 XML 注释对象
XmlSignificantWhitespace	表示带有空白的节点。只有 PreserveWhiteSpace 标志为 true 时，才能创建节点
XmlWhitespace	表示元素内容中的空白，只有 PreserveWhiteSpace 标志为 true 时，才能创建节点
XmlText	表示元素或属性的文本内容

最后，表 34-5 列出了扩展 XmlLinkedNode 的类。

表 34-5

类　名	说　明
XmlDeclaration	表示声明节点(例如<?xml version='1.0'...>)
XmlDocumentType	表示与文档类型声明相关的数据
XmlElement	表示一个 XML 元素对象
XmlEntityReferenceNode	表示一个实体引用节点
XmlProcessingInstruction	包含 XML 处理指令

可以看出，.NET 使其类适合于可能遇到的任何 XML 类型。因此，该工具集非常灵活和强大。本节不打算详细介绍每个类，而是用几个示例来说明可以完成什么任务。

使用 XmlDocument 类

XmlDocument 类及其派生类 XmlDataDocument(详见本章后面的内容)是用于在.NET 中表示 DOM 的类。与 XmlReader 类和 XmlWriter 类不同，XmlDocument 类具有读写功能，并可以随机访问 DOM 树。XmlDocument 类非常类似于 MSXML 中的 DOM 实现。如果你用 MSXML 编过程序，就会觉得使用 XmlDocument 类很合适。

下面介绍的示例创建一个 XmlDocument 对象，加载磁盘上的一个文档，再从标题元素中加载带有数据的文本框，这类似于 34.4.1 节的示例，区别是本例选择要使用的节点，而不是像基于 XmlReader 类的示例那样浏览整个文档。

下面是创建 XmlDocument 对象的代码，与 XmlReader 示例相比，这个示例比较简单(代码文件 frmXMLDOM.cs)：

```
private void button1_Click(object sender, System.EventArgs e)
{
    //doc is declared at the module level
    //change path to match your path structure
    _doc.Load("books.xml");
    //get only the nodes that we want.
    XmlNodeList nodeLst = _doc.GetElementsByTagName("title");
    //iterate through the XmlNodeList
    textBox1.Text = "";
```

```
        foreach (XmlNode node in nodeLst)
        {
            textBox1.Text += node.OuterXml + "\r\n";
        }
    }
```

注意，本节的示例添加了以下模块级的声明：

```
private XmlDocument doc=new XmlDocument();
```

如果这就是我们需要完成的全部工作，使用XmlReader类就是加载文本框的一种非常高效的方式，原因是我们只浏览一次文档，就完成了处理。这就是XmlReader类的工作方式。但如果要重新查看某个节点，则最好使用XmlDocument类。

下面的示例使用XPath语法从文档中检索一组节点：

```
private void button2_Click(object sender, EventArgs e)
{
  //doc is declared at the module level
  //change path to match your path structure
  doc.Load("books.xml");
  //get only the nodes that we want.
  XmlNodeList nodeLst = _doc.SelectNodes("/bookstore/book/title");
  textBox1.Text = "";
  //iterate through the XmlNodeList
  foreach (XmlNode node in nodeLst)
  {
     textBox1.Text += node.OuterXml + "\r\n";
  }
}
```

SelectNodes()方法返回一个NodeList或一个XmlNodes集合。这个列表只包含匹配作为SelectNodes()方法的参数传递的XPath语句。在这个示例中，只需要查看title节点。如果调用了SelectSingleNode()方法，就会接收到一个节点对象，它包含XmlDocument中满足XPath条件的第一个节点。

下面简要介绍一下SelectSingleNode()方法，它是XmlDocument类的XPath实现方式，SelectSingleNode()和SelectNodes()都是在XmlNode类中定义的方法，而XmlDocument类基于XmlNode类。SelectSingleNode()方法返回一个XmlNode，SelectNodes()方法返回一个XmlNodeList。System.Xml.XPath名称空间包含许多XPath实现方式。后面一节会介绍它们。

插入节点

前面的示例使用XmlTextWriter类新建一个文档。其局限性是它不能把节点插入到当前文档中。而使用XmlDocument类可以做到这一点。把上一个示例中的button1_Click()事件处理程序作如下改动：

```
private void button4_Click(object sender, System.EventArgs e)
{
      //change path to match your structure
      _doc.Load("books.xml");
      //create a new 'book' element
      XmlElement newBook = _doc.CreateElement("book");
      //set some attributes
```

```
        newBook.SetAttribute("genre", "Mystery");
        newBook.SetAttribute("publicationdate", "2001");
        newBook.SetAttribute("ISBN", "123456789");
        //create a new 'title' element
        XmlElement newTitle = _doc.CreateElement("title");
        newTitle.InnerText = "Case of the Missing Cookie";
        newBook.AppendChild(newTitle);
        //create new author element
        XmlElement newAuthor = _doc.CreateElement("author");
        newBook.AppendChild(newAuthor);
        //create new name element
        XmlElement newName = _doc.CreateElement("name");
        newName.InnerText = "Cookie Monster";
        newAuthor.AppendChild(newName);
        //create new price element
        XmlElement newPrice = _doc.CreateElement("price");
        newPrice.InnerText = "9.95";
        newBook.AppendChild(newPrice);
        //add to the current document
        _doc.DocumentElement.AppendChild(newBook);
        //write out the doc to disk
        XmlTextWriter tr = new XmlTextWriter("booksEdit.xml", null);
        tr.Formatting = Formatting.Indented;
        _doc.WriteContentTo(tr);
        tr.Close();
        //load listBox1 with all of the titles, including new one
        XmlNodeList nodeLst = _doc.GetElementsByTagName("title");
        textBox1.Text = "";
        foreach (XmlNode node in nodeLst)
        {
             textBox1.Text += node.OuterXml + "\r\n";
        }
}
```

在执行这段代码后，会实现与上一个示例相同的功能，但本例在文本框中添加了一本书：*The Case of the Missing Cookie*。仔细查看代码，就会发现这是一个相当简单的过程。首先，新建一个 book 元素：

```
XmlElement newBook = doc.CreateElement("book");
```

CreateElement()方法有 3 个重载版本，它们可以指定：

- 元素名
- 名称和名称空间 URI
- 前缀、本地名和名称空间

一旦创建该元素，就要添加属性：

```
newBook.SetAttribute("genre","Mystery");
newBook.SetAttribute("publicationdate","2001");
newBook.SetAttribute("ISBN","123456789");
```

既然创建了属性，就要添加书籍的其他元素：

```
XmlElement newTitle = doc.CreateElement("title");
```

```
newTitle.InnerText = "The Case of the Missing Cookie";
newBook.AppendChild(newTitle);
```

再次新建一个基于 XmlElement 的对象(newTitle)，然后把 InnerText 属性设置为新经典著作的书名，把该元素追加为 book 元素的一个子元素。对 book 元素中的其他元素重复这一操作。注意把 name 元素添加为 author 元素的一个子元素。这样就可以像其他 book 元素一样得到合适的嵌套关系。

最后把 newBook 元素追加到 doc.DocumentElement 节点上，它与所有其他 book 元素同级。现在用新元素更新已有文档。

最后，把新 XML 文档写入到磁盘中。在这个示例中，新建一个 XmlTextWriter，把它传递给 WriteContentTo()方法。WriteContentTo()和 WriteTo()方法都接受一个 XmlTextWriter 参数。WriteContentTo 方法把当前节点及其所有子节点都保存到 XmlTextWriter 中，而 WriteTo()方法只保存当前节点。因为 doc 是一个基于 XmlDocument 的对象，它表示整个文档，所以应保存它。还可以使用 Save()方法，它总是保存整个文档，Save()方法有 4 个重载版本，其参数分别是一个包含文件名和路径的字符串、一个基于 Stream 的对象、一个基于 TextWriter 的对象和一个基于 XmlWriter 的对象。

我们还在 XmlTextWriter 上调用了 Close()方法，刷新内部缓存，并关闭文件。在运行这个示例时，会得到如图 34-1 所示的对话框。注意列表框底部的新项。

图 34-1

本章的前面说明了如何使用 XmlTextWriter 类创建一个文档，还可以使用 XmlDocument 类。使用哪个类比较好？如果要写入 XML 流的数据可用且准备写入，那么最好选择 XmlTextWriter 类。但是，如果需要一次构建 XML 文档的一小部分，在不同的地方插入节点，那么用 XmlDocument 类创建文档比较好。为此，可以把下面的代码：

```
doc.Load("books.xml");
```

改为：

```
//create the declaration section
XmlDeclaration newDec = doc.CreateXmlDeclaration("1.0",null,null);
doc.AppendChild(newDec);
//create the new root element
XmlElement newRoot = doc.CreateElement("newBookstore");
doc.AppendChild(newRoot);
```

首先新建一个 XmlDeclaration，其参数是版本(目前是"1.0")、编码和独立标志。如果没有使用 null(null 默认为 UTF-8)，编码参数就应设置为一个字符串，该字符串应是 System.Text.Encoding 类的一部分。独立标志可以是 yes、no 或 null。如果它是 null，就不使用该特性，也不包含在文档中。

要创建的下一个元素是 DocumentElement。在本例中，它称为 newBookstore，这样区别就比较明显。代码的其余部分与前面的示例相同，工作原理也相同。下面是从以下代码中生成的 booksEdit.xml：

```
<?xml version="1.0"?>
<newBookstore>
  <book genre="Mystery" publicationdate="2001" ISBN="123456789">
    <title>The Case of the Missing Cookie</title>
    <author>
      <name>C. Monster</name>
    </author>
    <price>9.95</price>
  </book>
</newBookstore>
```

在希望随机访问文档时，可以使用 XmlDocument 类。在希望有一个流类型的模型时，可以使用基于 XmlReader 的类。基于 XmlNode 的 XmlDocument 类的灵活性要求的内存比较多，读取文档的性能也没有使用 XmlReader 类好。遍历 XML 文档还有另一种方式：使用 XPathNavigator 类。

34.7 使用 XPathNavigator 类

XPathNavigator 类用于从 XML 文档中选择、迭代和偶尔编辑数据。XPathNavigator 类可以从 XmlDocument 中创建，以支持编辑功能；它也可以从 XPathDocument 类中创建，此时只能用于读取。因为 XPathDocument 类是只读的，所以它执行得很好。与 XmlReader 类不同，XPathNavigator 类不是一个流模型，所以文档只读取和分析一次。

XPathNavigator 类在 System.Xml.XPath 名称空间中，XPath 是一种查询语言，可以从 XML 文档中选择特定的节点或元素，以进行处理。

34.7.1 System.Xml.XPath 名称空间

System.Xml.XPath 名称空间建立在速度的基础上，由于它提供了 XML 文档的一种只读视图，因此它没有编辑功能。这个名称空间中的类可以采用光标的方式在 XML 文档上进行快速迭代和选择操作。

表 34-6 列出了 System.Xml.XPath 名称空间中的重要类，并对每个类的功能进行了简单的说明。

表 34-6

类 名	说 明
XPathDocument	提供整个 XML 文档的视图，只读
XPathNavigator	提供 XPathDocument 的导航功能
XPathNodeIterator	提供节点集的迭代功能
XPathExpression	编译好的 XPath 表达式，由 SelectNodes、SelectSingleNodes、Evaluate 和 Matches 使用
XPathException	XPath 异常类

1. XPathDocument 类

XPathDocument 类没有提供 XmlDocument 类的任何功能，它唯一的功能是创建 XPathNavigator。

因此，这是 XPathDocument 类上唯一可用的方法(除了其他由 Object 提供的方法)。

XPathDocument 类可以用许多不同的方式创建。可以给构造函数传递 XmlReader、XML 文档的文件名或基于流的对象，其灵活性非常大。例如，可以使用 XmlValidatingReader 验证 XML，然后使用同一个对象创建 XPathDocument。

2. XPathNavigator 类

XPathNavigator 类包含移动和选择所需元素的所有方法，在该类中定义的一些“移动”方法如表 34-7 所示。

表 34-7

方 法 名	说 明
MoveTo()	把 XPathNavigator 作为参数，移动当前位置到 XPathNavigator 指定的地方
MoveToAttribute()	移动到指定的特性，其参数是属性名和名称空间
MoveToFirstAttribute()	移动到当前元素中的第一个特性上，如果成功，就返回 true
MoveToNextAttribute()	移动到当前元素中的下一个特性上，如果成功，就返回 true
MoveToFirst()	移动到当前节点中的第一个同级节点上，如果成功，就返回 true
MoveToLast()	移动到当前节点中的最后一个同级节点上，如果成功，就返回 true
MoveToNext()	移动到当前节点中的下一个同级节点上，如果成功，就返回 true
MoveToPrevious()	移动到当前节点中的上一个同级节点上，如果成功，就返回 true
MoveToFirstChild()	移动到当前元素中的第一个子元素上，如果成功，就返回 true
MoveToId()	移动到 ID 参数提供的元素上，文档中需要有一个架构，元素的数据类型必须是 ID 类型
MoveToParent()	移动到当前节点的父节点上，如果成功，就返回 true
MoveToRoot()	移动到文档的根节点上

要选择文档的一个子集，可以使用表 34-8 所示的 Select()方法。

表 34-8

方 法 名	说 明
Select()	使用 XPath 表达式选择一个节点集
SelectAncestors`()	根据 XPath 表达式选择当前节点的所有上级节点
SelectChildren()	根据 XPath 表达式选择当前节点的所有子节点
SelectDescendants()	根据 XPath 表达式选择当前节点的所有下级节点
SelectSingleNode()	使用 XPath 表达式选择一个节点

如果 XPathNavigator 类是从 XPathDocument 类中创建的，它就是只读的。如果 XPathNavigator 类是从 XmlDocument 类中创建的，它就可以用于编辑文档。查看 CanEdit 属性就可以验证这一点。如果该属性是 true，就可以使用某个“插入”方法。InsertBefore()和 InsertAfter()会分别在当前节点的前面和后面新建一个节点。新节点的源可以是 XmlReader 类或字符串。还可以返回一个 XmlWriter

类，它用于写入新节点信息。

使用 ValueAs 属性可以从节点中读取强类型化的值。注意这与 XmReader 类不同，XmReader 类使用 ReadValue()方法。

3. XPathNodeIterator 类

XPathNodeIterator 类可以看作是 XPath 中的一个 NodeList 或一个 NodeSet，这个对象有两个属性和 3 个方法：

- Clone()——新建它本身的一个副本。
- Count——XPathNodeIterator 对象中的节点数。
- Current——返回指向当前节点的 XPathNavigator。
- CurrentPosition()——返回表示当前位置的一个整数。
- MoveNext()——移动到匹配 XPath 表达式的下一个节点上，XPath 表达式用于创建 XPathNodeIterator。

XPathNodeIterator 类由 XPathNavigators 类的 Select()方法返回，使用它可以迭代 XPathNavigator 类的“选择”方法返回的节点集。使用 XPathNodeIterator 类的 MoveNext()方法不会改变创建它的 XPathNavigator 类的位置。

4. 使用 XPath 名称空间中的类

要理解这些类的用法，最好是查看一下迭代 books.xml 文档的代码，弄清楚导航是如何工作的。为了使用这些示例，首先需要添加对 System.Xml.Xsl 和 System.Xml.XPath 名称空间的引用，如下所示：

```
using System.Xml.XPath;
using System.Xml.Xsl;
```

对于这个示例，使用 booksxpath.xml 文件，它类似于前面使用的 books.xml 文件，但在 booksxpath.xml 文件中添加了两本书。下面是窗体代码，这段代码是 XmlSample 项目的一部分(代码文件 frmNavigator.cs)：

```
private void button1_Click(object sender, EventArgs e)
{
  //modify to match your path structure
  XPathDocument doc = new XPathDocument("books.xml");
  //create the XPath navigator
  XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
  //create the XPathNodeIterator of book nodes
  // that have genre attribute value of novel
  XPathNodeIterator iter = nav.Select("/bookstore/book[@genre='novel']");
  textBox1.Text = "";
  while (iter.MoveNext())
  {
    XPathNodeIterator newIter =
       iter.Current.SelectDescendants(XPathNodeType.Element, false);
    while (newIter.MoveNext())
    {
```

```
            textBox1.Text += newIter.Current.Name + ": " +
               newIter.Current.Value + "\r\n";
         }
      }
   }
```

在 button1_Click()方法中，首先创建 XPathDocument(命名为 doc)，其参数是要打开的文档的文件和路径字符串。下面一行代码创建 XPathNavigator：

```
XPathNavigator nav = doc.CreateNavigator();
```

本例用 Select()方法检索所有 genre 属性值为 novel 的一组节点，然后使用 MoveNext()方法迭代书籍列表中的所有小说。

要把数据加载到列表框中，使用 XPathNodeIterator.Current 属性。根据 XPathNodeIterator 指向的节点，新建一个 XPathNavigator 对象。在本例中，为文档中的一个 book 节点创建 XPathNavigator。

之后的循环接受这个 XPathNavigator，调用 Select()方法的另一个重载版本 SelectDescendants 创建另一个 XPathNodeIterator。这样，XPathNodeIterator 就包含了 book 节点的所有子节点。

然后，在这个 XPathNodeIterator 上执行另一个 MoveNext 循环，给文本框加载元素名和元素值。在运行代码后，屏幕显示如图 34-2 所示的对话框，注意只列出了小说。

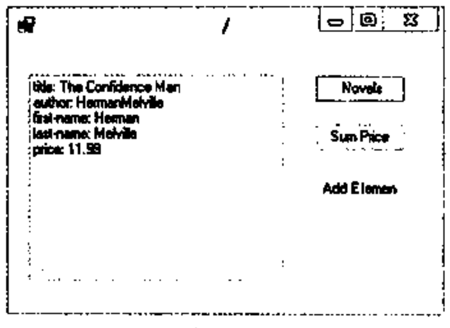


图 34-2

如果要把这些书的成本相加，该怎么办？XPathNavigator 类为此包含了 Evaluate()方法。Evaluate()有 3 个重载版本，第 1 个版本包含一个字符串，该字符串就是 XPath 函数调用。第 2 个重载版本的参数是 XPathExpression 对象，第 3 个重载版本的参数是 XPathExpression 和 XPathNodeIterator。下面的代码类似于前面的示例，但这次迭代文档中的所有节点。最后调用的 Evaluate()方法汇总了所有书籍的成本：

```
private void button2_Click(object sender, EventArgs e)
{
   //modify to match your path structure
   XPathDocument doc = new XPathDocument("books.xml");
   //create the XPath navigator
   XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
   //create the XPathNodeIterator of book nodes
   XPathNodeIterator iter = nav.Select("/bookstore/book");
   textBox1.Text = "";
   while (iter.MoveNext())
   {
      XPathNodeIterator newIter =
```

```
        iter.Current.SelectDescendants(XPathNodeType.Element, false);
      while (newIter.MoveNext())
      {
        textBox1.Text += newIter.Current.Name + ": " + newIter.Current.Value +
          "\r\n";
      }
    }
    textBox1.Text += "========================" + "\r\n";
    textBox1.Text += "Total Cost = " + nav.Evaluate("sum(/bookstore/book/price)");
}
```

这次，可以看到文本框中书籍的总成本，如图34-3所示。

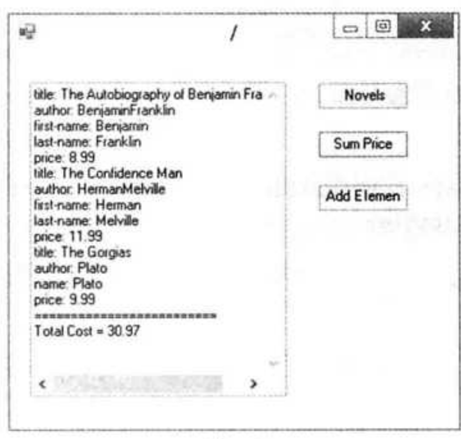


图34-3

现在，假定需要添加一个折扣节点。使用InsertAfter()方法可以很容易插入该节点。下面是代码：

```
private void button3_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  doc.Load("books.xml");
  XPathNavigator nav = doc.CreateNavigator();

  if (nav.CanEdit)
  {
    XPathNodeIterator iter = nav.Select("/bookstore/book/price");
    while (iter.MoveNext())
    {
      iter.Current.InsertAfter("<disc>5</disc>");
    }
  }
  doc.Save("newbooks.xml");
}
```

这段代码在价格元素的后面添加了<disc>5</disc>元素。首先选择前面所有的价格节点，使用XPathNodeIterator迭代节点，并插入新节点。修改后的文档保存为一个新名称newbooks.xml。下面是新文档的内容：

```
<?xml version="1.0"?>
<!--This file represents a fragment of a book store inventory database-->
<bookstore>
```

```
  <book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
    <disc>5</disc>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
    <disc>5</disc>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
     <name>Plato</name>
    </author>
    <price>9.99</price>
    <disc>5</disc>
  </book>
</bookstore>
```

节点可以插入到选中的节点之前或之后。还可以修改节点，并删除它们。如果对许多节点进行了改动，那么最好使用从 XmlDocument 中创建的 XPathNavigator。

34.7.2 System.Xml.Xsl 名称空间

System.Xml.Xsl 名称空间包含.NET Framework 用于支持 XSL 转换的类。这个名称空间中的类可以和任何实现 IXPathNavigable 接口的存储器一起使用。在目前的.NET Framework 中，包含 XmlDocument、XmlDataDocument 和 XPathDocument。与 XPath 一样，应使用最有效的存储器。如果计划创建一个自定义存储器，如文件系统的存储器，并希望能够进行一定的转换，就应在类中实现 IXPathNavigable 接口。

XSL 基于一个流式上拉模型(streaming pull model)上。因此，可以把几个转换链接在一起。根据需要，甚至可以在转换之间应用一个自定义读取器，这样在设计时就会有很大的灵活性。

1. 转换 XML

第一个示例接受 books.xml 文档，并使用 XSLT 文件 book.xsl 把它转换为一个简单的 HTML 文档，以进行显示(这段代码在 XSLSample01 文件夹中)，需要添加如下 using 语句：

```
using System.IO;
using System.Xml.Xsl;
using System.Xml.XPath;
```

下面是执行转换的代码(代码文件 XslSample01.sln)：

```
private void button1_Click(object sender, EventArgs e)
{
  XslCompiledTransform trans = new XslCompiledTransform();
  trans.Load("books.xsl");
  trans.Transform("books.xml", "out.html");
  webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "out.html");
}
```

这就是一个最简单的转换。首先新建一个 XmlCompiledTransform 对象。它加载 books.xsl 转换文档，然后执行转换。在本例中，把带文件名的字符串用作输入。输出是 out.html。之后把这个文件加载到窗体使用的 Web 浏览器控件上。这里不把文件名 books.xml 用作输入文档，而是使用一个基于 IXPathNavigable 的对象。它可以是创建 XPathNavigator 的任何对象。

创建 XmlCompiledTransform 对象，加载样式表后，就执行转换。Transform()方法的参数可以是 IXPathNavigable 对象、流、TextWriter、XmlWriter 和 URI 的任意组合，因此转换流上的灵活性很大。可以把转换的结果作为输入传递给下一个转换操作。

XsltArgumentLists 和 XmlResolver 对象也包含在参数选项中。下一节介绍 XsltArgumentList 对象。基于 XmlResolver 的对象用于解析当前文档外部的数据项，如架构、证书或样式表。

books.xsl 文档是一个很简单的样式表，如下所示：

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head>
      <title>Price List</title>
    </head>
    <body>
      <table>
        <xsl:apply-templates/>
      </table>
     </body>
    </html>
     </xsl:template>
    <xsl:template match="bookstore">
       <xsl:apply-templates select="book"/>
    </xsl:template>
    <xsl:template match="book">
       <tr><td>
         <xsl:value-of select="title"/>
       </td><td>
         <xsl:value-of select="price"/>
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

2. 使用 XsltArgumentList

XsltArgumentList 是把对象和方法绑定到名称空间上的一种方式。绑定之后，就可以在转换过程中调用该方法。下面是一个示例：

```
private void button3_Click(object sender, EventArgs e)
```

```
{
  //new XPathDocument
  XPathDocument doc = new XPathDocument("books.xml");
  //new XslTransform
  XslCompiledTransform trans = new XslCompiledTransform();
  trans.Load("booksarg.xsl");
  //new XmlTextWriter since we are creating a new xml document
  XmlWriter xw = new XmlTextWriter("argSample.xml", null);
  //create the XslArgumentList and new BookUtils object
  XsltArgumentList argBook = new XsltArgumentList();
  BookUtils bu = new BookUtils();
  //this tells the argumentlist about BookUtils
  argBook.AddExtensionObject("urn:XslSample", bu);
  //new XPathNavigator
  XPathNavigator nav = doc.CreateNavigator();
  //do the transform
  trans.Transform(nav, argBook, xw);
  xw.Close();
  webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "argSample.xml");
}
```

下面是 BooksUtil 类的代码，这是将在转换过程中调用的类(代码文件 BookUtils.cs)：

```
class BookUtils
{
  public BookUtils() { }

  public string ShowText()
  {
    return "This came from the ShowText method!";
  }
}
```

下面是转换的结果。其结果已进行了格式化，以便于查看(代码文件 argSample.xml)：

```
<books>
   <discbook>
      <booktitle>The Autobiography of Benjamin Franklin</booktitle>
      <showtext>This came from the ShowText method!</showtext>
   </discbook>
   <discbook>
      <booktitle>The Confidence Man</booktitle>
      <showtext>This came from the ShowText method!</showtext>
   </discbook>
   <discbook>
      <booktitle>The Gorgias</booktitle>
      <showtext>This came from the ShowText method!</showtext>
   </discbook>
   <discbook>
      <booktitle>The Great Cookie Caper</booktitle>
      <showtext>This came from the ShowText method!</showtext>
   </discbook>
   <discbook>
      <booktitle>A Really Great Book</booktitle>
      <showtext>This came from the ShowText method!</showtext>
```

```
    </discbook>
  </books>
```

本例定义一个新的 BookUtils 类。在这个类中有一个无用的方法，它返回字符串“This came from the ShowText method!”。在 button3_Click 事件中，创建 XPathDocument 和 XslTransform 对象，前面的示例把 XML 文档和转换文档直接加载到 XslComiledTransform 对象中，这次使用 XPathNavigator 来加载文档。

接着编写下面的代码：

```
XsltArgumentList argBook=new XsltArgumentList();
BookUtils bu=new BookUtils();
argBook.AddExtensionObject("urn:XslSample",bu);
```

这段代码创建 XsltArgumentList 对象。接着创建 BookUtils 对象的一个实例，在调用 AddExtension-Object()方法时，其参数是扩展的名称空间和要从中调用方法的对象。在调用 Transform()时，其参数是 XsltArgumentList(argBook)，以及前面创建的 XPathNavigator 和 XmlWriter 对象。

下面是 booksarg.xsl 文档(基于 books.xsl)：

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:bookUtil="urn:XslSample">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:element name="books">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <xsl:element name="discbook">
      <xsl:element name="booktitle">
        <xsl:value-of select="title"/>
      </xsl:element>
      <xsl:element name="showtext">
        <xsl:value-of select="bookUtil:ShowText()"/>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

两行重要的代码已突出显示。首先，在给 XsltArgumentList 添加对象时，添加前面创建的名称空间。然后，在调用对应方法时，使用标准的 XSLT 名称空间前缀语法。

另一种方式是使用 XSLT 脚本完成。可以在该样式表中包含 C#、VB 和 JavaScript 代码。最重要的是在 Transform.Load()调用中编译该脚本，这与目前的非.NET 实现方式不同。这样就可以执行已经编译的脚本。

下面对前面的 XSLT 文件也进行这样的修改。首先给样式表添加脚本，在 booksscript.xsl 中进行的修改如下所示：

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                              xmlns:msxsl="urn:schemas-microsoft-com:xslt"
                              xmlns:user="http://wrox.com">
  <msxsl:script language="C#" implements-prefix="user">
    string ShowText()
      {
         return"This came from the ShowText method!";
      }
  </msxsl:script>
  <xsl:output method="xml" indent="yes"/>
     <xsl:template match="/">
  <xsl:element name="books">
     <xsl:apply-templates/>
  </xsl:element>
     </xsl:template>
  <xsl:template match="bookstore">
     <xsl:apply-templates select="book"/>
  </xsl:template>
     <xsl:template match="book">
     <xsl:element name="discbook">
     <xsl:element name="booktitle">
        <xsl:value-of select="title"/>
     </xsl:element>
     <xsl:element name="showtext">
        <xsl:value-of select="user:ShowText()"/>
     </xsl:element>
   </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

同样，其中的改变已突出显示。设置脚本的名称空间，添加代码(可以从 VS.NET IDE 中复制和粘贴代码)，在样式表上执行调用。输出结果与前面示例的输出结果相同。

34.7.3 调试 XSLT

Visual Studio 2013 有调试转换功能。我们可以单步逐行地执行转换代码，查看变量，访问主调栈，设置断点，这些都与调试 C#源代码相同。调试转换代码有两种方式：仅使用样式表和 XML 输入文件，或者运行转换代码所在的应用程序。

1. 在不运行应用程序的情况下调试

第一次创建转换操作时，有时并不希望运行整个应用程序，只希望使样式表工作。Visual Studio 2013 允许使用 XSLT 编辑器进行这个操作。

把 books.xsl 样式表加载到 Visual Studio 2013 的 XSLT 编辑器中。在下面的代码上设置一个断点：

```
<xsl:value-of select="title"/>
```

现在选择 XML 菜单，再选择 Debug XSLT 命令。此时需要指定 XML 输入文档，这就是我们要转换的 XML。在默认配置下，结果如图 34-4 所示。

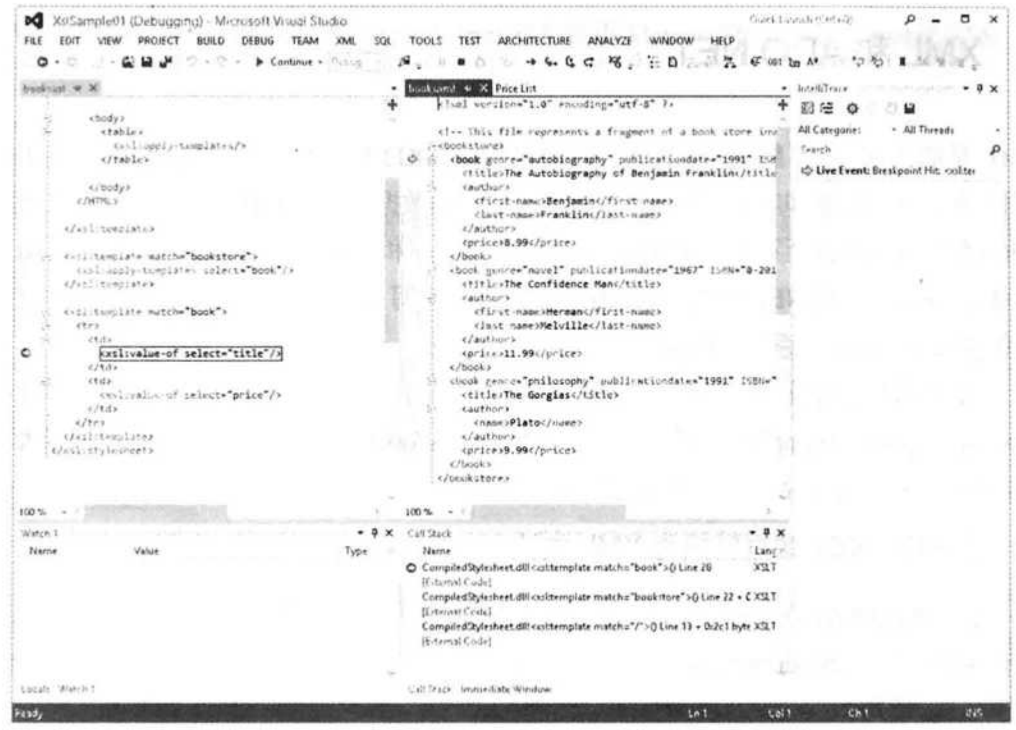


图 34-4

既然已经暂停转换，就可以查看与调试源代码时相同的几乎所有调试信息。注意调试器显示了 XSLT、输入文档、当前突出显示的元素和转换的结果。现在就可以单步执行转换代码。如果 XSLT 包含脚本代码，则还可以在脚本中设置断点，使用相同的调试功能。

2. 在运行应用程序的情况下调试

如果要同时调试转换代码和应用程序，就必须在创建 XslCompiledTransform 对象时进行一个小小的改动。构造函数的一个重载版本把一个布尔值作为参数，这个参数是 enableDebug，默认为 false，表示即使在转换代码中设置一个断点，如果运行调用转换代码的应用程序，它也不会中断。如果把该参数设置为 true，就会生成 XSLT 的调试信息，并在断点处暂停。所以在前面的示例中，创建 XlsCompiledTransform 的代码行应改为：

```
XslCompiledTransform trans = new XslCompiledTransform(true);
```

现在当应用程序运行在调试模式时，甚至 XSLT 会生成调试信息，同样可以在样式表中获得完整的 Visual Studio 调试功能。

总之，在进行转换时，一定要记住使用正确的 XML 数据存储器。如果不需要编辑功能，就使用 XPathDocument；如果要从 ADO.NET 中获得数据，就使用 XmlDataDocument；如果需要编辑数据，就使用 XmlDocument。其他过程都相同。

34.8 XML 和 ADO.NET

XML 是把 ADO.NET 绑定到其他语言中的纽带。ADO.NET 从一开始就在 XML 环境中工作。XML 用于在数据存储器和应用程序或网页之间来回传输数据。因为 ADO.NET 使用 XML 进行远程传输，所以数据可以在甚至不能识别 ADO.NET 的应用程序和系统之间交换。因为 XML 在 ADO.NET 中非常重要，所以 ADO.NET 提供了一些强大的功能来读写 XML 文档。System.Xml 名称空间也包含可以使用 ADO.NET 关系数据的类。

用于示例的数据库来自于 AdventureWorksLT 示例应用程序。示例数据库可以从 codeplex.com/SqlServerSamples 上下载。注意 AdventureWorks 数据库有几个版本，大多数版本都可以工作，但 LT 版本是简化版本，足以达到本章的目的。

34.8.1 将 ADO.NET 数据转换为 XML 文档

第一个示例使用 ADO.NET、流和 XML 把数据库中的一些数据推入 DataSet 中，从 DataSet 中加载带有 XML 的 XmlDocument 对象，并把 XML 加载到文本框中。为了运行下面几个示例，需要添加如下 using 语句：

```
using System.Data;
using System.Xml;
using System.Data.SqlClient;
using System.IO;
```

把连接字符串定义为模块级变量：

```
string _connectString = "Server=.\\SQLExpress;
                                    Database=adventureworkslt;Trusted_Connection=Yes";
```

对于 ADO.NET 示例，在窗体上添加一个 DataGrid 对象，这样就可以在 ADO.NET 的 DataSet 中查看数据，因为数据被绑定到网格上。还可以查看生成的 XML 文档中的数据，这些数据已加载到文本框中。下面是第一个示例的代码。本示例的第一步是创建标准的 ADO.NET 对象，以生成一个 DataSet 对象。之后把该数据集绑定到网格上(代码文件 frmADOXML.cs)。

```
private void button1_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  DataSet ds = new DataSet("XMLProducts");
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter da = new SqlDataAdapter
                    ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
  //fill the dataset
  da.Fill(ds, "Products");
  //load data into grid
  dataGridView1.DataSource = ds.Tables["Products"];
```

在创建了 ADO.NET 对象，并绑定到网格上后，再实例化 MemoryStream、StreamReader 和

StreamWriter 对象。StreamReader 和 StreamWriter 对象使用 MemoryStream 对象来浏览 XML 文档：

```
MemoryStream memStrm=new MemoryStream();
StreamReader strmRead=new StreamReader(memStrm);
StreamWriter strmWrite=new StreamWriter(memStrm);
```

使用 MemoryStream 的原因是不必把数据写入磁盘中。但还可以使用基于 Stream 类的其他对象，如 FileStream。

下一步是生成 XML。调用 DataSet 类的 WriteXml()方法，它生成一个 XML 文档。WriteXml()方法有两个重载版本，一个重载版本的参数是带有文件路径和名称的字符串，另一个重载版本还有一个模式参数。这个模式是一个 XmlWriteMode 枚举，其值可以是

- IgnoreSchema
- WriteSchema
- DiffGram

如果不希望 WriteXml()方法把内联架构写入 XML 文件的开头，就使用 IgnoreSchema 参数；如果要写入内联架构，就使用 WriteSchema 参数。DiffGram 显示在 DataSet 中编辑前后的数据。

```
//write the xml from the dataset to the memory stream
  ds.WriteXml(strmWrite, XmlWriteMode.IgnoreSchema);
  memStrm.Seek(0, SeekOrigin.Begin);
  //read from the memory stream to a XmlDocument object
  doc.Load(strmRead);
  //get all of the products elements
  XmlNodeList nodeLst = doc.SelectNodes("//XMLProducts/Products");
  textBox1.Text = "";

  foreach (XmlNode node in nodeLst)
  {
    textBox1.Text += node.InnerXml + "\r\n";
  }
```

在图 34-5 所示的对话框中，可以查看列表中的数据和绑定的数据网格。

如果只需要该架构，就应该调用 WriteXmlSchema()方法而不是 WriteXml()方法。这个方法有 4 个重载版本。第 1 个重载版本的参数是一个字符串，其中包含了写入 XML 文档的位置对应的路径和文件名。第 2 个重载版本使用基于 XmlWriter 类的一个对象。第 3 个重载版本使用基于 TextWriter 类的对象。第 4 个重载版本使用派生自 Stream 类的对象。

此外，如果要把 XML 文档持久化到磁盘中，就应执行下述操作：

```
string file = "c:\\test\\product.xml";
ds.WriteXml(file);
```

在磁盘上会生成一个格式良好的 XML 文档，它可以由另一个流或 DataSet 来读取，或者由另一个应用程序或 Web 站点使用。因为没有指定 XmlMode 参数，所以这个 XmlDocument 包含了该架构。在本例中，把流作为 XmlDocument.Load()方法的参数。

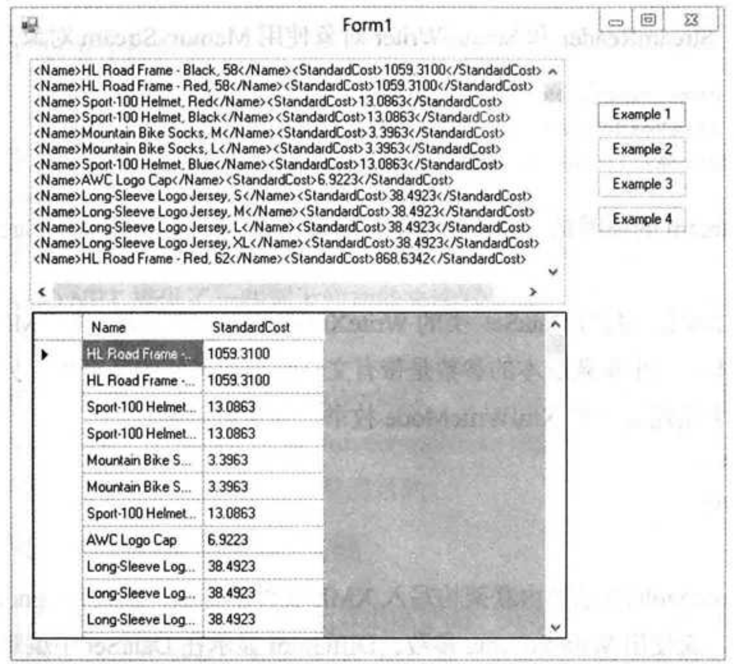


图 34-5

现在数据有两个视图，但更重要的是，可以使用两个不同的模型来处理数据。可以使用System.Data名称空间来操纵数据，也可以使用System.Xml名称空间来处理数据。这样应用程序就可以有某些非常灵活的设计，因为现在不必把一个对象模型绑定到程序上。这是ADO.NET和System.Xml组合的强大之处。相同数据可以有多个视图，访问数据也有多种方式。

下一个示例消除3个流，并使用内置于System.Xml名称空间中的一些ADO功能来简化过程。但需要修改模块级的代码行：

```
private XmlDocument doc = new XmlDocument();
```

为：

```
private XmlDataDocument doc;
```

这么做的原因是现在使用的是XmlDataDocument。下面是代码：

```
private void button3_Click(object sender, EventArgs e)
{
  XmlDataDocument doc;
  //create a dataset
  DataSet ds = new DataSet("XMLProducts");
  //connect to the northwind database and
  //select all of the rows from products table
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter da = new SqlDataAdapter
  ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
  //fill the dataset
  da.Fill(ds, "Products");
  ds.WriteXml("sample.xml", XmlWriteMode.WriteSchema);
  //load data into grid
  dataGridView1.DataSource = ds.Tables[0];
```

```
    doc = new XmlDataDocument(ds);
    //get all of the products elements
    XmlNodeList nodeLst = doc.GetElementsByTagName("Products");
    textBox1.Text = "";
    foreach (XmlNode node in nodeLst)
    {
      textBox1.Text += node.InnerXml + "\r\n";
    }
}
```

可以看出，把DataSet对象加载到XML文档中的代码已经进行了简化。它没有使用XmlDocument类，而是使用 XmlDataDocument 类，这个类是为了使用 DataSet 对象的数据而专门设计的。

因为 XmlDataDocument 类基于 XmlDocument 类，所以它拥有 XmlDocument 类的所有功能。一个主要区别是 XmlDataDocument 类有重载的构造函数。注意下面的代码实例化 XmlDataDocument 对象 doc：

```
doc = new XmlDataDocument(ds);
```

它传递的参数是我们创建的 DataSet 对象 ds，从 DataSet 对象中创建 XML 文档，而且不必使用 Load()方法。实际上，如果实例化一个新的 XmlDataDocument 对象而不把 DataSet 作为参数，该 XmlDataDocument 就会包含一个名为 NewDataSet 的 DataSet，其 table 集合中没有任何 DataTable。在创建了基于 XmlDataDocument 的对象后，还可以设置 DataSet 属性。

如果把下面一行代码添加到 DataSet.Fill 调用的后面：

```
ds.WriteXml("c:\\test\\sample.xml", XmlWriteMode.WriteSchema);
```

就会在文件夹 c:\test 中生成下面的 XML 文件 sample.xml：

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
  <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="XMLProducts" msdata:IsDataSet="true"
     msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Products">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Name" type="xs:string" minOccurs="0" />
                <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  <Products>
    <Name>HL Road Frame-Black, 58</Name>
    <StandardCost>1059.3100</StandardCost>
  </Products>
```

```
  <Products>
    <Name>HL Road Frame-Red, 58</Name>
    <StandardCost>1059.3100</StandardCost>
  </Products>
  <Products>
    <Name>Sport-100 Helmet, Red</Name>
    <StandardCost>13.0863</StandardCost>
  </Products>
</XMLProducts>
```

这里只显示了前几个 Products 元素。实际的 XML 文件应包含 Northwind 数据库中 Products 表的所有 Products 元素。

转换关系数据

这看起来非常简单，因为只有一个表。但对于关系数据，如 DataSet 中有多个 DataTable 和 Relation，该怎么办？其工作方式仍旧是这样。下面的示例使用了两个关系表(代码文件 frmADOXML.cs)：

```
private void button5_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  DataSet ds = new DataSet("XMLProducts");
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter daProduct = new SqlDataAdapter
  ("SELECT Name, StandardCost, ProductCategoryID FROM SalesLT.Product", conn);
  SqlDataAdapter daCategory = new SqlDataAdapter
        ("SELECT ProductCategoryID, Name from SalesLT.ProductCategory", conn);
  //Fill DataSet from both SqlAdapters
  daProduct.Fill(ds, "Products");
  daCategory.Fill(ds, "Categories");
  //Add the relation
  ds.Relations.Add(ds.Tables["Categories"].Columns["ProductCategoryID"],
  ds.Tables["Products"].Columns["ProductCategoryID"]);
  //Write the Xml to a file so we can look at it later
  ds.WriteXml("Products.xml", XmlWriteMode.WriteSchema);
  //load data into grid
  dataGridView1.DataSource = ds.Tables[0];
  //create the XmlDataDocument
  doc = new XmlDataDocument(ds);
  //Select the productname elements and load them in the grid
  XmlNodeList nodeLst = doc.SelectNodes("//XMLProducts/Products");
  textBox1.Text = "";
  foreach (XmlNode node in nodeLst)
  {
    textBox1.Text += node.InnerXml + "\r\n";
  }
}
```

在这个示例中，在 XMLProducts 数据集中创建了两个 DataTable：Products 和 Categories。在两个表的 ProductCategoryID 列上新建一个关系。

使用与上一个示例相同的 WriteXml()方法调用，得到如下 XML 文件(代码文件 SuppProd.xml)：

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
  <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="XMLProducts" msdata:IsDataSet="true"
     msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Products">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Name" type="xs:string" minOccurs="0" />
                <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
                <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="Categories">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
                <xs:element name="Name" type="xs:string" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
      <xs:unique name="Constraint1">
        <xs:selector xpath=".//Categories" />
        <xs:field xpath="ProductCategoryID" />
      </xs:unique>
      <xs:keyref name="Relation1" refer="Constraint1">
        <xs:selector xpath=".//Products" />
        <xs:field xpath="ProductCategoryID" />
      </xs:keyref>
    </xs:element>
  </xs:schema>
  <Products>
    <Name>HL Road Frame-Black, 58</Name>
    <StandardCost>1059.3100</StandardCost>
    <ProductCategoryID>18</ProductCategoryID>
  </Products>
  <Products>
    <Name>HL Road Frame-Red, 58</Name>
    <StandardCost>1059.3100</StandardCost>
    <ProductCategoryID>18</ProductCategoryID>
  </Products>
</XMLProducts>
```

该架构包含 DataSet 中的两个 DataTable。此外，数据包含两个表中的所有数据。为简洁起见，这里只显示第一个 Products 和 ProductCategory 记录。与以前一样，使用正确的 XmlWriteMode 参数可以只保存架构或数据。

34.8.2 把 XML 文档转换为 ADO.NET 数据

假设有一个 XML 文档，准备把它转换为 ADO.NET 的 DataSet。这样就可以把 XML 加载到数据库中，或者把数据绑定到.NET 数据控件上，如 DataGrid。此时实际上可以把 XML 文档用作数据存储器，完全消除数据库的系统开销。如果数据比较少，这完全有可能。看看下面这些代码：

```
private void button7_Click(object sender, EventArgs e)
{
  //create the DataSet
  DataSet ds = new DataSet("XMLProducts");

  //read in the xml document
  ds.ReadXml("Products.xml");

  //load data into grid
  dataGridView1.DataSource = ds.Tables[0];

  textBox1.Text = "";

  foreach (DataTable dt in ds.Tables)
  {
    textBox1.Text += dt.TableName + "\r\n";
    foreach (DataColumn col in dt.Columns)
    {
      textBox1.Text += "\t" + col.ColumnName + "-" + col.DataType.FullName +
       "\r\n";
    }
  }
}
```

这很简单。这个示例实例化一个新的 DataSet 对象，然后，从此处调用 ReadXml()方法，把 XML 放在 DataSet 的一个 DataTable 中。与 WriteXml()方法一样，ReadXml()方法的参数是 XmlReadMode。ReadXml()方法还可以使用 XmlReadMode 中的更多选项(如表 34-9 所示)。

表 34-9

选　项	说　明
Auto	把 XmlReadMode 设置为最合适的值。如果数据是 DiffGram 格式，就选择 DiffGram；如果已经读取了架构，或者检测到某个内联架构，就选择 ReadSchema。如果没有为 DataSet 指定架构，也没有检测到内联架构，就选择 IgnoreSchema
DiffGram	读取 DiffGram，并把变化应用到 DataSet 上
Fragment	读取包含 XDR 架构片段的文档，如 SQL Server 创建的类型
IgnoreSchema	忽略任何找到的内联架构，把数据读入当前的 DataSet 架构中，如果数据与 DataSet 架构不匹配，就删除它
InferSchema	忽略任何内联架构，根据 XML 文档中的数据创建架构。如果 DataSet 中已经有一个架构，就使用该架构，根据需要，可以用其他列或表来扩展它。如果存在一列，但其数据类型不符，就会抛出一个异常
ReadSchema	读取内联架构，并加载数据。不重写 DataSet 中的架构，但如果内联架构中的表已经存在于 DataSet 中，就抛出一个异常

这里也有 ReadXmlSchema()方法，它读取独立的架构，并创建相应的表、列和关系。如果架构没有和数据建立关联，就可以使用 ReadXmlSchema()方法。该方法也有 4 个重载版本：其参数分别是包含文件和路径名的字符串、基于 Stream 的对象、基于 TextReader 的对象和基于 XmlReader 的对象。

要说明如何正确地创建数据表，可以迭代表和列，并在文本框中显示名称。可以把它与最初的数据库比较一下，可以发现所有内容保持不变。最后一个 foreach 循环执行这个任务。图 34-6 显示了输出结果。

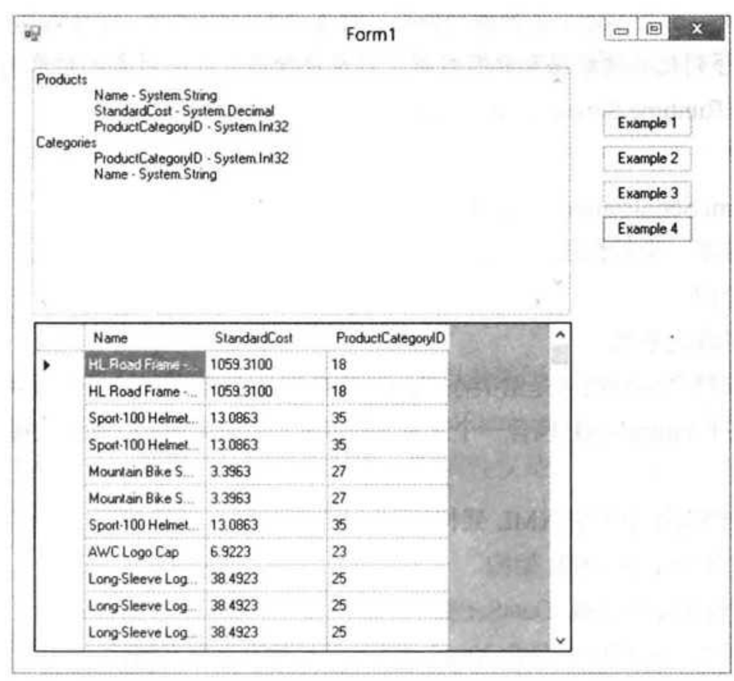


图 34-6

从列表框中可以看出，数据表中包含的所有列都有正确的名称和数据类型。

还要注意，前两个示例都没有在数据库中传入或传出任何数据，所以没有定义 SqlDataAdapter 或 SqlConnection。这说明了 System.Xml 名称空间和 ADO.NET 的灵活性：可以用多种格式查看相同的数据。如果需要进行转换，以 HTML 格式显示数据，或者如果需要把数据绑定到网格上，就应获取这些数据，用一个方法调用，把它们以需要的格式显示出来。

34.9　在 XML 中序列化对象

序列化是把一个对象持久化到磁盘中的过程。应用程序的另一部分，甚至另一个应用程序都可以反序列化对象，使它的状态与序列化之前相同。.NET Framework 为此提供了两种方式。

本节将介绍 System.Xml.Serialization 名称空间。它包含的类可用于把对象序列化为 XML 文档或流。这表示把对象的公共属性和公共字段转换为 XML 元素和/或属性。

System.Xml.Serialization 名称空间中最重要的类是 XmlSerializer。要序列化对象，首先需要实例化一个 XmlSerializer 对象，指定要序列化的对象类型，然后实例化一个流/写入器对象，以把文件写入流/文档中。最后一步是在 XmlSerializer 上调用 Serializer()方法，给它传递流/写入器对象和要序列化的对象。

被序列化的数据可以为基元类型的数据、字段、数组，以及 XmlElement 和 XmlAttribute 对象格式的内嵌 XML。为了从 XML 文档中反序列化对象，应执行上述过程的逆过程。即创建一个流/读取器对象和一个 XmlSerializer 对象，然后给 DeSerializer()方法传递该流/读取器对象。这个方法返回反序列化的对象，尽管它需要强制转换为正确的类型。

XML 序列化程序不能转换私有数据，只能转换公共数据，它也不能序列化对象图表。但是，这并不是一个严格的限制。对类进行仔细设计，就很容易避免这个问题。如果需要序列化公共数据和私有数据，以及包含许多嵌套对象的对象图表，就可以使用 System.Runtime.Serialization.Formatters.Binary 名称空间。

使用 System.Xml.Serialization 类可以完成的其他工作如下所示：

- 确定数据应是一个特性还是元素
- 指定名称空间
- 改变特性名或元素名

对象和 XML 文档之间的链接是给类添加注释的自定义 C#特性，这些属性可以告诉序列化程序如何输出数据。.NET Framework 包含一个工具 xsd.exe，它可以帮助创建这些特性。xsd.exe 可以完成如下任务：

- 从 XDR 架构文件中生成 XML 架构
- 从 XML 文件中生成 XML 架构
- 从 XSD 架构文件中生成 DataSet 类
- 生成运行库类，运行库类包含 XmlSerialization 类的自定义属性
- 从已经开发出来的类中生成 XSD 文件
- 限制在代码中创建的元素
- 确定生成代码的编程语言(C#、VB 或 JScript.NET)
- 在编译好的程序集中从类型中创建架构

参见.NET Framework 文档，了解 xsd.exe 命令行选项的详细内容。

尽管 xsd.exe 具备这些功能，但不一定用它为序列化创建类。这个过程很简单。下面介绍一个简单的应用程序，它序列化一个类。示例代码的起始部分比较简单，新建一个 Product 对象 pd，并给它填充一些数据(代码文件 frmSerial.cs)：

```
private void button1_Click(object sender, EventArgs e)
{
  //new products object
  Product pd = new Product();
  //set some properties
  pd.ProductID = 200;
  pd.CategoryID = 100;
  pd.Discontinued = false;
  pd.ProductName = "Serialize Objects";
  pd.QuantityPerUnit = "6";
  pd.ReorderLevel = 1;
  pd.SupplierID = 1;
  pd.UnitPrice = 1000;
```

```
    pd.UnitsInStock = 10;
    pd.UnitsOnOrder = 0;

}
```

XmlSerializer 类的 Serialize()方法实际上执行序列化，它有 9 个重载版本。一个必需的参数是要写入数据的流，它可以是 Stream、TextWriter 或 XmlWriter 参数。在本例中，创建一个基于 TextWriter 的对象 tr。接着创建基于 XmlSerializer 类的对象 sr。XmlSerializer 类需要知道正在序列化的对象的类型信息，所以对要序列化的类型使用 typeof 关键字。在创建 sr 对象后，调用 Serialize()方法，其参数是 tr(基于 Stream 的对象)和要序列化的对象，在本例中是 pd。确保完成后关闭该数据流。

```
//new TextWriter and XmlSerializer
TextWriter tr = new StreamWriter("serialprod.xml");
XmlSerializer sr = new XmlSerializer(typeof(Product));
//serialize object
sr.Serialize(tr, pd);
tr.Close();
webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "serialprod.xml");
```

下面介绍 Products 类，即要序列化的类。这个类与以前编写的任何其他类的唯一区别是给它添加了 C#特性。这些特性中的 XmlRootAttribute 类和 XmlElementAttribute 类继承自 System.Attribute 类。不要把这些特性与 XML 文档中的特性相混淆。C#属性仅是一些声明信息，在运行期间可以由 CLR 检索到。在本例中，特性描述了如何序列化对象：

```
//class that will be serialized.
//attributes determine how object is serialized
[System.Xml.Serialization.XmlRootAttribute()]
  public class Product {
    private int prodId;
    private string prodName;
    private int suppId;
    private int catId;
    private string qtyPerUnit;
    private Decimal unitPrice;
    private short unitsInStock;
    private short unitsOnOrder;
    private short reorderLvl;
    private bool discont;
    private int disc;
    //added the Discount attribute
    [XmlAttributeAttribute(AttributeName="Discount")]
    public int Discount {
      get {return disc;}
      set {disc=value;}
    }
    [XmlElementAttribute()]
    public int ProductID {
      get {return prodId;}
      set {prodId=value;}
    }
    [XmlElementAttribute()]
    public string ProductName {
```

```
      get {return prodName;}
      set {prodName=value;}
    }
    [XmlElementAttribute()]
    public int SupplierID {
      get {return suppId;}
      set {suppId=value;}
    }
    [XmlElementAttribute()]
    public int CategoryID {
      get {return catId;}
      set {catId=value;}
    }
    [XmlElementAttribute()]
    public string QuantityPerUnit {
      get {return qtyPerUnit;}
      set {qtyPerUnit=value;}
    }
    [XmlElementAttribute()]
    public Decimal UnitPrice {
      get {return unitPrice;}
      set {unitPrice=value;}
    }
    [XmlElementAttribute()]
    public short UnitsInStock {
      get {return unitsInStock;}
      set {unitsInStock=value;}
    }
    [XmlElementAttribute()]
    public short UnitsOnOrder {
      get {return unitsOnOrder;}
      set {unitsOnOrder=value;}
    }
    [XmlElementAttribute()]
    public short ReorderLevel {
      get {return reorderLvl;}
      set {reorderLvl=value;}
    }
    [XmlElementAttribute()]
    public bool Discontinued {
      get {return discont;}
      set {discont=value;}
    }
    public override string ToString()
    {
      StringBuilder outText = new StringBuilder();
      outText.Append(prodId);
      outText.Append(" ");
      outText.Append(prodName);
      outText.Append(" ");
      outText.Append(unitPrice);
      return outText.ToString();
    }
  }
```

在 Products 类定义上面的特性中调用的 XmlRootAttribute()方法把这个类标识为根元素(在 XML 文件中，在序列化时生成)。包含 XmlElementAttribute()方法的特性指出，该特性下面的成员表示一个 XML 元素。

注意重写了 ToString()方法，它提供了运行反序列化示例时显示在消息框中的字符串。

查看一下刚才在序列化过程中创建的 XML 文档，就会发现它与前面创建的其他 XML 文档非常类似。这就是本练习的目的。下面就是这个文档：

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  Discount="0">
  <ProductID>200</ProductID>
  <ProductName>Serialize Objects</ProductName>
  <SupplierID>1</SupplierID>
  <CategoryID>100</CategoryID>
  <QuantityPerUnit>6</QuantityPerUnit>
  <UnitPrice>1000</UnitPrice>
  <UnitsInStock>10</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>1</ReorderLevel>
  <Discontinued>false</Discontinued>
</Products>
```

这里没有任何不寻常的地方。可以以使用 XML 文档的任何方式来使用这个文档。可以对它进行转换，并以 HTML 格式显示它，使用 ADO.NET 将其加载到 DataSet 中，用它加载 XmlDocument，或者像在该示例中那样，对它进行反序列化，并创建一个对象，该对象的状态与序列化前 pd 的状态一样(这就是第二个按钮的作用)。

接着添加另一个按钮事件处理程序，以反序列化一个基于 Products 的新对象 newPd。这次使用 FileStream 对象读取 XML：

```
private void button2_Click(object sender, EventArgs e)
    {
     //create a reference to product type
     Product newPd;
     //new filestream to open serialized object
     FileStream f = new FileStream("serialprod.xml", FileMode.Open);
```

同样，传递 Product 的类型信息，新建一个 XmlSerializer 类。然后就可以调用 Deserialize()方法。注意在创建 newPd 对象时，仍需要进行显式的类型强制转换。此时 newPd 与 pd 的状态完全一样：

```
//new serializer
      XmlSerializer newSr = new XmlSerializer(typeof(Product));
      //deserialize the object
      newPd = (Product)newSr.Deserialize(f);
      f.Close();
      MessageBox.Show(newPd.ToString());
    }
```

消息框应显示产品 ID、产品名称和刚才反序列化的对象的单价。这是因为使用了在 Product 类中实现的 ToString()方法。

如果有派生的类和可能返回一个数组的属性，则也可以使用 XmlSerializer 类。下面介绍一个解决这些问题的复杂示例。

首先定义 3 个新类 Product、BookProduct (派生于 Product)和 Inventory (它包含其他两个类)。注意又重写了 ToString()方法，这次要列出 Inventory 类中的项：

```
public class BookProduct: Product
{
  private string isbnNum;
  public BookProduct() {}
  public string ISBN
  {
    get {return isbnNum;}
    set {isbnNum=value;}
  }
}

public class Inventory
{
  private Product[] stuff;
  public Inventory() {}
  //need to have an attribute entry for each data type
  [XmlArrayItem("Prod",typeof(Product)),
  XmlArrayItem("Book",typeof(BookProduct))]
  public Product[] InventoryItems
  {
    get {return stuff;}
    set {stuff=value;}
  }
  public override string ToString()
  {
    StringBuilder outText = new StringBuilder();
    foreach (Product prod in stuff)
    {
      outText.Append(prod.ProductName);
      outText.Append("\r\n");
    }
    return outText.ToString();
  }
}
```

这里只对 Inventory 类感兴趣。如果序列化这个类，就需要插入一个特性，该特性为每个要添加到数组中的类型包含一个 XmlArrayItem 构造函数。注意，XmlArrayItem 是由 XmlArrayItem-Attribute 类表示的.NET 特性的名称。

这些构造函数的第一个参数是在序列化过程中创建的 XML 文档中的元素名。如果不使用 ElementName 参数，元素名就会与对象类型名相同(在本例中，就是 Product 和 BookProduct)。必须指定的第二个参数是对象的类型。

如果属性返回一个对象数组或基元类型的数组，则还要使用 XmlArrayAttribute 类。因为要在数组中返回不同的类型，所以使用 XmlArrayItemAttribute 类，它允许进行更高级别的控制。

在 button4_Click()事件处理程序中，新建一个 Product 对象和一个 BookProduct 对象(newProd 和 newBook)。给每个对象的各种属性添加数据，再把这些对象添加到一个 Product 数组中。接下来新

建一个 Inventory 对象，并把这个数组作为参数，然后序列化 Inventory 对象，以便在以后重新创建它：

```
private void button4_Click(object sender, EventArgs e)
{
  //create the XmlAttributes object
  XmlAttributes attrs = new XmlAttributes();
  //add the types of the objects that will be serialized
  attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
  attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));
  XmlAttributeOverrides attrOver = new XmlAttributeOverrides();
  //add to the attributes collection
  attrOver.Add(typeof(Inventory), "InventoryItems", attrs);
  //create the Product and Book objects
  Product newProd = new Product();
  BookProduct newBook = new BookProduct();
  newProd.ProductID = 100;
  newProd.ProductName = "Product Thing";
  newProd.SupplierID = 10;
  newBook.ProductID = 101;
  newBook.ProductName = "How to Use Your New Product Thing";
  newBook.SupplierID = 10;
  newBook.ISBN = "123456789";
  Product[] addProd ={ newProd, newBook };
  Inventory inv = new Inventory();
  inv.InventoryItems = addProd;
  TextWriter tr = new StreamWriter("inventory.xml");
  XmlSerializer sr = new XmlSerializer(typeof(Inventory), attrOver);
  sr.Serialize(tr, inv);
  tr.Close();
  webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "inventory.xml");
}
```

XML 文档如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
  </Product>
  <Book Discount="0">
    <ProductID>101</ProductID>
    <ProductName>How to Use Your New Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
```

```
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
    <ISBN>123456789</ISBN>
  </Book>
</Inventory>
```

button2_Click()事件处理程序实现Inventory对象的反序列化。注意在新建的newInv对象中，我们迭代了数组，以说明其数据保持不变：

```
private void button2_Click(object sender, System.EventArgs e)
{
  Inventory newInv;
  FileStream f=new FileStream("order.xml",FileMode.Open);
  XmlSerializer newSr=new XmlSerializer(typeof(Inventory));
  newInv=(Inventory)newSr.Deserialize(f);
  foreach(Product prod in newInv.InventoryItems)
     listBox1.Items.Add(prod.ProductName);
  f.Close();
}
```

不能访问源代码的序列化

这些代码都很好地发挥了作用，但如果不能访问已经序列化的类型的源代码，该怎么办？如果没有源代码，就不能添加属性。此时可以采用另一种方式。可以使用XmlAttributes类和XmlAttributeOverrides类，这两个类可以完成刚才的任务，但不需要添加属性。下面的代码说明了这两个类的工作方式。

对于这个示例，假定Inventory、Product和派生的BookProduct类在一个单独的DLL中，而且没有源代码。Product和BookProduct类与前面的示例相同，但应注意Inventory类中没有添加属性：

```
public class Inventory
{
  private Product[] stuff;
  public Inventory() {}
  public Product[] InventoryItems
  {
    get {return stuff;}
    set {stuff=value;}
  }
}
```

下面处理button1_Click事件处理程序中的序列化：

```
private void button1_Click(object sender, System.EventArgs e)
{
```

序列化过程的第一步是创建一个XmlAttributes对象，为每个要重写的数据类型创建一个XmlElementAttribute对象：

```
XmlAttributes attrs=new XmlAttributes();
attrs.XmlElements.Add(new XmlElementAttribute("Book",typeof(BookProduct)));
attrs.XmlElements.Add(new XmlElementAttribute("Product",typeof(Product)));
```

从中可以看出，我们给 XmlAttributes 类的 XmlElements 集合添加了新的 XmlElementAttribute 对象。XmlAttributes 类的属性对应于可以应用的特性，前面示例中的 XmlArray 和 XmlArrayItems 仅是其中的几个属性而已。现在我们有一个 XmlAttributes 对象，并在 XmlElements 集合中添加了两个基于 XmlElementAttribute 的对象。

接着创建 XmlAttributeOverrides 对象：

```
XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
attrOver.Add(typeof(Inventory),"InventoryItems",attrs);
```

这个类的 Add()方法有两个重载版本。第一个重载版本的参数是要重写的对象的类型信息和前面创建的 XmlAttributes 对象，本例使用第二个重载版本，其参数也是一个字符串值，该字符串值是重写对象的成员。在本例中，要重写 Inventory 类中的 InventoryItems 成员。

创建 XmlSerializer 对象时，把 XmlAttributeOverrides 对象添加为参数。现在 XmlSerializer 类知道我们要重写的类和需要为这些类型返回的内容。

```
//create the Product and Book objects
   Product newProd=new Product();
   BookProduct newBook=new BookProduct();
   newProd.ProductID=100;
   newProd.ProductName="Product Thing";
   newProd.SupplierID=10;
   newBook.ProductID=101;
   newBook.ProductName="How to Use Your New Product Thing";
   newBook.SupplierID=10;
   newBook.ISBN="123456789";
   Product[] addProd={newProd,newBook};

   Inventory inv=new Inventory();
   inv.InventoryItems=addProd;
   TextWriter tr=new StreamWriter("inventory.xml");
   XmlSerializer sr=new XmlSerializer(typeof(Inventory),attrOver);
   sr.Serialize(tr,inv);
   tr.Close();
}
```

如果执行 Serialize()方法，就会得到如下 XML 输出：

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
  </Product>
  <Book Discount="0">
```

```
      <ProductID>101</ProductID>
      <ProductName>How to Use Your New Product Thing</ProductName>
      <SupplierID>10</SupplierID>
      <CategoryID>0</CategoryID>
      <UnitPrice>0</UnitPrice>
      <UnitsInStock>0</UnitsInStock>
      <UnitsOnOrder>0</UnitsOnOrder>
      <ReorderLevel>0</ReorderLevel>
      <Discontinued>false</Discontinued>
      <ISBN>123456789</ISBN>
    </Book>
  </Inventory>
```

可以看出，得到的 XML 与前面的示例完全相同。为了反序列化对象，重新创建基于 Inventory 的对象，需要创建在序列化对象时创建所有相同的 XmlAttributes、XmlElementAttribute 和 XmlAttributeOverrides 对象。之后就可以读取 XML，像以前那样重新创建 Inventory 对象。下面的代码反序列化 Inventory 对象：

```
private void button2_Click(object sender, System.EventArgs e)
{
  //create the new XmlAttributes collection
  XmlAttributes attrs=new XmlAttributes();
  //add the type information to the elements collection
  attrs.XmlElements.Add(new XmlElementAttribute("Book",typeof(BookProduct)));
  attrs.XmlElements.Add(new XmlElementAttribute("Product",typeof(Product)));

  XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
  //add to the Attributes collection
  attrOver.Add(typeof(Inventory),"InventoryItems",attrs);

  //need a new Inventory object to deserialize to
  Inventory newInv;

  //deserialize and load data into the listbox from deserialized object
  FileStream f=new FileStream(".\\.\\.\\inventory.xml",FileMode.Open);
  XmlSerializer newSr=new XmlSerializer(typeof(Inventory),attrOver);

  newInv=(Inventory)newSr.Deserialize(f);
  if(newInv!=null)
  {
    foreach(Product prod in newInv.InventoryItems)
    {
      listBox1.Items.Add(prod.ProductName);
    }
  }
  f.Close();
}
```

注意，前几行代码与序列化对象所用的代码相同。

System.Xml.XmlSerialize 名称空间提供了一个功能非常强大的工具集，可以把对象序列化到 XML 中。把对象序列化和反序列化到 XML 中替代了把对象保存为二进制格式，因此可以通过 XML 对对象进行其他处理。这将大大增强设计的灵活性。

34.10　LINQ to XML 和.NET

把 LINQ 引入.NET Framework 中时，其重点是便于访问要在应用程序中使用的数据。由于在应用程序存储空间中，一个主要的数据存储器是 XML，因此很自然地演变为创建 LINQ to XML。

在 LINQ to XML 发布之前，通过 System.Xml 使用 XML 并不是很容易实现。引入 System.Xml.Linq 名称空间后，就可以利用一系列功能在代码中方便地处理 XML。

在应用程序代码中创建 XML 时，许多开发人员以前都使用 XmlDocument 对象。这个对象可以创建 XML 文档，以层次结构的方式追加元素、特性和其他项。通过 LINQ to XML 和引入的 System.Xml.Linq 名称空间，一些工具使 XML 文档的创建更加简单。

34.11　使用不同的 XML 对象

除了.NET 4.5 包含的 LINQ 查询功能之外，.NET Framework 提供的 XML 对象也非常好，它们甚至可以独立于 LINQ。在该版本中可以使用 XML 对象取代 DOM 的直接处理。在 System.Xml.Linq 名称空间中，有一系列 LINQ to XML 帮助对象，简化了内存中的 XML 文档的处理。

下面几节介绍这个名称空间中可用的对象。

本章的许多示例都使用了 Hamlet.xml 文件。这个 XML 文件在 http://metalab.unc.edu/bosak/xml/eg/shaks200.zip 上，以 XML 文件格式包含莎士比亚的所有戏剧。

34.11.1　XDocument 对象

XDocument 对象替代了.NET 3.5 之前的 XmlDocument 对象，它更容易处理 XML 文档。XDocument 对象还和这个名称空间中的其他新对象一起使用，如 XNamespace、XComment、XElement 和 XAttribute 对象。

XDocument 对象的一个更重要的成员是 Load()方法：

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
```

这个操作会把 Hamlet.xml 文件的内容加载为内存中的一个 XDocument 对象。还可以给 Load()方法传递一个 TextReader 或 XmlReader 对象。现在就可以以编程方式处理 XML 了(代码文件 ConsoleApplication1.sln)：

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
Console.WriteLine(xdoc.Root.Name.ToString());
Console.WriteLine(xdoc.Root.HasAttributes.ToString());
```

输出的结果如下：

```
PLAY
False
```

另一个重要的成员是 Save()方法，它类似于 Load()方法，可以保存到一个物理磁盘位置，或一个 TextWriter 或 XmlWriter 对象中：

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");

xdoc.Save(@"C:\CopyOfHamlet.xml");
```

34.11.2 XElement 对象

一个常用的对象是 XElement。使用这个对象可以轻松地创建包含单个元素的对象，该对象可以是 XML 文档本身，也可以只是 XML 片段。例如，下面的例子写入一个 XML 元素及其相应的值：

```
XElement xe = new XElement("Company", "Lipper");
Console.WriteLine(xe.ToString());
```

在创建 XElement 对象时，可以定义该元素的名称和元素中使用的值。在这个例子中，元素的名称是<Company>，<Company>元素的值是 Lipper。在引用 System.Xml.Linq 名称空间的控制台应用程序中运行它，得到的结果如下：

```
<Company>Lipper</Company>
```

还可以使用多个 XElement 对象创建比较完整的 XML 文档，如下例所示：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XElement xe = new XElement("Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

运行这个应用程序，得到的结果如图 34-7 所示。

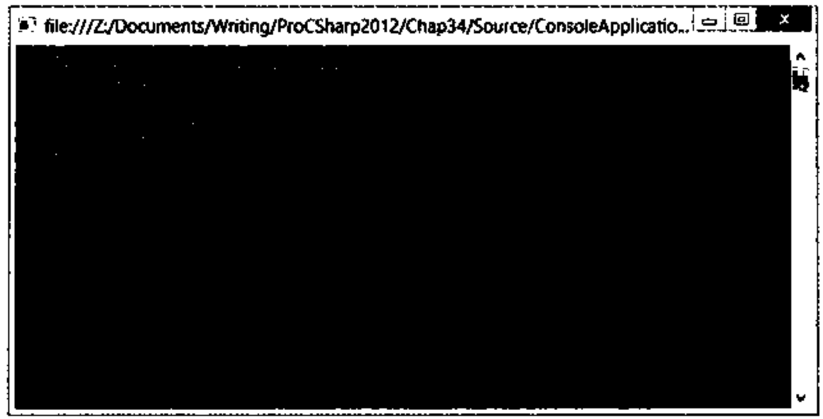


图 34-7

34.11.3　XNamespace 对象

XNamespace 对象表示 XML 名称空间，很容易应用于文档中的元素。例如，在前面的例子中，很容易给根元素应用一个名称空间：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XNamespace ns = "http://www.lipperweb.com/ns/1";

            XElement xe = new XElement(ns + "Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

在这个例子中，创建了一个 XNamespace 对象，具体方法是给它赋予 http://www.lipperweb.com/ns/1 的值。之后，就可以在根元素<company>中通过实例化 XElement 对象来使用它。

```
XElement xe = new XElement(ns + "Company", // .
```

这会生成如图 34-8 所示的结果。

file:///Z:/Documents/Writing/ProCSharp2012/Chap34/Source/ConsoleApplicatio...

图 34-8

除了仅处理根元素之外，还可以把名称空间应用于所有元素，如下例所示：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XNamespace ns1 = "http://www.lipperweb.com/ns/root";
            XNamespace ns2 = "http://www.lipperweb.com/ns/sub";

            XElement xe = new XElement(ns1 + "Company",
                new XElement(ns2 + "CompanyName", "Lipper"),
                new XElement(ns2 + "CompanyAddress",
                    new XElement(ns2 + "Address", "123 Main Street"),
                    new XElement(ns2 + "City", "St. Louis"),
                    new XElement(ns2 + "State", "MO"),
                    new XElement(ns2 + "Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

这会生成如图 34-9 所示的结果。

在这个例子中，子名称空间应用于指定的所有对象，但<Address>、<City>、<State>和<Country>元素除外，因为它们继承自其父对象<CompanyAddress>，而<CompanyAddress>有名称空间声明。

file:///Z:/Documents/Writing/ProCSharp2012/Chap34/Source/ConsoleApplicatio...

图 34-9

34.11.4　XComment 对象

XComment 对象可以轻松地把 XML 注释添加到 XML 文档中。下面的例子说明了如何把一条注释添加到文档的开头：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = new XDocument();

            XComment xc = new XComment("Here is a comment.");
            xdoc.Add(xc);

            XElement xe = new XElement("Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XComment("Here is another comment."),
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));
            xdoc.Add(xe);

            Console.WriteLine(xdoc.ToString());

            Console.ReadLine();
        }
    }
}
```

这个例子把包含两条 XML 注释的 XDocument 对象写到控制台中，其中一条注释写到文档的开头，另一条注释写到<CompanyAddress>元素内部，其结果如图 34-10 所示。

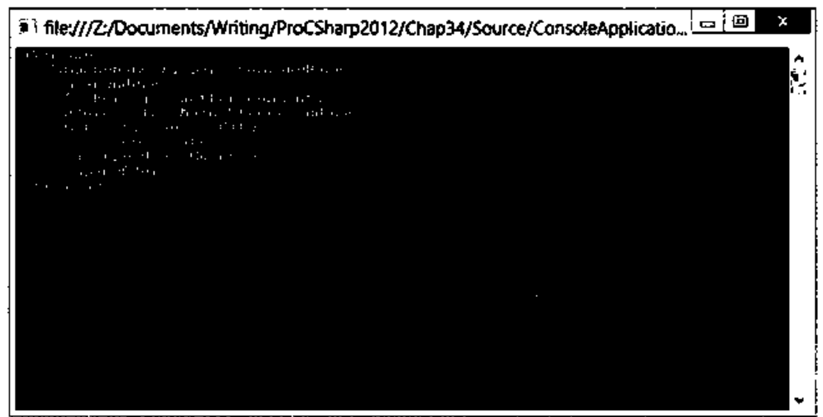


图 34-10

34.11.5 XAttribute 对象

除了元素之外，XML 的另一个要素是特性。通过 XAttribute 对象添加和使用特性。下面的例子说明了给根节点<Customers>添加一个特性的过程：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XElement xe = new XElement("Company",
                new XAttribute("MyAttribute", "MyAttributeValue"),
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

这里把 MyAttribute 特性及 MyAttributeValue 的值添加到 XML 文档的根元素中，结果如图 34-11 所示。

图 34-11

34.12　使用 LINQ 查询 XML 文档

现在可以把 XML 文档放在 XDocument 对象中，操作这个文档的各个部分。还可以使用 LINQ to XML 查询 XML 文档，操作其结果。

34.12.1　查询静态的 XML 文档

使用 LINQ to XML 查询静态的 XML 文档几乎不需要做任何工作。下面的例子就使用 hamlet.xml 文件和查询获得戏剧中的所有演员。每位演员都在 XML 文档中用<PERSONA>元素定义：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            var query = from people in xdoc.Descendants("PERSONA")
                        select people.Value;

            Console.WriteLine("{0} Players Found", query.Count());
            Console.WriteLine();

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

            Console.ReadLine();
        }
    }
}
```

在这个例子中，XDocument 对象加载了一个物理 XML 文件 hamlet.xml，对文档的内容执行一个 LINQ 查询：

```
var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;
```

people 对象表示在文档中找到的所有<PERSONA>元素。接着 select 语句获取这些元素的值。之后，使用 Console.WriteLine()方法输出通过 query.Count()方法找到的所有演员的总数。再在 foreach 循环中把每一项写到屏幕上。结果如下所示：

```
26 Players Found

CLAUDIUS, king of Denmark.
HAMLET, son to the late king, and nephew to the present king.
POLONIUS, lord chamberlain.
HORATIO, friend to Hamlet.
LAERTES, son to Polonius.
LUCIANUS, nephew to the king.
VOLTIMAND
CORNELIUS
ROSENCRANTZ
GUILDENSTERN
OSRIC
A Gentleman
A Priest.
MARCELLUS
BERNARDO
FRANCISCO, a soldier.
REYNALDO, servant to Polonius.
Players.
Two Clowns, grave-diggers.
FORTINBRAS, prince of Norway.
A Captain.
English Ambassadors.
GERTRUDE, queen of Denmark, and mother to Hamlet.
OPHELIA, daughter to Polonius.
Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other Attendants.
Ghost of Hamlet's Father.
```

34.12.2 查询动态的 XML 文档

目前，Internet 上有许多动态的 XML 文档。给指定的 URL 端点发送一个请求，就会找到博客种子、播客种子等许多提供 XML 文档的内容。这些种子可以在浏览器上查看，或者通过 RSS 聚合器查看，或用作纯粹的 XML。下面的示例说明了如何直接从代码中使用 RSS 种子：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
```

```
        static void Main()
        {
            XDocument xdoc =
               XDocument.Load(@"http://geekswithblogs.net/evjen/Rss.aspx");

            var query = from rssFeed in xdoc.Descendants("channel")
                        select new
                        {
                          Title = rssFeed.Element("title").Value,
                          Description = rssFeed.Element("description").Value,
                          Link = rssFeed.Element("link").Value,
                        };

            foreach (var item in query)
            {
              Console.WriteLine("TITLE: " + item.Title);
               Console.WriteLine("DESCRIPTION: " + item.Description);
               Console.WriteLine("LINK: " + item.Link);
            }

            Console.WriteLine();

            var queryPosts = from myPosts in xdoc.Descendants("item")
                             select new
                             {
                               Title = myPosts.Element("title").Value,
                               Published =
                                 DateTime.Parse(
                                   myPosts.Element("pubDate").Value),
                               Description =
                                 myPosts.Element("description").Value,
                               Url = myPosts.Element("link").Value,
                               Comments = myPosts.Element("comments").Value
                             };

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

            Console.ReadLine();
        }
    }
}
```

在这段代码中，XDocument 对象的 Load()方法指向一个 URL，从该 URL 中检索 XML 文档。第一个查询提取种子中<channel>元素的所有主要子元素，新建 Title、Description 和 Link 对象，以获取这些子元素的值。之后，运行一条 foreach 语句，迭代该查询找到的所有项，结果如下：

```
TITLE: Bill Evjen's Blog
DESCRIPTION: Code, Life and Community
LINK: http://geekswithblogs.net/evjen/Default.aspx
```

第二个查询遍历它找到的所有<item>元素和不同子元素(这些都是在博客中找到的博客项)。尽

管找到的许多项都出现在属性中，但在 foreach 循环中只使用了 Title 属性。这个查询的部分结果如下所示：

```
AJAX Control Toolkit Controls Grayed Out-HOW TO FIX
Welcome .NET 4.5!
Visual Studio
IIS 7.0 Rocks the House!
Word Issue-Couldn't Select Text
Microsoft Releases XML Schema Designer CTP1
Silverlight Book
Microsoft Tafiti as a beta
ReSharper on Visual Studio
Windows Vista Updates for Performance and Reliability Issues
First Review of Professional XML
Go to MIX07 for free!
Microsoft Surface and the Future of Home Computing?
Alas my friends-I'm *not* TechEd bound
```

34.13 XML 文档的更多查询技术

如果正在处理 XML 文档 hamlet.xml，就会注意到该文件相当大。在本章中，查询 XML 文档有两种方式，但下一节介绍 XML 文档的读写操作。

34.13.1 读取 XML 文档

使用 LINQ 查询语句查询 XML 文档非常简单，如下所示：

```
var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;
```

这个查询返回在文档中找到的所有演员。使用 XDocument 对象的 Element()方法，还可以获取 XML 文档中的特定值。例如，下面的 XML 片段说明了如何在 hamlet.xml 文档中表示标题：

```
<?xml version="1.0"?>

<PLAY>
  <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>

  <!-XML removed for clarity->

</PLAY>
```

可以看出，<TITLE>元素是<PLAY>元素的一个嵌套元素。在控制台应用程序中使用下面的代码可以轻松地获得标题：

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

Console.WriteLine(xdoc.Element("PLAY").Element("TITLE").Value);
```

这段代码把标题 The Tragedy of Hamlet, Prince of Denmark 输出到控制台屏幕上。在代码中，可以使用两个 Element()方法调用进入 XML 文档的层次结构，第一个 Element()方法调用<PLAY>元素，

然后第二个 Element()方法调用嵌套在<PLAY>元素中的<TITLE>元素。

仔细查看 hamlet.xml 文档，会发现它使用<PERSONA>元素定义了一个很大的演员列表：

```
<?xml version="1.0"?>

<PLAY>
   <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>

   <!--XML removed for clarity-->

   <PERSONAE>
     <TITLE>Dramatis Personae</TITLE>

     <PERSONA>CLAUDIUS, king of Denmark.</PERSONA>
     <PERSONA>HAMLET, son to the late king,
     and nephew to the present king.</PERSONA>
     <PERSONA>POLONIUS, lord chamberlain.</PERSONA>
     <PERSONA>HORATIO, friend to Hamlet.</PERSONA>
     <PERSONA>LAERTES, son to Polonius.</PERSONA>
     <PERSONA>LUCIANUS, nephew to the king.</PERSONA>

     <!--XML removed for clarity-->

   </PERSONAE>

</PLAY>
```

现在看看下面这个 C#查询：

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

Console.WriteLine(
  xdoc.Element("PLAY").Element("PERSONAE").Element("PERSONA").Value);
```

这段代码首先访问<PLAY>元素，再访问<PERSONA>元素，最后使用<PERSONA>元素。但是，其结果如下：

```
CLAUDIUS, king of Denmark
```

原因是尽管有一个<PERSONA>元素集合，但我们只处理使用 Element().Value 调用遇到的第一个<PERSONA>元素。

34.13.2 写入 XML 文档

除了读取 XML 文档之外，还可以同样轻松地写入该文档。例如，如果要改变《哈姆雷特》剧本的第一个演员的名称，就可以使用下面的代码：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
```

```
    {
        static void Main()
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            xdoc.Element("PLAY").Element("PERSONAE").
               Element("PERSONA").SetValue("Bill Evjen, king of Denmark");

            Console.WriteLine(xdoc.Element("PLAY").
               Element("PERSONAE").Element("PERSONA").Value);

            Console.ReadLine();
        }
    }
}
```

在这个例子中，使用 Element()对象的 SetValue()方法把<PERSONA>元素的第一个实例重写为 Bill Evjen，king of Denmark。调用 SetValue()方法并将该值应用于 XML 文档后，就使用与前面相同的方法检索该值。运行这段代码，会发现第一个<PERSONA>元素的值改变了。

修改文档的另一种方式(在这个例子中是给文档添加项)是把需要的元素创建为 XElement 对象，再把它们添加到文档中：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            XElement xe = new XElement("PERSONA",
               "Bill Evjen, king of Denmark");

            xdoc.Element("PLAY").Element("PERSONAE").Add(xe);

            var query = from people in xdoc.Descendants("PERSONA")
                        select people.Value;

            Console.WriteLine("{0} Players Found", query.Count());
            Console.WriteLine();

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

            Console.ReadLine();
        }
    }
}
```

在这个例子中，创建了一个 XElement 文档 xe。xe 的构造会提供如下 XML 结果：

```
<PERSONA>Bill Evjen, king of Denmark</PERSONA>
```

接着使用 XDocument 对象的 Element().Add()方法，可以添加所创建的元素：

```
xdoc.Element("PLAY").Element("PERSONAE").Add(xe);
```

现在查询所有演员时，会找到 27 个演员，而不是 26 个，新加的一个演员在列表的底部。除了 Add()方法之外，还可以使用 AddFirst()方法，顾名思义，它会把元素添加到列表的开头，而不是默认的末尾。

34.14 小结

本章探讨了.NET Framework 的 System.Xml 名称空间中的许多内容，其中包括如何使用基于 XMLReader 和 XmlWriter 的类快速读写 XML 文档，如何在.NET 中实现 DOM，如何使用 DOM 的强大功能。XML 和 ADO.NET 实际上有非常密切的关系。DataSet 和 XML 文档仅是相同底层体系结构的两个不同视图而已。另外，我们还介绍了 XPath 和 XSL 转换，以及添加到 VS 中的调试功能。最后，可以把对象序列化到 XML 中，还可以通过两个方法调用对其进行反序列化。

XML 是以后几年中应用程序开发的一个重要部分。.NET Framework 提供了操作 XML 的丰富而强大的工具集。

本章介绍了如何使用 LINQ to XML 和读写 XML 文件及 XML 源(无论是静态还是动态的)的一些选项。

使用 LINQ to XML 可以通过一系列强类型化的操作对 XML 文件及 XML 源执行 CRUD 操作。也可以联合使用 XmlReader、XmlWriter 和 LINQ to XML 功能编写代码。

本章还介绍了新的 LINQ to XML 帮助对象 XDocument、XElement、XNamespace、XAttribute 和 XComment。这些都是使 XML 操作比以前更方便的重要对象。

第Ⅴ部分

显　示

- 第 35 章　核心 WPF
- 第 36 章　用 WPF 编写业务应用程序
- 第 37 章　用 WPF 创建文档
- 第 38 章　Windows Store 应用程序：用户界面
- 第 39 章　Windows Store 应用程序：协定和设备
- 第 40 章　核心 ASP.NET
- 第 41 章　ASP.NET Web Forms
- 第 42 章　ASP.NET MVC

第35章

核 心 WPF

本章要点

- 用作基本绘图元素的形状和几何图形
- 利用转换功能实现缩放、旋转和倾斜
- 填充背景的画笔
- WPF 控件及其特性
- 用 WPF 面板定义布局
- 样式、模板和资源
- 触发器和 Visual State Manager
- 动画
- 3D

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Shapes Demo
- Geometry Demo
- Transformation Demo
- Brushes Demo
- Decorations Demo
- Layout Demo
- Styles and Resources
- Trigger Demo
- Template Demo
- Animation Demo
- Visual State Demo
- 3D Demo

35.1 理解WPF

Windows Presentation Foundation(WPF)是为智能客户端应用程序创建 UI 的一个库。本章介绍WPF的重要概念，讨论大量不同的控件及其类别，说明如何用面板安排控件，如何使用样式、资源和模板定制外观，如何用触发器和动画添加动态操作，并介绍WPF的3D操作。

WPF的一个主要优点是设计人员和开发人员的工作很容易分开。设计人员的工作成果可以直接供开发人员使用。为此，必须理解XAML。XAML语法可参见第29章。

本章的第一个主题是概述WPF使用的类层次结构和类别，包括理解XAML的原则的额外信息。WPF由几个包含了上千个类的程序集组成。为了帮助用户在这些类中导航，查找需要的类，本节介绍WPF中的类层次结构和名称空间。

35.1.1 名称空间

Windows 窗体类和WPF类很容易混淆。Windows 窗体类位于System.Windows.Forms名称空间，而WPF类位于System.Windows名称空间及其子名称空间中，但不位于System.Windows.Forms名称空间。Windows 窗体的Button类的全称是System.Windows. Forms.Button，而用于WPF的Button类的全称是System.Windows.Controls. Button。

WPF的名称空间及其功能如表35-1所述。

表35-1

名称空间	说明
System.Windows	这是WPF的核心名称空间，其中包含WPF的核心类，如Application类、用于依赖对象的类DependencyObject和DependencyProperty，所有WPF元素的基类FrameworkElement
System.Windows.Annotations	这个名称空间中的类用于用户在应用程序数据上创建的注释和备注，它们与文档分开存储。System.Windows.Annotations.Storage名称空间包含存储注释的类
System.Windows.Automation	System.Windows.Automation名称空间用于自动完成WPF应用程序。它有几个子名称空间。System.Windows.Automation.Peers名称空间提供用于自动化的WPF元素，如ButtonAutomationPeer和CheckBoxAutomationPeer。如果创建自定义自动化提供程序，就需要System.Windows.Automation. Provider名称空间
System.Windows.Baml2006	这个名称空间包含Baml2006Reader类，它用于读取二进制标记语言，生成XAML
System.Windows.Controls	这个名称空间包含所有WPF控件，如Button、Border、Canvas、ComboBox、Expander、Slider、ToolTip、TreeView等。System.Windows.Controls.Primitives名称空间包含在复杂控件中使用的类，如Popup、ScrollBar、StatusBar、TabPanel等
System.Windows.Converters	这个名称空间包含用于数据转换的类。但它没有包含所有转换类。核心转换类在System.Windows名称空间中定义

(续表)

名称空间	说明
System.Windows.Data	这个名称空间由 WPF 数据绑定使用。其中的一个重要类是 Binding 类，它用于定义 WPF 目标元素和 CLR 源之间的绑定。数据绑定参见第 36 章
System.Windows.Documents	在处理文档时，使用这个名称空间中的许多类很有帮助。内容元素 FixedDocument 和 FlowDocument 可以包含这个名称空间中的其他元素。System.Windows.Documents.Serialization 名称空间中的类可以将文档写入磁盘。这个名称空间中的类参见第 37 章
System.Windows.Ink	Windows Tablet PC 和 Ultra Mobile PC 使用得越来越多。在这些 PC 上，喷墨可以用于用户输入。System.Windows.Ink 名称空间包含处理喷墨输入的类
System.Windows.Input	这个名称空间包含的几个类用于命令处理、键盘输入、使用触笔等
System.Windows.Interop	这个名称空间中的类用于集成 WPF 与 Windows API 和 Windows 窗体中的本地 Windows 句柄
System.Windows.Markup	用于 XAML 标记代码的帮助类位于这个名称空间
System.Windows.Media	要使用图像、音频和视频内容，可以使用这个名称空间中的类
System.Windows.Navigation	这个名称空间包含在窗口之间导航的类
System.Windows.Resources	这个名称空间包含资源的支持类
System.Windows.Shapes	UI 的核心类位于这个名称空间，如 Line、Ellipse、Rectangle 等
System.Windows.Threading	把 WPF 元素绑定到单个线程上。这个名称空间中的类可以处理多个线程，如 Dispatcher 类就属于这个名称空间
System.Windows.Xps	XPS (XML Paper Specification，XML 纸张规范)是一个文档规范，Microsoft Word 也支持该规范。System.Windows.Xps、System.Windows.Xps.Packaging 和 System.Windows.Xps.Serialization 名称空间包含用于创建和流式传输 XPS 文档的类

35.1.2 类层次结构

WPF 包含上千个类，有很深的层次结构。为了帮助理解类之间的关系，图 35-1 列出了一些 WPF 类。表 35-2 描述了一些类及其功能。

表 35-2

类	说明
DispatcherObject	DispatcherObject 是一个抽象基类，用于绑定到一个线程上的类。WPF 控件要求仅从创建线程中调用方法和属性。派生自 DispatcherObject 的类有一个关联的 Dispatcher 对象，它可以用于切换线程
Application	在 WPF 应用程序中，会创建 Application 类的一个实例。这个类实现单一模式，用于访问应用程序的窗口、资源和属性

(续表)

类	说 明
DependencyObject	DependencyObject 是所有支持依赖属性的类的基类。依赖属性参见第 29 章
Visual	所有可见元素的基类是 Visual。这个类包含单击测试和转换等特性
UIElement	所有需要基本显示功能的 WPF 元素的抽象基类是 UIElement。这个类提供鼠标移动、拖放、按键单击的隧道和冒泡事件；提供可以由派生类重写的虚呈现方法；以及布局方法。WPF 不再使用 Window 句柄，这个类就可以用作 Window 句柄
FrameworkElement	FrameworkElement 派生自基类 UIElement，并实现由基类定义的方法的默认行为
Shape	Shape 是所有图形元素的基类，如 Line、Ellipse、Polygon、Rectangle
Control	Control 派生自 FrameworkElement，是所有用户交互元素的基类
ContentControl	ContentControl 是所有有单个内容的控件的基类，如 Label、Button。内容控件的默认样式是受限制的，但可以使用模板改变其外观
ItemsControl	ItemsControl 是所有包含一个项集合的控件的基类，如 ListBox、ComboBox
Panel	Panel 类派生自 FrameworkElement，是所有面板的抽象基类，这个类的 Children 属性用于面板中的所有 UI 元素，并定义了排列子控件的方法。派生自 Panel 的类为子控件的布局方式定义了不同的类，如 WrapPanel、StackPanel、Canvas、Grid

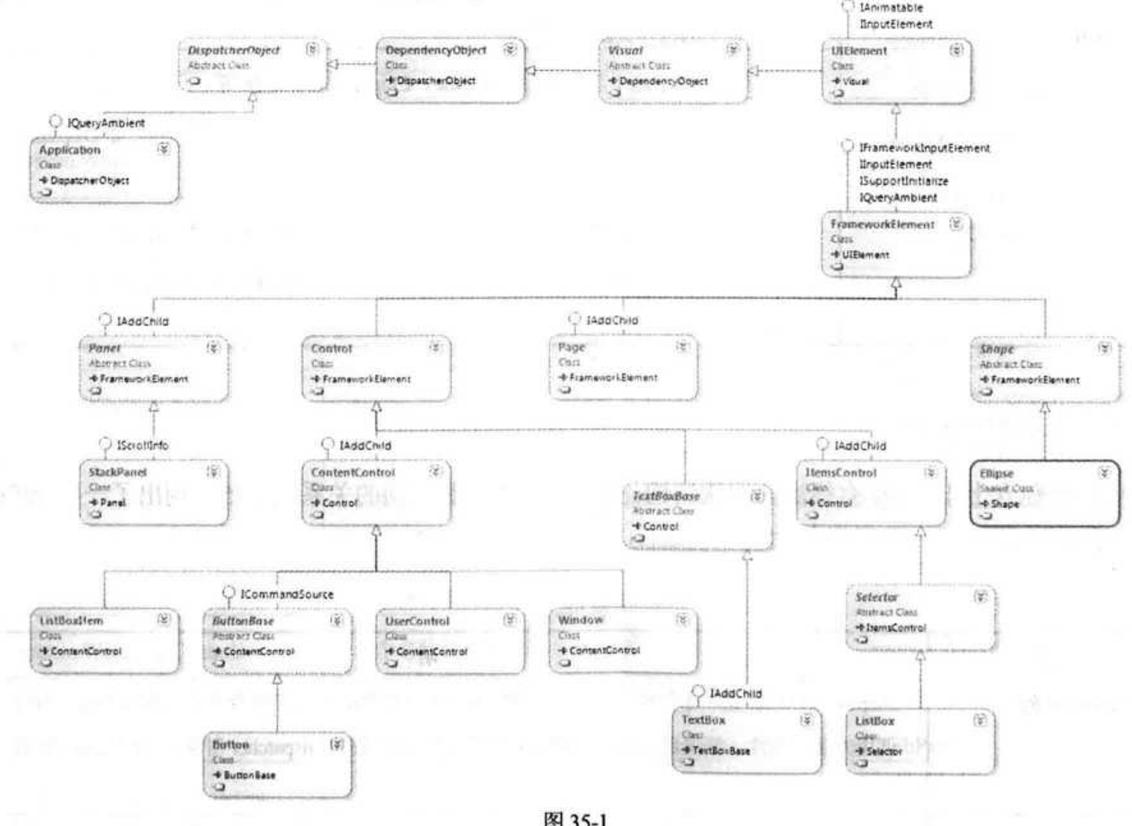


图 35-1

可以看出，WPF 类有非常深的层次结构。本章和后面几章将介绍提供核心功能的类，但不可能涵盖 WPF 的所有特性。

35.2　形状

形状是 WPF 的核心元素。利用形状，可以绘制矩形、线条、椭圆、路径、多边形和折线等二维图形，这些图形用派生自抽象类 Shape 的类表示。图形在 System.Windows.Shapes 名称空间中定义。

下面的 XAML 示例(代码文件 ShapesDemo/MainWindow.xaml)绘制了一个黄色笑脸，它用一个椭圆表示笑脸，两个椭圆表示眼睛，两个椭圆表示眼睛中的瞳孔，一条路径表示嘴型：

```
<Window x:Class="ShapesDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="300" Width="300">
  <Canvas>
    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
        Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
    <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
          <GradientStop Offset="0.1" Color="DarkGreen" />
          <GradientStop Offset="0.7" Color="Transparent" />
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
       StrokeThickness="3" Fill="White" />
    <Ellipse Canvas.Left="40" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
    <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
       StrokeThickness="3" Fill="White" />
    <Ellipse Canvas.Left="75" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
    <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
       Data="M 40,74 Q 57,95 80,74 " />
  </Canvas>

</Window>
```

图 35-2 显示了这些 XAML 代码的结果。

无论是按钮还是线条、矩形等图形，所有这些 WPF 元素都可以通过编程来访问。把 Path 元素的 Name 或 x:Name 属性设置为 mouth，就可以用变量名 mouth 以编程方式访问这个元素：

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
   Data="M 40,74 Q 57,95 80,74 " />
```

在 Path 元素的代码隐藏属性 Data 中(代码文件 ShapesDemo/MainWindow.xaml.cs)，把 mouth 设置为一个新的图形。为了设置路径，Path 类支持 PathGeometry 和路径标记语法。字母 M 定义了路径的起点，字母 Q 指定了二次贝塞尔曲线的一个控制点和一个终点。运行应用程序，会看到如图 35-3 所示的窗口。

```
public MainWindow()
{
  InitializeComponent();
  mouth.Data = Geometry.Parse("M 40,92 Q 57,75 80,92");
}
```

图 35-2

图 35-3

表 35-3 描述了 System.Windows.Shapes 名称空间中可用的图形。

表 35-3

Shape 类	说 明
Line	可以在坐标(X1,Y1)到(X2,Y2)之间绘制一条线
Rectangle	使用 Rectangle 类，通过指定 Width 和 Height 可以绘制一个矩形
Ellipse	使用 Ellipse 类，可以绘制一个椭圆
Path	使用 Path 类可以绘制一系列直线和曲线。Data 属性是 Geometry 类型。还可以使用派生自基类 Geometry 的类绘制图形，或使用路径标记语法来定义图形
Polygon	使用 Polygon 类可以绘制由线段连接而成的封闭图形。多边形由一系列赋予 Points 属性的 Point 对象定义
Polyline	类似于 Polygon 类，使用 Polyline 也可以绘制连接起来的线段。与多边形的区别是，折线不一定是封闭图形

35.3 几何图形

其中一种形状Path使用Geometry来绘图。Geometry元素也可用于其他地方，如用于DrawingBrush。

在某些方面，Geometry 元素非常类似于形状。与 Line、Ellipse 和 Rectangle 形状一样，也有绘制这些形状的 Geometry 元素：LineGeometry、EllipseGeometry 和 RectangleGeometry。形状与几何图形有显著的区别。Shape 是一个 FrameworkElement，可以用于把 UIElement 用作其子元素的任意类。FrameworkElement 派生自 UIElement。形状会参与系统的布局，并呈现自身。而 Geometry 类不呈现自身，特性和系统开销也比 Shpae 类少。Geometry 类派生自 Freezable 基类，可以在多个线程中共享。

Path 类使用 Geometry 来绘图。几何图形可以用 Path 的 Data 属性设置。可以设置的简单的几何图形元素有绘制椭圆的 EllipseGeometry、绘制线条的 LineGeometry 和绘制矩形的 RectangleGeometry。如下面的示例所示，使用 CombineGeometry 可以合并多个几何图形。

CombineGeometry 有 Geometry1 和 Geometry2 属性，使用 GeometryCombineMode 可以合并它们，构成 Union、Intersect、Xor 和 Exclude。Union 会合并两个几何图形，Intersect 只取两个几何图形都覆盖的区域，Xor 与 Intersect 相反，显示一个几何图形覆盖的区域，但不显示两个几何图形都覆盖的区域。Exclude 显示第一个几何图形减去第二个几何图形的区域。

下面的示例(代码文件 GeometryDemo/MainWindow.xaml)合并了一个 EllipseGeometry 和一个 RectangleGeometry，生成并集，如图 35-4 所示。

```
<Path Canvas.Top="0" Canvas.Left="250" Fill="Blue" Stroke="Black" >
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="30,60 105 50" />
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

也可以使用段来创建几何图形。几何图形类 PathGeometry 使用段来绘图。下面的代码段使用 BezierSegment 和 LineSegment 元素来绘制一个红色的图形和一个绿色的图形，如图 35-5 所示。第一个 BezierSegment 在图形的起点(70,40)、终点(150,63)、控制点(90,37)和(130,46)之间绘制了一条贝塞尔曲线。下面的 LineSegment 使用贝塞尔曲线的终点和(120,110)绘制了一条线段：

图 35-4

```
<Path Canvas.Left="0" Canvas.Top="0" Fill="Red" Stroke="Blue"
      StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="70,40" IsClosed="True">
            <PathFigure.Segments>
              <BezierSegment Point1="90,37" Point2="130,46" Point3="150,63" />
              <LineSegment Point="120,110" />
              <BezierSegment Point1="100,95" Point2="70,90" Point3="45,91" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>

<Path Canvas.Left="0" Canvas.Top="0" Fill="Green" Stroke="Blue"
      StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="160,70">
            <PathFigure.Segments>
              <BezierSegment Point1="175,85" Point2="200,99"
                             Point3="215,100" />
              <LineSegment Point="195,148" />
              <BezierSegment Point1="174,150" Point2="142,140"
                             Point3="129,115" />
              <LineSegment Point="160,70" />
```

```
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>
```

除了 BezierSegment 和 LineSegment 元素之外，还可以使用 ArcSegment 元素在两点之间绘制椭圆弧。使用 PolyLineSegment 可以绘制一组线段，PolyBezierSegment 由多条贝塞尔曲线组成，QuadraticBezierSegment 创建一条二次贝塞尔曲线，PolyQuadraticBezierSegment 由多条二次贝塞尔曲线组成。

图 35-5

图 35-6

使用 StreamGeometry 可以进行高效的绘图。通过编程，可以利用 StreamGeometryContext 类的成员创建线段、贝塞尔曲线和圆弧，以定义图形。通过 XAML 可以使用路径标记语法。路径标记语法可以与 Path 类的 Data 属性一起使用，来定义 StreamGeometry。特殊字符定义点的连接方式。在下面的示例中，M 标记起点，L 是到指定点的线条命令，Z 是闭合图形的闭合命令。图 35-6 显示了这个绘图操作的结果。路径标记语法允许使用更多的命令，如水平线(H)、垂直线(V)、三次贝塞尔曲线(C)、二次贝塞尔曲线(Q)、光滑的三次贝塞尔曲线(S)、光滑的二次贝塞尔曲线(T)，以及椭圆弧(A)：

```
<Path Canvas.Left="0" Canvas.Top="200" Fill="Yellow" Stroke="Blue"
      StrokeThickness="2.5"
      Data="M 120,5 L 128,80 L 220,50 L 160,130 L 190,220 L 100,150
            L 80,230 L 60,140 L0,110 L70,80 Z" StrokeLineJoin="Round">
</Path>
```

35.4　变换

因为 WPF 基于矢量，所以可以重置每个元素的大小。基于矢量的图形现在可以缩放、旋转和倾斜。不需要手工计算位置，就可以进行单击测试(如移动鼠标和鼠标单击)。

给 Canvas 元素的 LayoutTransform 属性添加 ScaleTransform 元素，如下所示(代码文件 TransformationDemo/MainWindow.xaml)，把整个画布的内容在 X 和 Y 方向上放大 1.5 倍。

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
</Canvas.LayoutTransform>
```

旋转与缩放的执行方式相同。使用 RotateTransform 元素，可以定义旋转的角度：

```
<Canvas.LayoutTransform>
  <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

对于倾斜，可以使用 SkewTransform 元素。此时可以指定 X 和 Y 方向的倾斜角度：

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

为了同时执行旋转和倾斜操作，可以定义一个 TransformGroup，它同时包含 RotateTransform 和 SkewTransform。也可以定义一个 MatrixTransform，其中 Matrix 元素指定了用于拉伸的 M11 和 M22 属性，以及用于倾斜的 M12 和 M21 属性。

```
<Canvas.LayoutTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix M11="0.8" M22="1.6" M12="1.3" M21="0.4" />
    </MatrixTransform.Matrix>
  </MatrixTransform>
</Canvas.LayoutTransform>
```

图 35-7 显示了这些变换的结果。这些图放在一个 StackPanel 中。从左到右，第 1 个图重置了大小，第 2 个图旋转了，第 3 个图倾斜了，第 4 个图使用一个矩阵进行变换。为了更容易看出它们的区别，可以把 Canvas 元素的 Background 属性设置为不同的颜色。

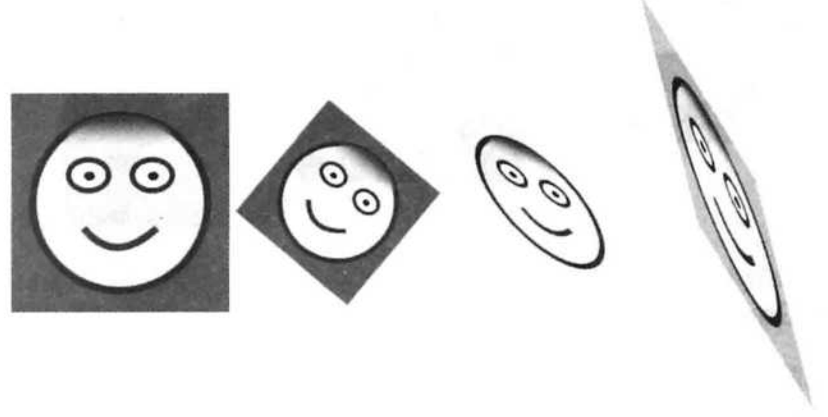


图 35-7

除了 LayerTransform 之外，还有一个 RenderTransform。LayerTransform 在布局阶段之前、RenderTransform 发生之后发生。

35.5 画笔

本节介绍如何使用 WPF 提供的画笔绘制背景和前景。本节将参考图 35-8，它显示了在 Button 元素的一条 Path 和 Background 属性上使用各种画笔的效果。

35.5.1 SolidColorBrush

图 35-8 中的第一个按钮使用了 SolidColorBrush，顾名思义，这支画笔使用纯色。全部区域用同一种颜色绘制。

把 Background 属性设置为定义纯色的字符串，就可以定义纯色。使用 BrushValueSerializer 把该

字符串转换为一个 SolidColorBrush 元素。

```
<Button Height="30" Background="PapayaWhip">Solid Color</Button>
```

当然，设置 Background 子元素，把 SolidColorBrush 元素添加为它的内容(代码文件 BrushesDemo/MainWindow.xaml)，也可以得到同样的效果。应用程序中的第一个按钮把 PapayaWhip 用作纯背景色：

```
<Button Content="Solid Color" Margin="10">
  <Button.Background>
    <SolidColorBrush Color="PapayaWhip" />
  </Button.Background>
</Button>
```

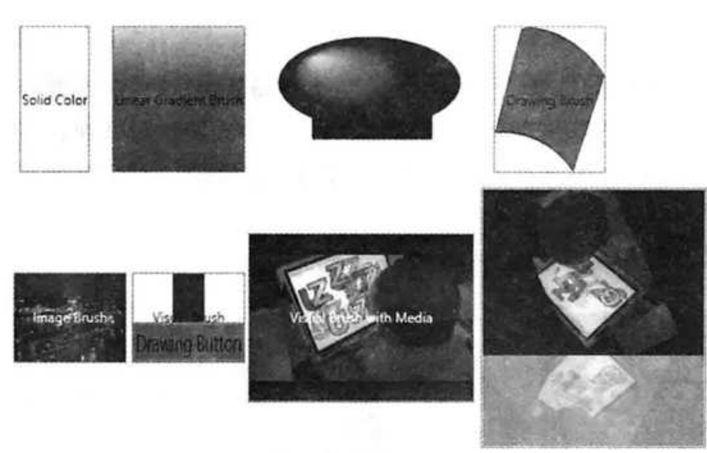


图 35-8

35.5.2 LinearGradientBrush

对于平滑的颜色变化，可以使用 LinearGradientBrush，如第二个按钮所示。这个画笔定义了 StartPoint 和 EndPoint 属性。使用这些属性可以为线性渐变指定 2D 坐标。默认的渐变方向是从(0,0)到(1,1)的对角线。定义其他值可以给渐变指定不同的方向。例如，StartPoint 指定为(0,0)，EndPoint 指定为(0,1)，就得到了一个垂直渐变。StartPoint 不变，EndPoint 值指定为(1,0)，就得到了一个水平渐变。

通过该画笔的内容，可以用 GradientStop 元素定义指定偏移位置的颜色值。在各个偏移位置之间，颜色是平滑过渡的(代码文件 BrushesDemo/MainWindow.xaml)。

```
<Button Content="Linear Gradient Brush" Margin="10">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Offset="0" Color="LightGreen" />
      <GradientStop Offset="0.4" Color="Green" />
      <GradientStop Offset="1" Color="DarkGreen" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

35.5.3 RadialGradientBrush

使用 RadialGradientBrush 可以以放射方式产生平滑的颜色渐变。在图 35-8 中，第 3 个元素 Path

使用了 RadialGradientBrush。该画笔定义了从 GradientOrigin 点开始的颜色(代码文件 BrushesDemo/MainWindow.xaml)。

```
<Canvas Width="200" Height="150">
  <Path Canvas.Top="0" Canvas.Left="20" Stroke="Black" >
    <Path.Fill>
      <RadialGradientBrush GradientOrigin="0.2,0.2">
        <GradientStop Offset="0" Color="LightBlue" />
        <GradientStop Offset="0.6" Color="Blue" />
        <GradientStop Offset="1.0" Color="DarkBlue" />
      </RadialGradientBrush>
    </Path.Fill>
    <Path.Data>
      <CombinedGeometry GeometryCombineMode="Union">
        <CombinedGeometry.Geometry1>
          <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
        </CombinedGeometry.Geometry1>
        <CombinedGeometry.Geometry2>
          <RectangleGeometry Rect="30,60 105 50" />
        </CombinedGeometry.Geometry2>
      </CombinedGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

35.5.4 DrawingBrush

DrawingBrush 可以定义用画笔绘制的图形。用画笔绘制的图形在 GeometryDrawing 元素中定义。Geometry 属性中的 GeometryGroup 元素包含本章前面讨论的 Geometry 元素(代码文件 BrushesDemo/MainWindow.xaml)。

```
<Button Content="Drawing Brush" Margin="10" Padding="10">
  <Button.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <GeometryDrawing Brush="Red">
          <GeometryDrawing.Pen>
            <Pen>
              <Pen.Brush>
                <SolidColorBrush>Blue</SolidColorBrush>
              </Pen.Brush>
            </Pen>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <PathGeometry>
              <PathGeometry.Figures>
                <PathFigure StartPoint="70,40">
                  <PathFigure.Segments>
                    <BezierSegment Point1="90,37" Point2="130,46"
                                   Point3="150,63" />
                    <LineSegment Point="120,110" />
```

```
                        <BezierSegment Point1="100,95" Point2="70,90"
                                       Point3="45,91" />
                        <LineSegment Point="70,40" />
                      </PathFigure.Segments>
                    </PathFigure>
                  </PathGeometry.Figures>
                </PathGeometry>
              </GeometryDrawing.Geometry>
            </GeometryDrawing>
          </DrawingBrush.Drawing>
        </DrawingBrush>
      </Button.Background>
    </Button>
```

35.5.5 ImageBrush

要把图像加载到画笔中，可以使用ImageBrush元素。通过这个元素，显示ImageSource属性定义的图像。图像可以从文件系统中访问，或者在程序集的资源中访问。在本例中(代码文件BrushesDemo/MainWindow.xaml)，图像添加为程序集的一个资源，并通过程序集和资源名来引用：

```
<Button Content="Image Brush" Width="100" Height="80" Margin="5"
        Foreground="White">
  <Button.Background>
    <ImageBrush ImageSource="/BrushesDemo;component/Budapest.jpg" />
  </Button.Background>
</Button>
```

35.5.6 VisualBrush

VisualBrush可以在画笔中使用其他WPF元素。下面(代码文件BrushesDemo/MainWindow.xaml)给Visual属性添加一个WPF元素。图35-8中的第6个元素包含一个矩形和一个按钮。

```
<Button Content="Visual Brush" Width="100" Height="80">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
        <StackPanel Background="White">
          <Rectangle Width="25" Height="25" Fill="Blue" />
          <Button Content="Drawing Button" Background="Red" />
        </StackPanel>
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
</Button>
```

可以给VisualBrush添加任意UIElement。一个例子是可以使用MediaElement播放视频：

```
<Button Content="Visual Brush with Media" Width="200" Height="150"
     Foreground="White">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
        <MediaElement Source="./Stephanie.wmv" />
```

```
        </VisualBrush.Visual>
      </VisualBrush>
    </Button.Background>
  </Button>
```

在 VisualBrush 中，还可以创建反射等有趣的效果。这里显示的按钮包含一个 StackPanel，它包含一个播放视频的 MediaElement 和一个 Border。Border 边框包含一个用 VisualBrush 填充的矩形。这支画笔定义了一个不透明值和一个变换。把 Visual 属性绑定到 Border 元素上。变换通过设置 VisualBrush 的 RelativeTransform 属性来完成。这个变换使用了相对坐标。把 ScaleY 设置为-1，完成 Y 方向上的反射。TranslateTransform 在 Y 方向上移动变换，从而使反射效果位于原始对象的下面。图 35-8 中的第 8 个元素显示了其效果。

这里使用的数据绑定和 Binding 元素详见第 36 章。

```
<Button Width="200" Height="200" Foreground="White">
  <StackPanel>
    <MediaElement x:Name="reflected" Source="./Stephanie.wmv" />
      <Border Height="100">
        <Rectangle>
          <Rectangle.Fill>
            <VisualBrush Opacity="0.35" Stretch="None"
                Visual="{Binding ElementName=reflected}">
              <VisualBrush.RelativeTransform>
                <TransformGroup>
                  <ScaleTransform ScaleX="1" ScaleY="-1" />
                  <TranslateTransform Y="1" />
                </TransformGroup>
              </VisualBrush.RelativeTransform>
            </VisualBrush>
          </Rectangle.Fill>
        </Rectangle>
      </Border>
    </StackPanel>
  </Button>
```

35.6 控件

可以对 WPF 使用上百个控件。下面把控件分为几组，在各小节中分别介绍。

35.6.1 简单控件

简单控件是没有 Content 属性的控件。例如，Button 类可以包含任意形状或任意元素，这对于简单控件没有问题。表 35-4 列出了简单控件及其功能。

表 35-4

简单控件	说　明
PasswordBox	PasswordBox 控件用于输入密码。这个控件有用于输入密码的专用属性，例如，PasswordChar 属性定义了在用户输入密码时显示的字符，Password 属性可以访问输入的密码。PasswordChanged 事件在修改密码时立即调用
ScrollBar	ScrollBar 控件包含一个 Thumb，用户可以从 Thumb 中选择一个值。例如，如果文档在屏幕中放不下，就可以使用滚动条。一些控件包含滚动条，如果内容过多，就显示滚动条
ProgressBar	使用 ProgressBar 控件，可以指示时间较长的操作的进度
Slider	使用 Slider 控件，用户可以移动 Thumb，选择一个范围的值。ScrollBar、ProgressBar 和 Slider 派生自同一个基类 RangeBase
Textbox	Textbox 控件用于显示简单的无格式文本
RichTextbox	RichTextbox 控件通过 FlowDocument 类支持带格式的文本。RichTextBox 和 TextBox 派生自同一个基类 TextBoxBase
Calendar	Calendar 控件可以显示年份、月份或 10 年。用户可以选择一个日期或日期范围
DatePicker	DatePicker 控件会打开 Calendar 屏幕，供用户选择日期

尽管简单控件没有 Content 属性，但通过定义模板，完全可以定制这些控件的外观。模板详见本章后面的内容。

35.6.2 内容控件

ContentControl 有 Content 属性，利用 Content 属性，可以给控件添加任意内容。因为 Button 类派生自基类 ContentControl，所以可以在这个控件中添加任意内容。在上面的例子中，在 Button 类中有一个 Canvas 控件。表 35-5 列出了内容控件。

表 35-5

ContentControl 控件	说　明
ButtonRepeat ButtonToggle ButtonCheckBox RadioButton	Button、RepeatButton、ToggleButton 和 GridViewColumnHeader 类派生自同一个基类 ButtonBase。所有这些按钮都响应 Click 事件。RepeatButton 类会重复引发 Click 事件，直到释放按钮为止。ToggleButton 是 CheckBox 和 RadioButton 的基类。这些按钮有开关状态。CheckBox 可以由用户选择和取消选择，RadioButton 可以由用户选择。清除 RadioButton 的选择必须通过编程方式实现
Label	Label 类表示控件的文本标签。这个类也支持访问键，如菜单命令
Frame	Frame 控件支持导航。使用 Navigate()方法可以导航到一个页面内容上。如果该内容是一个网页，就使用 WebBrowser 控件来显示
ListBoxItem	ListBoxItem 是 ListBox 控件中的一项
StatusBarItem	StatusBarItem 是 StatusBar 控件中的一项

(续表)

ContentControl 控件	说　明
ScrollViewer	ScrollViewer 控件是一个包含滚动条的内容控件，可以把任意内容放入这个控件中，滚动条会在需要时显示
ToolTip	ToolTip 创建一个弹出窗口，以显示控件的附加信息
UserControl	将 UserControl 类用作基类，可以为创建自定义控件提供一种简单方式。但是，基类 UserControl 不支持模板
Window	Window 类可以创建窗口和对话框。使用这个类，会获得一个带有最小化/最大化/关闭按钮和系统菜单的框架。在显示对话框时，可以使用 ShowDialog()方法，Show()方法会打开一个窗口
NavigationWindow	类 NavigationWindow 派生自 Window 类，支持内容导航

只有一个Frame控件包含在下面XAML代码的Window中(代码文件FrameDemo/MainWindow.xaml)。因为Source属性设置为http://www.cninnovation.com，所以Frame控件导航到这个网站上，如图 35-9 所示。

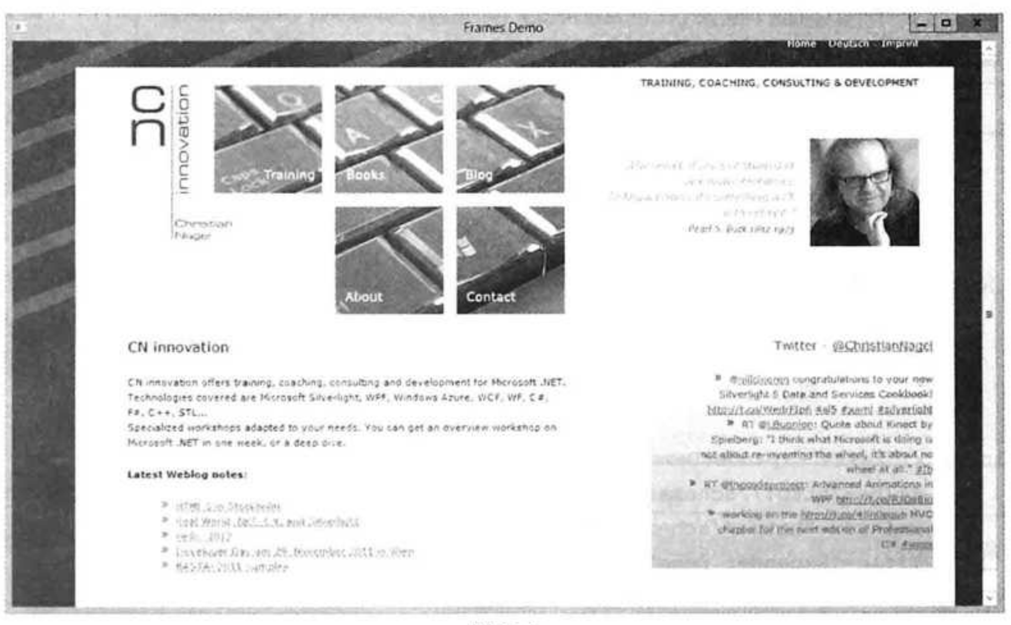


图 35-9

```
<Window x:Class="FrameDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Frames Demo" Height="240" Width="500">
  <Frame Source="http://www.cninnovation.com" />
</Window>
```

35.6.3 带标题的内容控件

带标题的内容控件派生自 HeaderContentControl 基类。HeaderContentControl 类又派生自基类 ContentControl。HeaderContentControl 类的 Header 属性定义了标题的内容，HeaderTemplate 属性可以对标题进行完全的定制。派生自基类 HeaderContentControl 的控件如表 35-6 所示。

表 35-6

HeaderContentControl	说　明
Expander	使用 Expander 控件，可以创建一个带对话框的“高级”模式，它在默认情况下不显示所有的信息，只有用户展开它，才会显示更多的信息。在未展开模式下，只显示标题信息；在展开模式下，显示内容
GroupBox	GroupBox 控件提供了边框和标题来组合控件
TabItem	TabItem 控件是 TabControl 类中的项。TabItem 的 Header 属性定义了标题的内容，这些内容用 TabControl 的标签显示

Expander 控件的简单用法如下面的例子所示。把 Expander 控件的 Header 属性设置为 Click for more。这个文本用于显示扩展。这个控件的内容只有在控件展开时才显示。图 35-10 中的应用程序包含折叠的 Expander 控件，图 35-11 中的同一个应用程序包含展开的 Expander 控件。代码如下(代码文件 ExpanderDemo/MainWindow.xaml)：

图 35-10

图 35-11

```
<Window x:Class="ExpanderDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Expander Demo" Height="240" Width="500">
  <StackPanel>
    <TextBlock>Short information</TextBlock>
    <Expander Header="Additional Information">
      <Border Height="200" Width="200" Background="Yellow">
        <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center">
          More information here!
        </TextBlock>
      </Border>
    </Expander>
  </StackPanel>

</Window>
```

在展开 Expander 控件时，如果要修改该控件的标题文本，就可以创建一个触发器。触发器详见本章后面的内容。

35.6.4 项控件

ItemsControl 类包含一个可以用 Items 属性访问的数据项列表。派生自 ItemsControl 的类如表 35-7 所示。

表 35-7

ItemsControl	说　明
Menu ContextMenu	Menu 类和 ContextMenu 类派生自抽象基类 MenuBase。把 MenuItem 元素放在数据项列表和相关联的命令中，就可以给用户提供菜单
StatusBar	StatusBar 控件通常显示在应用程序的底部，为用户提供状态信息。可以把 StatusBarItem 元素放在 StatusBar 列表中
TreeView	要分层显示数据项，可以使用 TreeView 控件
ListBox ComboBox TabControl	ListBox、ComboBox 和 TabControl 都有相同的抽象基类 Selector。这个基类可以从列表中选择数据项。ListBox 显示列表中的数据项，ComboBox 有一个附带的 Button 控件，只有单击该按钮，才会显示数据项。在 TabControl 中，内容可以排列为表格形式
DataGrid	DataGrid 控件是显示数据的可定制网格，这个控件详见下一章

35.6.5 带标题的项控件

HeaderItemsControl 是不仅包含数据项而且包含标题的控件的基类。HeaderItemsControl 类派生自 ItemsControl。

派生自 HeaderItemsControl 的类如表 35-8 所示。

表 35-8

HeaderedItemsControl	说　明
MenuItem	菜单类 Menu 和 ContextMenu 包含 MenuItem 类型的数据项。菜单项可以连接到命令上，因为 MenuItem 类实现了 ICommandSource 接口
TreeViewItem	TreeViewItem 类可以包含 TreeViewItem 类型的数据项
ToolBar	ToolBar 控件是一组控件(通常是 Button 和 Separator 元素)的容器。可以将 ToolBar 放在 ToolBarTray 中，它会重新排列 ToolBar 控件

35.6.6 修饰

给单个元素添加修饰可以使用 Decorator 类完成。Decorator 是一个基类，派生自它的类有 Border、Viewbox 和 BulletDecorator。主题元素如 ButtonChrome 和 ListBoxChrome 也是修饰器。

下面的例子(代码文件 DecorationsDemo/MainWindow.xaml) 说明了 Border、Viewbox 和 BulletDecorator 类，如图 35-12 所示。Border 类给子元素四周添加边框，以修饰子元素。可以给子元素定义画笔和边框的宽度、背景、圆角半径和填充图案：

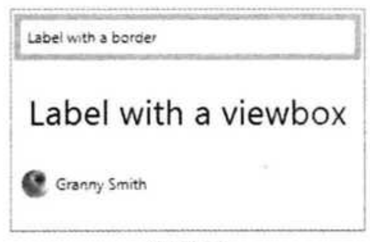


图 35-12

```
<Border BorderBrush="Violet" BorderThickness="5.5">
  <Label>Label with a border</Label>
</Border>
```

Viewbox 将其子元素拉伸并缩放到可用的空间中。StretchDirection 和 Stretch 属性专用于 Viewbox 的功能，它们允许设置子元素是否双向拉伸，以及是否保持宽高比：

```
<Viewbox StretchDirection="Both" Stretch="Uniform">
  <Label>Label with a viewbox</Label>
</Viewbox>
```

BulletDecorator 类用一个项目符号修饰其子元素。子元素可以是任意元素(在本例中是一个文本块)。同样，项目符号也可以是任意元素，本例使用一个 Image 元素，也可以使用任意 UIElement：

```
<BulletDecorator>
  <BulletDecorator.Bullet>
    <Image Width="25" Height="25" Margin="5" HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Source="/DecorationsDemo;component/images/apple1.jpg" />
  </BulletDecorator.Bullet>
  <BulletDecorator.Child>
    <TextBlock VerticalAlignment="Center" Padding="8">Granny Smith</TextBlock>
  </BulletDecorator.Child>
</BulletDecorator>
```

35.7 布局

为了定义应用程序的布局，可以使用派生自 Panel 基类的类。布局容器要完成两个主要任务：测量和排列。在测量时，容器要求其子控件有合适的大小。因为控件的整体大小不一定合适，所以容器需要确定和排列其子控件的大小和位置。这里讨论几个布局容器。

35.7.1 StackPanel

Window 可以只包含一个元素，作为其内容。如果要在其中包含多个元素，就可以将 StackPanel 用作 Window 的一个子元素，并在 StackPanel 的内容中添加元素。StackPanel 是一个简单的容器控件，只能逐个地显示元素。StackPanel 的方向可以是水平或垂直。ToolBarPanel 类派生自 StackPanel(代码文件 LayoutDemo/StackPanelWindow.xaml)。

```
<Window x:Class="LayoutDemo.StackPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StackPanelWindow" Height="300" Width="300">
  <StackPanel Orientation="Vertical">
    <Label>Label</Label>
    <TextBox>TextBox</TextBox>
    <CheckBox>CheckBox</CheckBox>
    <CheckBox>CheckBox</CheckBox>
    <ListBox>
      <ListBoxItem>ListBoxItem One</ListBoxItem>
      <ListBoxItem>ListBoxItem Two</ListBoxItem>
    </ListBox>
    <Button>Button</Button>
  </StackPanel>
</Window>
```

在图 35-13 中，可以看到 StackPanel 垂直显示的子控件。

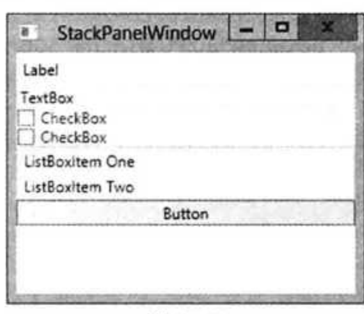


图 35-13

35.7.2 WrapPanel

WrapPanel 将子元素自左向右逐个排列，若一个水平行中放不下，就排在下一行。面板的方向可以是水平或垂直(代码文件 LayoutDemo/WrapPanelWindow.xaml)。

```
<Window x:Class="LayoutDemo.WrapPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WrapPanelWindow" Height="300" Width="300">
  <WrapPanel>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
  </WrapPanel>
</Window>
```

图 35-14 显示了面板的排列结果。如果重新设置应用程序的大小，按钮就会重新排列，以便填满一行。

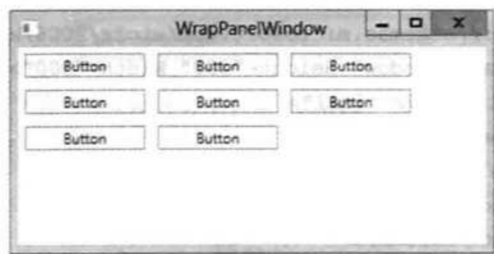


图 35-14

35.7.3 Canvas

Canvas是一个允许显式指定控件位置的面板。它定义了相关的Left、Right、Top和Bottom属性，这些属性可以由子元素在面板中定位时使用(代码文件LayoutDemo/CanvasWindow.xaml)。

```
<Window x:Class="LayoutDemo.CanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CanvasWindow" Height="300" Width="300">
  <Canvas Background="LightBlue">
    <Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
    <TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />
    <Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />
  </Canvas>
</Window>
```

图35-15显示了Canvas面板的结果，其中定位了子元素Label、TextBox和Button。

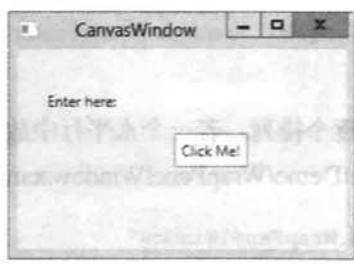


图 35-15

35.7.4 DockPanel

DockPanel非常类似于Windows窗体的停靠功能。DockPanel可以指定排列子控件的区域。DockPanel定义了相关的Dock属性，可以在控件的子控件中将它设置为Left、Right、Top和Bottom。图35-16显示了排列在DockPanel中的带边框的文本块。为了便于区别，为不同的区域指定了不同的颜色(代码文件LayoutDemo/DockPanelWindow.xaml)：

```
<Window x:Class="LayoutDemo.DockPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DockPanelWindow" Height="300" Width="300">
  <DockPanel>
    <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
      <TextBlock>Menu</TextBlock>
```

```
    </Border>
    <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
      <TextBlock>Ribbon</TextBlock>
    </Border>
    <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
      <TextBlock>Status</TextBlock>
    </Border>
    <Border Height="80" Background="Azure" DockPanel.Dock="Left">
      <TextBlock>Left Side</TextBlock>
    </Border>
    <Border Background="HotPink">
      <TextBlock>Remaining Part</TextBlock>
    </Border>
  </DockPanel>
</Window>
```

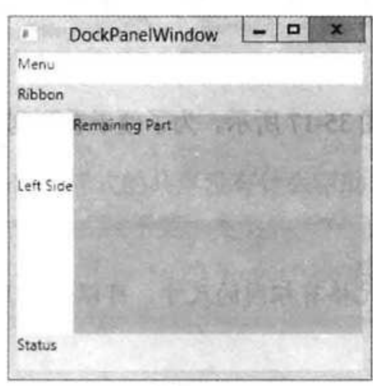


图 35-16

35.7.5 Grid

使用 Grid，可以在行和列中排列控件。对于每一列，可以指定一个 ColumnDefinition；对于每一行，可以指定一个 RowDefinition。下面的示例代码显示两列和三行(代码文件 LayoutDemo/GridWindow.xaml)。在每一列和每一行中，都可以指定宽度或高度。ColumnDefinition 有一个 Width 依赖属性，RowDefinition 有一个 Height 依赖属性。可以以像素、厘米、英寸或点为单位定义高度和宽度，或者把它们设置为 Auto，根据内容来确定其大小。Grid 还允许根据具体情况指定大小，即根据可用的空间以及与其他行和列的相对位置，计算行和列的空间。在为列提供可用空间时，可以将 Width 属性设置为"*"。要使某一列的空间是另一列的两倍，应指定"2*"。下面的示例代码定义了两列和三行，但没有定义列定义和行定义的其他设置，默认使用根据具体情况指定大小的设置。

这个 Grid 包含几个 Label 和 TextBox 控件。因为这些控件的父控件是 Grid，所以可以设置相关的 Column、ColumnSpan、Row 和 RowSpan 属性。

```
<Window x:Class="LayoutDemo.GridWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        Title="GridWindow" Height="300" Width="300">
  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
          VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title"
    />
    <Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center"
          Content="Firstname:" Margin="10" />
    <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />
    <Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center"
          Content="Lastname:" Margin="10" />
    <TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />
  </Grid>
</Window>
```

在 Grid 中排列控件的结果如图 35-17 所示。为了便于看到列和行，把 ShowGridLines 属性设置为 true。

要使 Grid 的每个单元格有相同的尺寸，可以使用 UniformGrid 类。

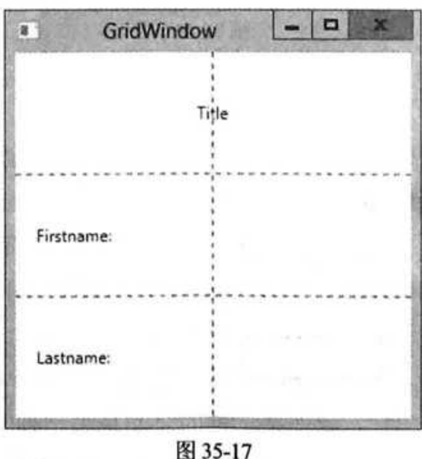


图 35-17

35.8 样式和资源

设置 Button 元素的 FontSize 和 Background 属性，就可以定义 WPF 元素的外观，如 Button 元素所示(代码文件 StylesAndResources/MainWindow.xaml)：

```
<Button Width="150" FontSize="12" Background="AliceBlue" Content="Click Me!" />
```

除了定义每个元素的外观之外，还可以定义用资源存储的样式。为了完全定制控件的外观，可以使用模板，再把它们存储到资源中。

35.8.1 样式

控件的Style属性可以赋予附带Setter的Style元素。Setter元素定义Property和Value属性，并给指定的属性设置一个值。下例设置Background、FontSize和FontWeight属性(代码文件StylesAndResources/MainWindow.xaml)。把Style设置为TargetType Button，以便直接访问Button的属性。如果没有设置样式的TargetType，就可以通过Button.Background、Button.FontSize访问属性。如果需要设置不同元素类型的属性，这就很重要。

```
<Button Width="150" Content="Click Me!">
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
</Button>
```

直接通过Button元素设置Style对样式的共享没有什么帮助。样式可以放在资源中。在资源中，可以把样式赋予指定的元素，把一个样式赋予某一类型的所有元素，或者为该样式使用一个键。要把样式赋予某一类型的所有元素，可使用Style的TargetType属性，指定x:Type标记扩展{x:Type Button}，从而将样式赋予一个按钮。要定义需要引用的样式，必须设置x:Key：

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    <Setter Property="Background" Value="LemonChiffon" />
    <Setter Property="FontSize" Value="18" />
  </Style>
  <Style x:Key="ButtonStyle">
    <Setter Property="Button.Background" Value="Red" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.FontSize" Value="18" />
  </Style>
</Window.Resources>
```

在下面的XAML代码中，第一个按钮没有用元素属性定义样式，而是使用为Button类型定义的样式。对于下一个按钮，把Style属性用StaticResource标记扩展设置为{StaticResource ButtonStyle}，而ButtonStyle指定了前面定义的样式资源的键值，所以该按钮的背景为红色，前景是白色。

```
<Button Width="200" Content="Uses named style"
        Style="{StaticResource ButtonStyle}" Margin="3" />
```

除了把按钮的Background设置为单个值之外，还可以将Background属性设置为定义了渐变色的LinearGradientBrush，如下所示：

```
<Style x:Key="FancyButtonStyle">
  <Setter Property="Button.FontSize" Value="22" />
```

```
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.Background">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0.0" Color="LightCyan" />
          <GradientStop Offset="0.14" Color="Cyan" />
          <GradientStop Offset="0.7" Color="DarkCyan" />
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
```

本例中下一个按钮的样式采用青色的线性渐变效果：

```
<Button Width="200" Content="Fancy button style"
    Style="{StaticResource FancyButtonStyle}" Margin="3" />
```

样式提供了一种继承方式。一个样式可以基于另一个样式。下面的 AnotherButtonStyle 样式基于 FancyButtonStyle 样式。它使用该样式定义的所有设置，且通过 BasedOn 属性引用，但 Foreground 属性除外，它设置为 LinearGradientBrush：

```
<Style x:Key="AnotherButtonStyle" BasedOn="{StaticResource FancyButtonStyle}"
    TargetType="Button">
  <Setter Property="Foreground">
    <Setter.Value>
      <LinearGradientBrush>
        <GradientStop Offset="0.2" Color="White" />
        <GradientStop Offset="0.5" Color="LightYellow" />
        <GradientStop Offset="0.9" Color="Orange" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

最后一个按钮应用了 AnotherButtonStyle：

```
<Button Width="200" Content="Style inheritance"
    Style="{StaticResource AnotherButtonStyle}" Margin="3" />
```

图 35-18 显示了所有这些按钮的效果。

图 35-18

35.8.2 资源

从样式示例可以看出，样式通常存储在资源中。可以在资源中定义任意可冻结的元素。例如，前面为按钮的背景样式创建了画笔，它本身就可以定义为一个资源，这样就可以在需要画笔的地方

使用它。

下面的示例(代码文件 StylesAndResources/MainWindow.xaml)在 StackPanel 资源中定义一个 LinearGradientBrush，它的键名是 MyGradientBrush。Button1 使用 StaticResource 标记扩展将 Background 属性赋予 MyGradientBrush 资源。图 35-19 显示了这段 XAML 代码的结果。

```
<StackPanel x:Name="myContainer">
  <StackPanel.Resources>
    <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0,0"
        EndPoint="0.3,1">
      <GradientStop Offset="0.0" Color="LightCyan" />
      <GradientStop Offset="0.14" Color="Cyan" />
      <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>
  </StackPanel.Resources>
  <Button Width="200" Height="50" Foreground="White" Margin="5"
      Background="{StaticResource MyGradientBrush}" Content="Click Me!" />
</StackPanel>
```

图 35-19

这里，资源用 StackPanel 定义。在上面的例子中，资源用 Window 元素定义。基类 FrameworkElement 定义 ResourceDictionary 类型的 Resources 属性。这就是资源可以用派生自 FrameworkElement 的所有类(任意 WPF 元素)来定义的原因。

资源按层次结构来搜索。如果用 Window 元素定义资源，它就会应用于 Window 的所有子元素。如果 Window 包含一个 Grid，该 Grid 包含一个 StackPanel，且资源是用 StackPanel 定义的，该资源就会应用于 StackPanel 中的所有控件。如果 StackPanel 包含一个按钮，但只用该按钮定义资源，这个样式就只对该按钮有效。

对于层次结构，需要注意是否为样式使用了没有 Key 的 TargetType。如果用 Canvas 元素定义一个资源，并把样式的 TargetType 设置为应用于 TextBox 元素，该样式就会应用于 Canvas 中的所有 TextBox 元素。如果 Canvas 中有一个 ListBox，该样式甚至会应用于 ListBox 包含的 TextBox 元素。

如果需要将同一个样式应用于多个窗口，就可以用应用程序定义样式。在一个 Visual Studio WPF 项目中，创建 App.xaml 文件，以定义应用程序的全局资源。应用程序样式对其中的每个窗口都有效；每个元素都可以访问用应用程序定义的资源。如果通过父窗口找不到资源，就可以通过 Application 继续搜索资源(代码文件 StylesAndResources/App.xaml)。

```
<Application x:Class="StylesAndResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <Application.Resources>

  </Application.Resources>
</Application>
```

35.8.3 系统资源

还有一些系统范围内的颜色和字体资源，它们可用于所有应用程序。这些资源用 SystemColors、SystemFonts 和 SystemParameters 类定义。

- 使用 SystemColors 类可以获得边框、控件、桌面和窗口的颜色设置，例如，ActiveBorderColor、ControlBrush、DesktopColor、WindowColor 和 WindowBrush 等。
- SystemFonts 类返回菜单、状态栏、消息框等的字体设置，例如，CaptionFont、DialogFont、MenuFont、MessageBoxFont 和 StatusFont 等。
- SystemParameters 类设置菜单按钮、光标、图标、边框、标题、时间信息、键盘设置的大小。例如，BorderWidth、CaptionHeight、CaptionWidth、MenuButtonWidth、MenuPopupAnimation、MenuShowDelay、SmallIconHeight 和 SmallIconWidth 等。

35.8.4 从代码中访问资源

要从代码隐藏中访问资源，因为基类 FrameworkElement 实现 FindResource()方法，所以可以通过每个 WPF 对象调用 FindResource()方法。为此，button1 没有指定背景，但将 Click 事件赋予 button1_Click()方法(代码文件 StylesAndResources/ResourceDemo.xaml)。

```
<Button Name="button1" Width="220" Height="50" Margin="5"
   Click="button1_Click" Content="Apply Resource Programmatically" />
```

通过 button1_Click()方法的实现方式，给单击的按钮调用 FindResource()方法。然后按照层次结构搜索 MyGradientBrush 资源，将该画笔应用于控件的 Background 属性。前面在 StackPanel 的资源中创建了资源 MyGradientBrush(代码文件 StylesAndResources/ResourceDemo.xaml.cs)：

```
public void button1_Click(object sender, RoutedEventArgs e)
{
  Control ctrl = sender as Control;
  ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

如果 FindResource()方法没有找到资源键，就会抛出一个异常。如果不知道资源是否可用，就可以使用 TryFindResource()方法来替代。如果找不到资源，TryFindResource()方法就返回 null。

35.8.5 动态资源

通过 StaticResource 标记扩展，在加载期间搜索资源。如果在运行程序的过程中改变了资源，就应使用 DynamicResource 标记扩展。

下面的例子(代码文件 StylesAndResources/ResourceDemo.xaml)使用与前面定义的相同资源。前面的示例使用 StaticResource，而这个按钮通过 DynamicResource 标记扩展使用 DynamicResource。这个按钮的事件处理程序以编程方式改变资源。把处理程序方法 button2_Click 赋予 Click 事件处理程序。

```
<Button Name="button2" Width="200" Height="50" Foreground="White" Margin="5"
    Background="{DynamicResource MyGradientBrush}" Content="Change Resource"
    Click="button2_Click" />
```

button2_Click()方法的实现代码清除了 StackPanel 的资源，并用相同的名称 MyGradientBrush 添加了一个新资源。这个新资源非常类似于在 XAML 代码中定义的资源，它只定义了不同的颜色(代码文件 StylesAndResources/ResourceDemo.xaml.cs)。

```
private void button2_Click(object sender, RoutedEventArgs e)
{
  myContainer.Resources.Clear();
  var brush = new LinearGradientBrush
  {
    StartPoint = new Point(0, 0),
    EndPoint = new Point(0, 1)
  };

  brush.GradientStops = new GradientStopCollection()
  {
    new GradientStop(Colors.White, 0.0),
    new GradientStop(Colors.Yellow, 0.14),
    new GradientStop(Colors.YellowGreen, 0.7)
  };
  myContainer.Resources.Add("MyGradientBrush", brush);
}
```

运行应用程序时，单击 Change Resource 按钮，可以动态地更改资源。使用通过动态资源定义的按钮会获得动态创建的资源，而用静态资源定义的按钮看起来与以前一样。

35.8.6 资源字典

如果相同的资源可用于不同的应用程序，把资源放在一个资源字典中就比较有效。使用资源字典，可以在多个应用程序之间共享文件，也可以把资源字典放在一个程序集中，供应用程序共享。

要共享程序集中的资源字典，应创建一个库。可以把资源字典文件(这里是 Dictionary1.xaml)添加到程序集中。这个文件的构建动作必须设置为 Resource，从而把它作为资源添加到程序集中。

Dictionary1.xaml 定义了两个资源：一个是包含 CyanGradientBrush 键的 LinearGradientBrush；另一个是用于按钮的样式，它可以通过 PinkButtonStyle 键来引用(代码文件 ResourcesLib/Dictionary1.xaml)：

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <LinearGradientBrush x:Key="CyanGradientBrush" StartPoint="0,0"
      EndPoint="0.3,1">
    <GradientStop Offset="0.0" Color="LightCyan" />
    <GradientStop Offset="0.14" Color="Cyan" />
    <GradientStop Offset="0.7" Color="DarkCyan" />
  </LinearGradientBrush>

  <Style x:Key="PinkButtonStyle" TargetType="Button">
    <Setter Property="FontSize" Value="22" />
    <Setter Property="Foreground" Value="White" />
```

```
      <Setter Property="Background">
        <Setter.Value>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0.0" Color="Pink" />
            <GradientStop Offset="0.3" Color="DeepPink" />
            <GradientStop Offset="0.9" Color="DarkOrchid" />
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
  </ResourceDictionary>
```

对于目标项目，需要引用这个库，并把资源字典添加到这个字典中。通过 ResourceDictionary 的 MergedDictionaries 属性，可以使用添加进来的多个资源字典文件。可以把一个资源字典列表添加到合并的字典中。使用 ResourceDictionary 的 Source 属性，可以引用字典。在引用时，使用包 URI 语法。包 URI 语法可以指定为绝对的，其中 URI 以 pack://开头，也可以指定为相对的，如本例所示。使用相对语法，包含字典的引用程序集 ResourceLib 跟在"/"的后面，其后是";component"。Component 表示，该字典包含为程序集中的一个资源。之后添加字典文件的名称 Dictionary1.xaml。如果把字典添加到子文件夹中，则必须声明子文件夹名(代码文件 StylesAndResources/App.xaml)。

```
<Application x:Class="StylesAndResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/ResourceLib;component/Dictionary1.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

现在可以像本地资源那样使用引用程序集中的资源了(代码文件 StylesAndResources/ResourceDemo.xaml)：

```
<Button Width="300" Height="50" Style="{StaticResource PinkButtonStyle}"
   Content="Referenced Resource" />
```

35.9　触发器

使用触发器，可以动态地更改控件的外观，因为一些事件或属性值改变了。例如，用户把鼠标移动到按钮上，按钮就会改变其外观。通常，这必须在 C#代码中实现。使用 WPF，也可以用 XAML 实现，而这只会影响 UI。

XAML 有几个触发器。属性触发器在属性值改变时激活。多触发器基于多个属性值。事件触发器在事件发生时激活。数据触发器在绑定的数据改变时激活。本节讨论属性触发器、多触发器和数据触发器。事件触发器在后面与动画一起论述。

35.9.1 属性触发器

Style 类有一个 Triggers 属性，通过它可以指定属性触发器。下面的示例(代码文件 TriggerDemo/PropertyTriggerWindow.xaml)将一个 Button 元素放在一个 Grid 面板中。利用 Window 资源定义 Button 元素的默认样式。这个样式指定，将 Background 属性设置为 LightBlue，将 FontSize 属性设置为 17。这是应用程序启动时 Button 元素的样式。使用触发器可以改变控件的样式。触发器在 Style.Triggers 元素中用 Trigger 元素定义。将一个触发器赋予 IsMouseOver 属性，另一个触发器赋予 IsPressed 属性。这两个属性通过应用了样式的 Button 类定义。如果 IsMouseOver 属性的值是 true，就会激活触发器，将 Foreground 属性设置为 Red，将 FontSize 属性设置为 22。如果按下该按钮，IsPressed 属性就是 true，激活第二个触发器，并将 TextBox 的 Foreground 属性设置为 Yellow。

如果把 IsPressed 属性设置为 true，IsMouseOver 属性就也是 true。按下该按钮也需要把鼠标放在按钮上。按下该按钮会激活 IsMouseOver 属性触发器，并改变属性。这里触发器的激活顺序很重要。如果 IsPressed 属性触发器在 IsMouseOver 属性触发器之前激活，IsMouseOver 属性触发器就会覆盖 IsPressed 属性触发器设置的值。

```
<Window x:Class="TriggerDemo.PropertyTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PropertyTriggerWindow" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="Button">
      <Setter Property="Background" Value="LightBlue" />
      <Setter Property="FontSize" Value="17" />
      <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
          <Setter Property="Foreground" Value="Red" />
          <Setter Property="FontSize" Value="22" />
        </Trigger>
        <Trigger Property="IsPressed" Value="True">
          <Setter Property="Foreground" Value="Yellow" />
          <Setter Property="FontSize" Value="22" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
   <Button Width="200" Height="30" Content="Click me!" />
  </Grid>
</Window>
```

当激活触发器的原因不再有效时，就不必将属性值重置为原始值。例如，不必定义 IsMouseOver=true 和 IsMouseOver=false 的触发器。只要激活触发器的原因不再有效，触发器操作进行的修改就会自动重置为原始值。

Click me!

图 35-20

图 35-20 显示了触发器示例应用程序，其中，鼠标指向按钮时，按钮的前景和字体大小就会不同于其原始值。

使用属性触发器，很容易改变控件的外观、字体、颜色、不透明度等。在鼠标滑过控件时，键盘设置焦点时——都不需要编写任何代码。

Trigger 类定义了表 35-9 中的属性，以指定触发器操作。

表 35-9

Trigger 属性	说 明
Property Value	使用属性触发器，Property 和 Value 属性用于指定触发器的激活时间，例如，Property = "IsMouseOver"，Value = "True"
Setters	一旦激活触发器，就可以使用 Setters 定义一个 Setter 元素集合，来改变属性值。Setter 类定义 Property、TargetName 和 Value 属性，以修改对象属性
EnterActions ExitActions	除了定义 Setters 之外，还可以定义 EnterActions 和 ExitActions。使用这两个属性，可以定义一个 TriggerAction 元素集合。EnterActions 在启动触发器时激活(此时通过属性触发器应用 Property/Value 组合)。ExitActions 在触发器结束之前激活(此时不再应用 Property/Value 组合)。用这些操作指定的触发器操作派生自基类 TriggerAction，如 SoundPlayerAction 和 BeginStoryboard。使用 SoundPlayerAction 基类可以开始播放声音，BeginStoryboard 基类用于动画，详见本章后面的内容

35.9.2 多触发器

属性的值变化时，就会激活属性触发器，如果因为两个或多个属性有特定的值，而需要设置触发器，就可以使用 MultiTrigger。

MultiTrigger 有一个 Conditions 属性，可以在其中设置属性的有效值。它还有一个 Setters 属性，可以在其中指定需要设置的属性。在下面的示例中(代码文件 TriggerDemo/MultiTriggerWindow.xaml)，给 TextBox 元素定义了一个样式，如果 IsEnabled 属性是 True，Text 属性的值是 Test，就应用触发器。如果应用这两个触发器，就把 TextBox 的 Foreground 属性设置为 Red：

```
<Window x:Class="TriggerDemo.MultiTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultiTriggerWindow" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="TextBox">
      <Style.Triggers>
        <MultiTrigger>
          <MultiTrigger.Conditions>
            <Condition Property="IsEnabled" Value="True" />
            <Condition Property="Text" Value="Test" />
          </MultiTrigger.Conditions>
          <MultiTrigger.Setters>
            <Setter Property="Foreground" Value="Red" />
          </MultiTrigger.Setters>
        </MultiTrigger>
      </Style.Triggers>
```

```
    </Style>
  </Window.Resources>
  <Grid>
    <TextBox />
  </Grid>
</Window>
```

35.9.3 数据触发器

如果绑定到控件上的数据满足指定的条件，就激活数据触发器。下面的例子(代码文件TriggerDemo/Book.cs)使用Book类，它根据图书的出版社显示不同的内容。

Book类定义Title和Publisher属性，还重载ToString()方法：

```
public class Book
{
  public string Title { get; set; }
  public string Publisher { get; set; }

  public override string ToString()
  {
    return Title;
  }
}
```

在XAML代码中，给ListBoxItem元素指定了一个样式。该样式包含DataTrigger元素，它绑定到用于列表项的类的Publisher属性上。如果Publisher属性的值是Wrox Press，Background就设置为Red。对于Dummies和Sybex出版社，把Background分别设置为Yellow和LightBlue(代码文件TriggerDemo/DataTriggerWindow.xaml)：

```
<Window x:Class="TriggerDemo.DataTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Data Trigger Window" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="ListBoxItem">
      <Style.Triggers>
        <DataTrigger Binding="{Binding Publisher}" Value="Wrox Press">
          <Setter Property="Background" Value="Red" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Publisher}" Value="Dummies">
          <Setter Property="Background" Value="Yellow" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Publisher}" Value="Wiley">
          <Setter Property="Background" Value="DarkGray" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
    <ListBox x:Name="list1" />
  </Grid>
</Window>
```

在代码隐藏文件中(代码文件 TriggerDemo/DataTriggerWindow.xaml.cs)，把列表 list1 初始化为包含几个 Book 对象：

```
public DataTriggerWindow()
{
  InitializeComponent();
  list1.Items.Add(new Book
  {
    Title = "Professional C# 4.0 and .NET 4",
    Publisher = "Wrox Press"
  });
  list1.Items.Add(new Book
  {
    Title = "C# 2010 for Dummies",
    Publisher = "For Dummies"
  });
  list1.Items.Add(new Book
  {
    Title = "HTML and CSS: Design and Build Websites",
    Publisher = "Wiley"
  });
}
```

运行应用程序，ListBoxItem 元素就会根据 Publisher 的值进行格式化，如图 35-21 所示。

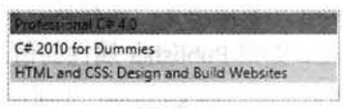


图 35-21

使用 DataTrigger 时，必须给 MultiDataTrigger 设置多个属性(类似于 Trigger 和 MultiTrigger)。

35.10 模板

本章前面介绍过，Button 控件可以包含任何内容，如简单的文本，还可以给按钮添加 Canvas 元素，Canvas 元素可以包含形状，也可以给按钮添加 Grid 或视频。然而，按钮还可以完成更多的操作。

控件的外观及其功能在 WPF 中是完全分离的。虽然按钮有默认的外观，但可以用模板完全定制其外观。

如表 35-10 所示，WPF 提供了几个模板类型，它们派生自基类 FrameworkTemplate。

表 35-10

模板类型	说 明
ControlTemplate	使用 ControlTemplate 可以指定控件的可视化结构，重新设计其外观
ItemsPanelTemplate	对于 ItemsControl，可以赋予一个 ItemsPanelTemplate，以指定其项的布局。每个 ItemsControl 都有一个默认的 ItemsPanelTemplate。MenuItem 使用 WrapPanel，StatusBar 使用 DockPanel，ListBox 使用 VirtualizingStackPanel

(续表)

模 板 类 型	说　　明
DataTemplate	DataTemplates 非常适用于对象的图形表示。给列表框指定样式，默认情况下，列表框中的项根据 ToString()方法的输出来显示。应用 DataTemplate，可以重写其操作，定义项的自定义表示
HierarchicalDataTemplate	HierarchicalDataTemplate 用于排列对象的树型结构。这个控件支持 HeaderedItemsControls，如 TreeViewItem 和 MenuItem

35.10.1　控件模板

本章前面介绍了如何给控件的属性定义样式。如果设置控件的简单属性得不到需要的外观，就可以修改 Template 属性。使用 Template 属性可以定制控件的整体外观。下面的例子说明了定制按钮的过程，后面逐步地说明了列表框的定制，以便显示出改变的中间结果。

Button 类型的定制在一个单独的资源字典文件 Styles.xaml 中进行。这里定义了键名为 RoundedGelButton 的样式。GelButton 样式设置 Background、Height、Foreground、Margin 和 Template 属性。Template 属性是这个样式中最有趣的部分，它指定一个仅包含一行一列的网格。

在这个单元格中，有一个名为 GelBackground 的椭圆。这个椭圆给笔触设置了一个线性渐变画笔。包围矩形的笔触非常细，因为把 StrokeThickness 设置为 0.5。

因为第二个椭圆 GelShine 比较小，其尺寸由 Margin 属性定义，所以在第一个椭圆内部是可见的。因为其笔触是透明的，所以该椭圆没有边框。这个椭圆使用一个线性渐变填充画笔，从部分透明的浅色变为完全透明，这使椭圆具有“亦真亦幻”的效果(代码文件 TemplateDemo/Styles.xaml)。

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Style x:Key="RoundedGelButton" TargetType="Button">
    <Setter Property="Width" Value="100" />
    <Setter Property="Height" Value="100" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
          <Grid>
            <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
              <Ellipse.Stroke>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#ff7e7e7e" />
                  <GradientStop Offset="1" Color="Black" />
                </LinearGradientBrush>
              </Ellipse.Stroke>
            </Ellipse>
            <Ellipse Margin="15,5,15,50">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#aaffffff" />
                  <GradientStop Offset="1" Color="Transparent" />
                </LinearGradientBrush>
              </Ellipse.Fill>
            </Ellipse>
          </Grid>
```

```
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

从 app.xaml 文件中，引用资源字典，如下所示(代码文件 TemplateDemo/App.xaml)：

```
<Application x:Class="TemplateDemo.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary Source="Styles.xaml" />
  </Application.Resources
</Application>
```

现在可以把 Button 控件关联到样式上。按钮的新外观如图 35-22 所示（代码文件 TemplateDemo/StyledButtonWindow.xaml）。

```
<Button Style="{StaticResource RoundedGelButton}" Content="Click Me!" />
```

按钮现在的外观完全不同，但按钮的内容未在图 35-22 中显示出来。必须扩展前面创建的模板，以把按钮的内容显示在新外观上。为此需要添加一个 ContentPresenter。ContentPresenter 是控件内容的占位符，并定义了放置这些内容的位置。这里(代码文件 TemplateDemo/StyledButtonWindow.xaml)把内容放在网格的第一行上，即 Ellipse 元素所在的位置。ContentPresenter 的 Content 属性定义了内容的外观。把内容设置为 TemplateBinding 标记表达式。TemplateBinding 绑定父模板，这里是 Button 元素。{TemplateBinding Content}指定，Button 控件的 Content 属性值应作为内容放在占位符内。图 35-23 显示了带内容的按钮：

图 35-22

```
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type Button}">
      <Grid>
        <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
          <Ellipse.Stroke>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
              <GradientStop Offset="0" Color="#ff7e7e7e" />
              <GradientStop Offset="1" Color="Black" />
            </LinearGradientBrush>
          </Ellipse.Stroke>
        </Ellipse>
        <Ellipse Margin="15,5,15,50">
          <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
              <GradientStop Offset="0" Color="#aaffffff" />
              <GradientStop Offset="1" Color="Transparent" />
            </LinearGradientBrush>
          </Ellipse.Fill>
        </Ellipse>
        <ContentPresenter Name="GelButtonContent"
```

```
                              VerticalAlignment="Center"
                              HorizontalAlignment="Center"
                              Content="{TemplateBinding Content}" />
        </Grid>
      </ControlTemplate>
    </Setter.Value>
```

现在这样一个样式化的按钮在屏幕上看起来很漂亮。但仍有一个问题：如果用鼠标单击该按钮，或使鼠标滑过该按钮，则它不会有任何动作。这不是用户操作按钮时的一般情况。解决方法如下：对于模板样式的按钮，必须给它指定触发器，使按钮在响应鼠标移动和鼠标单击时有不同的外观。

图 35-23

使用属性触发器(参见前面的内容)，就很容易解决这个问题。只需要把该触发器添加到 ControlTemplate 的 Triggers 集合中，如下所示。这里定义了两个触发器，一个属性触发器在按钮的 IsMouseOver 属性为 true 时激活，接着把椭圆 GelBackground 的 Fill 属性改为 RadialGradientBrush，其值从 Lime 改为 DarkGreen。使用 IsPressed 属性给 RadialGradientBrush 指定其他颜色：

```
<ControlTemplate.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter Property="Ellipse.Fill" TargetName="GelBackground">
      <Setter.Value>
        <RadialGradientBrush>
          <GradientStop Offset="0" Color="Lime" />
          <GradientStop Offset="1" Color="DarkGreen" />
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Trigger>
  <Trigger Property="IsPressed" Value="True">
    <Setter Property="Ellipse.Fill" TargetName="GelBackground">
      <Setter.Value>
        <RadialGradientBrush>
          <GradientStop Offset="0" Color="#ffcc34" />
          <GradientStop Offset="1" Color="#cc9900" />
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Trigger>

</ControlTemplate.Triggers>
```

现在就可以运行应用程序，只要把鼠标悬停在按钮上，或者单击鼠标，就会看到按钮的可见反馈。

35.10.2 数据模板

ContentControl 元素的内容可以是任意内容——不仅可以是 WPF 元素，还可以是.NET 对象。例如，可以把 Country 类型的对象赋予 Button 类的内容。下面的示例(代码文件 TemplateDemo/Country.cs)创建 Country 类，以表示国家名称和国旗(用一幅图像的路径表示)。这个类定义 Name 和 ImagePath 属性，并重写 ToString()方法，用于默认的字符串表示：

```
public class Country
{
```

```
        public string Name { get; set; }
        public string ImagePath { get; set; }

        public override string ToString()
        {
            return Name;
        }
    }
```

这些内容在按钮或任何其他ContentControl中会如何显示？默认情况下会调用ToString()方法，显示对象的字符串表示。要获得自定义外观，还可以为Country类型创建一个DataTemplate。

这里在Window的资源中创建一个DataTemplate。这个DataTemplate没有指定键，因此默认是Country.src类型——它也是引用.NET程序集的XML名称空间和.NET名称空间的别名。在DataTemplate内部，主元素是一个文本框，其Text属性绑定到Country的Name属性上，Source属性的Image绑定到Country的ImagePath属性上。Grid、Border和Rectangle元素定义了布局和可见外观(代码文件TemplateDemo/StyledButtonWindow.xaml)：

```
<Window.Resources>
  <DataTemplate DataType="{x:Type src:Country}">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="60" />
    </Grid.RowDefinitions>
    <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
        Text="{Binding Name}" FontWeight="Bold" Grid.Column="0" />
      <Border Margin="4,0" Grid.Column="1" BorderThickness="2"
          CornerRadius="4">
        <Border.BorderBrush>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#aaa" />
            <GradientStop Offset="1" Color="#222" />
          </LinearGradientBrush>
        </Border.BorderBrush>
        <Grid>
          <Rectangle>
            <Rectangle.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#444" />
                <GradientStop Offset="1" Color="#fff" />
              </LinearGradientBrush>
            </Rectangle.Fill>
          </Rectangle>
          <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
        </Grid>
      </Border>
    </Grid>
  </DataTemplate>
</Window.Resources>
```

在 XAML 代码中，定义一个简单的 Button 元素 button1：

```
<Button Grid.Row="1" x:Name="button1" Margin="10" />
```

在代码隐藏文件中(代码文件 TemplateDemo/StyledButtonWindow.xaml.cs)，实例化一个新的 Country 对象，并把它赋予 button1 的 Content 属性：

```
public StyledButtonWindow()
{
  InitializeComponent();
  button1.Content = new Country
  {
    Name = "Austria",
    ImagePath = "images/Austria.bmp"
  };
}
```

运行这个应用程序，可以看出，DataTemplate 应用于 Button，因为 Country 数据类型有默认的模板，如图 35-24 所示。

图 35-24

当然，也可以创建一个控件模板，并从中使用数据模板。

35.10.3 样式化列表框

更改按钮或标签的样式是一个简单的任务，例如改变包含一个元素列表的父元素的样式。如何更改列表框？列表框也有操作方式和外观。列表框可以显示一个元素列表，用户可以从列表中选择一个或多个元素。至于操作方式，ListBox 类定义了方法、属性和事件。ListBox 的外观与其操作是分开的。ListBox 元素有一个默认的外观，但可以通过创建模板，改变这个外观。

对于列表框，ControlTemplate 定义了整个控件的外观，ItemTemplate 定义了列表项的外观，DataTemplate 定义了项中的类型。为了给列表框填充一些项，静态类 Countries 返回几个要显示出来的国家(代码文件 TemplateDemo/Countries.cs)：

```
public class Countries
{
  public static IEnumerable<Country> GetCountries()
  {
    return new List<Country>
    {
      new Country { Name = "Austria", ImagePath = "Images/Austria.bmp" },
      new Country { Name = "Germany", ImagePath = "Images/Germany.bmp" },
      new Country { Name = "Norway", ImagePath = "Images/Norway.bmp" },
      new Country { Name = "USA", ImagePath = "Images/USA.bmp" }
    };
  }
}
```

在代码隐藏文件中(代码文件TemplateDemo/StyledListBoxWindow1.xaml.cs)，在StyledListBoxWindow1类的构造函数中，将StyledListBoxWindow1实例的DataContext属性设置为要从Countries.GetCountries()方法中返回的国家列表(DataContext属性是数据绑定的一个功能，详见第36章)。

```
public partial class StyledListBoxWindow1 : Window
{
  public StyledListBoxWindow1()
  {
    InitializeComponent();
    this.DataContext = Countries.GetCountries();
  }
}
```

在XAML代码(代码文件TemplateDemo/StyledListBoxWindow1.xaml)中，定义了countryList1列表框。countryList1没有不同的样式，它使用ListBox元素的默认外观。把ItemsSource属性设置为Binding标记扩展，它由数据绑定使用。从代码隐藏文件中，可以看到数据绑定用于一个Country对象数组。图35-25显示了ListBox的默认外观。在默认情况下，只在一个简单的列表中显示ToString()方法返回的国家的名称。

图35-25

```
<Window x:Class="TemplateDemo.StyledListBoxWindow1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:TemplateDemo"
    Title="StyledListBoxWindow1" Height="300" Width="300">
  <Grid>
    <ListBox ItemsSource="{Binding}" Margin="10" />
  </Grid>
</Window>
```

35.10.4 ItemTemplate

Country对象有名称和国旗。当然，还可以在列表框中显示这两个值。为此，必须定义一个模板。

ListBox元素包含ListBoxItem元素。使用ItemTemplate可以定义列表项的内容。listBoxStyle1样式定义一个ItemTemplate，其值为DataTemplate。DataTemplate用于将数据绑定到元素上。Binding标记扩展可以用于DataTemplate元素。

DataTemplate包含一个三列的栅格。第一列包含字符串“Country:”，第二列包含国家的名称，第三列包含该国家的国旗。因为国家名称的长度不同，但看起来每个国家名称应有相同的大小，所以给第二列定义设置SharedSizeGroup属性。只有这一列使用共享大小的信息，因为也设置了Grid.IsSharedSizeScope属性。

行和列定义之后，就可以看到两个TextBlock元素。第一个TextBlock元素包含文本“Country:”，第二个TextBlock元素绑定到Country类定义的Name属性上。

第三列的内容是一个包含栅格的Border元素。栅格包含一个带线性渐变画笔的矩形和一个绑定到Country类的ImagePath属性上的Image元素。图35-26显示了列表框中的国家，其效果与前面完全不同(代码文件TemplateDemo/Styles.xaml)。

图35-26

```
<Style x:Key="ListBoxStyle1" TargetType="{x:Type ListBox}" >
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
           <RowDefinition Height="60" />
          </Grid.RowDefinitions>
          <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                     FontStyle="Italic" Grid.Column="0" Text="Country:" />
          <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                     Text="{Binding Name}" FontWeight="Bold" Grid.Column="1" />
          <Border Margin="4,0" Grid.Column="2" BorderThickness="2"
                  CornerRadius="4">
            <Border.BorderBrush>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#aaa" />
                <GradientStop Offset="1" Color="#222" />
              </LinearGradientBrush>
            </Border.BorderBrush>
            <Grid>
              <Rectangle>
                <Rectangle.Fill>
                  <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Offset="0" Color="#444" />
                    <GradientStop Offset="1" Color="#fff" />
                  </LinearGradientBrush>
                </Rectangle.Fill>
              </Rectangle>
              <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
            </Grid>
          </Border>
        </Grid>
      </DataTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="Grid.IsSharedSizeScope" Value="True" />
</Style>
```

35.10.5　列表框元素的控件模板

列表框中的项不一定垂直排列，还可以给同一个功能提供另一种视图。下一个样式 ListBoxStyle2 定义了一个模板，使列表项水平显示，且带一个滚动条。

在前面的示例中，只创建了一个 ItemTemplate，来定义列表项在默认列表框中的外观。下面的代码(代码文件 TemplateDemo/Styles.xaml)创建一个模板，以定义另一个列表框。该模板包含一个 ControlTemplate 元素，它定义了列表框的元素。ControlTemplate 元素现在是一个包含 StackPanel 的 ScrollViewer——带滚动条的视图。因为列表项现在应水平排列，所以把 StackPanel 的 Orientation 设

置为 Horizontal。StackPanel 还包含用 ItemTemplate 定义的列表项，于是把 StackPanel 元素的 IsItemsHost 属性设置为 true。IsItemsHost 属性可以用于包含一组列表项的所有 Panel 元素。

ItemTemplate 定义了 StackPanel 中数据项的外观。ItemTemplate 使用 ListBoxStyle1 样式，ListBoxStyle2 基于 ListBoxStyle1 样式。

图 35-27 显示了使用 listBoxStyle2 样式的列表框，当视图太小，不能显示列表中的所有项时，会自动显示滚动条。

```
<Style x:Key="ListBoxStyle2" TargetType="{x:Type ListBox}"
    BasedOn="{StaticResource ListBoxStyle1}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer HorizontalScrollBarVisibility="Auto">
          <StackPanel Name="StackPanel1" IsItemsHost="True"
              Orientation="Horizontal" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="VerticalAlignment" Value="Center" />
</Style>
```

显然，将控件的外观与其行为分开很有好处。也许用户有许多方式，能很好地显示列表中的项，使之满足应用程序的要求。也许用户只想在窗口中显示尽量多的项，先水平排列，一行放不下时，就继续在下一行竖直排列。此时就可以使用 WrapPanel。当然，可以给列表框的模板定义一个 WrapPanel，如 listBoxStyle3 所示。图 35-28 显示了使用 WrapPanel 的结果：

```
<Style x:Key="ListBoxStyle3" TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer VerticalScrollBarVisibility="Auto"
            HorizontalScrollBarVisibility="Disabled">
          <WrapPanel IsItemsHost="True" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
```

```
                    Source="{Binding ImagePath}" />
                <TextBlock Grid.Row="1" FontSize="14"
                    HorizontalAlignment="Center" Margin="5" Text="{Binding Name}" />
            </Grid>
          </DataTemplate>
        </Setter.Value>
      </Setter>
    </Style>
```

图 35-27

图 35-28

35.11 动画

在动画中，可以使用移动的元素、颜色变化、变换等制作平滑的变换效果。WPF 使动画的制作非常简单。还可以连续改变任意依赖属性的值。不同的动画类可以根据其类型，连续改变不同属性的值。

动画的主要元素如下：

- 时间轴——定义了值随时间的变化方式。有不同类型的时间轴，可用于改变不同类型的值。所有时间轴的基类都是 Timeline。为了连续改变 double 值，可以使用 DoubleAnimation 类。Int32Animation 类是 int 值的动画类。PointAnimation 类用于连续改变点，ColorAnimation 类用于连续改变颜色。
- 故事板——用于合并动画。Storyboard 类派生自基类 TimelineGroup，TimelineGroup 又派生自基类 Timeline。使用 DoubleAnimation 类，可以连续改变 double，使用 Storyboard 类可以合并所有应连接在一起的动画。
- 触发器——通过触发器可以启动和停止动画。前面介绍了属性触发器。当属性值变化时，属性触发器就会激活。还可以创建事件触发器，当事件发生时，事件触发器就会激活。

动画类的名称空间是 System.Windows.Media.Animation。

35.11.1 时间轴

时间轴定义了值随时间变化的方式。下面的示例连续改变椭圆的大小。在下面的代码中(代码文件 AnimationDemo/EllipseWindow.xaml)，DoubleAnimation 时间轴改变为 double 值。把 Ellipse 类的 Triggers 属性设置为 EventTrigger。椭圆加载时，就激活用 EventTrigger 的 RoutedEvent 属性定义的事件触发器。BeginStoryboard 是启动故事板的触发器动作。在故事板中，DoubleAnimation 元素用于连续改变 Ellipse 类的 Width 属性。动画在 3 秒内把椭圆的宽度从 100 改为 300，在之后的 3 秒内再恢复原状。

ColorAnimation 连续改变 ellipseBrush 中用于填充椭圆的颜色：

```
<Ellipse Height="50" Width="100">
  <Ellipse.Fill>
    <SolidColorBrush x:Name="ellipseBrush" Color="Yellow" />
  </Ellipse.Fill>
  <Ellipse.Triggers>
    <EventTrigger RoutedEvent="Ellipse.Loaded" >
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
            <DoubleAnimation Storyboard.TargetProperty="(Ellipse.Width)"
                Duration="0:0:3" AutoReverse="True" FillBehavior="Stop"
                RepeatBehavior="Forever" AccelerationRatio="0.9"
                DecelerationRatio="0.1" From="100" To="300" />
            <ColorAnimation Storyboard.TargetName="ellipseBrush"
                Storyboard.TargetProperty="(SolidColorBrush.Color)"
                Duration="0:0:3" AutoReverse="True"
                FillBehavior="Stop" RepeatBehavior="Forever"
                From="Yellow" To="Red" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Ellipse.Triggers>
</Ellipse>
```

图 35-29 和图 35-30 显示了具有动画效果的椭圆的两个状态。

图 35-29　　　　图 35-30

动画并不仅仅是一直和立刻显示在屏幕上的一般窗口动画。还可以给业务应用程序添加动画，使用户界面的响应性更好。

下面的例子(代码文件 AnimationDemo/ButtonAnimationWindow.xaml)演示了一个相当好的动画，还说明了如何在样式中定义动画。在 Window 资源中，有一个用于按钮的 AnimatedButtonStyle 样式。在模板中定义了一个矩形 outline。这个模板使用很细的笔触，把其宽度设置为 0.4。

该模板为 IsMouseOver 属性定义了一个属性触发器。当鼠标移过按钮时，就应用这个触发器的 EnterActions 属性。启动动作是 BeginStoryboard，它是一个触发器动作，可以包含并启动 Storyboard 元素。Storyboard 元素定义了一个 DoubleAnimation，可以为 double 值制作动画。在这个动画中改变的属性值是 Rectangle 元素 outline 的 StrokeThickness 属性。该值平滑地增加 1.2，因为 By 属性指定，该属性变化的时间长度是 Duration 属性设置的 0.3 秒。在动画结束时，笔触的宽度重置为其初始值，因为 AutoReverse="True"。总之，只要鼠标移过按钮，outline 的边框就在 0.3 秒内增加 1.2。图 35-31 显示了没有改变的按钮。图 35-32 显示了鼠标移过按钮 0.3 秒后的按钮。在纸质媒介上，不可能显示平滑的动画和按钮外观的中间状态。

图 35-31

图 35-32

```
<Window x:Class="AnimationDemo.ButtonAnimationWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ButtonAnimationWindow" Height="300" Width="300">
  <Window.Resources>
    <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="{x:Type Button}">
            <Grid>
              <Rectangle Name="outline" RadiusX="9" RadiusY="9"
                  Stroke="Black" Fill="{TemplateBinding Background}"
                  StrokeThickness="1.6">
              </Rectangle>
              <ContentPresenter VerticalAlignment="Center"
                  HorizontalAlignment="Center" />
            </Grid>
            <ControlTemplate.Triggers>
              <Trigger Property="IsMouseOver" Value="True">
                <Trigger.EnterActions>
                  <BeginStoryboard>
                    <Storyboard>
                      <DoubleAnimation Duration="0:0:0.3" AutoReverse="True"
                        Storyboard.TargetProperty="(Rectangle.StrokeThickness)"
                        Storyboard.TargetName="outline" By="1.2" />
                    </Storyboard>
                  </BeginStoryboard>
                </Trigger.EnterActions>
              </Trigger>
            </ControlTemplate.Triggers>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </Window.Resources>
  <Grid>
    <Button Style="{StaticResource AnimatedButtonStyle}" Width="200"
       Height="100" Content="Click Me!" />
  </Grid>
</Window>
```

Timeline 可以完成的任务如表 35-11 所示。

表 35-11

Timeline 属性	说 明
AutoReverse	使用 AutoReverse 属性，可以指定连续改变的值在动画结束后是否返回初始值

(续表)

Timeline 属性	说 明
SpeedRatio	使用 SpeedRatio，可以改变动画的移动速度。在这个属性中，可以定义父子元素的相对关系。默认值为 1；将速率设置为较小的值，会使动画移动较慢；将速率设置为高于 1 的值，会使动画移动较快
BeginTime	使用 BeginTime，可以指定从触发器事件开始到动画开始移动之间的时间长度。其单位可以是天、小时、分钟、秒和几分之秒。根据 SpeedRatio，这可以不是真实的时间。例如，如果把 SpeedRatio 设置为 2，把开始时间设置为 6 秒，动画就在 3 秒后开始
AccelerationRatio DecelerationRatio	在动画中，值不一定是线性变化。可以指定 AccelerationRatio 和 DecelerationRatio，定义加速度和减速度。这两个值的总和不能超过 1
Duration	使用 Duration 属性，可以指定动画重复一次的时间长度
RepeatBehavior	给 RepeatBehavior 属性指定一个 RepeatBehavior 结构，可以定义动画的重复次数或重复时间
FillBehavior	如果父元素的时间轴有不同的持续时间，FillBehavior 属性就很重要。例如，如果把父元素的时间轴比实际动画的持续时间短，将 FillBehavior 设置为 Stop 就表示实际动画停止。如果父元素的时间轴比实际动画的持续时间长，HoldEnd 就会一直执行动画，直到把它重置为初始值为止(假定把 AutoReverse 设置为 true)

根据 Timeline 类的类型，还可以使用其他一些属性。例如，使用 DoubleAnimation，可以为动画的开始和结束设置 From 和 To 属性。还可以设置 By 属性，用 Bound 属性的当前值启动动画，该属性值会递增由 By 属性指定的值。

35.11.2 非线性动画

定义非线性动画的一种方式是设置 AccelerationRatio 和 DecelerationRatio，指定动画在开始和结束时的速度。.NET 4.5 有比这更灵活的方式。

几个动画类有 EasingFunction 属性。这个属性接受一个实现了 IEasingFunction 接口的对象。通过这个接口，缓动函数对象可以定义值随着时间如何变化。有几个缓动函数可用于创建非线性动画，如 ExponentialEase，它给动画使用指数公式；QuadraticEase、CubicEase、QuarticEase 和 QuinticEase 的指数分别是 2、3、4、5，PowerEase 的指数是可以配置的。特别有趣的是 SineEase，它使用正弦曲线，BounceEase 创建弹跳效果，ElasticEase 用弹簧的来回震荡来模拟动画值。

要在 XAML 中指定这种缓动效果，可以把该缓动效果添加到动画的 EasingFunction 属性中，如下所示(代码文件 AnimationDemo/EllipseWindow.xaml)。添加不同的缓动函数，就会看到动画的有趣效果：

```
<DoubleAnimation Storyboard.TargetProperty="(Ellipse.Width)"
                 Duration="0:0:3" AutoReverse="True"
                 FillBehavior=" RepeatBehavior="Forever"
                 From="100" To="300">
  <DoubleAnimation.EasingFunction>
    <BounceEase EasingMode="EaseInOut" />
  </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

35.11.3 事件触发器

除了使用属性触发器之外，还可以定义一个事件触发器，来启动动画。属性触发器在属性改变其值时激活，事件触发器在事件发生时激活。这种事件的例子包括控件的 Load 事件、按钮的 Click 事件，以及 MouseMove 事件等。

下一个例子将为前面通过形状创建的笑脸创建一个动画。创建动画后，一旦激活按钮的 Click 事件，眼睛就会移动。

在 Window 元素中，定义了一个 DockPanel 元素，来排列笑脸和控制动画的按钮。包含 3 个按钮的 StackPanel 停靠在顶部。包含笑脸的 Canvas 元素占用了 DockPanel 的其余部分。

第 1 个按钮用于启动眼睛的动画，第 2 个按钮用于停止动画，第 3 个按钮用于启动另一个重置笑脸大小的动画。

动画在 DockPanel.Triggers 部分中定义。这里没有使用属性触发器，而使用了事件触发器。RoutedEvent 和 SourceName 属性定义了 buttonBeginMoveEyes 按钮，一旦该按钮的 Click 事件发生，就激活第 1 个事件触发器。触发器动作由 BeginStoryboard 元素定义，它启动所包含的 Storyboard。BeginStoryboard 定义了一个名称，它用于控制故事板的暂停、继续和停止动作。Storyboard 元素包含 4 个动画。前两幅动画连续改变左眼，后两幅动画连续改变右眼。第 1 幅和第 3 幅动画改变眼睛的 Canvas.Left 位置，第 2 幅和第 4 幅动画改变 Canvas.Top。动画在 x 和 y 方向上有不同的时间值，使用指定的重复行为使眼睛的运动更有趣。

一旦 buttonStopMoveEyes 按钮的 Click 事件发生，就激活第 2 个事件触发器。在这里，故事板用 StopStoryboard 元素停止，该元素引用了起始故事板 beginMoveEye。

第 3 个事件触发器通过单击 buttonResize 按钮来激活。在这幅动画中，改变了 Canvas 元素的变换。因为这幅动画不会无限制地运行下去，所以它不需要停止。这个故事板也使用前面介绍的 EaseFunction(代码文件 AnimationDemo/EventTriggerWindow.xaml)：

```
<Window x:Class="AnimationDemo.EventTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EventTriggerWindow" Height="300" Width="300">
  <DockPanel>
    <DockPanel.Triggers>
      <EventTrigger RoutedEvent="Button.Click" SourceName="buttonBeginMoveEyes">
        <BeginStoryboard x:Name="beginMoveEyes">
          <Storyboard>
            <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                AutoReverse="True" By="6" Duration="0:0:1"
                Storyboard.TargetName="eyeLeft"
                Storyboard.TargetProperty="(Canvas.Left)" />
            <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                By="6" Duration="0:0:5"
                Storyboard.TargetName="eyeLeft"
                Storyboard.TargetProperty="(Canvas.Top)" />
            <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                AutoReverse="True" By="-6" Duration="0:0:3"
                Storyboard.TargetName="eyeRight"
                Storyboard.TargetProperty="(Canvas.Left)" />
            <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                By="6" Duration="0:0:6"
```

```
            Storyboard.TargetName="eyeRight"
            Storyboard.TargetProperty="(Canvas.Top)" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.Click" SourceName="buttonStopMoveEyes">
    <StopStoryboard BeginStoryboardName="beginMoveEyes" />
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.Click" SourceName="buttonResize">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
            From="0.1" To="3" Duration="0:0:5">
          <DoubleAnimation.EasingFunction>
            <ElasticEase />
          </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
        <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="(ScaleTransform.ScaleY)"
            From="0.1" To="3" Duration="0:0:5">
          <DoubleAnimation.EasingFunction>
            <BounceEase />
          </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</DockPanel.Triggers>
<StackPanel Orientation="Vertical" DockPanel.Dock="Top">
  <Button x:Name="buttonBeginMoveEyes" Content="Start Move Eyes" Margin="5" />
  <Button x:Name="buttonStopMoveEyes" Content="Stop Move Eyes" Margin="5" />
  <Button x:Name="buttonResize" Content="Resize" Margin="5" />
</StackPanel>
<Canvas>
  <Canvas.LayoutTransform>
    <ScaleTransform x:Name="scale1" ScaleX="1" ScaleY="1" />
</Canvas.LayoutTransform>
<Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
    Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
<Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
  <Ellipse.Fill>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
      <GradientStop Offset="0.1" Color="DarkGreen" />
      <GradientStop Offset="0.7" Color="Transparent" />
    </LinearGradientBrush>
  </Ellipse.Fill>
</Ellipse>
<Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
    Stroke="Blue" StrokeThickness="3" Fill="White" />
<Ellipse x:Name="eyeLeft" Canvas.Left="40" Canvas.Top="43" Width="6"
    Height="5" Fill="Black" />
<Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
```

```
            Stroke="Blue" StrokeThickness="3" Fill="White" />
        <Ellipse x:Name="eyeRight" Canvas.Left="75" Canvas.Top="43" Width="6"
            Height="5" Fill="Black" />
        <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
            Data="M 40,74 Q 57,95 80,74 " />
      </Canvas>
    </DockPanel>
</Window>
```

图 35-33 显示了运行应用程序的结果。

图 35-33

除了直接从 XAML 的事件触发器中启动和停止动画之外，还可以从代码隐藏文件中控制动画。只需要给故事板指定一个名称，并调用 Begin()、Stop()、Pause()和 Resume()方法。

35.11.4 关键帧动画

如前所述，使用加速比、减速比和缓动函数，就可以用非线性的方式制作动画。如果需要为动画指定几个值，就可以使用关键帧动画。与正常的动画一样，关键帧动画也有不同的动画类型，它们可以改变不同类型的属性。

DoubleAnimationUsingKeyFrames 是双精度类型的关键帧动画。其他关键帧动画类型有 Int32AnimationUsingKeyFrames、PointAnimationUsingKeyFrames、ColorAnimationUsingKeyFrames、SizeAnimationUsingKeyFrames 和 ObjectAnimationUsingKeyFrames。

示例 XAML 代码(代码文件 AnimationDemo/KeyFrameWindow.xaml)连续地改变 TranslateTransform 元素的 X 值和 Y 值，从而改变椭圆的位置。把 EventTrigger 定义为 RountedEvent 的 Ellipse.Loaded，动画就会在加载椭圆时启动。事件触发器用 BeginStoryboard 元素启动一个 Storyboard。该 Storyboard 包含两个 DoubleAnimationUsingKeyFrame 类型的关键帧动画。关键帧动画由帧元素组成。第一幅关键帧动画使用一个 LinearKeyFrame、一个 DiscreteDoubleKeyFrame 和一个 SplineDoubleKeyFrame；第二幅关键帧动画是一个 EasingDoubleKeyFrame。LinearDoubleKeyFrame 使对应值线性变化。KeyTime 属性定义了动画应何时达到 Value 属性的值。

这里 LinearDoubleKeyFrame 用 3 秒的时间使 X 属性到达值 30。DiscreteDoubleKeyFrame 在 4 秒后立即改变为新值。SplineDoubleKeyFrame 使用贝塞尔曲线，其中的两个控制点由 KeySpline 属性指定。EasingDoubleKeyFrame 是一个帧类，它支持设置缓动函数(如 BounceEase)来控制动画值：

```
<Canvas>
  <Ellipse Fill="Red" Canvas.Left="20" Canvas.Top="20" Width="25" Height="25">
    <Ellipse.RenderTransform>
      <TranslateTransform X="50" Y="50" x:Name="ellipseMove" />
    </Ellipse.RenderTransform>
      <Ellipse.Triggers>
        <EventTrigger RoutedEvent="Ellipse.Loaded">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="X"
                  Storyboard.TargetName="ellipseMove">
                <LinearDoubleKeyFrame KeyTime="0:0:2" Value="30" />
                <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="80" />
                <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                    KeyTime="0:0:10" Value="300" />
                <LinearDoubleKeyFrame KeyTime="0:0:20" Value="150" />
              </DoubleAnimationUsingKeyFrames>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Y"
                  Storyboard.TargetName="ellipseMove">
                <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                    KeyTime="0:0:2" Value="50" />
                <EasingDoubleKeyFrame KeyTime="0:0:20" Value="300">
                  <EasingDoubleKeyFrame.EasingFunction>
                    <BounceEase />
                  </EasingDoubleKeyFrame.EasingFunction>
                </EasingDoubleKeyFrame>
              </DoubleAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Ellipse.Triggers>
  </Ellipse>
</Canvas>
```

35.12 可见状态管理器

自从.NET 4 以来，Visual State Manager 提供了控制动画的另一种方式。控件可以有特定的状态。状态定义了到达该状态时应用于控件的外观。状态切换定义了一种状态变成另一种状态时会发生什么。

在数据网格中，可以使用 Read、Selected 和 Edit 状态，根据用户的选择给行定义不同的外观。MouseOver 和 IsPressed 可以是状态，也可以用来替代前面讨论的触发器。

下面的示例(代码文件 VisualStateDemo/Style.xaml)为 Button 类型创建了一个定制模板，其中使用可见的状态，而不是前面使用的触发器。这个代码段中的 XAML 代码给 Button 类型定义了一个模板，其中包含使用渐变画笔的 Ellipse 元素。如代码所示，用户把鼠标移动到按钮上或单击它时，什么都不会发生。这将使用可见的状态来改变。

```
<Style TargetType="Button">
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Button}">
        <Grid>
          <Ellipse Name="GelBackground" StrokeThickness="0.5">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="Black" />
                <GradientStop Offset="1" Color="Black" />
              </LinearGradientBrush>
            </Ellipse.Fill>
            <Ellipse.Stroke>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#ff7e7e7e" />
                <GradientStop Offset="1" Color="Black" />
              </LinearGradientBrush>
            </Ellipse.Stroke>
          </Ellipse>
          <Ellipse Margin="15,5,15,50">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#aaffffff" />
                <GradientStop Offset="1" Color="Transparent" />
              </LinearGradientBrush>
            </Ellipse.Fill>
          </Ellipse>
          <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
              HorizontalAlignment="Center"
              Content="{TemplateBinding Content}" />
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

35.12.1 可见的状态

Buttom 类型定义了几个状态组和状态。状态组 CommonStates 定义了状态 Normal、MouseOver 和 Pressed。状态组 FocusedStates 定义了 Focused 和 Unfocused。如下面的代码所示(代码文件 VisualStateDemo/Style.xaml)，Buttom 类的实现代码使用 VisualStateManager 改变了状态——只需要给这些状态定义外观。

为了使用可见状态给控件定义不同的外观，应在模板中定义附加属性 VisualStateManager.VisualStateGroups。定义的第一组是 CommonStates。在这个组中，定义了 MouseOver 和 Pressed 的外观。在 MouseOver 状态中，一个关键帧动画将椭圆的填充色从绿黄色逐渐改变为暗绿色。Pressed 状态有类似的实现方式：填充色从 ffcc34 逐渐改变为 cc9900。

```
<ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Content="{TemplateBinding Content}" />
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
    <VisualState Name="Normal" />
    <VisualState Name="MouseOver">
      <Storyboard>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[0].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="Lime"/>
        </ColorAnimationUsingKeyFrames>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[1].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="DarkGreen"/>
        </ColorAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
    <VisualState Name="Pressed">
      <Storyboard>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[0].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="#ffcc34"/>
        </ColorAnimationUsingKeyFrames>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[1].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="#cc9900"/>
        </ColorAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
  <VisualStateGroup Name="FocusedStates">
    <VisualState Name="Focused" />
    <VisualState Name="Unfocused" />
```

```
  </VisualStateGroup>

</VisualStateManager.VisualStateGroups>
```

状态的改变是很明显的。把鼠标移动到Button上，或者单击Button，就会改变其用户界面。接着添加状态变换之间的动画。

35.12.2 变换

在状态变换中，可以定义状态改变时应该发生什么。变换使用VisualStateManager. Transitions来添加。在下面的示例代码中(代码文件VisualStateDemo/Style.xaml)，第一个变换是全局性的，它指定状态改变应持续0.2秒，并对这个动画使用QuadraticEase函数。如果状态变成MouseOver，就指定第二个定义的变换。这个状态变换的实现代码把椭圆中GelBackground元素的厚度在0.5秒内增加2，动画完成后，就返回其初始状态。

```
<ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Content="{TemplateBinding Content}" />
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
    <!-- ... -->
    <VisualStateGroup.Transitions>
      <VisualTransition GeneratedDuration="0:0:0.2">
        <VisualTransition.GeneratedEasingFunction>
          <QuadraticEase EasingMode="EaseOut" />
        </VisualTransition.GeneratedEasingFunction>
      </VisualTransition>
      <VisualTransition GeneratedDuration="0:0:0.5" To="MouseOver">
        <Storyboard>
          <DoubleAnimation By="2" Duration="0:0:0.5"
              AutoReverse="True"
              Storyboard.TargetProperty="(Shape.StrokeThickness)"
              Storyboard.TargetName="GelBackground" />
        </Storyboard>
      </VisualTransition>
    </VisualStateGroup.Transitions>
  </VisualStateGroup>
  <VisualStateGroup Name="FocusedStates">
    <VisualState Name="Focused" />
    <VisualState Name="Unfocused" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

使用定制状态，很容易调用VisualStateManager类的方法GoToElementState来改变状态。

35.13 3-D

本章的最后一节介绍 WPF 的 3D 功能，其中包含了开始使用该功能的信息。

WPF 的 3D 特性在 System.Windows.Media.Media3D 名称空间中。

为了理解 WPF 的 3D 功能，一定要知道坐标系统之间的区别。图 35-34 显示了 WPF 中的 3D 坐标系统。原点位于中心。x 轴的正值在右边，负值在左边。y 轴是垂直的，正值在上边，负值在下边。z 轴在指向观察者的方向上定义正值。

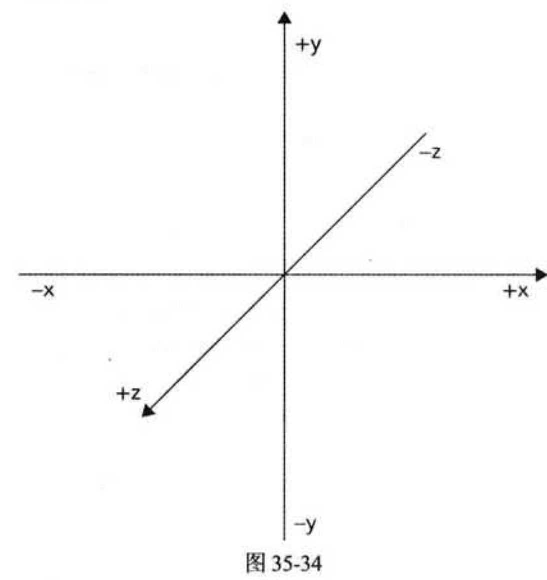


图 35-34

为了理解 WPF 的 3D 特性，最重要的概念是模型、照相机和光线。模型定义了使用三角形显示的内容，照相机定义了查看模型的位置和方式，而没有光线，模型就是黑暗的。光线定义了完整的场景如何照明。本节介绍如何使用 WPF 定义模型、照相机和光线，以及有哪些不同的选项。另外，还讨论如何为场景制作动画。

35.13.1 模型

本节将创建一个具有 3D 效果的图书模型。因为 3D 模型由三角形组成，所以最简单的模型仅是一个三角形。更复杂的模型由多个三角形组成。矩形由两个三角形组成，球由多个三角形组成，所使用的三角形越多，球就越光滑。

对于图书模型，每一面都是矩形，矩形仅由两个三角形组成。但是，因为书的封面有 3 种不同的材质，所以要使用 6 个三角形。

三角形用 MeshGeometry3D 的 Positions 属性定义。这里仅是书的封面的一部分。MeshGeometry3D 定义两个三角形。可以只计算 5 个点的坐标，因为第一个三角形的第三个点就是第二个三角形的第一个点。还可以进行优化，以减小模型的尺寸。所有这些点都使用相同的 z 坐标 0，而(x,y)坐标分别是(0,0)，

(10,0)，(0,10)，(10,10)和(10,0)。TriangleIndices 属性指定这些点的顺序。第一个三角形按顺时针定义，第二个三角形按逆时针定义。使用这个属性可以指定三角形的哪一边是可见的。三角形的一边显示 GeometryModel3D 类的 Material 属性定义的颜色，另一边显示 BackMaterial 属性。

3D 的渲染表面是包围模型的 ModelVisual3D，如下所示(代码文件 3DDemo/MainWindow.xaml)：

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <Model3DGroup>

      <!-- front -->
      <GeometryModel3D>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D
              Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
              TriangleIndices="0, 1, 2, 2, 4, 3" />
        </GeometryModel3D.Geometry>
```

GeometryModel3D 类的 Material 属性指定模型使用什么材质。根据视点的不同，Material 或 BackMaterial 属性很重要。

WPF 提供不同的材质类型：DiffuseMaterial、EmissiveMaterial 和 SpecularMaterial。材质和用于给场景照明的光线会影响模型的外观。计算 EmissiveMaterial 和应用于材质画笔的颜色可定义光线，以显示模型。SpecularMaterial 在发生高光反射时会添加眩目的高光反射效果。示例代码使用 DiffuseMaterial，并引用 mainCover 资源中的画笔：

```
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="{StaticResource mainCover}" />
  </GeometryModel3D.Material>
</GeometryModel3D>
```

用于主封面的画笔是 VisualBrush。VisualBrush 有一个 Border 元素，它是由两个 Label 元素组成的 Grid。一个 Label 元素定义了文本 Professional C# 4，并写在封面上：

```
<VisualBrush x:Key="mainCover">
  <VisualBrush.Visual>
    <Border Background="Red">
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="30" />
          <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Label Grid.Row="0" HorizontalAlignment="Center">
           Professional C# 5</Label>
        <Label Grid.Row="1"></Label>
      </Grid>
    </Border>
  </VisualBrush.Visual>
</VisualBrush>
```

因为画笔用2D坐标系统定义，而模型在3D坐标系统中定义，所以需要对它们进行转换。这个转换使用MeshGeometry3D的TextureCoordinates属性来完成。TextureCoordinates属性指定了三角形的每个点，指定这些点如何映射到2D上。第一个点(0,0,0)映射到(0,1)上，第二个点(10,0,0)映射到(1,1)上，以此类推。注意y在3D和2D坐标系统中有不同的方向。图35-35显示了2D坐标系统。

```
<MeshGeometry3D Positions="0 0 0, 10 0 0, 0 10
0, 10 10 0, 10 0 0"
    TriangleIndices="0, 1, 2, 2, 4, 3"
    TextureCoordinates="0 1, 1 1, 0 0, 1 0, 1 1" />
```

0/0　1　x　1　y

图35-35

35.13.2 照相机

需要对3D模型使用照相机，以查看模型。本例(代码文件3DDemo/MainWindow.xaml)使用PerspectiveCamera，它需要指定位置和方向。把照相机的位置向左移动，会使模型向右移动，反之亦然。改变照相机的y坐标，模型会显得更大或更小。使照相机与模型相距越远，模型就越小：

```
<Viewport3D.Camera>
  <PerspectiveCamera Position="0,0,25" LookDirection="15,6,-50" />
</Viewport3D.Camera>
```

WPF还有一个OrtographicCamera，因为它在场景中没有地平线，所以如果它移动得远一些，元素的大小不会改变。使用MatrixCamera，可以精确地指定照相机的行为。

35.13.3 光线

没有光线，就是一片黑暗。3D场景需要光线使模型可见。可以使用不同的光线。AmbientLight会均匀地照亮场景。DirectionalLight只照亮一个方向，类似于阳光。PointLight在空间中有一个位置，它会照亮所有方向。SpotLight也有一个位置，但其照亮的区域仅限于一个圆锥。

示例代码使用SpotLight，指定了位置、方向和锥角：

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <SpotLight Color="White" InnerConeAngle="20" OuterConeAngle="60"
               Direction="15,6,-50" Position="0,0,25" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

35.13.4 旋转

为了得到模型的3D外观，模型还应能旋转。要进行旋转，可以使用RotateTransform3D元素，来定义旋转的中心和旋转角度：

```
<Model3DGroup.Transform>
  <RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">
    <RotateTransform3D.Rotation>
```

```
      <AxisAngleRotation3D x:Name="angle" Axis="-1,-1,-1" Angle="70" />
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</Model3DGroup.Transform>
```

要旋转已完成的模型，可以通过一个事件触发器启动动画。该动画连续地改变 AxisAngleRotation3D 元素的 Angle 属性：

```
<Window.Triggers>
  <EventTrigger RoutedEvent=f"Window.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation From="0" To="360" Duration="00:00:10"
            Storyboard.TargetName="angle"
            Storyboard.TargetProperty="Angle"
            RepeatBehavior="Forever" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Window.Triggers>
```

运行应用程序，结果如图 35-36 所示。

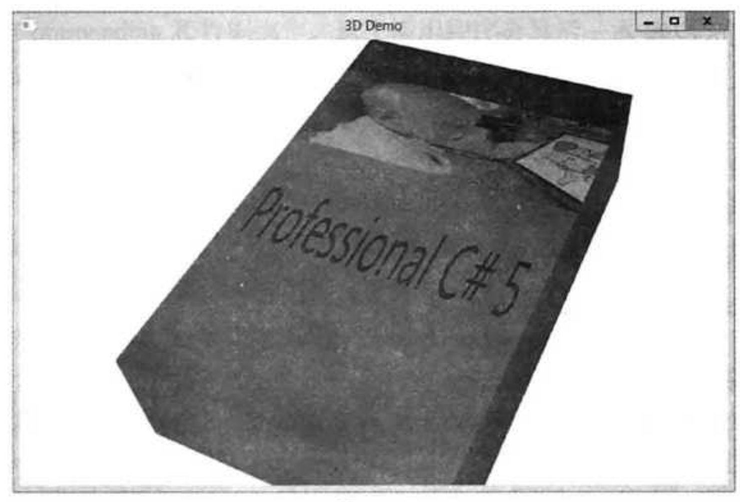


图 35-36

35.14 小结

本章介绍了 WPF 的许多功能。WPF 便于分开开发人员和设计人员的工作。所有 UI 功能都可以使用 XAML 创建，其功能用代码隐藏文件创建。

我们还探讨了基于矢量图形的许多控件和容器。因为 WPF 元素是矢量图形，所以可以缩放、倾斜和旋转。因为内容控件的内容非常灵活，所以事件处理机制基于冒泡和隧道事件。

可以使用不同类型的画笔绘制背景和前景元素，不仅可以使用纯色画笔、线性渐变或放射性渐变画笔，而且可以使用可视化画笔完成反射功能或显示视频。

样式和模板可以定制控件的外观。触发器可以动态地改变 WPF 控件的属性。连续改变 WPF 控件的属性值，就可以轻松地制作出动画。第 36 章继续介绍 WPF，主要探讨数据绑定、命令、导航和其他几个功能。

第36章 用WPF编写业务应用程序

本章要点

- 菜单和功能区控件
- 给处理输入使用 Commanding
- 绑定到元素、对象、列表和 XML 的数据
- 值的转换和验证
- 使用 TreeView 显示层次数据
- 使用 DataGrid 显示和组合数据
- 使用 CollectionView Source 实时成型

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Books Demo
- Multi Binding Demo
- Priority Binding Demo
- XML Binding Demo
- Validation Demo
- Formula 1 Demo
- Live Shaping

36.1 概述

第 35 章介绍了 WPF 的一些核心功能，本章继续用 WPF 编程，介绍创建完整应用程序的一些重要方面，如数据绑定和命令处理，以及 DataGrid 控件。数据绑定是把.NET 类中的数据提供给用户界面的一个重要概念，还允许用户修改数据。WPF 不仅允许绑定到简单的项或列表，还可以利用

本章介绍的多绑定和优先绑定功能，把一个 UI 属性绑定到类型可能不同的多个属性上。在数据绑定的过程中，验证用户输入的数据也很重要。本章将学习数据验证的不同方式，包括.NET 4.5 新增的 INotifyDataErrorInfo 接口。Commanding 可以把 UI 的事件映射到代码上。与事件模型相反，它更好地分隔开了 XAML 和代码，我们还将学习使用预定义的命令和创建定制的命令。

TreeView 和 DataGrid 控件是显示绑定数据的 UI 控件。TreeView 控件可以在树型结构中显示数据，其中数据根据用户的选择动态加载。通过 DataGrid 控件学习如何使用过滤、排序和分组，以及.NET 4.5 的一个新增功能——实时成型，它可以实时改变排序或过滤选项。

首先介绍菜单和功能区控件。功能区控件是.NET 4.5 中新增的。

36.2　菜单和功能区控件

许多数据驱动的应用程序都包含菜单和工具栏或功能区控件，允许用户控制操作。在 WPF 4.5 中，也可以使用功能区控件，所以这里介绍菜单和功能区控件。

本节会创建一个新的 WPF 应用程序 BooksDemo，本章将一直使用它——它不仅包含菜单和功能区控件，还包含 Commanding 和数据绑定。这个应用程序会显示一本书、一个图书列表和一个图书网格。操作由与命令关联的菜单或功能区控件来启动。

36.2.1　菜单控件

在 WPF 中，菜单很容易使用 Menu 和 MenuItem 元素创建，如下面的代码段所示，其中包含两个主菜单 File 和 Edit，以及一个子菜单项列表。字符前面的_标识了可不使用鼠标而直接访问菜单项的特定字符。使用 Alt 键可以使这些字符可见，并使用该字符访问菜单。其中一些菜单项指定了一个命令，如下一节所示(XAML 文件 BooksDemo/MainWindow.xaml)：

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Books Demo App" Height="400" Width="600">
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_File">
        <MenuItem Header="Show _Book" />
        <MenuItem Header="Show Book_s" />
        <Separator />
        <MenuItem Header="E_xit" />
      </MenuItem>
      <MenuItem Header="_Edit">
        <MenuItem Header="Undo" Command="Undo" />
        <Separator />
        <MenuItem Header="Cut" Command="Cut" />
        <MenuItem Header="Copy" Command="Copy" />
        <MenuItem Header="Paste" Command="Paste" />
      </MenuItem>
    </Menu>
  </DockPanel>
</Window>
```

运行应用程序，得到的菜单如图 36-1 所示。菜单目前还不能用，因为命令还没有激活。

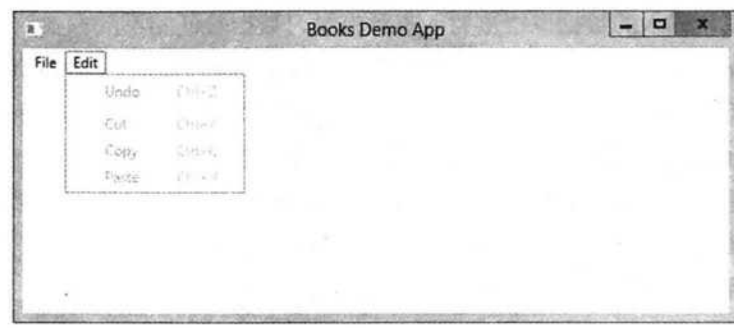


图 36-1

36.2.2 功能区控件

Microsoft Office 是 Microsoft 引入新开发的功能区控件后发布的第一个应用程序。引入这个新功能之后不久，Office 以前版本的许多用户抱怨在新的 UI 中找不到需要的操作按钮了。新 Office 用户没有使用过以前的用户界面，却在新的 UI 中得到了很好的体验，很容易找到以前版本的用户难以找到的操作按钮。

当然，目前功能区控件在许多应用程序中都很常见。在 Windows 8 中，功能区在随操作系统发布的工具中，例如 Windows 资源管理器、画图和写字板。

WPF 功能区控件在 System.Windows.Controls.Ribbon 名称空间中，需要引用程序集 system.windows.controls.ribbon。

图 36-2 显示了示例应用程序的功能区控件。在顶行上，标题左边是快速访问工具栏，第二行中最左边的项是应用程序菜单，其后是两个功能区标签 Home 和 Ribbon Controls。选择了 Home 标签，会显示两个组 Clipboard 和 Show。这两个组都包含一些按钮控件。

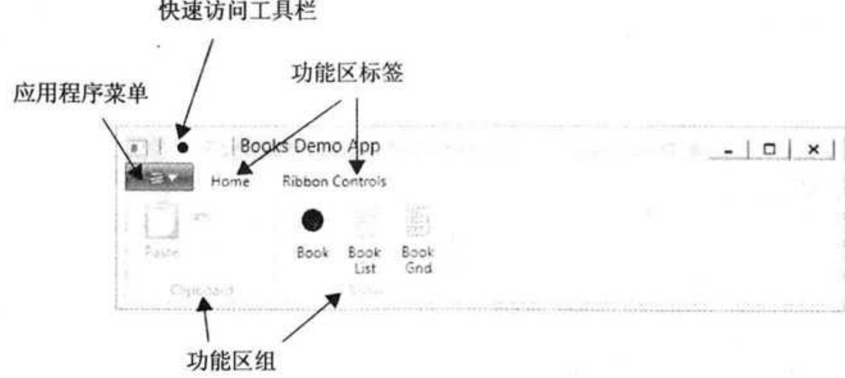


图 36-2

功能区控件在下面的代码段中定义。Ribbon 元素的第一个子元素由 QuickAccessToolBar 属性定义。这个工具栏包含两个引用了小图像的 RibbonButton 控件，这些按钮允许用户直接、快速、方便地执行操作：

```
<Ribbon DockPanel.Dock="Top">
  <Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar>
      <RibbonButton SmallImageSource="Images/one.png" />
```

```
      <RibbonButton SmallImageSource="Images/list.png" />
    </RibbonQuickAccessToolBar>
  </Ribbon.QuickAccessToolBar>
```

为了直接把快速访问工具栏中的这些按钮放在窗口的边框中，需要把基类改为 RibbonWindow，而不是 Window 类(代码文件 BooksDemo/MainWindow.xaml.cs)：

```
public partial class MainWindow : RibbonWindow
{
```

修改带后台代码的基类时，还需要修改 XAML 代码，以使用 RibbonWindow 元素：

```
<RibbonWindow x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Books Demo App" Height="400" Width="600">
```

应用程序菜单使用 ApplicationMenu 属性定义。应用程序菜单定义了两个菜单项——第一个显示一本书；第二个关闭应用程序：

```
<Ribbon.ApplicationMenu>
  <RibbonApplicationMenu SmallImageSource="Images/books.png" >
    <RibbonApplicationMenuItem Header="Show _Book" />
    <RibbonSeparator />
    <RibbonApplicationMenuItem Header="Exit" Command="Close" />
  </RibbonApplicationMenu>
</Ribbon.ApplicationMenu>
```

在应用程序菜单的后面，使用 RibbonTab 元素定义功能区控件的内容。该标签的标题用 Header 属性定义。RibbonTab 包含两个 RibbonGroup 元素，每个 RibbonGroup 元素都包含 RibbonButton 元素。在按钮中，可以设置 Label 来显示文本，设置 SmallImageSource 或 LargeImageSource 属性来显示图像：

```
<RibbonTab Header="Home">
  <RibbonGroup Header="Clipboard">
    <RibbonButton Command="Paste" Label="Paste"
        LargeImageSource="Images/paste.png" />
    <RibbonButton Command="Cut" SmallImageSource="Images/cut.png" />
    <RibbonButton Command="Copy" SmallImageSource="Images/copy.png" />
    <RibbonButton Command="Undo" LargeImageSource="Images/undo.png" />
  </RibbonGroup>
  <RibbonGroup Header="Show">
    <RibbonButton LargeImageSource="Images/one.png" Label="Book" />
    <RibbonButton LargeImageSource="Images/list.png" Label="Book List" />
    <RibbonButton LargeImageSource="Images/grid.png" Label="Book Grid" />
  </RibbonGroup>
</RibbonTab>
```

第二个 RibbonTab 元素仅用于演示可以在功能区控件中使用的不同控件，例如文本框、复选框、组合框、微调按钮和图库元素。图 36-3 打开了这个选项卡。

```
<RibbonTab Header="Ribbon Controls">
```

```
    <RibbonGroup Header="Sample">
      <RibbonButton Label="Button" />
      <RibbonCheckBox Label="Checkbox" />
      <RibbonComboBox Label="Combo1">
        <Label>One</Label>
        <Label>Two</Label>
      </RibbonComboBox>
      <RibbonTextBox>Text Box </RibbonTextBox>
      <RibbonSplitButton Label="Split Button">
        <RibbonMenuItem Header="One" />
        <RibbonMenuItem Header="Two" />
      </RibbonSplitButton>
      <RibbonComboBox Label="Combo2" IsEditable="False">
        <RibbonGallery SelectedValuePath="Content" MaxColumnCount="1"
            SelectedValue="Green">
          <RibbonGalleryCategory>
            <RibbonGalleryItem Content="Red" Foreground="Red" />
            <RibbonGalleryItem Content="Green" Foreground="Green" />
            <RibbonGalleryItem Content="Blue" Foreground="Blue" />
          </RibbonGalleryCategory>
        </RibbonGallery>
      </RibbonComboBox>
    </RibbonGroup>
  </RibbonTab>
```

图 36-3

功能区控件的更多信息看参见第 30 章，其中介绍了如何动态建立功能区项。

36.3 Commanding

Commanding 是一个 WPF 概念，它在动作源(如按钮)和执行动作的目标(如处理程序方法)之间创建松散耦合。这个概念基于 Gang of Four 中的命令模式。事件是紧密耦合的(至少在 XAML 2006 中是这样)。编译包含事件引用的 XAML 代码，要求代码隐藏已实现一个处理程序方法，且在编译期间可用。而对于命令，这个耦合是松散的。

要执行的动作用命令对象定义。命令实现 ICommand 接口。WPF 使用的命令类是 RoutedCommand 及其派生类 RoutedUICommand。RoutedUICommand 类定义一个 ICommand 接口未定义的附加 Text 属性，这个属性可以在用户界面中用作文本信息。ICommand 定义 Execute()和 CanExecute()方法，它们都在目标对象上执行。

命令源是调用命令的对象。命令源实现 ICommandSource 接口。这种命令源的例子有派生自 ButtonBase 的按钮类、Hyperlink 和 InputBinding。KeyBinding 和 MouseBinding 是派生自 InputBinding 的类。命令源有一个 Command 属性，其中可以指定实现 ICommand 接口的命令对象。在使用控件(如单击按钮)时，这会激活命令。

命令目标是实现了处理程序的对象，用于执行动作。通过命令绑定，定义一个映射，把处理程序映射到命令上。命令绑定指定在命令上调用哪个处理程序。命令绑定通过 UIElement 类中实现的 CommandBinding 属性来定义。因此派生自 UIElement 的每个类都有 CommandBinding 属性。这样，查找映射的处理程序就是一个层次化的过程。例如，在 StackPanel 中定义的一个按钮可以激活命令，而 StackPanel 位于 ListBox 中，ListBox 位于 Grid 中。处理程序在该树型结构的某个位置上通过命令绑定来指定，如 Window 的命令绑定。下面修改 BooksDemo 项目的实现方式，改为使用命令替代事件模型。

36.3.1 定义命令

.NET 提供了返回预定义命令的类。ApplicationCommands 类定义了静态属性 New、Open、Close、Print、Cut、Copy、Paste 等。这些属性返回可用于特殊目的的 RoutedUICommand 对象。提供了命令的其他类有 NavigationCommands 和 MediaCommands。NavigationCommands 提供了用于导航的常见命令，如 GoToPage、NextPage 和 PreviousPage，MediaCommand 提供的命令可用于运行媒体播放器，媒体播放器包含 Play、Pause、Stop、Rewind 和 Record 等按钮。

定义执行应用程序域的特定动作的自定义命令并不难。为此，创建一个 BooksCommands 类，它通过 ShowBook 和 ShowBookslist 属性返回一个 RoutedUICommand。也可以给命令指定一个输入手势，如 KeyGesture 或 MouseGesture。这里指定一个 KeyGesture，用 ALT 修饰符定义 B 键。因为输入手势是命令源，所以按 Alt+B 组合键会调用该命令(代码文件 BooksDemo/BooksCommands.cs):

```
public static class BooksCommands
{
  private static RoutedUICommand showBook;
  public static ICommand ShowBook
  {
    get
    {
      return showBook ?? (showBook = new RoutedUICommand("Show Book",
          "ShowBook", typeof(BooksCommands)));
    }
  }

  private static RoutedUICommand showBooksList;
  public static ICommand ShowBooksList
  {
    get
    {
      if (showBooksList == null)
      {
        showBooksList = new RoutedUICommand("Show Books", "ShowBooks",
            typeof(BooksCommands));
        showBook.InputGestures.Add(new KeyGesture(Key.B, ModifierKeys.Alt));
      }
```

```
        return showBooksList;
      }
    }
  }
```

36.3.2 定义命令源

每个实现 ICommandSource 接口的类都可以是命令源，如 Button 和 MenuItem。在前面创建的功能区控件中，把 Command 属性赋予几个 RibbonButton 元素，例如快速访问工具栏，如下所示(XAML 文件 BooksDemo/MainWindow.xaml)：

```
<Ribbon.QuickAccessToolBar>
  <RibbonQuickAccessToolBar>
    <RibbonButton SmallImageSource="Images/one.png"
       Command="local:BooksCommands.ShowBook" />
    <RibbonButton SmallImageSource="Images/list.png"
       Command="local:BooksCommands.ShowBooksList" />
  </RibbonQuickAccessToolBar>
</Ribbon.QuickAccessToolBar>
```

一些预定义的命令，例如 ApplicationCommands.Cut、Copy 和 Paste 也赋予 RibbonButton 元素的 Command 属性，对于预定义的命令，使用简写表示法：

```
<RibbonGroup Header="Clipboard">
  <RibbonButton Command="Paste"Label="Paste"
     LargeImageSource="Images/paste.png" />
  <RibbonButton Command="Cut"SmallImageSource="Images/cut.png" />
  <RibbonButton Command="Copy"SmallImageSource="Images/copy.png" />
  <RibbonButton Command="Undo"LargeImageSource="Images/undo.png" />
</RibbonGroup>
```

36.3.3 命令绑定

必须添加命令绑定，才能把它们连接到处理程序方法上。这里在 Window 元素中定义命令绑定，这样这些绑定就可用于窗口中的所有元素。执行 ApplicationCommands.Close 命令时，会调用 OnClose()方法。执行 BooksCommands.ShowBooks 命令时，会调用 OnShowBooks()方法：

```
<Window.CommandBindings>
  <CommandBinding Command="Close" Executed="OnClose" />
  <CommandBinding Command="local:BooksCommands.ShowBooksList"
     Executed="OnShowBooksList" />
</Window.CommandBindings>
```

通过命令绑定，还可以指定 CanExecute 属性，在该属性中，会调用一个方法来验证命令是否可用。例如，如果文件没有变化，ApplicationCommands.Save 命令就是不可用的。

需要用两个参数定义处理程序，它们分别是 sender 的 object 和可以从中访问命令信息的 ExecutedRoutedEventArgs(代码文件 BooksDemo/MainWindow.xaml.cs)：

```
private void OnClose(object sender, ExecutedRoutedEventArgs e)
{
  Application.Current.Shutdown();
}
```

还可以通过命令传递参数。为此，可以通过命令源(如 MenuItem)指定 CommandParameter 属性。使用 ExecutedRoutedEventArgs 的 Parameter 属性可以访问该参数。

命令绑定也可以通过控件来定义。TextBox 控件给 ApplicationCommands.Cut、ApplicationCommands.Copy、ApplicationCommands.Paste 和 ApplicationCommands.Undo 定义了绑定。这样，就只需指定命令源，并使用 TextBox 控件中的已有功能。

36.4 数据绑定

与以前的技术相比，WPF 数据绑定向前迈了一大步。数据绑定把数据从.NET 对象传递给 UI，或从 UI 传递给.NET 对象。简单对象可以绑定到 UI 元素、对象列表和 XAML 元素上。在 WPF 数据绑定中，目标可以是 WPF 元素的任意依赖属性，CLR 对象的每个属性都可以是绑定源。因为 WPF 元素作为.NET 类实现，所以每个 WPF 元素也可以用作绑定源。图 36-4 显示了绑定源和绑定目标之间的连接。Binding 对象定义了该连接。

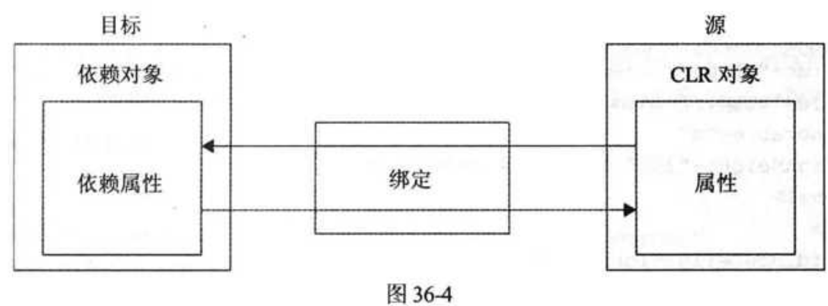


图 36-4

Binding 对象支持源与目标之间的几种绑定模式。绑定可以是单向的，即从源信息指向目标，但如果用户在用户界面上修改了该信息，则源不会更新。要更新源，需要双向绑定。

表 36-1 列出了绑定模式及其要求。

表 36-1

绑定模式	说明
一次性	绑定从源指向目标，且仅在应用程序启动时，或数据环境改变时绑定一次。通过这种模式可以获得数据的快照
单向	绑定从源指向目标。这对于只读数据很有用，因为它不能从用户界面中修改数据。要更新用户界面，源必须实现 INotifyPropertyChanged 接口
双向	在双向绑定中，用户可以从 UI 中修改数据。绑定是双向的——从源指向目标，从目标指向源。源对象需要实现读写属性，才能把改动的内容从 UI 更新到源对象上
指向源的单向	采用这种绑定模式，如果目标属性改变，源对象也会更新

除了绑定模式之外，WPF 数据绑定还涉及许多方面。本节详细介绍与 XAML 元素、简单的.NET 对象和列表的绑定。通过更改通知，可以使用绑定对象中的更改更新 UI。本节还会讨论从对象数据提供程序中获取数据和直接从代码中获取数据。多绑定和优先绑定说明了与默认绑定不同的绑定可能性，本节也将论述如何动态地选择数据模板，以及绑定值的验证。

下面从 BooksDemo 示例应用程序开始。

36.4.1 BooksDemo 应用程序内容

上一节在 BooksDemo 应用程序中定义了一个功能区和命令，现在添加内容。修改 XAML 文件 MainWindow.xaml，并添加 ListBox、Hyperlink 和 TabControl(XAML 文件 BooksDemo/MainWindow.xaml)。

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <Hyperlink Click="OnShowBook">Show Book</Hyperlink>
</ListBox>
<TabControl Margin="5" x:Name="tabControl1">
</TabControl>
```

现在添加一个 WPF 用户控件 BookUC。这个用户控件包含一个 DockPanel、一个几行几列的 Grid、一个 Label 和多个 TextBox 控件(XAML 文件 BooksDemo/BookUC.xaml)：

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <DockPanel>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
          HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
          Margin="10,0,5,0" HorizontalAlignment="Left"
          VerticalAlignment="Center" />
      <Label Content="Isbn" Grid.Row="2" Grid.Column="0"
          Margin="10,0,5,0" HorizontalAlignment="Left"
          VerticalAlignment="Center" />
      <TextBox Grid.Row="0" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="1" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="2" Grid.Column="1" Margin="5" />
      <StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
        <Button Content="Show Book" Margin="5" Click="OnShowBook" />
```

```
      </StackPanel>
    </Grid>
  </DockPanel>
</UserControl>
```

在 MainWindow.xaml.cs 的 OnShowBook 处理程序中，新建 BookUC 用户控件的一个实例，给 TabControl 添加一个新的 TabItem。接着修改 TabControl 的 SelectedIndex 属性，以打开新的选项卡(代码文件 BooksDemo/MainWindow.xaml.cs)：

```
private void OnShowBook(object sender, ExecutedRoutedEventArgs e)
{
  var bookUI = new BookUC();
  this.tabControl1.SelectedIndex = this.tabControl1.Items.Add(
      new TabItem { Header = "Book", Content = bookUI });
}
```

构建项目后，就可以启动应用程序，单击超链接，打开 TabControl 中的用户控件。

36.4.2 用 XAML 绑定

WPF 元素不仅是数据绑定的目标，它还可以是绑定的源。可以把一个 WPF 元素的源属性绑定到另一个 WPF 元素的目标属性上。

在下面的代码示例中，数据绑定用于通过一个滑块重置用户控件中控件的大小。给用户控件 BookUC 添加一个 StackPanel 控件，该 StackPanel 控件包含一个标签和一个滑块控件。滑块控件定义了 Minimum 和 Maximum 值，以指定缩放比例，把其初始值 1 赋予 Value 属性(XAML 文件 BooksDemo/BookUC.xaml)：

```
<DockPanel>
  <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
      HorizontalAlignment="Right">
    <Label Content="Resize" />
    <Slider x:Name="slider1" Value="1" Minimum="0.4" Maximum="3"
        Width="150" HorizontalAlignment="Right" />
  </StackPanel>
```

设置 Grid 控件的 LayoutTransform 属性，并添加一个 ScaleTransform 元素。通过 ScaleTransform 元素，对 ScaleX 和 ScaleY 属性进行数据绑定。这两个属性都用 Binding 标记扩展来设置。在 Binding 标记扩展中，把 ElementName 设置为 slider1，以引用前面创建的滑块控件。把 Path 设置为 Value，从 Value 属性中获取滑块的值。

```
<Grid>
  <Grid.LayoutTransform>
    <ScaleTransform x:Name="scale1"
        ScaleX="{Binding Path=Value, ElementName=slider1}"
        ScaleY="{Binding Path=Value, ElementName=slider1}" />
  </Grid.LayoutTransform>
```

运行应用程序时，可以移动滑块，从而重置 Grid 中的控件，如图 36-5 和图 36-6 所示。

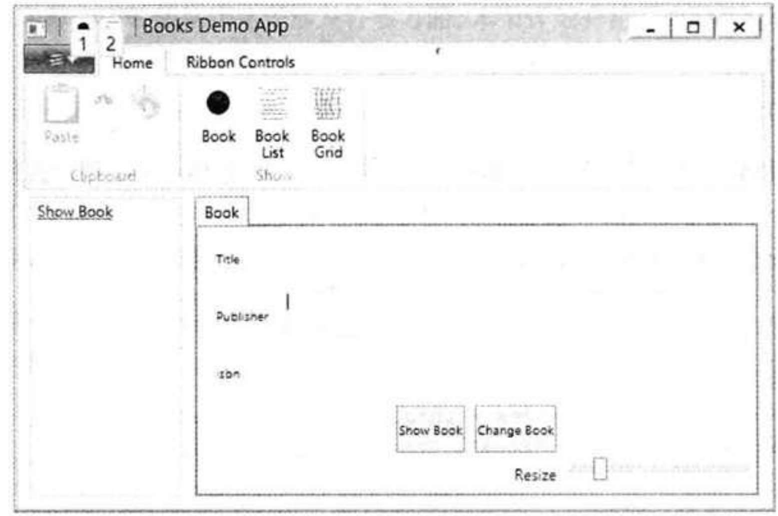


图 36-5

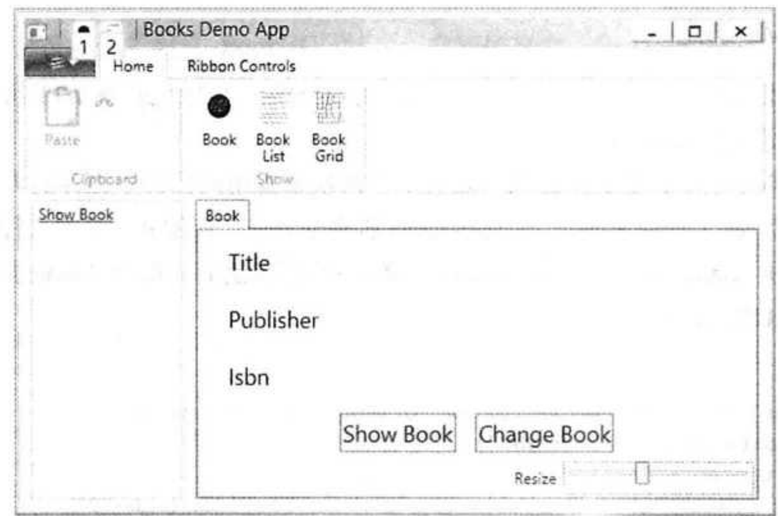


图 36-6

除了用 XAML 代码定义绑定信息之外，如上述代码使用 Binding 元数据扩展来定义，还可以使用代码隐藏。在代码隐藏中，必须新建一个 Binding 对象，并设置 Path 和 Source 属性。必须把 Source 属性设置为源对象，这里是 WPF 对象 slider1。把 Path 属性设置为一个 PropertyPath 实例，它用源对象的 Value 属性名进行初始化。对于派生自 FrameworkElement 的控件，可以调用 SetBinding()方法来定义绑定。但是，ScaleTransform 不派生自 FrameworkElement，而派生自 Freezable 基类。使用辅助类 BindingOperations 可以绑定这类控件。BindingOperations 类的 SetBinding()方法需要一个 DependencyObject，在本例中是 ScaleTransform 实例。对于第二和第三个参数，SetBinding()方法还需要绑定目标的 dependency 属性和 Binding 对象。

```
var binding = new Binding
{
  Path = new PropertyPath("Value"),
  Source = slider1
};
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleXProperty,
```

```
    binding);
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleYProperty,
    binding);
```

派生自 DependencyObject 的所有类都可以有依赖属性。依赖属性参见第 29 章。

使用 Binding 类，可以配置许多绑定选项，如表 36-2 所示。

表 36-2

Binding 类的成员	说　明
Source	使用 Source 属性，可以定义数据绑定的源对象
RelativeSource	使用 RelativeSource 属性，可以指定与目标对象相关的源对象。当错误来源于同一个控件时，它对于显示错误消息很有用
ElementName	如果源对象是一个 WPF 元素，就可以用 ElementName 属性指定源对象
Path	使用 Path 属性，可以指定到源对象的路径。它可以是源对象的属性，但也支持子元素的索引器和属性
XPath	使用 XML 数据源时，可以定义一个 XPath 查询表达式，来获得要绑定的数据
Mode	模式定义了绑定的方向。Mode 属性是 BindingMode 类型。BindingMode 是一个枚举，其值如下：Default、OneTime、OneWay、TwoWay、OneWayToSource。默认模式依赖于目标：对于文本框，默认是双向绑定；对于只读的标签，默认为单向。OneTime 表示数据仅从源中加载一次；OneWay 将对源对象的修改更新到目标对象中。TwoWay 绑定表示，对 WPF 元素的修改可以写回源对象中。OneWayToSource 表示，从不读取数据，但总是从目标对象写入源对象中
Converter	使用 Converter 属性，可以指定一个转换器类，该转换器类来回转换 UI 的数据。转换器类必须实现 IvalueConverter 接口，它定义了 Convert()和 ConvertBack()方法。使用 ConverterParameter 属性可以给转换方法传递参数。转换器区分区域性，区域性可以用 ConverterCultrue 属性设置
FallbackValue	使用 FailbackValue 属性，可以定义一个在绑定没有返回值时使用的默认值
ValidationRules	使用 ValidationRules 属性，可以定义一个 ValidationRule 对象集合，在从 WPF 目标元素更新源对象之前检查该集合。ExceptionValidationRule 类派生自 ValidationRule 类，负责检查异常
Delay	这个属性是 WPF 4.5 新增的，它可以指定更新绑定源之前等待的时间。在开始验证之前，希望给用户一些时间来输入更多的字符时，就可以使用这个属性

36.4.3 简单对象的绑定

要绑定 CLR 对象，只需要使用.NET 类定义属性，如下面的例子就使用 Book 类定义了 Title、Publisher、Isbn 和 Authors 属性。这个类在 BooksDemo 项目的 Data 文件夹中(代码文件 BooksDemo/Data/Book.cs)。

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

```
namespace Wrox.ProCSharp.WPF.Data
{
  public class Book
  {
    public Book(string title, string publisher, string isbn,
                params string[] authors)
    {
      this.Title = title;
      this.Publisher = publisher;
      this.Isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book()
      : this("unknown", "unknown", "unknown")
    {
    }
    public string Title { get; set; }
    public string Publisher { get; set; }
    public string Isbn { get; set; }

    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
      get
      {
        return authors.ToArray();
      }
    }

    public override string ToString()
    {
      return Title;
    }
  }
}
```

在用户控件 BookUC 的 XAML 代码中，定义了几个标签和文本框控件，以显示图书信息。使用 Binding 标记扩展，将文本框控件绑定到 Book 类的属性上。在 Binding 标记扩展中，仅定义了 Path 属性，将它绑定到 Book 类的属性上。不需要定义源对象，因为通过指定 DataContext 来定义源对象，如下面的代码隐藏所示。对于 TextBox 元素，模式定义为其默认值，即双向绑定(XAML 文件 BooksDemo//BookUC.xaml)：

```
<TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
```

在代码隐藏中定义一个新的 Book 对象，并将其赋予用户控件的 DataContext 属性。DataContext 是一个依赖属性，它用基类 FrameworkElement 定义。指定用户控件的 DataContext 属性表示，用户控件中的每个元素都默认绑定到同一个数据上下文上(代码文件 BooksDemo/MainWindow.xaml.cs)。

```
private void OnShowBook(object sender, ExecutedRoutedEventArgs e)
{
  var bookUI = new BookUC();
  bookUI.DataContext = new Book
  {
    Title = "Professional C# 4 and .NET 4",
    Publisher = "Wrox Press",
    Isbn = "978-0-470-50225-9"
  };
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header = "Book", Content = bookUI });
}
```

启动应用程序后，就会看到图 36-7 所示的绑定数据。

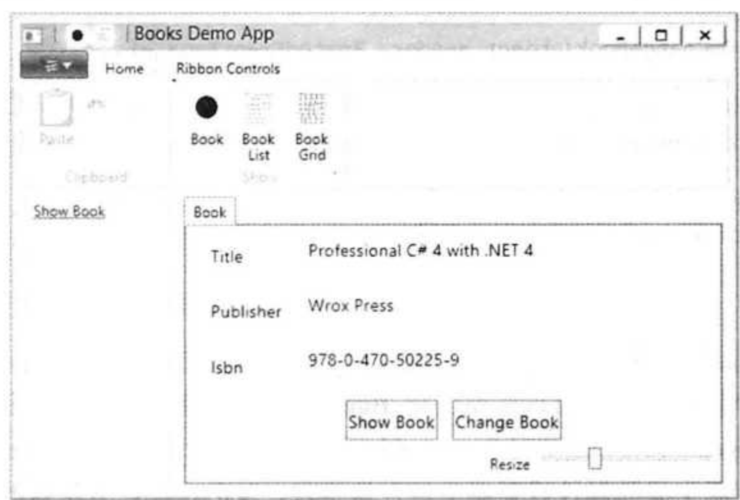


图 36-7

为了实现双向绑定(对输入的 WPF 元素的修改反映到 CLR 对象中)，实现了用户控件中按钮的 Click 事件处理程序——OnShowBook()方法。在实现时，会弹出一个消息框，显示 book1 对象的当前标题和 ISBN 号。图 36-8 显示了在运行过程中修改该输入后消息框的输出(代码文件 BooksDemo/BookUC.xaml.cs)。

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
  Book theBook = this.DataContext as Book;
  if (theBook != null)
    MessageBox.Show(theBook.Title, theBook.Isbn);
}
```

图 36-8

36.4.4 更改通知

使用当前的双向绑定，可以读写对象中的数据。但如果数据不由用户修改，而是直接在代码中修改，用户界面就接收不到更改信息。只要在用户控件中添加一个按钮，并实现 Click 事件处理程序 OnChangeBook，就可以验证这一点(XAML 文件 BooksDemo/BookUC.xaml)。

```
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2"
            Orientation="Horizontal" HorizontalAlignment="Center">
  <Button Content="Show Book" Margin="5" Click="OnShowBook" />
  <Button Content="Change Book" Margin="5" Click="OnChangeBook" />
</StackPanel>
```

在处理程序的实现代码中，数据上下文中的图书变化了，但用户界面没有显示这个变化(代码文件 BooksDemo/BookUC.xaml.cs)。

```
private void OnChangeBook(object sender, RoutedEventArgs e)
{
  Book theBook = this.DataContext as Book;
  if (theBook != null)
  {
    theBook.Title = "Professional C# 5";
    theBook.Isbn = "978-0-470-31442-5";
  }
}
```

为了把更改信息传递给用户界面，实体类必须实现 INotifyPropertyChanged 接口。这里不是实现每个需要这个接口的类，而只需要创建 BindableObject 抽象基类。这个基类实现了接口 INotifyPropertyChanged。该接口定义了 PropertyChanged 事件，该事件在 OnPropertyChanged 方法中触发。为了便于在派生类的属性设置器中触发该事件，SetProperty 方法修改了该属性，调用 OnPropertyChanged 方法，来触发该事件。这个方法在 C#中通过 CallerMemberName 属性来使用调用者信息。propertyName 参数通过这个属性定义为可选参数，C#编译器就会通过这个参数传递属性名，所以不需要在代码中添加硬编码字符串(代码文件 BooksDemo/Data/BindableObject.cs)：

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Wrox.ProCSharp.WPF.Data
{
  public abstract class BindableObject : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
      var propertyChanged = PropertyChanged;
      if (propertyChanged != null)
      {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
      }
    }
```

```
      protected void SetProperty<T>(ref T item, T value,
        [CallerMemberName] string propertyName = null)
      {
        if (!EqualityComparer<T>.Default.Equals(item, value))
        {
          item = value;
          OnPropertyChanged(propertyName);
        }
      }
    }
  }
```

调用者信息参见第 16 章。

类 Book 现在改为派生自基类 BindableObject，来继承 INotifyPropertyChanged 接口的实现代码。属性设置器改为调用 SetProperty 方法，如下所示(代码文件 BooksDemo/Data/Book.cs)：

```
using System.ComponentModel;
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
  public class Book : BindableObject
  {
    public Book(string title, string publisher, string isbn,
                params string[] authors)
    {
      this.title = title;
      this.publisher = publisher;
      this.isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book()
      : this("unknown", "unknown", "unknown")
    {
    }

    private string title;
    public string Title {
      get
      {
        return title;
      }
      set
      {
        SetProperty(ref title, value);
      }
    }

    private string publisher;
    public string Publisher
```

```
        {
          get
          {
            return publisher;
          }
          set
          {
            SetProperty(ref publisher, value);
          }
        }
        private string isbn;
        public string Isbn
        {
          get
          {
            return isbn;
          }
          set
          {
            SetProperty(ref isbn, value);
          }
        }

        private readonly List<string> authors = new List<string>();
        public string[] Authors
        {
          get
          {
            return authors.ToArray();
          }
        }

        public override string ToString()
        {
          return this.title;
        }
      }
    }
```

进行了这个修改后，就可以再次启动应用程序，以验证用户界面从事件处理程序中接收到更改信息。

36.4.5 对象数据提供程序

除了在代码隐藏中实例化对象之外，还可以用 XAML 定义对象实例。为了在 XAML 中引用代码隐藏中的类，必须引用在 XML 根元素中声明的名称空间。XML 属性 xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"将.NET 名称空间 Wrox.ProCSharp.WPF 赋予 XML 名称空间别名 local。

现在，在 DockPanel 资源中用 Book 元素定义 Book 类的一个对象。给 XML 属性 Title、Publisher 和 Isbn 赋值，就可以设置 Book 类的属性值。x:Key="theBook"定义了资源的标识符，以便引用 book 对象(XAML 文件 BooksDemo/BookUC.xaml)：

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <DockPanel>
    <DockPanel.Resources>
      <local:Book x:Key="theBook" Title="Professional C# 4 and .NET 4"
          Publisher="Wrox Press" Isbn="978-0-470-50225-9" />
    </DockPanel.Resources>
```

如果要引用的.NET 名称空间在另一个程序集中，就必须把该程序集添加到 XML 声明中。

```
xmlsn:sys="clr-namespace:System;assembly=mscorlib"
```

在 TextBox 元素中，用 Binding 标记扩展定义 Source，Binding 标记扩展引用 theBook 资源。

```
<TextBox Text="{Binding Path=Title, Source={StaticResource theBook}}"
   Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher, Source={StaticResource theBook}}"
   Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn, Source={StaticResource theBook}}"
   Grid.Row="2" Grid.Column="1" Margin="5" />
```

因为所有 TextBox 元素都包含在同一个控件中，所以可以用父控件指定 DataContext 属性，用 TextBox 绑定元素设置 Path 属性。因为 Path 属性是默认的，所以也可以在下面的代码中删除 Binding 标记扩展：

```
<Grid x:Name="grid1" DataContext="{StaticResource theBook}">
  <!-- ... -->
  <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
     Margin="5" />
  <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
     Margin="5" />
  <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
     Margin="5" />
```

除了直接在 XAML 代码中定义对象实例外，还可以定义一个对象数据提供程序，该提供程序引用类，以调用方法。为了使用 ObjectDataProvider，最好创建一个返回要显示的对象的工厂类，如下面的 BookFactory 类所示(代码文件 BooksDemo/Data/BookFactory.cs)：

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
  public class BookFactory
```

```
{
  private List<Book> books = new List<Book>();

  public BookFactory()
  {
    books.Add(new Book
    {
      Title = "Professional C# 4 and .NET 4",
      Publisher = "Wrox Press",
      Isbn = "978-0-470-50225-9"
    });
  }

  public Book GetTheBook()
  {
    return books[0];
  }
}
}
```

ObjectDataProvider 元素可以在资源部分中定义。XML 特性 ObjectType 定义了类的名称，MethodName 指定了获得 book 对象要调用的方法的名称(XAML 文件 BooksDemo/BookUC.xaml)：

```
<DockPanel.Resources>
  <ObjectDataProvider x:Key="theBook" ObjectType="local:BookFactory"
     MethodName="GetTheBook" />
</DockPanel.Resources>
```

用 ObjectDataProvider 类指定的属性如表 36-3 所示。

表 36-3

ObjectDataProvider	说　明
ObjectType	ObjectType 属性定义了要创建的实例类型
ConstrutorParameters	使用 ConstructorParameters 集合可以在类中添加创建实例的参数
MethodName	MethodName 属性定义了由对象数据提供程序调用的方法的名称
MethodParameters	使用 MethodParameters 属性，可以给通过 MethodName 属性定义的方法指定参数
ObjectInstance	使用 ObjectInstance 属性，可以获取和设置由 ObjectDataProvider 类使用的对象。例如，可以用编程方式指定已有的对象，而不是定义 ObjectType 以便用 ObjectDataProvider 实例化一个对象
Data	使用 Data 属性，可以访问用于数据绑定的底层对象。如果定义了 MethodName，则使用 Data 属性，可以访问从指定的方法返回的对象

36.4.6　列表绑定

绑定到列表上比绑定到简单对象上更常见，这两种绑定非常类似。可以从代码隐藏中将完整的列表赋予 DataContext，也可以使用 ObjectDataProvider 访问一个对象工厂，以返回一个列表。对于支持绑定到列表上的元素(如列表框)，会绑定整个列表。对于只支持绑定一个对象上的元素(如文本

框)，只绑定当前项。

使用 BookFactory 类，现在返回一个 Book 对象列表(代码文件 BooksDemo/Data/BookFactory.cs)：

```
public class BookFactory
{
  private List<Book> books = new List<Book>();

  publicBookFactory()
  {
    books.Add(new Book("Professional C# 4 with .NET 4", "Wrox Press",
                       "978-0-470-50225-9", "Christian Nagel", "Bill Evjen",
                       "Jay Glynn", "Karli Watson", "Morgan Skinner"));
    books.Add(new Book("Professional C# 2008", "Wrox Press",
                       "978-0-470-19137-8", "Christian Nagel", "Bill Evjen",
                       "Jay Glynn", "Karli Watson", "Morgan Skinner"));
    books.Add(new Book("Beginning Visual C# 2010", "Wrox Press",
                       "978-0-470-50226-6", "Karli Watson", "Christian Nagel",
                       "Jacob Hammer Pedersen", "Jon D. Reid",
                       "Morgan Skinner", "Eric White"));
    books.Add(new Book("Windows 7 Secrets", "Wiley", "978-0-470-50841-1",
                       "Paul Thurrott", "Rafael Rivera"));
    books.Add(new Book("C# 2008 for Dummies", "For Dummies",
                       "978-0-470-19109-5", "Stephen Randy Davis",
                       "Chuck Sphar"));
  }

  public IEnumerable<Book> GetBooks()
  {
    return books;
  }
}
```

要使用列表，应新建一个 BooksUC 用户控件。这个控件的 XAML 代码包含的标签和文本框控件可以显示一本书的值，它包含的列表框控件可以显示一个图书列表。ObjectDataProvider 调用 BookFactory 的 GetBooks()方法，这个提供程序用于指定 DockPanel 的 DataContext。DockPanel 把绑定的列表框和文本框作为其子控件(XAML 文件 BooksDemo/BooksUC.xaml)。

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <UserControl.Resources>
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                        MethodName="GetBooks" />
  </UserControl.Resources>
  <DockPanel DataContext="{StaticResource books}">
    <ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
        MinWidth="120" />
    <Grid>
```

```
        <Grid.RowDefinitions>
          <RowDefinition />
          <RowDefinition />
          <RowDefinition />
          <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto" />
          <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
            HorizontalAlignment="Left" VerticalAlignment="Center" />
        <Label Content="Publisher" Grid.Row="1" Grid.Column="0" Margin="10,0,5,0"
            HorizontalAlignment="Left" VerticalAlignment="Center" />
        <Label Content="Isbn" Grid.Row="2" Grid.Column="0" Margin="10,0,5,0"
            HorizontalAlignment="Left" VerticalAlignment="Center" />
        <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
        <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
            Margin="5" />
        <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
      </Grid>
    </DockPanel>
  </UserControl>
```

新的用户控件通过给 MainWindow.xaml 添加一个 Hyperlink 来启动。它使用 Command 属性来指定 ShowBooks 命令。该命令绑定必须也指定为调用 OnShowBooksList 事件处理程序(XAML 文件 BooksDemo/BooksUC.xaml)：

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <ListBoxItem>
    <Hyperlink Command="local:BooksCommands.ShowBook">Show Book</Hyperlink>
  </ListBoxItem>
  <ListBoxItem>
    <Hyperlink Command="local:ShowCommands.ShowBooksList">
        Show Books List</Hyperlink>
  </ListBoxItem>
</ListBox>
```

事件处理程序的实现代码给 TabControl 添加一个新的 TabItem 控件，把 Content 指定为用户控件 BooksUC，将 TabControl 的选择设置为新建的 TabItem(代码文件 BooksDemo/BooksUC.xaml.cs)：

```
private void OnShowBooks(object sender, ExecutedRoutedEventArgs e)
{
  var booksUI = new BooksUC();
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header="Books List", Content=booksUI});
}
```

因为 DockPanel 将 Book 数组赋予 DataContext，列表框放在 DockPanel 中，所以列表框会用默认模板显示所有图书，如图 36-9 所示。

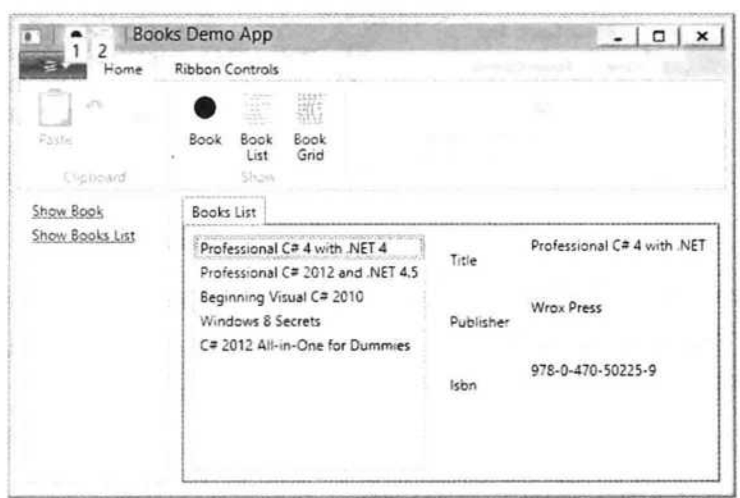


图 36-9

为了使列表框有更灵活的布局，必须定义一个模板，就像前一章为列表框定义样式那样。列表框的 ItemTemplate 定义了一个带标签元素的 DataTemplate。标签的内容绑定到 Title 上。列表项模板重复应用于列表中的每一项，当然也可以把列表项模板添加到资源内部的样式中(XAML 文件 BooksDemo/BooksUC.xaml)。

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
    MinWidth="120">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

36.4.7　主从绑定

除了显示列表中的所有元素之外，还应能显示选中项的详细信息。这不需要做太多的工作。标签和文本框控件已经定义好了，当前它们只显示列表中的第一个元素。

这里必须对列表框进行一个重要的修改。在默认情况下，把标签绑定到列表的第一个元素上。设置列表框的属性 IsSynchronizedWithCurrentItem = “True”，就会把列表框的选项设置为当前项(XAML 文件 BooksDemo/BooksUC.xaml)。

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
        MinWidth="120" IsSynchronizedWithCurrentItem="True">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

在图 36-10 中显示了结果：选中的项显示在详细信息标签中。

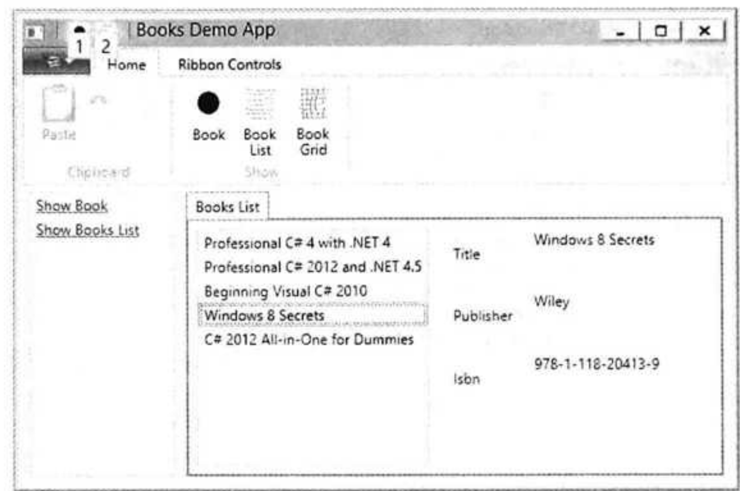


图 36-10

36.4.8　多绑定

Binding 是可用于数据绑定的类之一。BindingBase 是所有绑定的抽象基类，有不同的具体实现方式。除了 Binding 之外，还有 MuitlBinding 和 PriorityBinding。MuitlBinding 允许把一个 WPF 元素绑定到多个源上。例如，Person 类有 LastName 和 FirstName 属性，把这两个属性绑定到一个 WPF 元素上会比较有趣(代码文件 MultiBindingDemo/Person.cs)：

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

对于 MuitlBinding，标记扩展不可用——因此必须用 XAML 元素语法来指定绑定。MuitlBinding 的子元素是指定绑定到各种属性上的 Binding 元素。这里使用了 LastName 和 FirstName 属性。数据上下文用 Grid 元素设置，以便引用 person1 资源。

为了把属性连接在一起，MuitlBinding 使用一个 Converter 把多个值转换为一个。这个转换器使用一个参数，并可以根据参数进行不同的转换(XAML 文件 MultiBindingDemo/MainWindow.xaml)：

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="MainWindow" Height="240" Width="500">
<Window.Resources>
  <local:Person x:Key="person1" FirstName="Tom" LastName="Turbo" />
  <local:PersonNameConverter x:Key="personNameConverter" />
</Window.Resources>
<Grid DataContext="{StaticResource person1}">
  <TextBox>
    <TextBox.Text>
      <MultiBinding Converter="{StaticResource personNameConverter}" >
        <MultiBinding.ConverterParameter>
            <system:String>FirstLast</system:String>
```

```
          </MultiBinding.ConverterParameter>
          <Binding Path="FirstName" />
          <Binding Path="LastName" />
        </MultiBinding>
      </TextBox.Text>
    </TextBox>
  </Grid>
</Window>
```

多值转换器实现 IMuitlValueConverter 接口。这个接口定义了两个方法：Convert 和 ConvertBack()。Convert()方法通过第一个参数从数据源中接收多个值，并把一个值返回给目标。在实现代码中，根据参数的值是 FirstName 还是 LastName，生成不同的结果(代码文件 MultiBindingDemo/MainWindow.xaml.cs)：

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF
{
  public class PersonNameConverter : IMultiValueConverter
  {
    public object Convert(object[] values, Type targetType, object parameter,
                          CultureInfo culture)
    {
      switch (parameter as string)
      {
        case"FirstLast":
          return values[0] +"" + values[1];
        case"LastFirst":
          return values[1] + ", " + values[0];
        default:
          throw new ArgumentException(String.Format(
               "invalid argument {0}", parameter));
      }
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
                                object parameter, CultureInfo culture)
    {
      throw new NotSupportedException();
    }
  }
}
```

在这个简单的情形中，只把一些字符串与 MultiBinding 合并起来，并不需要实现 IMultiValueConverter，定义一个格式字符串就足够了，如下面的 XAML 代码段所示。用 MultiBinding 定义的格式字符串首先需要一个{}前缀，在 XAML 中，花括号通常定义一个标记表达式。把{}用作前缀会转义这个符号，不定义标记表达式，而是表示它后面的是一个通常的字符串。该示例指定，两个 Binding 元素用一个逗号和一个空白分隔开(XAML 文件 MultiBindingDemo/MainWindow.xaml)：

```
<TextBox>
    <TextBox.Text>
      <MultiBinding StringFormat="{}{0}, {1}">
        <Binding Path="LastName" />
        <Binding Path="FirstName" />
      </MultiBinding>
    </TextBox.Text>
  </TextBox>
```

36.4.9 优先绑定

PriorityBinding 非常便于绑定还不可用的数据。如果通过 PriorityBinding 需要一定的时间才能得到结果，就可以通知用户目前的进度，让用户知道需要等待。

为了说明优先绑定，使用 PriorityBindingDemo 项目来创建 Data 类。调用 Thread.Sleep()方法，来模拟访问 ProcessSomedata 属性需要一些时间(代码文件 PriorityBindingDemo/Data.cs)：

```
public class Data
{
  public string ProcessSomeData
  {
    get
    {
      Thread.Sleep(8000);
      return "the final result is here";
    }
  }
}
```

Information 类给用户提供信息。从 Info2 属性返回信息 5 秒后，立刻返回 Info1 属性的信息。在实际的实现代码中，这个类可以与处理数据的类关联起来，从而给用户提供估计的时间范围(代码文件 PriorityBindingDemo/Information.cs)：

```
public class Information
{
  public string Info1
  {
    get
    {
     return "please wait...";
    }
  }
  public string Info2
  {
    get
    {
      Thread.Sleep(5000);
      return "please wait a little more";
    }
  }
}
```

在 MainWindow.xaml 文件中，在 Window 的资源内部引用并初始化 Data 类和 Information 类(XAML 文件 PriorityBindingDemo/MainWindow.xaml)：

```
<Window.Resources>
    <local:Data x:Key="data1" />
    <local:Information x:Key="info" />
</Window.Resources>
```

PriorityBinding 在 Label 的 Content 属性中替代了正常的绑定。PriorityBinding 包含多个 Binding 元素，其中除了最后一个元素之外，其他元素都把 IsAsyncs 属性设置为 True。因此，如果第一个绑定表达式的结果不能立即使用，绑定进程就选择下一个绑定。第一个绑定引用 Data 类的 ProcessSomedata 属性，这需要一些时间。所以，选择下一个绑定，并引用 Information 类的 Info2 属性。Info2 属性没有立刻返回结果，而且因为设置了 IsAsyncs 属性，所以绑定进程不等待，而是继续处理下一个绑定。最后一个绑定使用 Info1 属性。如果它没有立刻返回结果，就要等待，因为它的 IsAsyncs 属性设置为默认值 False。

```
<Label>
  <Label.Content>
    <PriorityBinding>
      <Binding Path="ProcessSomeData" Source="{StaticResource data1}"
           IsAsync="True" />
      <Binding Path="Info2" Source="{StaticResource info}"
           IsAsync="True" />
      <Binding Path="Info1" Source="{StaticResource info}"
           IsAsync="False" />
    </PriorityBinding>
  </Label.Content>
</Label>
```

启动应用程序，会在用户界面中看到消息"please wait"...。几秒后从 Info2 属性返回结果 please wait a little more。它替换了 Info1 的输出。最后，ProcessSomedata 的结果再次替代了 Info2 的结果。

36.4.10　值的转换

返回到 BooksDemo 应用程序中。图书的作者还没有显示在用户界面中。如果将 Authors 属性绑定到标签元素上，就要调用 Array 类的 ToString()方法，它只返回类型的名称。一种解决方法是将 Authors 属性绑定到一个列表框上。对于该列表框，可以定义一个模板，以显示特定的视图。另一种解决方法是将 Authors 属性返回的字符串数组转换为一个字符串，再将该字符串用于绑定。

StringArrayConverter 类可以将字符串数组转换为字符串。WPF 转换器类必须实现 System.Windows.Data 名称空间中的 IValueConverter 接口。这个接口定义了 Convert()和 ConvertBack()方法。在 StringArrayConverter 类中，Convert()方法会通过 String.Join()方法把 value 变量中的字符串数组转换为字符串。从 Convert()方法接收的 parameter 变量中提取 Join()方法的分隔符参数(代码文件 BooksDemo/Utilities/Information.cs)。

String 类的方法的更多信息参见第 9 章。

```
using System;
using System.Diagnostics.Contracts;
using System.Globalization;
using System.Windows.Data;
```

```
namespace Wrox.ProCSharp.WPF.Utilities

{
  [ValueConversion(typeof(string[]), typeof(string))]
  class StringArrayConverter : IValueConverter
  {
    public object Convert(object value, Type targetType, object parameter,
                          CultureInfo culture)
    {
      if (value == null) return null;

      string[] stringCollection = (string[])value;
      string separator = parameter == null;

      return String.Join(separator, stringCollection);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                              CultureInfo culture)
    {
      throw new NotImplementedException();
    }
  }
}
```

在 XAML 代码中，StringArrayConverter 类可以声明为一个资源，以便从 Binding 标记扩展中引用它(XAML 文件 BooksDemo/BooksUC.xaml)：

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    xmlns:utils="clr-namespace:Wrox.ProCSharp.WPF.Utilities"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <UserControl.Resources>
    <utils:StringArrayConverter x:Key="stringArrayConverter" />
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                        MethodName="GetBooks" />
  </UserControl.Resources>
  <!-- -->
```

为了输出多行结果，声明一个 TextBlock 元素，将其 TextWrapping 属性设置为 Wrap，以便可以显示多个作者。在 Binding 标记扩展中，将 Path 设置为 Authors，它定义为一个返回字符串数组的属性。Converter 属性指定字符串数组从 stringArrayConverter 资源中转换。转换器实现的 Convert()方法接收 ConverterParameter=', '作为输入来分隔多个作者。

```
<TextBlock Text="{Binding Authors,
          Converter={StaticResource stringArrayConverter},
          ConverterParameter=', '}"
          Grid.Row="3" Grid.Column="1" Margin="5"
               VerticalAlignment="Center" TextWrapping="Wrap" />
```

图 36-11 显示了图书的详细信息，包括作者。

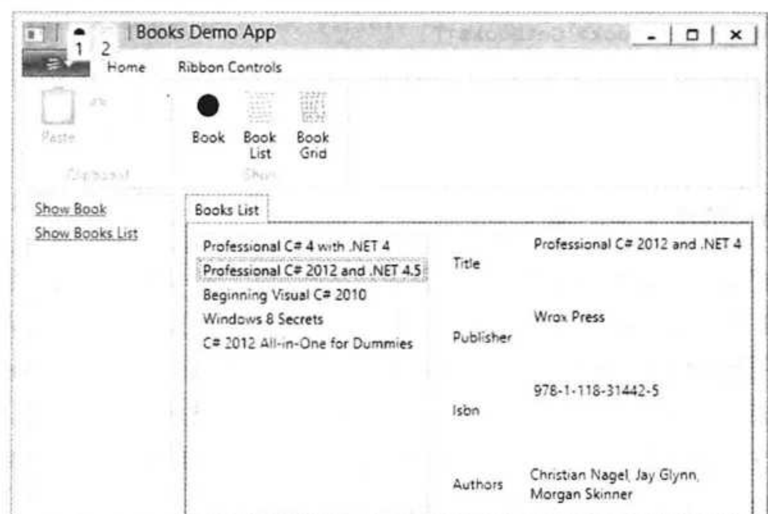


图 36-11

36.4.11　动态添加列表项

如果列表项要动态添加，就必须通知 WPF 元素：要在列表中添加元素。

在 WPF 应用程序的 XAML 代码中，要给 StackPanel 添加一个按钮元素。给 Click 事件指定 OnAddBook()方法(XAML 文件 BooksDemo/BooksUC.xaml)：

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
            HorizontalAlignment="Center">
  <Button Margin="5" Padding="4" Content="Add Book" Click="OnAddBook" />
</StackPanel>
```

在 OnAddBook()方法中，将一个新的 Book 对象添加到列表中。如果用 BookFactory 测试应用程序(因为它已实现)，就不会通知 WPF 元素：已在列表中添加了一个新对象(代码文件 BooksDemo/BooksUC.xaml.cs)。

```
private void OnAddBook(object sender, RoutedEventArgs e)
{
  ((this.FindResource("books") as ObjectDataProvider).Data as IList<Book>).
     Add(new Book("HTML and CSS: Design and Build Websites",
         "Wiley", "978-1118-00818-8"));
}
```

赋予 DataContext 的对象必须实现 INotifyCollectionChanged 接口。这个接口定义了由 WPF 应用程序使用的 CollectionChanged 事件。除了用自定义集合类实现这个接口之外，还可以使用泛型集合类 ObservableCollection<T>，该类在 WindowsBase 程序集的 System.Collections.ObjectModel 名称空间中定义。现在，把一个新列表项添加到集合中，这个新列表项会立即显示在列表框中(代码文件 BooksDemo/Utilities/BookFactory.cs)。

```
public class BookFactory
{
  private ObservableCollection<Book> books = new ObservableCollection<Book>();
```

```
    // ...

    public IEnumerable<Book> GetBooks()
    {
      return books;
    }
  }
```

36.4.12 动态添加选项卡中的项

在原则上，动态添加列表项与在选项卡控件中动态添加用户控件是一样的。目前，选项卡中的项使用 TabControl 类中 Items 属性的 Add 方法来动态添加。下面的示例直接从代码隐藏中引用 TabControl。而使用数据绑定，选项卡中的项信息可以添加到 ObservableCollection<T>中。

BookSample 应用程序中的代码现在改为给 TabControl 使用数据绑定。首先，定义类 UIControlInfo，这个类包含的属性在 TabControl 中用于数据绑定。Title 属性用于给选项卡中的项显示标题信息，Content 属性用于显示该项的内容：

```
using System.Windows.Controls;

namespace Wrox.ProCSharp.WPF
{
  public class UIControlInfo
  {
    public string Title { get; set; }
    public UserControl Content { get; set; }
  }
}
```

现在需要一个可观察的集合，以允许选项卡控件刷新其项的信息。userControls 是 MainWindow 类的一个成员变量。属性 Controls 用于数据绑定，它返回集合(代码文件 BooksDemo/MainWindow.xaml.cs)：

```
private ObservableCollection<UIControlInfo> userControls =
    new ObservableCollection<UIControlInfo>();
public IEnumerable<UIControlInfo> Controls
{
  get { return userControls; }
}
```

在 XAML 代码中修改了 TabControl。ItemsSource 属性绑定到 Controls 属性上。现在，需要指定两个模板，一个模板 ItemTemplate 定义了项控件的标题，用 ItemTemplate 指定的 DataTemplate 使用一个 TextBlock 元素，在项控件的标题中显示 Text 属性的值。另一个模板是 ContentTemplate，它指定使用 ContentPresenter 将绑定被绑定项的 Content 属性(XAML 文件 BooksDemo/MainWindow.xaml)：

```
<TabControl Margin="5" x:Name="tabControl1" ItemsSource="{Binding Controls}">
  <TabControl.ContentTemplate>
    <DataTemplate>
      <ContentPresenter Content="{Binding Content}" />
    </DataTemplate>
  </TabControl.ContentTemplate>
  <TabControl.ItemTemplate>
```

```
        <DataTemplate>
          <StackPanel Margin="0">
            <TextBlock Text="{Binding Title}" Margin="0" />
          </StackPanel>
        </DataTemplate>
      </TabControl.ItemTemplate>
    </TabControl>
```

现在，事件处理程序可以改为创建新的 UIControlInfo 对象，把它们添加到可观察的集合中，而不是创建 TabItem 控件。与使用代码隐藏相比，修改项和内容模板是定制外观的一种更简单方式：

```
private void OnShowBooksList(object sender, ExecutedRoutedEventArgs e)
{
  var booksUI = new BooksUC();
  userControls.Add(new UIControlInfo
  {
    Title = "Books List",
    Content = booksUI
  });
}
```

36.4.13 数据模板选择器

第 35 章介绍了如何用模板来定制控件，还讨论了如何创建数据模板，为特定的数据类型定义外观。数据模板选择器可以为同一个数据类型动态地创建不同的数据模板。数据模板选择器在派生自 DataTemplateSelector 基类的类中实现。

下面实现的数据模板选择器根据发布者选择另一个模板。在用户控件的资源中，定义这些模板。一个模板可以通过键名 wroxTemplate 来访问；另一个模板的键名是 dummiesTemplate；第 3 个模板的键名是 bookTemplate(XAML 文件 BooksDemo/BooksUC.xaml)：

```
<DataTemplate x:Key="wroxTemplate" DataType="{x:Type local:Book}">
  <Border Background="Red" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="dummiesTemplate" DataType="{x:Type local:Book}">
  <Border Background="Yellow" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="bookTemplate" DataType="{x:Type local:Book}">
  <Border Background="LightBlue" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
```

```
    </StackPanel>
  </Border>
</DataTemplate>
```

要选择模板，BookDataTemplateSelector 类必须重写来自基类 DataTemplateSelector 的 SelectTemplate 方法。其实现方式根据 Book 类的 Publisher 属性选择模板(代码文件 BooksDemo/Utilities/BookTemplateSelector.cs)：

```
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.WPF.Data;

namespace Wrox.ProCSharp.WPF.Utilities
{
  public class BookTemplateSelector : DataTemplateSelector
  {
    public override DataTemplate SelectTemplate(object item,
        DependencyObject container)
    {
      if (item != null && item is Book)
      {
        var book = item as Book;
        switch (book.Publisher)
        {
          case "Wrox Press":
            return (container as FrameworkElement).FindResource(
                "wroxTemplate") as DataTemplate;
          case "For Dummies":
            return (container as FrameworkElement).FindResource(
                "dummiesTemplate") as DataTemplate;
          default:
            return (container as FrameworkElement).FindResource(
                "bookTemplate") as DataTemplate;
        }
      }
      return null;
    }
  }
}
```

要从XAML代码中访问BookDataTemplateSelector类，这个类必须在Window资源中定义(XAML文件 BooksDemo/BooksUC.xaml)：

```
<src:BookDataTemplateSelector x:Key="bookTemplateSelector" />
```

现在可以把选择器类赋予 ListBox 的 ItemTemplateSelector 属性：

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
        MinWidth="120" IsSynchronizedWithCurrentItem="True"
        ItemTemplateSelector="{StaticResource bookTemplateSelector}">
```

运行这个应用程序，可以看到基于不同发布者的不同数据模板，如图 36-12 所示。

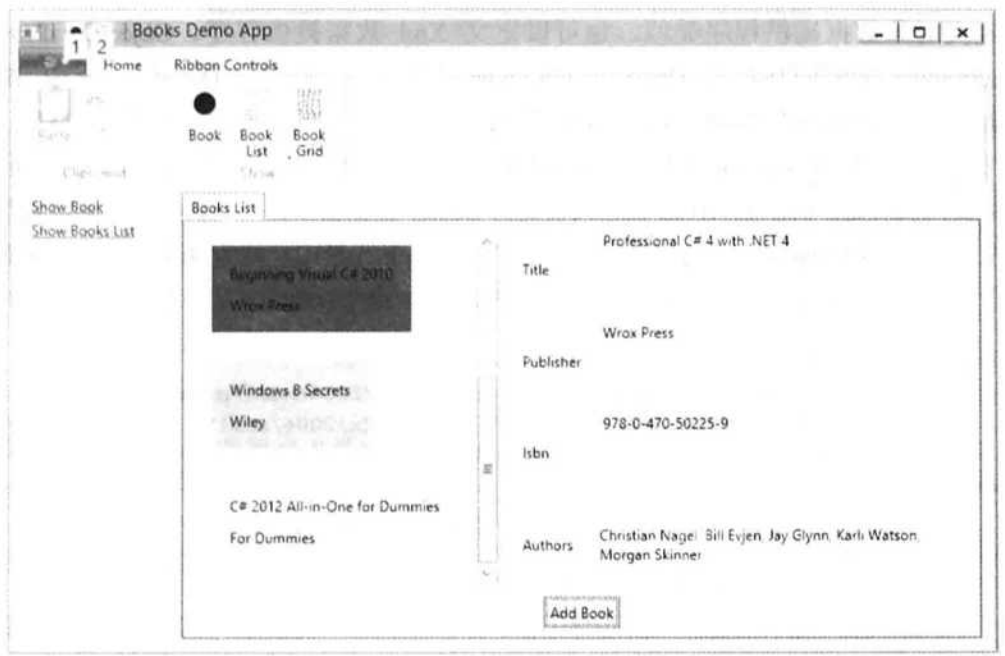


图 36-12

36.4.14　绑定到 XML 上

WPF 数据绑定还专门支持绑定到 XML 数据上。可以将 XmlDataProvider 用作数据源，使用 XPath 表达式绑定元素。为了以层次结构显示，可以使用 TreeView 控件，通过 HierarchicalDataTemplate 为对应项创建视图。

下面包含 Book 元素的 XML 文件将用作下一个例子的数据源(XAML 文件 XmlBindingDemo/Books.xml)：

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-1-118-31442-5">
    <Title>Professional C# 2012</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Jay Glynn</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-470-50226-6">
    <Title>Beginning Visual C# 2010</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>Christian Nagel</Author>
    <Author>Jacob Hammer Pedersen</Author>
    <Author>John D. Reid</Author>
    <Author>Morgan Skinner</Author>
  </Book>
</Books>
```

与定义对象数据提供程序类似，也可以定义 Xml 数据提供程序。ObjectDataProvider 和 XmlDataProvider 都派生自同一个 DataSourceProvider 基类。在示例的 XmlDataProvider 中，把 Source 属性设置为引用 XML 文件 books.xml。XPath 属性定义了一个 XPath 表达式，以引用 XML 根元素 Books。Grid 元素通过 DataContext 属性引用 XML 数据源。通过栅格的数据上下文，因为所有 Book 元素都需要列表绑定，所以把 XPath 表达式设置为 Book。在栅格中，把列表框元素绑定到默认的数据上下文中，并使用 DataTemplate 将标题包含在 TextBlock 元素中，作为列表框的项。在栅格中，还有 3 个标签元素，把它们的数据绑定设置为 XPath 表达式，以显示标题、出版社和 ISBN 号。

```
<Window x:Class="XmlBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Main Window" Height="240" Width="500">
  <Window.Resources>
    <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
    <DataTemplate x:Key="listTemplate">
      <TextBlock Text="{Binding XPath=Title}" />
    </DataTemplate>

    <Style x:Key="labelStyle" TargetType="{x:Type Label}">
      <Setter Property="Width" Value="190" />
      <Setter Property="Height" Value="40" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Window.Resources>

  <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <ListBox IsSynchronizedWithCurrentItem="True" Margin="5"
        Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
        ItemTemplate="{StaticResource listTemplate}" />

      <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
         Grid.Row="0" Grid.Column="1" />
      <Label Style="{StaticResource labelStyle}"
         Content="{Binding XPath=Publisher}" Grid.Row="1" Grid.Column="1" />
      <Label Style="{StaticResource labelStyle}"
         Content="{Binding XPath=@isbn}" Grid.Row="2" Grid.Column="1" />
  </Grid>
</Window>
```

图 36-13 显示了 XML 绑定的结果。

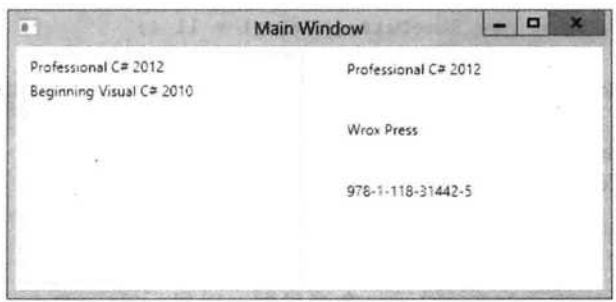


图 36-13

如果 XML 数据应以层次结构的方式显示，就可以使用 TreeView 控件。

36.4.15　绑定的验证和错误处理

在把数据用于.NET 对象之前，有几个选项可用于验证用户的数据，这些选项如下：

- 处理异常
- 数据错误信息的处理
- 数据错误信息的通知
- 自定义验证规则

1. 处理异常

这里说明的一个选项是，如果在 SomeData 类中设置了无效值，则这个.NET 类就抛出一个异常。Value1 属性只接受大于等于 5 且小于 12 的值(代码文件 ValidationDemo/SomeData.cs)：

```
public class SomeData
{
  private int value1;
  public int Value1 {
    get { return value1; }
    set
    {
      if (value < 5 || value > 12)
      {
        throw new ArgumentException(
             "value must not be less than 5 or greater than 12");
      }
      value1 = value;
    }
  }
}
```

在 MainWindow1 类的构造函数中，初始化 SomeData 类的一个新对象，并把它传递给 DataContext，用于数据绑定(代码文件 ValidationDemo/MainWindow.xaml.cs)：

```
public partial class MainWindow: Window
{
```

```
    private SomeData p1 = new SomeData { Value1 = 11 };

    public MainWindow()
    {
      InitializeComponent();
      this.DataContext = p1;

    }
```

事件处理程序方法 OnShowValue 显示一个消息框，以显示 SomeData 实例的实际值：

```
    private void OnShowValue(object sender, RoutedEventArgs e)
    {
      MessageBox.Show(p1.Value1.ToString());
    }
  }
```

通过简单的数据绑定，把文本框的 Text 属性绑定到 Value1 属性上。如果现在运行应用程序，并试图把该值改为某个无效值，那么单击 Submit 按钮可以验证该值永远不会改变。WPF 会捕获并忽略 Value1 属性的 set 访问器抛出的异常(XAML 文件 ValidationDemo/MainWindow.xaml)。

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1}" />
```

要在输入字段的上下文发生变化时尽快显示错误，可以把 Binding 标记扩展的 ValidatesOnException 属性设置为 True。输入一个无效值(设置该值时，会很快抛出一个异常)，文本框就会以红线框出，如图 36-14 所示。

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}" />
```

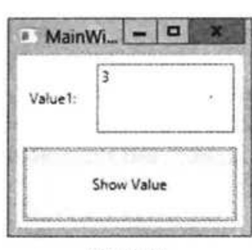
要以另一种方式给用户返回错误信息，可以把 Validation 类定义的附加属性 ErrorTemplate 赋予一个为错误定义 UI 的模板。这里标记错误的新模板用 validationTemplate 键表示。ControlTemplate 键在已有的控件内容前面添加了一个红色的感叹号。

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="40">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

用 Validation.ErrorTemplate 附加属性设置 validationTemplate 会激活带文本框的模板：

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
     Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
     Validation.ErrorTemplate="{StaticResource validationTemplate}" />
```

应用程序的新外观如图 36-15 所示。



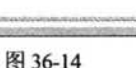


图 36-14

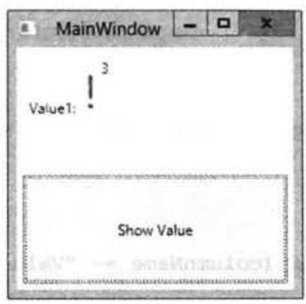


图 36-15

自定义错误消息的另一个选项是注册到 Validation 类的 Error 事件。这里必须把 NotifyOnValidationError 属性设置为 true。

可以从 Validation 类的 Errors 集合中访问错误信息。要在文本框的工具提示中显示错误信息，可以创建一个属性触发器，如下所示。只要把 Validation 类的 HasError 属性设置为 True，就激活触发器。触发器设置文本框的 ToolTip 属性：

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
          Value="{Binding RelativeSource={x:Static RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

2. 数据错误信息

处理错误的另一种方式是确定.NET 对象是否执行了 IDateErrorInfo 接口。SomeData 类现在改为实现 IDateErrorInfo 接口。这个接口定义了 Error 属性和带字符串参数的索引器。在数据绑定的过程中验证 WPF 时，会调用索引器，并把要验证的属性名作为 columnName 参数传递。在实现代码中，如果有效，会验证其值，如果无效，就传递一个错误字符串。下面验证 Value2 属性，它使用 C#简单属性标记实现(代码文件 ValidationDemo/SomeData.cs)。

```
public class SomeData: IDataErrorInfo
{
  //...

  public int Value2 { get; set; }

  string IDataErrorInfo.Error
  {
    get
    {
      return null;
```

```
      }
    }

    string IDataErrorInfo.this[string columnName]
    {
      get
      {
        if (columnName == "Value2")
        {
          if (this.Value2 < 0 || this.Value2 > 80)
              return "age must not be less than 0 or greater than 80";

        }
        return null;
      }
    }
}
```

在.NET 对象中，索引器返回什么内容并不清楚，例如，调用索引器，会从 Person 类型的对象中返回什么？因此最好在 IDataErrorInfo 接口中包含显式的实现代码。这样，这个索引器只能使用接口来访问，.NET 类可以有另一种实现方式，以实现其他目的。

如果把 Binding 类的 ValidationOnDataErrors 属性设置为 true，就在数据绑定过程中使用 IDataErrorInfo 接口。这里，改变文本框时，绑定机制会调用接口的索引器，并把 Value2 传递给 columnName 变量(XAML 文件 ValidationDemo/MainWindow.xaml)：

```
<Label Margin="5" Grid.Row="1" Grid.Column="0" >Value2:</Label>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
   Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />
```

3. 数据错误信息的通知

除了支持利用异常和 IDataErrorInfo 接口进行验证之外，.NET 4.5 附带的 WPF 还支持利用接口 INotifyDataErrorInfo 进行验证。在 IDataErrorInfo 接口中，属性的索引器可以返回一个错误，而在 INotifyDataErrorInfo 中，可以把多个错误关联到一个属性上。这些错误可以使用 GetErrors 方法来访问。如果实体有错误，HasErrors 属性就返回 true。这个接口的另一个很好的功能是使用事件 ErrorsChanged 通知出了错误。这样，错误就可以在客户端异步检索，例如，可以调用一个 Web 服务来验证用户输入。此时，在检索结果时，用户可以继续处理输入表单，并获得不匹配情况的异步通知。

下面的示例使用 INotifyDataErrorInfo 进行验证。该示例定义基类 NotifyDataErrorInfoBase，这个基类实现了接口 INotifyDataErrorInfo。它派生于基类 BindableObject，来获得 INotifyPropertyChanged 接口的实现，如本章前面所示。NotifyDataErrorInfoBase 使用字典 errors 来包含一个列表，列表中的每个属性都用于存储错误信息。如果任何属性有错误，HasErrors 属性就返回 true。GetErrors 方法返回一个属性的错误列表；事件 ErrorsChanged 在每次改变错误信息时触发。除了接口 INotifyDataErrorInfo 中的成员之外，这个基类还实现了方法 SetErrors、ClearErrors 和 ClearAllErrors，

以便于处理设置错误(代码文件 ValidationDemo/NotifyDataErrorInfoBase.cs)：

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace ValidationDemo
{
  public abstract class NotifyDataErrorInfoBase : BindableObject,
      INotifyDataErrorInfo
  {
    public void SetError(string errorMessage,
        [CallerMemberName] string propertyName = null)
    {
      List<string> errorList;
      if (errors.TryGetValue(propertyName, out errorList))
      {
        errorList.Add(errorMessage);
      }
      else
      {
        errorList = new List<string> { errorMessage };
        errors.Add(propertyName, errorList);
      }
      HasErrors = true;
      OnErrorsChanged(propertyName);
    }

    public void ClearErrors([CallerMemberName] string propertyName = null)
    {
      if (hasErrors)
      {
        List<string> errorList;
        if (errors.TryGetValue(propertyName, out errorList))
        {
          errors.Remove(propertyName);
        }
        if (errors.Count == 0)
        {
          HasErrors = false;
        }
        OnErrorsChanged(propertyName);
      }
    }

    public void ClearAllErrors()
    {
      if (HasErrors)
      {
        errors.Clear();
        HasErrors = false;
        OnErrorsChanged(null);
      }
```

```
    }

    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    private Dictionary<string, List<string>> errors =
        new Dictionary<string, List<string>>();
    public IEnumerable GetErrors(string propertyName)
    {
      List<string> errorsForProperty;
      bool err = errors.TryGetValue(propertyName, out errorsForProperty);
      if (!err) return null;
      return errorsForProperty;
    }

    private bool hasErrors = false;
    public bool HasErrors
    {
      get { return hasErrors; }
      protected set {
        if (SetProperty(ref hasErrors, value))
        {
          OnErrorsChanged(propertyName: null);
        }
      }
    }

    protected void OnErrorsChanged([CallerMemberName] string propertyName = null)
    {
      var errorsChanged = ErrorsChanged;
      if (errorsChanged != null)
      {
        errorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
      }
    }
  }
}
```

类 SomeDataWithNotifications 是绑定到 XAML 代码上的数据对象。这个类派生于基类 NotifyDataErrorInfoBase，继承了 INotifyDataErrorInfo 接口的实现代码。属性 Val1 是异步验证的。对于验证，应在设置属性后调用 CheckVal1 方法，这个方法会异步调用方法 ValidationSimulator.Validate。调用这个方法后，UI 线程就可以返回，来处理其他事件；一旦返回了结果，若返回了一个错误，就调用基类的 SetErrors 方法。很容易把异步调用改为调用 Web 服务或执行另一个异步操作(代码文件 ValidationDemo/SomeDataWithNotifications.cs)：

```
using System.Runtime.CompilerServices;
using System.Threading.Tasks;

namespace ValidationDemo
{
  public class SomeDataWithNotifications : NotifyDataErrorInfoBase
  {
    private int val1;
    public int Val1
    {
```

```
        get { return val1; }
        set
        {
          SetProperty(ref val1, value);
          CheckVal1(val1, value);
        }
      }

      private async void CheckVal1(int oldValue, int newValue,
          [CallerMemberName] string propertyName = null)
      {
        ClearErrors(propertyName);

        string result = await ValidationSimulator.Validate(newValue, propertyName);
        if (result != null)
        {
          SetError(result, propertyName);
      }
    }
  }
```

ValidationSimulator 的 Validate 方法在检查值之前推迟了 3 秒，如果该值大于 50，就返回一个错误消息：

```
  public static class ValidationSimulator
  {
    public static Task<string> Validate(int val,
        [CallerMemberName] string propertyName = null)
    {
      return Task<string>.Run(async () =>
        {
          await Task.Delay(3000);
          if (val > 50) return "bad value";
          else return null;
        });
    }
  }
```

在数据绑定中，只有 ValidatesOnNotifyDataErrors 属性必须设置为 True，才能使用接口 INotifyDataErrorInfo 的异步验证功能(XAML 文件 ValidationDemo/NotificationWindows.xaml)：

```
  <TextBox Grid.Row="0" Grid.Column="1"
      Text="{Binding Val1, ValidatesOnNotifyDataErrors=True}" Margin="8" />
```

运行应用程序，就可以看到在输入错误的信息后，文本框被默认的红色矩形包围了 3 秒。以不同的方式显示错误信息也可以用以前的方式实现——使用错误模板和触发器，来访问验证错误。

4. 自定义验证规则

为了更多地控制验证方式，可以实现自定义验证规则。实现自定义验证规则的类必须派生自基类 ValidationRule。在前面的两个例子中，也使用了验证规则。派生自 ValidationRule 抽象基类的两个类是 DataErrorValidationRule 和 ExceptionValidationRule。设置 ValidatesOnDataErrors 属性，并使用 IDataErrorInfo 接口，就可以激活 DataErrorValidationRule。ExceptionValidationRule 处理异常，设置

ValidationOnException 属性会激活 ExceptionValidationRule。

下面实现一条验证规则，来验证正则表达式。RegularExpressionValidationRule 类派生自基类 ValidationRule，并重写基类定义的抽象方法 Validate。在其实现代码中，使用 System.Text.RegularExpressions 名称空间中的 RegEx 类验证 Expression 属性定义的表达式。

```
public class RegularExpressionValidationRule : ValidationRule
{
  public string Expression { get; set; }
  public string ErrorMessage { get; set; }

  public override ValidationResult Validate(object value,
      CultureInfo cultureInfo)
  {
    ValidationResult result = null;
    if (value != null)
    {
      var regEx = new Regex(Expression);
      bool isMatch = regEx.IsMatch(value.ToString());
      result = new ValidationResult(isMatch, isMatch ?
          null: ErrorMessage);
    }
    return result;
  }
}
```

正则表达式参见第 9 章。

这里没有使用 Binding 标记扩展，而是把绑定作为 TextBox.Text 元素的一个子元素。绑定的对象现在定义一个 Email 属性，它用简单的属性语法来实现。UpdateSourceTrigger 属性定义绑定源何时应更新。更新绑定源的可能选项如下：

- 属性值变化时更新，即用户输入属性值中的每个字符时更新
- 失去焦点时更新
- 显式指定更新时间

ValidationRules 是 Binding 类的一个属性，Binding 类包含 ValidationRule 元素。这里使用的验证规则是自定义类 RegularExpressionValidationRule，其中把 Expression 属性设置为一个正则表达式，正则表达式用于验证输入是否是有效的电子邮件，ErrorMessage 属性给出 TextBox 中的输入数据无效时显示的错误消息：

```
<Label Margin="5" Grid.Row="2" Grid.Column="0">Email:</Label>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="Email" UpdateSourceTrigger="LostFocus">
      <Binding.ValidationRules>
        <src:RegularExpressionValidationRule
            Expression="^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.
                        [0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|
                        [0-9]{1,3})(\]?)$"
            ErrorMessage="Email is not valid" />
```

```
        </Binding.ValidationRules>
      </Binding>
    </TextBox.Text>
  </TextBox>
```

36.5 TreeView

TreeView 控件可以显示层次数据。绑定到 TreeView 非常类似于前面的绑定到 ListBox，其区别是绑定 TreeView 会显示分层数据——可以使用 HierarchicalDataTemplate。

下面的示例使用分层显示方式和 DataGrid 控件。Formula1 样本数据库通过 ADO.NET Entity Framework 来访问。所使用的映射如图 36-16 所示。Race 类包含竞赛日期的信息，且关联到 Circuit 类上。Circuit 类包含 Country 和竞赛环形跑道的信息。Race 类还与 RaceResult 类关联起来。RaceResult 类包含 Racer 和 Team 的信息。

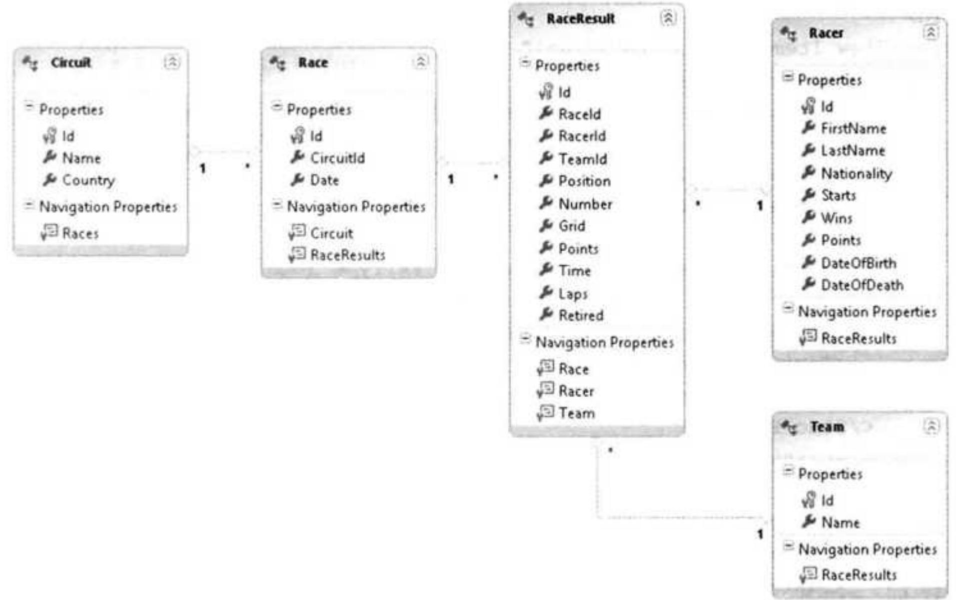


图 36-16

ADO.NET Entity Framework 参见第 33 章。

使用 XAML 代码声明一个 TreeView。TreeView 派生自基类 ItemsControl，其中，与列表的绑定可以通过 ItemsSource 属性来完成。把 ItemsSource 属性绑定到数据上下文上。数据上下文在代码隐藏中指定，如下所示。当然，这也可以通过 ObjectDataProvider 来实现。为了定义分层数据的自定义显示方式，定义了 HierarchicalDataTemplate 元素。这里的数据模板是用 DataType 属性为特定的数据类型定义的。第一个 HierarchicalDataTemplate 是 Championship 类的模板，它把这个类的 Year 属性绑定到 TextBlock 的 Text 属性上。ItemsSource 属性定义了该数据模板本身的绑定，以指定数据层次结构中的下一层。如果 Championship 类的 Races 属性返回一个集合，就直接把 ItemsSource 属性绑定到 Races 上。但是，因为这个属性返回一个 Lazy<T>对象，所以绑定到 Races.Value 上。Lazy<T>

类的优点在本章后面讨论。

第二个 HierarchicalDataTemplate 元素定义 F1Race 类的模板，并绑定这个类的 Country 和 Date 属性。利用 Date 属性，通过绑定定义一个 StringFormat。把 ItemsSource 属性绑定到 Races.Value 上，来定义层次结构中的下一层。

因为 F1RaceResult 类没有子集合，所以层次结构到此为止。对于这个数据类型，定义一个正常的 DataTemplate，来绑定 Position、Racer 和 Car 属性(XAML 文件 Formula1Demo/TreeUC.xaml)：

```
<UserControl x:Class="Formula1Demo.TreeUC"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:Formula1Demo"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <TreeView ItemsSource="{Binding}" >
      <TreeView.Resources>
        <HierarchicalDataTemplate DataType="{x:Type local:Championship}"
                                  ItemsSource="{Binding Races.Value}">
          <TextBlock Text="{Binding Year}" />
        </HierarchicalDataTemplate>

        <HierarchicalDataTemplate DataType="{x:Type local:F1Race}"
                                  ItemsSource="{Binding Results.Value}">
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Country}" Margin="5,0,5,0" />
            <TextBlock Text="{Binding Date, StringFormat=d}" Margin="5,0,5,0" />
          </StackPanel>
        </HierarchicalDataTemplate>

        <DataTemplate DataType="{x:Type local:F1RaceResult}">
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Position}" Margin="5,0,5,0" />
            <TextBlock Text="{Binding Racer}" Margin="5,0,0,0" />
            <TextBlock Text=", " />
            <TextBlock Text="{Binding Car}" />
          </StackPanel>
        </DataTemplate>
      </TreeView.Resources>
    </TreeView>
  </Grid>
</UserControl>
```

下面是填充分层控件的代码。在 XAML 代码的代码隐藏文件中，把 DataContext 赋予 Years 属性。Years 属性使用一个 LINQ 查询，而不是 ADO.NET Entity Framework 数据上下文，来获取数据库中所有一级方程式比赛的年份，并为每个年份新建一个 Championship 对象。通过 Championship 类的实例设置 Year 属性。这个类也有一个 Races 属性，可返回该年份的比赛信息，但这些信息还没有填充(代码文件 Formula1Demo/TreeUC.xaml.cs)。

LINQ 参见第 11 章和第 33 章。

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

namespace Formula1Demo
{
  public partial class TreeUC : UserControl
  {
    private Formula1Entities data = new Formula1Entities();

    public TreeUC()
    {
      InitializeComponent();
        this.DataContext = Years;
    }

    public IEnumerable<Championship> Years
    {
      get
      {
        F1DataContext.Data = data;
        return data.Races.Select(r => new Championship
        {
          Year = r.Date.Year
        }).Distinct().OrderBy(c => c.Year);
      }
    }
  }
}
```

Championship 类有一个用于返回年份的简单的自动属性。Races 属性的类型是 Lazy<IEnumerable<F1Race>>。Lazy<T>类是.NET 4 新增的，用于懒惰初始化。对于 TreeView 控件，这个类非常方便。如果表达式树中的数据非常多，且不希望提前加载整个表达式树，但仅在用户做出选择时加载，就可以使用懒惰加载方式。在 Lazy<T>类的构造函数中使用 Func<IEnumerable<F1Race>>委托。在这个委托中，需要返回 IEnumerable<F1Race>。赋予该委托的 lambda 表达式的实现方式使用一个 LINQ 查询，来创建一个 F1Race 对象列表，并指定它们的 Date 和 Country 属性(代码文件 Formula1Demo/Championship.cs)：

```
public class Championship
{
  public int Year { get; set; }
  public Lazy<IEnumerable<F1Race>> Races
  {
    get
    {
      return new Lazy<IEnumerable<F1Race>>(() =>
      {
```

```
            return from r in F1DataContext.Data.Races
                   where r.Date.Year == Year
                   orderby r.Date
                   select new F1Race
                   {
                     Date = r.Date,
                     Country = r.Circuit.Country
                   };
        });
      }
    }
  }
```

F1Race 类也定义了 Results 属性，该属性使用 Lazy<T>类型返回一个 F1RaceResult 对象列表(代码文件 Formula1Demo/F1Race.cs)：

```
public class F1Race
{
  public string Country { get; set; }
  public DateTime Date { get; set; }
  public Lazy<IEnumerable<F1RaceResult>> Results
  {
    get
    {
      return new Lazy<IEnumerable<F1RaceResult>>(() =>
      {
        return from rr in F1DataContext.Data.RaceResults
               where rr.Race.Date == this.Date
               select new F1RaceResult
               {
                 rr.Position,
                 Racer = rr.Racer.FirstName + " " + rr.Racer.LastName,
                 Car = rr.Team.Name
               };
      });
    }
  }
}
```

层次结构中的最后一个类是 F1RaceResult，它是 Position、Racer 和 Car 的简单数据存储器(代码文件 Formula1Demo/Championship.cs)：

```
public class F1RaceResult
{
  public int Position { get; set; }
  public string Racer { get; set; }
  public string Car { get; set; }
}
```

运行应用程序，首先会在树型视图中看到所有年份的冠军。因为使用了绑定，所以也访问了下一层——每个 Championship 对象已经关联到 F1Race 对象。用户不需要等待，就可以看到年份下面的第一级，也不需要使用默认显示的小三角形来打开某个年份的信息。图 36-17 打开了 1984 年的信息。只要用户单击某个年份，就会看到第二级绑定，第三级也绑定了，并检索出比赛结果。

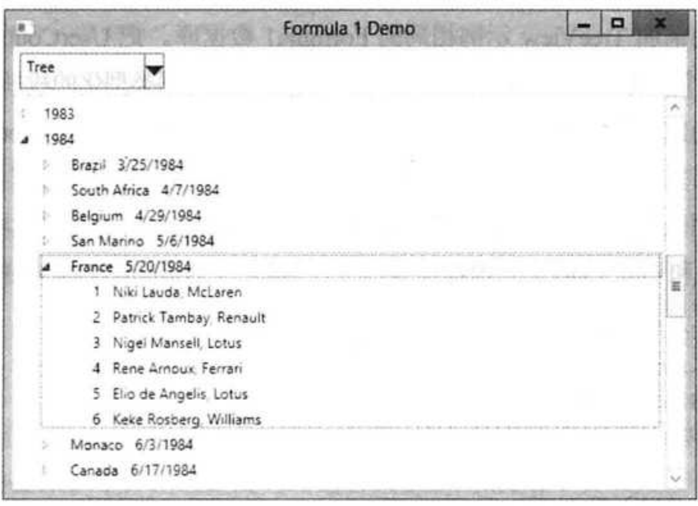


图 36-17

当然也可以定制 TreeView 控件，并为整个模板或视图中的项定义不同的样式。

36.6　DataGrid

通过 DataGrid 控件，可以把信息显示在行和列中，还可以编辑它们。DataGrid 控件是一个 ItemsControl，定义了绑定到集合上的 ItemsSource 属性。这个用户界面的 XAML 代码也定义了两个 RepeatButton 控件，用于实现分页功能。这里不是一次加载所有比赛信息，而是使用分页功能，这样用户就可以翻看各个页面。在简单的场景中，只需要指定 DataGrid 的 ItemsSource 属性。默认情况下，DataGrid 会根据绑定数据的属性来创建列(XAML 文件 Formula1Demo/GridUC.xaml)：

```
<UserControl x:Class="Formula1Demo.GridUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <Grid.RowDefinitions>
      <RepeatButton Margin="5" Click="OnPrevious">Previous</RepeatButton>
      <RepeatButton Margin="5" Click="OnNext">Next</RepeatButton>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.Row="0">
      <Button Click="OnPrevious">Previous</Button>
      <Button Click="OnNext">Next</Button>
    </StackPanel>
    <DataGrid Grid.Row="1" ItemsSource="{Binding}" />
  </Grid>
</UserControl>
```

代码隐藏使用与前面 TreeView 示例相同的 Formula1 数据库。把 UserControl 的 DataContext 设置为 Races 属性。这个属性返回 IEnumerable<object>。这里不指定强类型化的枚举，而使用一个 object，以通过 LINQ 查询创建一个匿名类。该 LINQ 查询使用 Year、Country、Position、Racer 和 Car 属性创建匿名类，并使用复合语句访问 Races 和 RaceResults 属性。它还访问 Races 的其他关联属性，以获取国籍、赛手和团队信息。使用 Skip()和 Take()方法实现分页功能。页面的大小固定为 50 项，当前页面使用 OnNext 和 OnPrevious 处理程序来改变(代码文件 Formula1Demo/GridUC.xaml.cs)：

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

namespace Formula1Demo
{
  public partial class GridUC : UserControl
  {
    private int currentPage = 0;
    private int pageSize = 50;
    private Formula1Entities data = new Formula1Entities();
    public GridUC()
    {
      InitializeComponent();
      this.DataContext = Races;
    }

    public IEnumerable<object> Races
    {
      get
      {
        return (from r in data.Races
                from rr in r.RaceResults
                orderby r.Date ascending
                select new
                {
                  r.Date.Year,
                  r.Circuit.Country,
                  rr.Position,
                  Racer = rr.Racer.FirstName + " " + rr.Racer.LastName,
                  Car = rr.Team.Name
                }).Skip(currentPage * pageSize).Take(pageSize);
      }
    }

    private void OnPrevious(object sender, RoutedEventArgs e)
    {
      if (currentPage > 0)
      {
        currentPage--;
```

```
            this.DataContext = Races;
          }
        }

        private void OnNext(object sender, RoutedEventArgs e)
        {
          currentPage++;
          this.DataContext = Races;
        }
      }
    }
```

图 36-18 显示了正在运行的应用程序，其中使用了默认的网格样式和标题。

在下一个 DataGrid 示例中，用自定义列和组合来定制网格。

Formula 1 Demo

DataGrid

Previous　Next

Year	Country	Position	Racer	Car
1970	Monaco	1	Jochen Rindt	Lotus
1970	Monaco	2	Jack Brabham	Brabham
1970	Monaco	3	Henri Pescarolo	Matra
1970	Monaco	4	Denny Hulme	McLaren
1970	Monaco	5	Graham Hill	Lotus
1970	Monaco	6	Pedro Rodriguez	BRM
1970	Belgium	1	Pedro Rodriguez	BRM
1970	Belgium	2	Chris Amon	March
1970	Belgium	3	Jean-Pierre Beltoise	Matra
1970	Belgium	4	Ignazio Giunti	Ferrari
1970	Belgium	5	Rolf Stommelen	Brabham
1970	Belgium	6	Henri Pescarolo	Matra
1970	Netherlands	1	Jochen Rindt	Lotus

图 36-18

36.6.1　自定义列

把 DataGrid 的 AutoGenerateColumns 属性设置为 False，就不会生成默认的列。使用 Columns 属性可以创建自定义列。还可以指定派生自 DataGridColumn 的元素，也可以使用预定义的类。DataGridTextColumn 可以用于读取和编辑文本。DataGridHyperlinkColumn 可显示超链接。DataGrid-CheckBoxColumn 可给布尔数据显示复选框。如果某列有一个项列表，就可以使用 DataGridCombo-BoxColumn。将来会有更多的 DataGridColumn 类型，但如果现在就需要其他表示方式，可以使用 DataGridTempalteColumn 定义并绑定任意需要的元素。

下面的示例代码使用 DataGridTextColumn 来绑定到 Position 和 Racer 属性。把 Header 属性设置为要显示的字符串，当然也可以使用模板给列定义完全自定义的标题(XAML 文件 Formula1Demo/GridUC.xaml.cs)：

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Position, Mode=OneWay}"
                        Header="Position" />
```

```
        <DataGridTextColumn Binding="{Binding Racer, Mode=OneWay}"
                            Header="Racer" />
    </DataGrid.Columns>
```

36.6.2 行的细节

选择一行时，DataGrid可以显示该行的其他信息。为此，需要指定DataGrid的RowDetailsTemplate。把一个DataTemplate赋予这个RowDetailsTemplate，其中包含几个显示汽车和赛点的TextBlock元素(XAML文件Formula1Demo/GridUC.xaml.cs)：

```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <StackPanel Orientation="Horizontal">
      <TextBlock Text="Car:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Car}" Margin="5,0,0,0" />
      <TextBlock Text="Points:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Points}" />
    </StackPanel>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

36.6.3 用DataGrid进行分组

一级方程式比赛有几行包含相同的信息，如年份和国籍。对于这类数据，可以使用分组功能，给用户组织信息。

对于分组功能，可以在XAML代码中使用CollectionViewSource来支持分组、排序和筛选功能。在代码隐藏中，也可以使用ListCollectionView类，它仅由CollectionViewSource使用。

CollectionViewSource在Resources集合中定义。CollectionViewSource的源是ObjectDataProvider的结果。ObjectDataProvider调用F1Races类型的GetRaces()方法。这个方法有两个int参数，它们从MethodParameters集合中指定。CollectionViewSource给分组使用了两个描述，分别用于Year属性和Country属性(XAML文件Formula1Demo/GridGroupingUC.xaml)：

```
<Grid.Resources>
  <ObjectDataProvider x:Key="races" ObjectType="{x:Type local:F1Races}"
                      MethodName="GetRaces">
    <ObjectDataProvider.MethodParameters>
      <sys:Int32>0</sys:Int32>
      <sys:Int32>20</sys:Int32>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  <CollectionViewSource x:Key="viewSource"
                        Source="{StaticResource races}">
    <CollectionViewSource.GroupDescriptions>
      <PropertyGroupDescription PropertyName="Year" />
      <PropertyGroupDescription PropertyName="Country" />
    </CollectionViewSource.GroupDescriptions>
  </CollectionViewSource>
</Grid.Resources>
```

这里显示的组使用 DataGrid 的 GroupStyle 属性定义。对于 GroupStyle 元素，需要自定义 ContainsStyle、HeaderTemplate 和整个面板。为了动态选择 GroupStyle 和 HeaderStyle，还可以编写一个容器样式选择器和一个标题模板选择器。它们的功能非常类似于前面的数据模板选择器。

示例中的 GroupStyle 设置了 GroupStyle 的 ContainsStyle 属性。在这个样式中，用模板定制 GroupItem。使用分组功能时，GroupItem 显示为组的根元素。在组中使用 Names 属性显示名字，使用 ItemCount 属性显示项数。Grid 的第 3 列使用 ItemPresenter 包含所有正常的项。如果行按国籍分组，Name 属性的标签就会有不同的宽度，这看起来不太好。因此，使用 Grid 的第二列设置 SharedSizeGroup 属性，使所有的项有相同的大小。还需要设置共享的尺寸范围，使所有的元素有相同的大小，为此在 DataGrid 中设置 Grid.IsSharedSizeScope="True"。

```
<DataGrid.GroupStyle>
  <GroupStyle>
    <GroupStyle.ContainerStyle>
      <Style TargetType="{x:Type GroupItem}">
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate >
              <StackPanel Orientation="Horizontal" >
                <Grid>
                  <Grid.ColumnDefinitions>
                    <ColumnDefinition SharedSizeGroup="LeftColumn" />
                    <ColumnDefinition />
                    <ColumnDefinition />
                  </Grid.ColumnDefinitions>
                  <Label Grid.Column="0" Background="Yellow"
                         Content="{Binding Name}" />
                  <Label Grid.Column="1" Content="{Binding ItemCount}" />
                  <Grid Grid.Column="2" HorizontalAlignment="Center"
                         VerticalAlignment="Center">
                    <ItemsPresenter/>
                  </Grid>
                </Grid>
              </StackPanel>
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </GroupStyle.ContainerStyle>
  </GroupStyle>
</DataGrid.GroupStyle>
```

ObjectDataProvider 使用了类 F1Races，F1Races 使用 LINQ 访问 Formula1 数据库，并返回一个匿名类型列表，以及 Year、Country、Position、Racer、Car 和 Points 属性。这里再次使用 Skip()和 Take()方法访问部分数据(代码文件 Formula1Demo/F1Races.cs)：

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

```
namespace Formula1Demo
{
  public class F1Races
  {
    private int lastpageSearched = -1;
    private IEnumerable<object>cache = null;
    private Formula1Entities data = new Formula1Entities();

    public IEnumerable<object> GetRaces(int page, int pageSize)
    {
      if (lastpageSearched == page)
        return cache;
      lastpageSearched = page;

      var q = (from r in data.Races
               from rr in r.RaceResults
               orderby r.Date ascending
               select new
               {
                 Year = r.Date.Year,
                 Country = r.Circuit.Country,
                 Position = rr.Position,
                 Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                 Car = rr.Team.Name,
                 Points = rr.Points
               }).Skip(page * pageSize).Take(pageSize);
      cache = q;
      return cache;
    }
  }
}
```

现在只需要为用户设置页码，修改 ObjectDataProvider 的参数。在用户界面中，定义一个文本框和一个按钮(XAML 文件 Formula1Demo/GridGroupingUC.xaml):

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
  <TextBlock Margin="5" Padding="4" VerticalAlignment="Center">
    Page:
  </TextBlock>
  <TextBox Margin="5" Padding="4" VerticalAlignment="Center"
    x:Name="textPageNumber" Text="0" />
  <Button Click="OnGetPage">Get Page</Button>
</StackPanel>
```

在代码隐藏中，按钮的 OnGetPage 处理程序访问 ObjectDataProvider，并修改方法的第一个参数。接着调用 Refresh()方法，以便 ObjectDataProvider 请求新页面(代码文件 Formula1Demo/GridGrouping-UC.xaml.cs):

```
private void OnGetPage(object sender, RoutedEventArgs e)
{
```

```
  int page = int.Parse(textPageNumber.Text);
  var odp = (sender as FrameworkElement).FindResource("races")
          as ObjectDataProvider;
  odp.MethodParameters[0] = page;
  odp.Refresh();
}
```

运行应用程序，就会看到分组和行的细节信息，如图 36-19 所示。

图 36-19

36.6.4　实时成型

WPF 4.5 的一个新功能是实时成型。前面介绍了集合视图源及其对排序、过滤和分组的支持。但是如果因为排序、过滤和分组返回不同的结果，而使集合随时间变化，CollectionViewSource 就没有什么帮助了。对于实时成型功能，应使用新接口 ICollectionViewLiveShaping。这个接口定义了属性 CanChangeLiveFiltering、CanChangeLiveGrouping 和 CanChangeLiveSorting，用于检查数据源能否使用实时成型功能。属性 IsLiveFiltering、IsLiveGrouping 和 IsLiveSorting 启用实时成型功能(如果可用)。有了 LiveFilteringProperties、LiveGroupingProperties 和 LiveSortingProperties，就可以定义源中可用于实时过滤、分组和排序的属性。

示例应用程序展示了一级方程式比赛的结果——这次是 2012 年巴塞罗那的比赛——如何变化。

赛手用 Racer 类表示，这个类型只有简单的属性 Name、Team 和 Number，这些属性使用自动属性来实现，因为这个类型的值不会在应用程序运行期间改变(代码文件 LiveShaping/Racer.cs)：

```
public class Racer
{
  public string Name { get; set; }
  public string Team { get; set; }
  public int Number { get; set; }

  public override string ToString()
  {
    return Name;
```

```
    }
}
```

类 Formula1 返回所有参加 2012 年巴塞罗那比赛的赛手(代码文件 LiveShaping/Formula1.cs):

```
public class Formula1
{
  private List<Racer> racers;
  public IEnumerable<Racer> Racers
  {
    get
    {
      return racers ?? (racers = GetRacers());
    }
  }

  private List<Racer> GetRacers()
  {
    return new List<Racer>()
    {
      new Racer { Name="Sebastian Vettel", Team="Red Bull Racing", Number=1 },
      new Racer { Name="Mark Webber", Team="Red Bull Racing", Number=2 },
      new Racer { Name="Jenson Button", Team="McLaren", Number=3 },
      new Racer { Name="Lewis Hamilton", Team="McLaren", Number=4 },
      new Racer { Name="Fernando Alonso", Team="Ferrari", Number=5 },
      new Racer { Name="Felipe Massa", Team="Ferrari", Number=6 },
      new Racer { Name="Michael Schumacher", Team="Mercedes", Number=7 },
      new Racer { Name="Nico Rosberg", Team="Mercedes", Number=8 },
      new Racer { Name="Kimi Raikkonen", Team="Lotus", Number=9 },
      new Racer { Name="Romain Grosjean", Team="Lotus", Number=10 },
      new Racer { Name="Paul di Resta", Team="Force India", Number=11 },
      new Racer { Name="Nico Hülkenberg", Team="Force India", Number=12 },
      new Racer { Name="Kamui Kobayashi", Team="Sauber", Number=14 },
      new Racer { Name="Sergio Perez", Team="Sauber", Number=15 },
      new Racer { Name="Daniel Riccardio", Team="Toro Rosso", Number=16 },
      new Racer { Name="Jean-Eric Vergne", Team="Toro Rosso", Number=17 },
      new Racer { Name="Pastor Maldonado", Team="Williams", Number=18 },

      //... more racers in the source code download
    };
  }
}
```

现在这个示例就更有趣了。LapRacerInfo 类是在 DataGrid 控件中显示的类型，这个类派生于基类 BindableObject，获得了如前所述的 INotifyPropertyChanged 的实现代码。属性 Lap、Position 和 PositionChange 随时间而变化。Lap 给出了赛车当前已跑过的圈数，Position 提供了赛车在特定圈时的位置，PositionChange 给出了赛车在当前圈数与前一圈的位置变化信息。如果赛车的位置没有变化，状态就是 None，如果赛车的位置比上一圈低，状态就是 Up，如果赛车的位置比上一圈高，状态就是 Down，如果赛手退出了比赛，PositionChange 就是 Out。这些信息可以在 UI 中用于不同的表示(代码文件 LiveShaping/LapRacerInfo.cs):

```
public enum PositionChange
{
```

```
    None,
    Up,
    Down,
    Out
  }

  public class LapRacerInfo : BindableObject
  {
    public Racer Racer { get; set; }
    private int lap;
    public int Lap
    {
      get { return lap; }
      set { SetProperty(ref lap, value); }
    }
    private int position;
    public int Position
    {
      get { return position; }
      set { SetProperty(ref position, value); }
    }
    private PositionChange positionChange;
    public PositionChange PositionChange
    {
      get { return positionChange; }
      set { SetProperty(ref positionChange, value); }
    }
  }
```

类 CapChart 包含所有圈和赛手的信息。这个类可以改为访问一个实时 Web 服务，来检索这些信息，然后应用程序就可以显示当前比赛的实时结果。

方法 SetLapInfoForStart 创建 LapRacerInfo 项的初始列表，并在网格 position 上填充赛手的位置。网格 position 是 List<int>集合中添加到 positions 字典中的第一个数字。接着每次调用 NextLap 方法时，lapInfo 集合中的项都会改为一个新位置，并设置 PositionChange 状态信息(代码文件 LiveShaping/LapChar.cs)：

```
  public class LapChart
  {
    private Formula1 f1 = new Formula1();
    private List<LapRacerInfo> lapInfo;
    private int currentLap = 0;
    private const int PostionOut = 999;
    private int maxLaps;
    public LapChart()
    {
      FillPositions();
      SetLapInfoForStart();
    }

    private Dictionary<int, List<int>> positions =
        new Dictionary<int, List<int>>();
    private void FillPositions()
    {
```

```
    positions.Add(18, new List<int> { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 3, 3, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1 });
    positions.Add(5, new List<int> { 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 1, 1, 1, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2 });
    positions.Add(10, new List<int> { 3, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 9, 7, 6,
        6, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
        4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
        4, 4, 4, 4, 4 });
    // more position information with the code download

    maxLaps = positions.Select(p => p.Value.Count).Max() - 1;
  }

  private void SetLapInfoForStart()
  {
    lapInfo = positions.Select(x => new LapRacerInfo
    {
      Racer = f1.Racers.Where(r => r.Number == x.Key).Single(),
      Lap = 0,
      Position = x.Value.First(),
      PositionChange = PositionChange.None
    }).ToList();
  }

  public IEnumerable<LapRacerInfo> GetLapInfo()
  {
    return lapInfo;
  }

  public bool NextLap()
  {
    currentLap++;
    if (currentLap > maxLaps) return false;

    foreach (var info in lapInfo)
    {
      int lastPosition = info.Position;
      var racerInfo = positions.Where(x => x.Key == info.Racer.Number).Single();

      if (racerInfo.Value.Count > currentLap)
      {
        info.Position = racerInfo.Value[currentLap];
      }
      else
      {
        info.Position = lastPosition;
      }
      info.PositionChange = GetPositionChange(lastPosition, info.Position);

      info.Lap = currentLap;
    }
```

```
      return true;
    }

    private PositionChange GetPositionChange(int oldPosition, int newPosition)
    {
      if (oldPosition == PositionOut ||| newPosition == PositionOut)
        return PositionChange.Out;
      else if (oldPosition == newPosition)
        return PositionChange.None;
      else if (oldPosition < newPosition)
        return PositionChange.Down;
      else
        return PositionChange.Up;
    }
  }
```

在主窗口中指定 DataGrid，其中包含一些 DataGridTextColumn 元素，这些元素绑定到 LapRacerInfo 类的属性上，LapRacerInfo 类是从前面所示的集合中返回的。DataTrigger 元素根据赛手的位置与上一圈相比是提高了还是落后了，使用 PositionChange 属性的枚举值，给行定义不同的背景色(XAML 文件 LiveShaping/MainWindow.xaml)：

```
<DataGrid IsReadOnly="True" ItemsSource="{Binding}"
    DataContext="{StaticResource cvs}" AutoGenerateColumns="False">
  <DataGrid.CellStyle>
    <Style TargetType="DataGridCell">
      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="{x:Null}" />
          <Setter Property="BorderBrush" Value="{x:Null}" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </DataGrid.CellStyle>
  <DataGrid.RowStyle>
    <Style TargetType="DataGridRow">
      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="{x:Null}" />
          <Setter Property="BorderBrush" Value="{x:Null}" />
        </Trigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="None">
          <Setter Property="Background" Value="LightGray" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Up">
          <Setter Property="Background" Value="LightGreen" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Down">
          <Setter Property="Background" Value="Yellow" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Out">
          <Setter Property="Background" Value="Red" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
```

```
  </DataGrid.RowStyle>
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Position}" />
    <DataGridTextColumn Binding="{Binding Racer.Number}" />
    <DataGridTextColumn Binding="{Binding Racer.Name}" />
    <DataGridTextColumn Binding="{Binding Racer.Team}" />
    <DataGridTextColumn Binding="{Binding Lap}" />
  </DataGrid.Columns>
</DataGrid>
```

数据触发器参见第 35 章。

用 DataGrid 指定的数据上下文在带有 CollectionViewSource 的窗口资源中找到。该集合视图源绑定到后面用后台代码指定的数据上下文上。这里设置的重要属性是 IsLiveSortingRequested，其值设置为 true，会改变元素在用户界面上的顺序。用于排序的属性是 Position。位置变化时，项会实时重排序：

```
<Window.Resources>
  <CollectionViewSource x:Key="cvs" Source="{Binding}"
      IsLiveSortingRequested="True">
    <CollectionViewSource.SortDescriptions>
      <scm:SortDescription PropertyName="Position" />
    </CollectionViewSource.SortDescriptions>
  </CollectionViewSource>
</Window.Resources>
```

现在，只需要进入后台源代码中，找到设置数据上下文、动态改变实时值的代码段。在主窗口的构造函数中，DataContext 属性设置为 LapRacerInfo 类型的初始集合。接着一个后台任务每隔 3 秒调用一次 NextLap 方法，用新位置修改 UI 中的值。后台任务使用了一个异步的 lambda 表达式。实现代码可以改为从 Web 服务中获得实时数据(代码文件 LiveShaping/MainWindow.xaml.cs)：

```
public partial class MainWindow : Window
{
  private LapChart lapChart = new LapChart();
  public MainWindow()
  {
    InitializeComponent();
    this.DataContext = lapChart.GetLapInfo();

    Task.Run(async () =>
      {
        bool raceContinues = true;
        while (raceContinues)
        {
          await Task.Delay(3000);
          raceContinues = lapChart.NextLap();
        }
      });
  }
}
```

图 36-20 显示了赛手在第 14 圈时的应用程序，领头的赛手是开着法拉利的 Fernando Alonso。

Live Shaping

1	5	Fernando Alonso	Ferrari	14	None
2	18	Pastor Maldonado	Williams	14	None
3	9	Kimi Raikkonen	Lotus	14	None
4	4	Lewis Hamilton	McLaren	14	None
5	8	Nico Rosberg	Mercedes	14	None
6	10	Romain Grosjean	Lotus	14	None
7	1	Sebastian Vettel	Red Bull Racing	14	None
8	3	Jenson Button	McLaren	14	None
9	14	Kamui Kobayashi	Sauber	14	None
10	2	Mark Webber	Red Bull Racing	14	Up
11	11	Paul di Resta	Force India	14	Up
12	17	Jean-Eric Vergne	Toro Rosso	14	Up
13	6	Felipe Massa	Ferrari	14	Up
14	12	Nico Hulkenberg	Force India	14	Up
15	16	Daniel Riccardio	Toro Rosso	14	Up
16	15	Sergio Perez	Sauber	14	Up
17	20	Heikki Kovalainen	Caterham	14	Down
18	24	Timo Glock	Marussia	14	Down
19	25	Charles Pic	Marussia	14	None
20	21	Witali Petrow	Caterham	14	None
21	22	Pedro de la Rosa	HRT	14	None
22	23	Narain Karthikeyan	HRT	14	None
99	19	Bruno Senna	Williams	14	Out
99	7	Michael Schumacher	Mercedes	14	Out

图 36-20

36.7　小结

本章介绍了 WPF 中对于业务应用程序非常重要的一些功能。在清晰而方便地与数据交互操作方面，WPF 的数据绑定功能前进了一大步。可以把.NET 类的任意属性绑定到 WPF 元素的属性上。绑定模式定义了绑定的方向。可以绑定.NET 对象和列表，定义数据模板，从而通过数据模板为.NET 类创建默认的外观。

命令绑定可以把处理程序的代码映射到菜单和工具栏上。还可以用 WPF 进行复制和粘贴，因为这个技术的命令处理程序已经包含在 TextBox 控件中。本章还介绍了其他 WPF 功能，例如使用 DataGrid、CollectionViewSource 进行排序和分组，所有这些功能也可以通过实时成型来完成。

下一章讨论 WPF 的另一个方面：处理文档。

第37章

用 WPF 创建文档

本章要点

- 创建流文档
- 创建固定的文档
- 创建 XPS 文档
- 打印文档

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 显示字体
- 文本效果
- 表
- 流文档
- 创建 XPS
- 打印

37.1 简介

创建文档是 WPF 的一个主要部分。System.Windows.Documents 名称空间支持创建流文档和固定文档。这个名称空间包含的元素可以利用类似于 Word 的方式创建流文档，也可以创建 WYSIWYG(所见即所得)固定文档。

流文档面向屏幕读取；文档的内容根据窗口的大小来排列，如果窗口重置了大小，文档的流就会改变。固定文档主要用于打印和面向页面的内容，其内容总是按照相同的方式排列。

本章讨论如何创建、打印流文档和固定文档，并涵盖 System.Windows.Documents、System.Windows.Xps 和 System.IO.Packaging 名称空间。

37.2　文本元素

要构建文档的内容，需要文档元素。这些元素的基类是 TextElement。这个类定义了字体设置、前景和背景，以及文本效果的常见属性。TextElement 是 Block 类和 Inline 类的基类，这两个类的功能在后面的几节中介绍。

37.2.1　字体

文本的一个重要方面是文本的外观，即字体。通过 TextElement，可以用 FontWeight、FontStyle、FontStretch、FontSize 和 FontFamily 属性指定字体。

- FontWeight——预定义的 FontWeight 值由 FontWeights 类定义，这个类提供的值包括 UltraLight、Light、Medium、Normal、Bold、UltraBold 和 Heavy。
- FontStyle——FontStyle 的值由 FontStyles 类定义，可以是 Normal、Italic 和 Oblique。
- FontStretch——利用 FontStretch 可以指定字体相对于正常宽高比的拉伸程度。FontStretch 指定了预定义的拉伸率从 50%(UltraCondensed)到 200%(UltraExpanded)。在这个范围之间的预定义值是 ExtraCondensed(62.5%)、Condensed(75%)、SemiCondensed(87.5%)、Normal(100%)、SemiExpanded(112.5%)、Expanded(125%)和 ExtraExpanded(150%)。
- FontSize——FontSize 是 double 类型，可以用于指定字体的大小，其单位与设备无关，如英寸、厘米和点。
- FontFamily——利用 FontFamily 可以定义首选字体系列的名称，如 Arial 或 Times New Roman。使用这个属性可以指定一个字体系列名列表，这样，如果某个字体不可用，就使用列表中的下一个字体(如果所选字体和备用字体都不可用，流文档就使用默认的 MessageFontFamily)。还可以从资源中引用字体系列，或者使用 URI 引用服务器上的字体。对于固定的文档，不会出现字体不可用的情况，因为字体是通过文档提供的。

为了了解不同字体的外观，下面的示例 WPF 应用程序包含一个列表框。该列表框为列表中的每一项定义了一个 ItemTemplate。这个模板使用 4 个 TextBlock 元素，这些元素的 FontFamily 绑定到 FontFamily 对象的 Source 属性上。给不同的 TextBlock 元素设置 FontWeight 和 FontStyle(XAML 文件 ShowFonts/ShowFontsWindow.xaml)：

```
<ListBox ItemsSource="{Binding}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" >
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" Text="{Binding Path=Source}" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontStyle="Italic" Text="Italic" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraBold" Text="UltraBold" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraLight" Text="UltraLight" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

在代码隐藏中，数据上下文设置为 System.Windows.Media.Font 类的 SystemFontFamilies 属性值，这会返回所有可用的字体(代码文件 ShowFonts/ShowFontsWindow.xaml.cs)：

```
public partial class ShowFontsWindow : Window
{
  public ShowFontsWindow()
  {
    InitializeComponent();

    this.DataContext = Fonts.SystemFontFamilies;
  }
}
```

运行应用程序，会显示一个很长的列表，其中包含系统字体系列的斜体、黑体、UltraBold 和 UltraLight 样式，如图 37-1 所示。

图 37-1

37.2.2 TextEffect

下面看看 TextEffect，因为它也是所有文档元素共有的。TextEffect 在名称空间 System.Windows.Media 中定义，派生自基类 Animatable，允许生成文本的动画效果。

TextEffect 可以为裁剪区域、前景画笔和变换创建动画效果。利用 PositionStart 和 PositionCount 属性可以指定在文本中应用动画的位置。

要应用文本效果，应设置 Run 元素的 TextEffects 属性。该属性内部指定的 TextEffect 元素定义了前景和变换效果。对于前景，使用名为 brush1 的 SolidColorBrush 画笔，通过 ColorAnimation 元素生成动画效果。转换使用名为 scale1 的 ScaleTransformation，从两个 DoubleAnimation 元素中制作动画效果(XAML 文件 TextEffectsDemo/MainWindow.xaml)。

```
<TextBlock>
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation AutoReverse="True" RepeatBehavior="Forever"
              From="Blue" To="Red" Duration="0:0:16"
              Storyboard.TargetName="brush1"
              Storyboard.TargetProperty="Color" />
          <DoubleAnimation AutoReverse="True"
              RepeatBehavior="Forever"
              From="0.2" To="12" Duration="0:0:16"
              Storyboard.TargetName="scale1"
```

```
                Storyboard.TargetProperty="ScaleX" />
            <DoubleAnimation AutoReverse="True"
                RepeatBehavior="Forever"
                From="0.2" To="12" Duration="0:0:16"
                Storyboard.TargetName="scale1"
                Storyboard.TargetProperty="ScaleY" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </TextBlock.Triggers>
    <Run FontFamily="Segoe UI">
        cn|elements
      <Run.TextEffects>
        <TextEffect PositionStart="0" PositionCount="30">
          <TextEffect.Foreground>
            <SolidColorBrush x:Name="brush1" Color="Blue" />
          </TextEffect.Foreground>
          <TextEffect.Transform>
            <ScaleTransform x:Name="scale1" ScaleX="3" ScaleY="3" />
          </TextEffect.Transform>
        </TextEffect>
      </Run.TextEffects>
    </Run>
  </TextBlock>
```

运行应用程序，会看到大小和颜色的变化，如图 37-2 和图 37-3 所示。

图 37-2

图 37-3

37.2.3　内联

所有内联流内容元素的基类都是 Inline。可以在流文档的段落中使用 Inline 元素。因为在段落中，Inline 元素可以一个跟着一个，所以 Inline 类提供了 PreviousInline 和 NextInline 属性，从一个元素导航到另一个元素，也可以使用 SiblingNextInlines 获取所有同级内联元素的集合。

前面用于输出一些文本的 Run 元素是一个 Inline 元素，它可输出格式化或非格式化的文本，还有许多其他的 Inline 元素。Run 元素后的换行可以用 LineBreak 元素来获得。

Span 元素派生自 Inline 类，它允许组合 Inline 元素。在 Span 的内容中只能包含 Inline 元素。含义明确的 Bold、Hyperlink、Italic 和 Underline 类都派生自 Span，因此允许 Inline 元素和其中的内容具有相同的功能，但对这些元素的操作不同。下面的 XAML 代码说明了 Bold、Italic、Underline 和 LineBreak 的用法，如图 37-4 所示(XAML 文件 FlowDocumentsDemo/FlowDocument1.xaml)。

```
<Paragraph FontWeight="Normal">
  <Span>
    <Span>Normal</Span>
    <Bold>Bold</Bold>
    <Italic>Italic</Italic>
    <LineBreak />
    <Underline>Underline</Underline>
  </Span>
</Paragraph>
```

AnchoredBlock 是一个派生自 Inline 的抽象类，用于把 Block 元素锚定到流内容上。Figure 和 Floater 是派生自 AnchoredBlock 的具体类。因为这两个内联元素在涉及块时比较有趣，所以本章后面讨论它们。

另一个映射到 UI 元素上的 Inline 元素是 InlineUIContainer，它在前面的章节使用过。InlineUIContainer 允许给文档添加所有的 UIElement 对象(如按钮)。下面的代码段给文档添加了一个 InlineUIContainer，其中包含组合框、单选按钮和文本框元素，结果如图 37-5 所示(XAML 文件 FlowDocumentsDemo/FlowDocument2.xaml)。

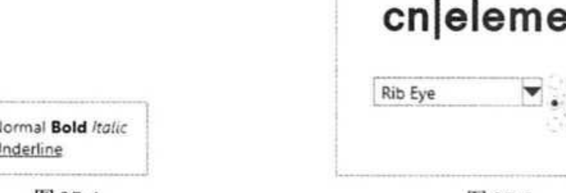


图 37-4　　图 37-5

当然，也可以给 UI 元素添加样式，参见第 35 章。

```
<Paragraph TextAlignment="Center">
  <Span FontSize="36">
    <Italic>cn|elements</Italic>
  </Span>
  <LineBreak />
  <LineBreak />
  <InlineUIContainer>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
```

```
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition />
          <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <ComboBox Width="40" Margin="3" Grid.Row="0">
          <ComboBoxItem>Filet Mignon</ComboBoxItem>
          <ComboBoxItem>Rib Eye</ComboBoxItem>
          <ComboBoxItem>Sirloin</ComboBoxItem>
        </ComboBox>
        <StackPanel Grid.Row="0" Grid.RowSpan="2" Grid.Column="1">
          <RadioButton>Raw</RadioButton>
          <RadioButton>Medium</RadioButton>
          <RadioButton>Well done</RadioButton>
        </StackPanel>
        <TextBox Grid.Row="1" Grid.Column="0" Width="140"></TextBox>
      </Grid>
    </InlineUIContainer>
  </Paragraph>
```

37.2.4 块

Block 是块级元素的抽象基类。块可以把包含其中的元素组合到特定的视图上。所有块通用的属性有 PreviousBlock、NextBlock 和 SiblingBlocks，它们允许从一个块导航到下一个块。在块开始之前设置 BreakPageBefore 换页符和 BreakColumnBefore 换行符。块还使用 BorderBrush 和 BorderThickness 属性定义边框。

派生自 Block 的类有 Paragraph、Section、List、Table 和 BlockUIContainer。BlockUIContainer 类似于 InlineUIContainer，其中也可以添加派生自 UIElement 的元素。

Paragraph 和 Section 是简单的块，其中 Paragraph 包含内联元素；Section 用于组合其他 Block 元素。使用 Paragraph 块可以确定在段落内部或段落之间是否允许添加换页符或换行符。KeepTogether 可用于禁止在段落内部换行，KeepWithNext 尝试把一个段落与下一个段落合并起来。如果段落用换页符或换行符隔开，那么 MinWindowLines 会定义分隔符之后的最小行数，MinOrphanLines 定义分隔符之前的最小行数。

Paragraph 块也允许在段落内部用 TextDecoration 元素装饰文本。预定义的文本装饰由 TextDecoration 定义：Baseline、Overline、Strikethrough 和 Underline。

下面的 XAML 代码显示了多个 Paragraph 元素。一个 Paragraph 元素包含标题，其后的另一个 Paragraph 元素包含属于上述标题的内容。这两个段落通过特性 KeepWithNext 连接起来。把 KeepTogether 设置为 True，也确保包含内容的段落不被隔开，结果如图 37-6 所示(XAML 文件 FlowDocumentsDemo/ParagraphDemo.xaml)。

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              ColumnWidth="300" FontSize="16" FontFamily="Georgia">
  <Paragraph FontSize="36">
    <Run>Lyrics</Run>
  </Paragraph>
  <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
    <Bold>
      <Run>Mary had a little lamb</Run>
```

```
      </Bold>
    </Paragraph>
    <Paragraph KeepTogether="True">
      <Run>Mary had a little lamb,</Run>
      <LineBreak />
      <Run>little lamb, little lamb,</Run>
      <LineBreak />
      <Run>Mary had a little lamb,</Run>
      <LineBreak />
      <Run>whose fleece was white as snow.</Run>
      <LineBreak />
      <Run>And everywhere that Mary went,</Run>
      <LineBreak />
      <Run>Mary went, Mary went,</Run>
      <LineBreak />
      <Run>and everywhere that Mary went,</Run>
      <LineBreak />
      <Run>the lamb was sure to go.</Run>
    </Paragraph>
    <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
      <Bold>
        <Run>Humpty Dumpty</Run>
      </Bold>
    </Paragraph>
    <Paragraph KeepTogether="True">
      <Run>Humpty dumpty sat on a wall</Run>
      <LineBreak />
      <Run>Humpty dumpty had a great fall</Run>
      <LineBreak />
      <Run>All the King's horses</Run>
      <LineBreak />
      <Run>And all the King's men</Run>
      <LineBreak />
      <Run>Couldn't put Humpty together again</Run>
    </Paragraph>
</FlowDocument>
```

结果如图37-6所示。

Lyrics

Mary had a little lamb

Mary had a little lamb,
little lamb, little lamb,
Mary had a little lamb,
whose fleece was white as snow.
And everywhere that Mary went,
Mary went, Mary went,
and everywhere that Mary went,
the lamb was sure to go.

Humpty Dumpty

Humpty dumpty sat on a wall
Humpty dumpty had a great fall
All the King's horses
And all the King's men
Couldn't put Humpty together again

图 37-6

37.2.5　列表

List类用于创建无序或有序的文本列表。List通过设置MarkerStyle属性，定义了其列表项的项目符号样式。MarkerStyle 的类型是TextMarkerStyle，它可以是数字(Decimal)、字母(LowerLatin 和 UpperLatin)、罗马数字(LowerRoman 和 UpperRoman)或图片(Disc、Circle、Square、Box)。List 只能包含 ListItem 元素，ListItem 只能包含 Block 元素。

❑ Monday
❑ Tuesday
❑ Wednesday

图 37-7

用XAML 定义如下列表，结果如图37-7 所示(XAML 文件 FlowDocumentsDemo/ListDemo.xaml)。

```
<List MarkerStyle="Square">
  <ListItem>
    <Paragraph>Monday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Tuesday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Wednesday</Paragraph>
  </ListItem>
</List>
```

37.2.6　表

Table 类非常类似于第 35 章讨论的 Grid 类，它也定义行和列。下面的例子说明了如何使用 Table 创建 FlowDocument。要创建表，可以给 Columns 属性添加 TableColumn 对象。而利用 TableColumn 可以指定宽度和背景。

Table 还包含 TableRowGroup 对象。TableRowGroup 有一个 Rows 属性，可以在 Rows 属性中添加 TableRow 对象。TableRow 类定义了一个 Cells 属性，在 Cells 属性中可以添加 TableCell 对象。TableCell 对象可以包含任意 Block 元素。这里使用了一个 Paragraph 元素，其中包含 Inline 元素 Run(代码文件 TableDemo/MainWindow.xaml)：

```
var doc = new FlowDocument();
var t1 = new Table();
t1.Columns.Add(new TableColumn
{ Width = new GridLength(50, GridUnitType.Pixel) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });

var titleRow = new TableRow { Background = Brushes.LightBlue };
var titleCell = new TableCell
{ ColumnSpan = 3, TextAlignment = TextAlignment.Center };
titleCell.Blocks.Add(
    new Paragraph(new Run("Formula 1 Championship 2011")
    { FontSize=24, FontWeight = FontWeights.Bold }));
titleRow.Cells.Add(titleCell);

var headerRow = new TableRow
```

```
{ Background = Brushes.LightGoldenrodYellow };
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Pos"))
{ FontSize = 14, FontWeight=FontWeights.Bold}));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Name"))
{ FontSize = 14, FontWeight = FontWeights.Bold }));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Points"))
{ FontSize = 14, FontWeight = FontWeights.Bold }));

var rowGroup = new TableRowGroup();
rowGroup.Rows.Add(titleRow);
rowGroup.Rows.Add(headerRow);

string[][] results = new string[][]
{
  new string[] { "1.", "Sebastian Vettel", "392" },
  new string[] { "2.", "Jenson Button", "270" },
  new string[] { "3.", "Mark Webber", "258" },
  new string[] { "4.", "Fernando Alonso", "257" },
  new string[] { "5.", "Lewis Hamilton", "227"}
};

List<TableRow> rows = results.Select(row =>
  {
    var tr = new TableRow();
    foreach (var cell in row)
    {
      tr.Cells.Add(new TableCell(new Paragraph(new Run(cell))));
    }
    return tr;
  }).ToList();
  rows.ForEach(r => rowGroup.Rows.Add(r));

  t1.RowGroups.Add(rowGroup);
  doc.Blocks.Add(t1);

  reader.Document = doc;
```

运行应用程序，会显示一个格式化好的表，如图37-8所示。

Formula 1 Championship
2011

Pos	Name	Points
1.	Sebastian Vettel	392
2.	Jenson Button	270
3.	Mark Webber	258
4.	Fernando Alonso	257
5.	Lewis Hamilton	227

图37-8

37.2.7 块的锚定

既然学习了Inline和Block元素，就可以使用AnchoredBlock类型的Inline元素合并它们。AnchoredBlock是一个抽象基类，它有两个具体的实现方式Figure和Floater。

Floater使用属性HorizontalAlignment和Width同时显示其内容和主要内容。

从上面的例子开始，添加一个包含 Floater 的新段落。这个 Floater 采用左对齐方式，宽度为 120。如图 37-9 所示，下一个段落将环绕它(XAML 文件 FlowDocumentsDemo/ParagraphKeepTogether.xaml)。

```
<Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
  <Bold>
    <Run>Mary had a little lamb</Run>
  </Bold>
</Paragraph>
<Paragraph>
  <Floater HorizontalAlignment="Left" Width="120">
    <Paragraph Background="LightGray">
      <Run>Sarah Josepha Hale</Run>
    </Paragraph>
  </Floater>
</Paragraph>
<Paragraph KeepTogether="True">
  <Run>Mary had a little lamb</Run>
  <LineBreak />
  <!-- ... -->
</Paragraph>
```

Mary had a little lamb

Sarah
Josepha Hale

Mary had a little lamb,
little lamb, little lamb,
Mary had a little lamb,
whose fleece was white as snow.
And everywhere that Mary went,
Mary went, Mary went,
and everywhere that Mary went,
the lamb was sure to go.

图 37-9

Figure 采用水平和垂直对齐方式，可以锚定到页面、内容、列或段落上。下面代码中的 Figure 锚定到页面中心处，但水平和垂直方向有偏移。设置 WrapDirection，使左列和右列环绕着图片，图 37-10 显示了环绕的结果(XAML 文件 FlowDocumentsDemo/FigureAlignment.xaml)。

```
<Paragraph>
  <Figure HorizontalAnchor="PageCenter" HorizontalOffset="20"
      VerticalAnchor="PageCenter" VerticalOffset="20" WrapDirection="Both" >
    <Paragraph Background="LightGray" FontSize="24">
      <Run>Lyrics Samples</Run>
    </Paragraph>
  </Figure>
</Paragraph>
```

Mary had a little lamb

Mary had a little lamb,
little lamb, little lamb,
Mary had a little lamb,
whose fleece was white as snow.
And everywhere that Mary went,
Mary went, Mary went,
and everywhere that Mary went,
the lamb was sure to go.

Lyrics Samples

Humpty Dumpty

Humpty dumpty sat on a wall
Humpty dumpty had a great fall
All the King's horses
And all the King's men
Couldn't put Humpty
together again

图 37-10

Figure 和 Floater 都用于添加不在主流中的内容，尽管这两个功能看起来类似，但它们的特征大不相同。表 37-1 列出了 Figure 和 Floater 之间的区别。

表 37-1

特　征	Floater	Figure
位置	Floater 不能定位，在空间可用时显示它	Figure 可以用水平和垂直锚点来定位，它可以停靠在页面、内容、列或段落上
宽度	Floater 只能放在一列中。如果它设置的宽度大于列宽，就忽略它	Figure 可以跨越多列。Figure 的宽度可以设置为半页或两列
分页	如果 Floater 高于列高，就分解 Floater，放到下一列或下一页上	如果 Figure 大于列高，就只显示列中的部分，其他内容会丢失

37.3 流文档

前面介绍了所有 Inline 和 Block 元素，现在我们知道应该把什么内容放在流文档中。FlowDocument 类可以包含 Block 元素，Block 元素可以包含 Block 或 Inline 元素，这取决于 Block 的类型。

FlowDocument 类的一个主要功能是把流分解为多个页面。这是通过 FlowDocument 实现的 IDocumentPaginatorSource 接口实现。

FlowDocument 的其他选项包括建立默认字体、前景画笔和背景画笔，以及配置页面和列的大小。下面 FlowDocument 的 XAML 代码定义了默认字体、字体大小、列宽和列之间的标尺：

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ColumnWidth="300" FontSize="16" FontFamily="Georgia"
    ColumnRuleWidth="3" ColumnRuleBrush="Violet">
```

现在需要一种方式来查看文档。以下列表描述了几个查看器：

- RichTextBox——一个简单的查看器，还允许编辑(只要 IsReadOnly 属性没有设置为 true)。RichTextBox 不在多列中显示文档，而以滚动模式显示文档。这类似于 Microsoft Word 中的 Web 布局。把 HorizontalScrollbarVisibility 设置为 ScrollbarVisibility.Auto，就可以启用滚动条。

- FlowDocumentScrollViewer——一个读取器，只能读取文档，不能编辑文档，这个读取器允许放大文档，工具栏中的滑块可以使用 IsToolbarEnabled 属性，来启用其缩放功能。设置 CanIncreaseZoom、CanDecreaseZoom、MinZoom 和 MaxZoom 都允许设置缩放功能。
- FlowDocumentPageViewer——给文档分页的查看器。使用这个查看器，不仅可以通过其工具栏放大文档，还可以在页面之间切换。
- FlowDocumentReader——这个查看器合并了 FlowDocumentScrollViewer 和 FlowDocument-PageViewer 的功能，它支持不同的查看模式，这些模式可以在工具栏中设置，或者使用 FlowDocumentReaderViewingMode 类型的 ViewingMode 属性来设置。这个枚举的值可以是 Page、TwoPage 和 Scroll，也可以根据需要禁用查看模式。

演示流文档的示例应用程序定义了几个读取器，以便动态选择其中一个读取器。在 Grid 元素中包含 FlowDocumentReader、RichTextBox、FlowDocumentScrollViewer 和 FlowDocumentPageViewer。所有的读取器都把 Visibility 属性设置为 Collapsed，这样在启动时，就不会显示任何读取器。网格中的第一个子元素是一个组合框，它允许用户选择活动的读取器。组合框的 ItemsSource 属性绑定到 Readers 属性上，来显示读取器列表。选择了一个读取器后，就调用 OnReaderSelectionChanged 方法(XAML 文件 FlowDocumentsDemo/MainWindow.xaml)：

```
<Grid x:Name="grid1">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <ComboBox ItemsSource="{Binding Readers}"Grid.Row="0" Grid.Column="0"
      Margin="4" SelectionChanged="OnReaderSelectionChanged"
     SelectedIndex="0">
    <ComboBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding Name}" />
        </StackPanel>
      </DataTemplate>
    </ComboBox.ItemTemplate>
  </ComboBox>
  <Button Grid.Column="1" Margin="4" Padding="3" Click="OnOpenDocument">
      Open Document</Button>
  <FlowDocumentReader ViewingMode="TwoPage" Grid.Row="1"
      Visibility="Collapsed" Grid.ColumnSpan="2" />
  <RichTextBox IsDocumentEnabled="True" HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto" Visibility="Collapsed"
      Grid.Row="1" Grid.ColumnSpan="2" />
  <FlowDocumentScrollViewer Visibility="Collapsed" Grid.Row="1"
      Grid.ColumnSpan="2" />
  <FlowDocumentPageViewer Visibility="Collapsed" Grid.Row="1"
```

```
        Grid.ColumnSpan="2" />
</Grid>
```

MainWindow类的Readers属性调用GetReaders方法，把读取器返回给ComboBox数据绑定。GetReaders方法返回赋予documentReaders变量的列表。万一没有指定documentReaders，就使用LogicalTreeHelper类获取网格grid1中的所有流文档读取器。由于流文档读取器没有基类，也没有所有读取器都实现的接口，因此LogicalTreeHelper查找类型为FrameworkElement、有Document属性的所有元素。所有流文档读取器都有Document属性。对于每个读取器，用Name和Instance属性创建一个新的匿名对象。Name属性用于显示在组合框中，允许用户选择活动的读取器，Instance属性包含对读取器的引用，如果读取器应是活动的，就显示它(代码文件FlowDocumentsDemo/MainWindow.xaml.cs)：

```
public IEnumerable<object> Readers
{
  get
  {
   return GetReaders();
  }
}

private List<object> documentReaders = null;
private IEnumerable<object> GetReaders()
{
  return documentReaders ?? (documentReaders =
    LogicalTreeHelper.GetChildren(grid1).OfType<FrameworkElement>()
      .Where(el => el.GetType().GetProperties()
          .Where(pi => pi.Name == "Document").Count() > 0)
      .Select(el =>new
      {
        Name = el.GetType().Name,
        Instance = el
      }).Cast<object>().ToList());
}
```

用户选择一个流文档读取器时，就调用OnReaderSelectionChanged方法。引用这个方法的XAML代码如上所示。在这个方法中，把以前选择的流文档读取器设置为折叠，使之隐藏起来，并把变量activeDocumentReader设置为选中的读取器：

```
private void OnReaderSelectionChanged(object sender,
                                      SelectionChangedEventArgs e)
{
  dynamic item = (sender as ComboBox).SelectedItem;

  if (activedocumentReader != null)
  {
   activedocumentReader.Visibility = Visibility.Collapsed;
  }
  activedocumentReader = item.Instance;
}

private dynamic activedocumentReader = null;
```

示例代码使用了 dynamic 关键字——activeDocumentReader 变量声明为 dynamic 类型。使用 dynamic 关键字是因为，ComboBox 中的 SelectedItem 会返回 FlowDocumentReader、FlowDocumentScrollViewer、FlowDocumentPageViewer 或 RichTextBox。所有这些类型都是流文档读取器，它们都提供了 FlowDocument 类型的 Document 属性。但是，要定义这个属性，并没有通用的基类或接口。dynamic 关键字允许从同一个变量中访问这些不同的类型，并使用 Document 属性。dynamic 关键字详见第 12 章。

用户单击按钮，打开文档时，就会调用 OnOpenDocument 方法。在这个方法中，使用 XamlReader 类加载选中的 XAML 文件，如果读取器返回 FlowDocument(此时 XAML 的根元素是 FlowDocument)，就给 activeDocumentReader 的 Document 属性赋值，把 Visibility 设置为 visible：

```
private void OnOpenDocument(object sender, RoutedEventArgs e)
{
  try
  {
    var dlg = new OpenFileDialog();
    dlg.DefaultExt = "*.xaml";
    dlg.InitialDirectory = Environment.CurrentDirectory;
    if (dlg.ShowDialog() == true)
    {
      using (FileStream xamlFile = File.OpenRead(dlg.FileName))
      {
        var doc = XamlReader.Load(xamlFile) as FlowDocument;
        if (doc != null)
        {
          activedocumentReader.Document = doc;
          activedocumentReader.Visibility = Visibility.Visible;
        }
      }
    }
  }
  catch (XamlParseException ex)
  {
    MessageBox.Show(string.Format("Check content for a Flow document, {0}",
         ex.Message));
  }
}
```

运行的应用程序如图 37-11 所示。在该图的流文档中，FlowDocumentReader 采用 TwoPage 模式。

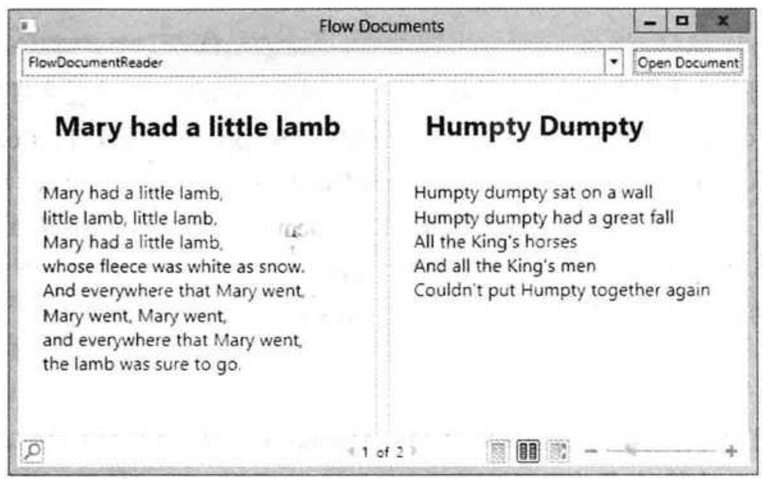


图 37-11

37.4　固定文档

无论固定文档在哪里复制或使用，它总是定义相同的外观、相同的分页方式，并使用相同的字体。WPF 定义了用于创建固定文档的 FixedDocument 类，和用于查看固定文档的 DocumentViewer 类。

本章使用一个示例应用程序，通过编程方式创建一个固定文档，该程序要求用户输入一个用于创建固定文档的菜单规划。菜单规划的数据就是固定文档的内容。图 37-12 显示了这个应用程序的主用户界面，用户可以在其中用 DatePicker 类选择某一天，在 DataGrid 中输入一周的菜单，再单击 Create Doc 按钮，新建一个 FixedDocument。这个应用程序使用 Page 对象在 NavigationWindow 中导航。单击 Create Doc 按钮会导航到一个包含固定文档的新页面上。

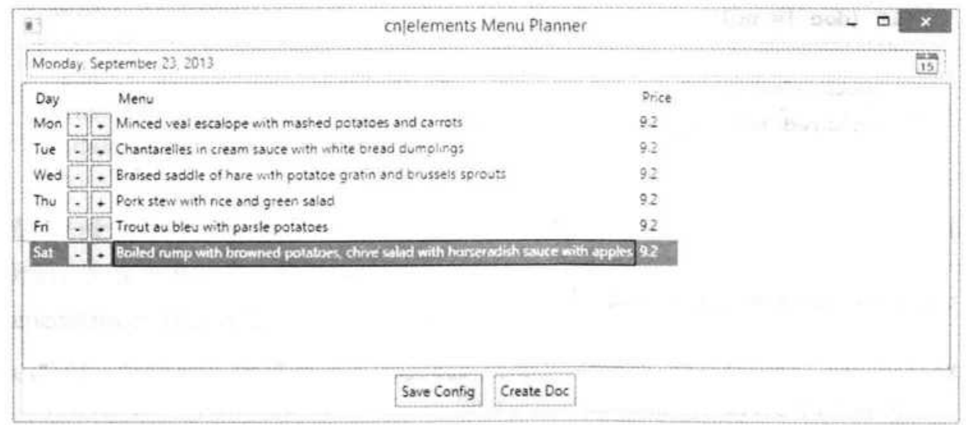


图 37-12

Create Doc 按钮的事件处理程序 OnCreateDoc 导航到一个新页面上。为此，处理程序实例化新页面 DocumentPage。这个页面包含一个 NavigationService_LoadCompleted 处理程序，把它赋予 NavigationService 的 LoadCompleted 事件。在这个处理程序中，新页面可以访问传送给页面的内容。接着调用 Navigate()方法导航到 page2。新页面接收对象 menus，该对象包含了构建固定页面所需的所有菜单信息。menus 变量的类型是 ObservableCollection<MenuEntry>(代码文件 CreateXps/MenuPlannerPage.xaml.cs)。

```
private void OnCreateDoc(object sender, RoutedEventArgs e)
{
  if (menus.Count == 0)
  {
    MessageBox.Show("Select a date first", "Menu Planner",
        MessageBoxButton.OK);
    return;
  }
  var page2 = new DocumentPage();
  NavigationService.LoadCompleted +=
      page2.NavigationService_LoadCompleted;
  NavigationService.Navigate(page2, menus);
}
```

在 DocumentPage 中，使用 DocumentViewer 获取对固定文档的读取访问权限。固定文档在 NavigationService_LoadCompleted()方法中创建。在这个事件处理程序中，从第一个页面传递的数据通过 NavigationEventArgs 的 ExtraData 属性接收。

把接收到的 ObservableCollection<MenuEntry>赋予 menus 变量，该变量用于构建固定页面(代码文件 CreateXps/DocumentPage.xaml.cs)：

```
internal void NavigationService_LoadCompleted(object sender,
    NavigationEventArgs e)
{
  menus = e.ExtraData as ObservableCollection<MenuEntry>;
  fixedDocument = new FixedDocument();
  var pageContent1 = new PageContent();
  fixedDocument.Pages.Add(pageContent1);
  var page1 = new FixedPage();
  pageContent1.Child = page1;
  page1.Children.Add(GetHeaderContent());
  page1.Children.Add(GetLogoContent());
  page1.Children.Add(GetDateContent());
  page1.Children.Add(GetMenuContent());
  viewer.Document = fixedDocument;
  NavigationService.LoadCompleted -= NavigationService_LoadCompleted;
}
```

固定文档用 FixedDocument 类创建。FixedDocument 元素只包含可通过 Pages 属性访问的 PageContent 元素。PageContent 元素必须按它们显示在页面上的顺序添加到文档中。PageContent 定义了单个页面的内容。

PageContent 有一个 Child 属性，因此可以把 PageContent 关联到 FixedPage 上。在 FixedPage 上可以把 UIElement 类型的元素添加到 Children 集合中。在这个集合中可以添加前两章介绍的所有元素，包括 TextBlock，它本身可以包含 Inline 和 Block 元素。

在示例代码中，FixedPage 的子元素用辅助方法 GetHeaderContent()、GetLogoContent()、GetDateContent()和 GetMenuContent()创建。

GetHeaderContent()方法创建一个 TextBlock，并返回它。给 TextBlock 添加 Inline 元素 Bold，又给 Bold 添加 Run 元素。Run 元素包含文档的标题文本。利用 FixedPage.SetLeft()和 FixedPage.SetTop()，定义 TextBox 在固定页面中的位置。

```
private static UIElement GetHeaderContent()
{
  var text1 = new TextBlock
  {
    FontFamily = new FontFamily("Segoe UI"),
    FontSize = 34,
    HorizontalAlignment = HorizontalAlignment.Center
  };
  text1.Inlines.Add(new Bold(new Run("cn|elements")));
  FixedPage.SetLeft(text1, 170);
  FixedPage.SetTop(text1, 40);
  return text1;
}
```

GetLogoContent()方法在固定文档中使用 RadialGradientBrush 添加一个 Ellipse 形状的徽标：

```
private static UIElement GetLogoContent()
{
  var ellipse = new Ellipse
  {
    Width = 90,
    Height = 40,
    Fill = new RadialGradientBrush(Colors.Yellow, Colors.DarkRed)
  };
  FixedPage.SetLeft(ellipse, 500);
  FixedPage.SetTop(ellipse, 50);
  return ellipse;
}
```

GetDateContent()方法访问 menus 集合，把一个日期范围添加到文档中：

```
private UIElement GetDateContent()
{
  Contract.Requires(menus != null);
  Contract.Requires(menus.Count > 0);

  string dateString = String.Format("{0:d} to {1:d}",
     menus[0].Day, menus[menus.Count - 1].Day);
  var text1 = new TextBlock
  {
    FontSize = 24,
    HorizontalAlignment = HorizontalAlignment.Center
  };
  text1.Inlines.Add(new Bold(new Run(dateString)));
  FixedPage.SetLeft(text1, 130);
  FixedPage.SetTop(text1, 90);
  return text1;
}
```

最后，GetMenuContent()方法创建并返回一个 Grid 控件，这个网格中的列和行包含日期、菜单和价格信息：

```
private UIElement GetMenuContent()
{
  var grid1 = new Grid { ShowGridLines = true };
```

```
    grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width= new GridLength(50)});
    grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width = new GridLength(300)});
    grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width = new GridLength(70) });
    for (int i = 0; i < menus.Count; i++)
    {
      grid1.RowDefinitions.Add(new RowDefinition
      { Height = new GridLength(40) });
      var t1 = new TextBlock(new Run(String.Format(
                            "{0:ddd}", menus[i].Day)));
      var t2 = new TextBlock(new Run(menus[i].Menu));
      var t3 = new TextBlock(new Run(menus[i].Price.ToString()));
      var textBlocks = new TextBlock[] { t1, t2, t3 };

      for (int column = 0; column < textBlocks.Length; column++)
      {
        textBlocks[column].VerticalAlignment = VerticalAlignment.Center;
        textBlocks[column].Margin = new Thickness(5, 2, 5, 2);
        Grid.SetColumn(textBlocks[column], column);
        Grid.SetRow(textBlocks[column], i);
        grid1.Children.Add(textBlocks[column]);
      }
    }
    FixedPage.SetLeft(grid1, 100);
    FixedPage.SetTop(grid1, 140);
    return grid1;
  }
```

运行应用程序，所创建的固定文档如图 37-13 所示。

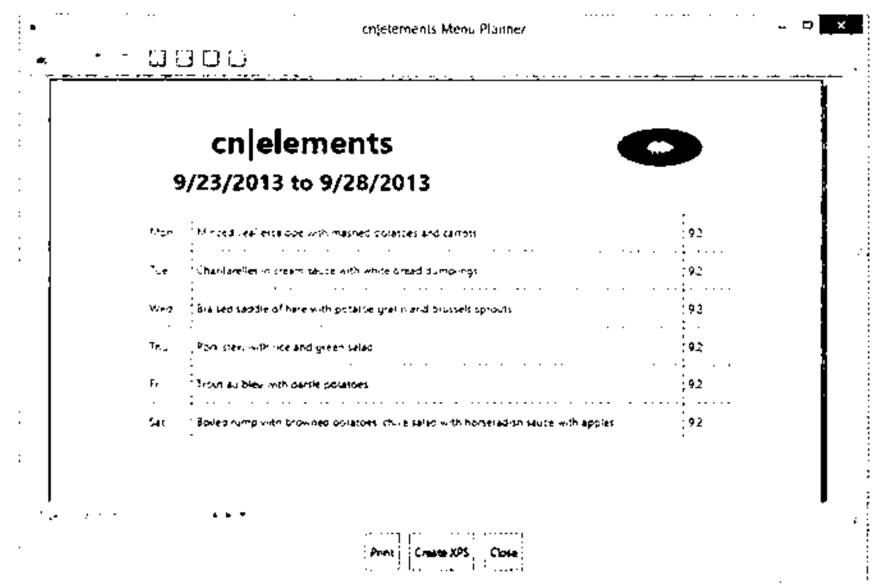


图 37-13

37.5 XPS文档

使用Microsoft Word，可以把文档另存为PDF或XPS文件。XPS是XML纸张规范(XML Paper Specification)，是WPF的一个子集。Windows包含一个XPS读取器。

.NET在System.Windows.Xps、System.Windows.Xps.Packaging和System.IO.Packaging名称空间中包含读写XPS文档的类和接口。

因为XPS以ZIP文件格式打包，所以很容易把扩展名为.xps文件重命名为.zip，打开该归档文件，来分析XPS文档。

XPS文件需要在.zip文档中有XML纸张规范(可从http://www.microsoft.com/ whdc/xps/xpsspec.mspx上下载)定义的特定结构。这个结构基于OPC(Open Packaging Convention，开放打包约定)，Word文档(OOXML或Office Open XML)也基于OPC。在这个文件中，可以包含用于元数据、资源(如字体和图片)和文档本身的不同文件夹。在XPS文档的文档文件夹中，可以找到表示XAML的XPS子集的XAML代码。

要创建XPS文档，可使用System.Windows.Xps.Packaging名称空间中的XpsDocument类。要使用这个类，也需要引用程序集ReachFramework。通过这个类可以给文档添加缩略图(AddThumbnail())和固定文档序列(AddFixedDocumentSequence())，还可以给文档加上数字签名。固定文档序列使用IXpsFixedDocumentSequenceWriter接口写入，该接口使用IXpsFixedDocumentWriter在序列中写入文档。

如果FixedDocument已经存在，写入XPS文档就有一个更简单的方法。不需要添加每个资源和每个文档页，而可以使用System.Windows.Xps名称空间中的XpsDocumentWriter类。要使用这个类，必须引用System.Printing程序集。

下面的代码段包含一个创建XSP文档的处理程序。首先创建用于菜单规划的文件名，它使用星期几和名称menuplan。星期几用GregorianCalender类来计算。接着打开SaveFileDialog，让用户覆盖已创建的文件名，并选择在其中存储文件的目录。SaveFileDialog类在名称空间Microsoft.Win32中定义，它封装本地文件对话框。接着新建一个XpsDocument，其中将文件名传送给构造函数。因为XPS文件使用ZIP格式压缩内容，所以使用CompressionOption可以指定该压缩是在时间上还是空间上进行优化。

之后使用静态方法XpsDocument.CreateXpsDocumentWriter()创建一个XpsDocumentWriter。重载XpsDocumentWriter的Write()方法，从而接受不同的内容或将内容部分写入文档中。Write()方法可接受的选项有FixedDocumentSequence、FixedDocument、FixedPage、string和DocumentPaginator。在示例代码中，仅传送了前面创建的fixedDocument：

```
private void OnCreateXPS(object sender, RoutedEventArgs e)
{
  var c = new GregorianCalendar();
  int weekNumber = c.GetWeekOfYear(menus[0].Day,
     CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
  string fileName = String.Format("menuplan{0}", weekNumber);
  var dlg = new SaveFileDialog
  {
    FileName = fileName,
    DefaultExt = "xps",
    Filter = "XPS Documents|*.xps|All Files|*.*",
```

```
    AddExtension = true
  };
  if (dlg.ShowDialog() == true)
  {
    var doc = new XpsDocument(dlg.FileName, FileAccess.Write,
                              CompressionOption.Fast);
    XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(doc);
    writer.Write(fixedDocument);
    doc.Close();
  }
}
```

运行应用程序，以存储 XPS 文档，就可以用 XPS 查看器查看文档，如图 37-14 所示。

图 37-14

还可以给 XpsDocumentWriter 的一个重载 Write()方法传递 Visual，Visual 是 UIElement 的基类，因此可以给写入器传递任意 UIElement，从而方便地创建 XPS 文档。这个功能在下面的打印示例中使用。

37.6　打印

用 DocumentViewer 打印显示在屏幕上的 FixedDocument，最简单的方法是使用关联到该文档上的 DocumentViewer 的 Print()方法。对于菜单规划应用程序，这都在 OnPrint 处理程序中完成。DocumentViewer 的 Print()方法会打开 PrintDialog，把关联的 FixedDocument 发送给选中的打印机(代码文件 CreateXps/DocumentPage.xaml.cs)：

```
private void OnPrint(object sender, RoutedEventArgs e)
{
  viewer.Print();
}
```

37.6.1 用 PrintDialog 打印

如果希望更多地控制打印过程，就可以实例化 PrintDialog，并用 PrintDocument()方法打印文档。PrintDocument()方法需要把 DocumentPaginator 作为第一个参数。FixedDocument 通过 DocumentPaginator 属性返回一个 DocumentPaginator 对象。第二个参数定义了当前打印机在“打印机”对话框中为打印作业显示的字符串：

```
var dlg = new PrintDialog();
if (dlg.ShowDialog() == true)
{
  dlg.PrintDocument(fixedDocument.DocumentPaginator, "Menu Plan");
}
```

37.6.2 打印可见元素

创建 UIElement 对象也很简单。下面的 XAML 代码定义了一个椭圆、一个矩形和一个用两个椭圆元素表示的按钮。利用该按钮的 Click 处理程序 OnPrint()，会启动可见元素的打印作业(XAML 文件 PrintingDemo/MainWindow.xaml)：

```
<Canvas x:Name="canvas1">
  <Ellipse Canvas.Left="10" Canvas.Top="20" Width="180" Height="60"
      Stroke="Red" StrokeThickness="3" >
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Offset="0" Color="LightBlue" />
        <GradientStop Offset="1" Color="DarkBlue" />
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Rectangle Width="180" Height="90" Canvas.Left="50" Canvas.Top="50">
    <Rectangle.LayoutTransform>
      <RotateTransform Angle="30" />
    </Rectangle.LayoutTransform>
    <Rectangle.Fill>
      <LinearGradientBrush>
        <GradientStop Offset="0" Color="Aquamarine" />
        <GradientStop Offset="1" Color="ForestGreen" />
      </LinearGradientBrush>
    </Rectangle.Fill>
    <Rectangle.Stroke>
      <LinearGradientBrush>
        <GradientStop Offset="0" Color="LawnGreen" />
        <GradientStop Offset="1" Color="SeaGreen" />
      </LinearGradientBrush>
    </Rectangle.Stroke>
  </Rectangle>
  <Button Canvas.Left="90" Canvas.Top="190" Content="Print" Click="OnPrint">
    <Button.Template>
      <ControlTemplate TargetType="Button">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
```

```
            </Grid.RowDefinitions>
            <Ellipse Grid.Row="0" Grid.RowSpan="2" Width="60"
               Height="40" Fill="Yellow" />
            <Ellipse Grid.Row="0" Width="52" Height="20"
               HorizontalAlignment="Center">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                  <GradientStop Color="White" Offset="0" />
                  <GradientStop Color="Transparent" Offset="0.9" />
                </LinearGradientBrush>
              </Ellipse.Fill>
            </Ellipse>
            <ContentPresenter Grid.Row="0" Grid.RowSpan="2"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />
          </Grid>
        </ControlTemplate>
      </Button.Template>
    </Button>
  </Canvas>
```

在 OnPrint()处理程序中，调用 PrintDialog 的 PrintVisual()方法可启动打印作业。PrintVisual()接受派生自 Visual 基类的任意对象(代码文件 PrintingDemo/MainWindow.xaml.cs)：

```
private void OnPrint(object sender, RoutedEventArgs e)
{
    var dlg = new PrintDialog();
    if (dlg.ShowDialog() == true)
    {
        dlg.PrintVisual(canvas1, "Print Demo");
    }
}
```

为了通过编程方式来打印，而无须用户干涉，System.Printing 名称空间中的 PrintDialog 类可用于创建一个打印作业，并调整打印设置。LocalPrintServer 类提供了打印队列的信息，并用 DefaultPrintQueue 属性返回默认的 PrintQueue。使用 PrintTicket 可以配置打印作业。PrintQueue.DefaultPrintTicket 返回与队列关联的默认 PrintTicket。PrintQueue 的 GetPrintCapabilities()方法返回打印机的功能，根据该功能可以配置 PrintTicket，如下面的代码段所示。配置完 PrintTicket 后，静态方法 PrintQueue.CreateXpsDocumentWriter()返回一个 XpsDocumentWriter 对象。XpsDocumentWriter 类以前用于创建 XPS 文档，也可以使用它启动打印作业。XpsDocumentWriter 的 Write()方法不仅接受 Visual 或 FixedDocument 作为第一个参数，还接受 PrintTicket 作为第二个参数。如果用第二个参数传递 PrintTicket，写入器的目标就是与对应标记关联的打印机，因此写入器把打印作业发送给打印机。

```
var printServer = new LocalPrintServer();
PrintQueue queue = printServer.DefaultPrintQueue;
PrintTicket ticket = queue.DefaultPrintTicket;
PrintCapabilities capabilities =
    queue.GetPrintCapabilities(ticket);
if (capabilities.DuplexingCapability.Contains(
    Duplexing.TwoSidedLongEdge))
  ticket.Duplexing = Duplexing.TwoSidedLongEdge;
```

```
if (capabilities.InputBinCapability.Contains(InputBin.AutoSelect))
  ticket.InputBin = InputBin.AutoSelect;
if (capabilities.MaxCopyCount > 3)
  ticket.CopyCount = 3;
if (capabilities.PageOrientationCapability.Contains(
     PageOrientation.Landscape))
  ticket.PageOrientation = PageOrientation.Landscape;
if (capabilities.PagesPerSheetCapability.Contains(2))
  ticket.PagesPerSheet = 2;
if (capabilities.StaplingCapability.Contains(Stapling.StapleBottomLeft))
  ticket.Stapling = Stapling.StapleBottomLeft;
XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(queue);
writer.Write(canvas1, ticket);
```

37.7 小结

本章学习了如何把 WPF 功能用于文档，如何创建根据屏幕大小自动调整的流文档，以及如何创建外观总是不变的固定文档。我们还讨论了如何打印文档，如何把可见元素发送给打印机。

第 38 章继续讨论 WPF 技术，并揭示它如何与 Windows Store 应用程序一起使用。

第38章

Windows Store 应用程序：用户界面

本章要点

- Windows Store 应用程序和 Windows 桌面应用程序的区别
- 定义应用程序条
- 在页面之间导航
- 响应布局的变化
- 使用存储器和选择器
- 创建磁贴

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章的代码只包含一个大示例，它展示了本章的各个方面：

- 菜单卡片(Menu Card)

38.1 概述

阅读了第 31 章，就会熟悉 Windows Store 应用程序的基础知识，以及 Windows 运行库如何关联到.NET。掌握第 31 章介绍的基础知识，就可以开始编写 Windows Store 应用程序了。本章介绍 Windows Store 应用程序的设计原则和 WPF 不支持的一些特殊 XAML 功能，演示操作 Windows Store 应用程序的几个方面，例如响应布局的变化，使用 Windows Runtime 存储 API 和文件选择器读写文件，使用协议与其他应用程序通信等。

除了第 31 章之外，还应熟悉第 29、35 和 36 章介绍的 XAML 基本信息，这里只介绍专用于 Windows Store 应用程序的功能。

38.2 Microsoft 的现代设计

Windows Store 应用程序中第一个引人注目的地方是，它们看起来与桌面应用程序不同。UI(用户界面)设计着重强调了一点，就是用户在使用应用程序时应感到愉悦和享受。Windows Store 应用程序如此注重设计，源自一些旧有的理念。其中一个理念是瑞士风格的图形设计，这是 20 世纪 50 年代提出的，强调干净(不凌乱)和易于理解。例如，飞机场和火车站的信号就基于这个概念，用户可以尽快处理信息。

德国著名的包浩斯(Bauhaus)学校是现代 UI 设计的另一个发源地，它在 1919 年~1933 年间极有影响力。这所学院的目标是将艺术、工艺和技术结合起来，根据功能而不是装饰来设计——没有任何多余的修饰。

第三个基础是电影艺术定义的动作。动画是给应用程序带来生气的重要工具，Windows 运行库框架提供了丰富的动画功能，使用户在传递信息时具有身历其境的体验，把使用应用程序变成一种享受。

38.2.1 内容，不是边框

设计 Windows Store 应用程序的指导原则是注重内容，这意味着，在任何时刻，都只向用户显示他们需要的信息，而不是用他们不需要的信息(即边框，如菜单、工具栏等)来分散其注意力。用户打开 Internet Explorer 时，内容就会占满整个视图。菜单是隐藏的，除非用户显式激活了它。

例如，网页占满了整个屏幕，用户可以快速搜寻需要的内容，而没有各种菜单和工具栏的打扰。图 38-1 显示了一个天气应用程序的主视图。注意大图形更便于用户迅速把注意力集中在需要的信息上。

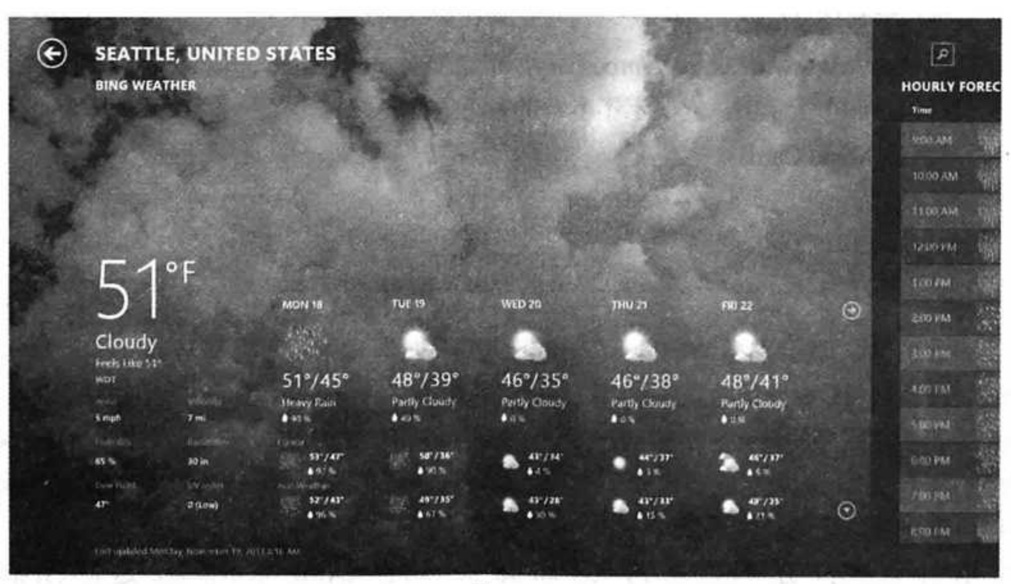


图 38-1

当然，在应用程序内，用户也可以改变设置，使用命令。要修改设置，可以使用新的 Charms 工具栏(Charms bar)。要激活 Charms 工具栏，用户可以单击屏幕的右边界，调用改变应用程序设置所需的控件。

命令放在屏幕顶部或底部的 app 工具栏中，要激活这些命令，用户可以使用类似的方式，即单击屏幕的顶部或底部，打开它们。图 38-2 显示了 Windows Store 命令栏。在该例中，命令位于屏幕顶部。导航命令应放在顶部的 app 工具栏中。

如图 38-3 所示的 Weather 应用程序把命令放在顶部或底部。导航命令放在顶部，操作命令放在底部。

图 38-2

图 38-3

38.2.2　快速流畅

快速流畅是 Windows Store 应用程序的另一个重要原则。在传统的用户界面中使用鼠标时，用户习惯于有一点儿延迟。同样，单击按钮或在屏幕上移动某些对象时，也习惯于有一点儿延迟。这种延迟在触摸模式下是不可接受的。如果在触摸后屏幕没有立即产生什么反映，或者 UI 没有响应，用户的体验就会很差。

新的 Windows Runtime 规定，如果一个方法的执行时间超过 50ms，就只能异步执行。在.NET

框架中，许多 API 调用既有同步版本，又有异步版本。因为同步编程比异步编程更容易创建，所以一般使用 API 的同步版本。使用 C# 5.0 中新的异步功能，以及 async 和 await 关键字，异步 API 调用也非常容易使用。第 13 章介绍了这些新关键字的所有细节。除了使用异步 API 之外，还应在应用程序中为执行很长时间的任务创建异步 API。

异步编程仅是实现快速流畅这一原则的一部分。如前所述，Windows Store 应用程序能很好地支持动画，因为它们以自然、真实的方式把用户体验联系在一起，且不会分散注意力。内置控件已经有动画，允许编写出流畅的切换效果，而不是很突然的变化。使用这些内置控件，就不需要定义自定义动画，但如果愿意，也是可以定义自定义动画的。

38.2.3　可读性

可读性对于任何应用程序都很重要，而 Microsoft 提供了完整的样式应用规则集。这些规则覆盖了用户体验的所有排版方面，包括可读的字体、颜色和字母间距。例如，Segoe UI 字体应用于 UI 元素(按钮、日期选择器)，Calibri 字体用于用户读写的文本，Cambria 字体用于大文本块。

38.3　示例应用程序的核心功能

本节开发的 Windows Store 示例应用程序用于创建菜单卡。后面看到的菜单和图片都来自作者妻子在维也纳市中心开的饭店 http://www.kantine.at，欢迎读者光临该饭店。

在该应用程序中，饭店可以创建菜单卡，例如早餐、午餐卡和汤羹卡等。通过这个功能，应用程序使用 XAML 和 C#获得用户的信息，以写入数据，处理菜单卡的图像，以及与应用程序相关的其他任务。

示例应用程序的创建从 Windows Store 类别中的 Blank App (XAML)模板开始，如图 38-4 所示。

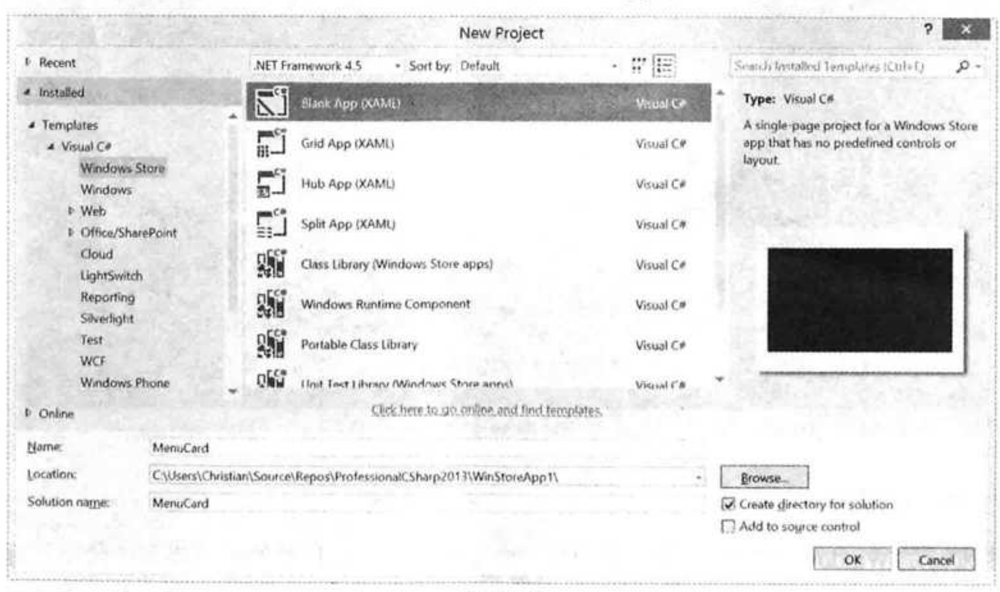


图 38-4

38.3.1　文件和目录

在根据模板创建的项目中，解决方案包含几个目录和一些文件。Assets 目录包含应用程序的徽

标图像和一个闪屏。如果在项目中添加其他 Windows Store Visual Studio 项，就创建 Common 目录。Common 目录包含标准样式和实用工具类。在 Blank App (XAML)模板中，Common 目录还不存在。继续处理这个应用程序，就会创建并填充这个目录。项目中最重要的文件是 App.xaml 及其代码文件 App.xaml.cs，MainPage.xaml 及其代码文件 MainPage.xaml.cs，以及 Package.appxmanifest。XAML 和代码文件非常类似于第 35 章中 WPF 的结构。

Package.appxmanifest 是一个 XML 文件，描述了应用程序的打包和功能。用 Visual Studio 打开这个文件，会打开清单设计器，如图 38-5 所示。其中定义了应用程序名、徽标图像和闪屏。单击 Visual Assets 选项卡后，会看到徽标的所有不同选项。图像所需的像素尺寸显示在这个编辑器中。徽标需要 150×150 像素，小徽标、大徽标和宽徽标是可选的。对于每种尺寸的徽标，都可以根据系统的比例指定不同的尺寸，分辨率为 1920×1080、屏幕尺寸为 10.6 英寸的 Microsoft Surface 设备使用 140%的比例。分辨率相同但屏幕尺寸为 23 英寸的设备使用 100%的比例。在 Visual Studio 中使用 Simulator 运行应用程序，很容易测试不同的分辨率和比例。可以给徽标和闪屏添加 PNG 或 JPG 文件。

应用程序的入口点是 App 类。在这个类中实例化了主页。除了 UI 的定义之外，也使用该软件包指定了功能和声明。在 Capabilities 选项卡，应用程序指定是否希望访问麦克风或网络摄像头等设备。在 Windows Store 中安装应用程序时，会告诉用户应用程序有什么需求。如果没有声明，应用程序就不能使用这些设备。在 Declarations 选项卡，应用程序声明了它支持的功能，例如，它是否可用于搜索系统；或者它是否提供共享目标，以允许其他应用程序为它提供一些数据。

下面给应用程序添加一些页面。

图 38-5

38.3.2 应用程序页面

现在向应用程序添加一些 UI 页面。从模板添加的第一个页面是 MainPage.xaml。根据 Blank App

(XAML)模板，该页面没有提供任何结构，内容是完全可定制的。如果不创建 Windows Store 游戏或需要特殊布局的其他应用程序，最好使用标准的格式和样式，把应用程序名放在 Windows Store 样式规则准确定义的位置上。在开始运行不同的 Windows Store 应用程序时，会发现它们有一些相似之处。为了不重复工作，可以直接通过 Visual Studio 项模板，使用预定义的样式，如图 38-6 所示。

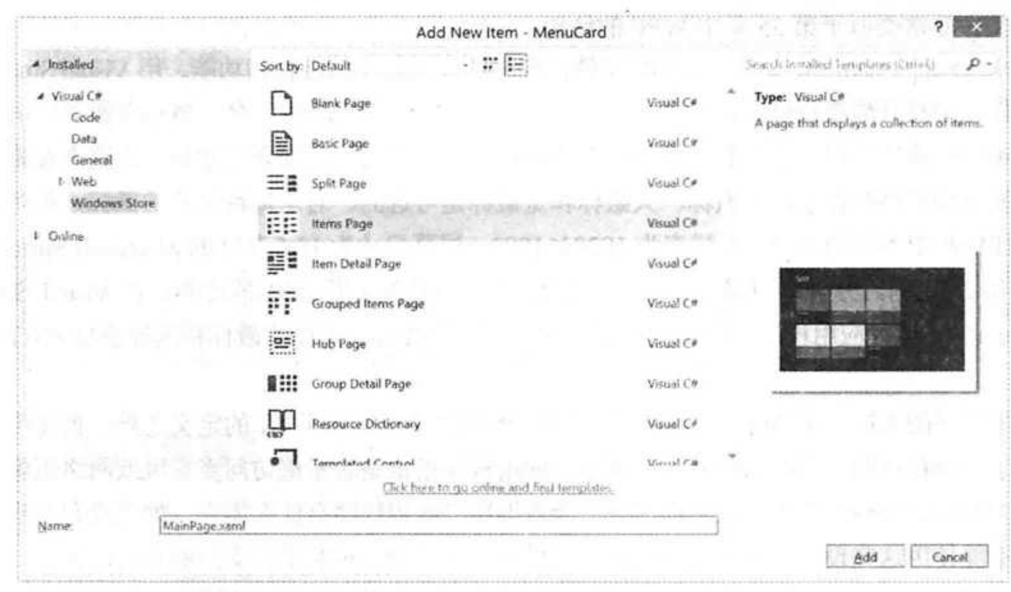


图 38-6

在示例应用程序中，前面创建的主页用 Items Page 模板替代。该应用程序创建的其他页面有基本页面(Basic Page)AddMenuCardPage 和项页面(Items Page)MenuItemsPage。

基本页面提供了一个布局，把应用程序名放在用户习惯查看的顶部位置。分割页面(Split Page)把页面分成两半，一半是列表，另一半是细节。项页面包含 GridView 控件，它在一个网格中显示项列表。要使用项组，可以使用模板 Grouped Items Page、Group Detail Page 和 Item Detail Page。Grouped Items Page 用于显示不同的项组，并使用带 GroupStyle 设置的 ListView，以及带分组的 CollectionViewSource。Group Detail Page 显示一个组及其详细信息，于是为这个任务使用 GridView。Item Detail Page 使用 RichTextBlock 显示一项的细节，Hub Page 是 Windows 8.1 中的新页面，在一个页面上提供了几个看起来完全不同的部分。

当然，总是可以从 Blank Page 开始，根据需要添加控件，定义布局。但是，使用预定义的模板，再根据需要调整 XAML 代码，可以节省许多时间和精力。

使用这些模板在项目的 Common 目录中再添加更多的类：NavigationHelper 帮助在导航页面时管理状态，ObservableDictionary 实现了 IObservableMap 接口，可以用作未类型化的视图模型。RelayCommand 用于通过 Action 和 Func 委托将命令委托给方法。最后，SuspensionManager 用于在应用程序挂起时，存储和加载应用程序的状态。

1. 主页

应用程序的主页如图 38-7 所示，它显示了每个菜单卡的标题和图像。

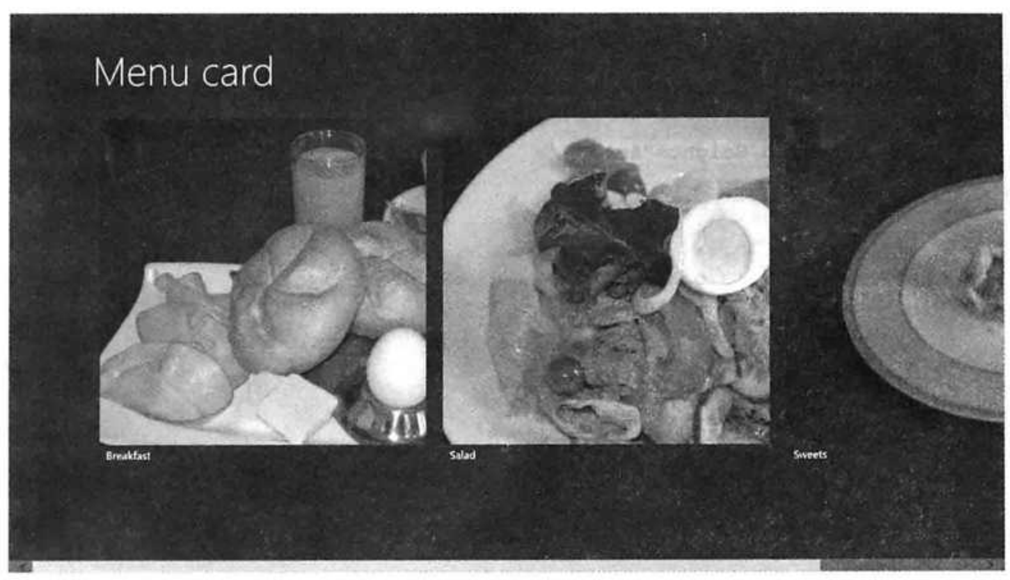


图 38-7

为此，只需要对 XAML 代码(代码文件 MainPage.xaml)进行一些小的调整，如下代码所示。在 Items Page 模板中，XAML 代码包含一个 GridView 作为子元素，有变化的是单击一项时触发的 ItemClick 事件处理程序：

```
<GridView
  x:Name="itemGridView"
  AutomationProperties.AutomationId="ItemsGridView"
  AutomationProperties.Name="Items"
  TabIndex="1"
  Grid.RowSpan="2"
  Padding="116,136,116,46"
  ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
  SelectionMode="None"
  IsItemClickEnabled="True"
  ItemClick="OnMenuCardClick">
```

XAML 模板和项模板参见第 35 和 36 章。

GridView 的源代码用 ItemsSource 属性定义，它引用了静态资源 itemsViewSource。itemsViewSource 是在刚刚绑定到 Items 属性的页面资源中指定的一个简单 CollectionViewSource：

```
<CollectionViewSource
  x:Name="itemsViewSource"
  Source="{Binding Items}"/>
```

菜单卡的项模板直接在 GridView 中定义为 DataTemplate。默认模板中的项使用两列建立，而这里的项由两行组成。尺寸较大，且绑定到 Image 和 Title 属性上。记住，前面定义的 MenuCard 类实现了这些属性：

```
<GridView.ItemTemplate>
```

```
<DataTemplate>
  <Grid Margin="6">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Border Background=
      "{StaticResource
      ListViewItemPlaceholderBackgroundThemeBrush}"
      Width="450" Height="450">
      <Image Source="{Binding Image}" Stretch="UniformToFill"/>
    </Border>
    <StackPanel Grid.Column="1" Margin="10,0,0,0">
      <TextBlock Text="{Binding Title}"
        Style="{StaticResource TitleTextBlockStyle}"
        MaxHeight="40"/>
      </StackPanel>
    </Grid>
  </DataTemplate>
</GridView.ItemTemplate>
```

前面定义的 CollectionViewSource 绑定到 Items 集合上。Items 集合在 MainPage 类的 LoadState 方法中赋值(代码文件 MenuCard/MainPage.xaml.cs)。LoadState 方法的实现代码指定了 LayoutAwarePage 基类的 DefaultViewModel 属性。这个属性返回 IObservableMap<string, object>，其中任意数据对象都可以赋予一个键名。键名在 XAML 中用于引用数据。

```
private async void navigationHelper_LoadState(object sender,
    LoadStateEventArgs e)
{
  var storage = new MenuCardStorage();
  MenuCardFactory.Instance.InitMenuCards(
    new ObservableCollection<MenuCard>(
      await storage.ReadMenuCardsAsync()));
  this.DefaultViewModel["Items"] = MenuCardRepository.Instance.Cards;
}
```

代码使用自定义类 MenuCardStorage 在移动存储器中读写数据。这个类在本章后面介绍。

2. 添加菜单卡页面

为了添加新的菜单卡，添加了 AddMenuCardPage。这里使用的模板只是 Basic Page 模板。但是，其中没有太多的内容要定义。用户只需要给菜单卡指定标题、描述和图像。UI 如图 38-8 所示，只需两个文本框、一个按钮和一个 Image 控件。

在 AddMenuCard.xaml 文件中定义主要控件的 XAML 代码如下所示。注意这里的两个要点：控件绑定到 Image、Title 和 Description 属性上，赋予父控件(Grid)的数据上下文被设置为 Item 属性：

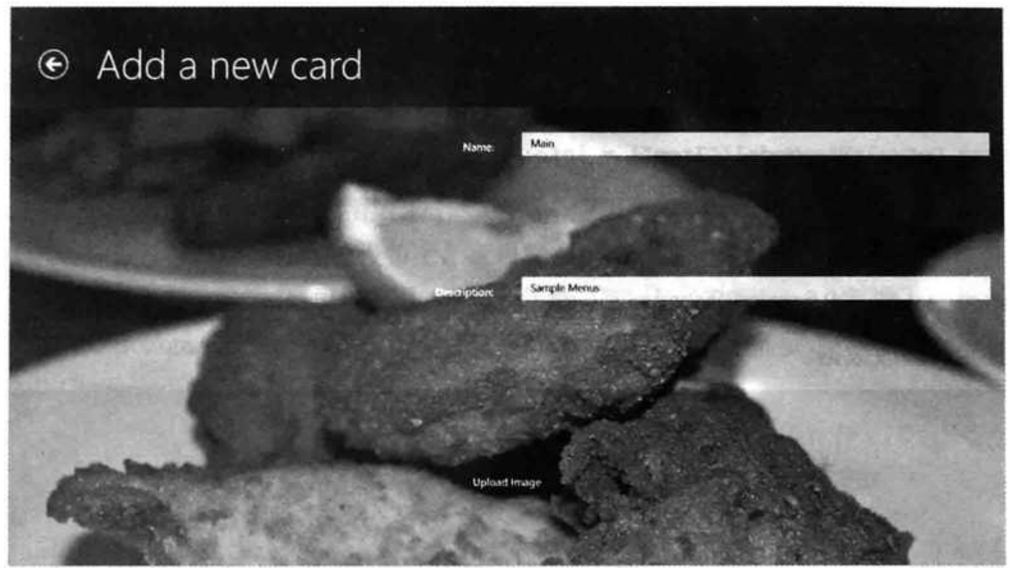


图 38-8

```
<Grid Grid.Row="1" DataContext="{Binding Item}">
  <Grid.RowDefinitions>
    <RowDefinition Height="100" />
    <RowDefinition Height="300" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Border Grid.Row="0" Grid.RowSpan="3" Grid.Column="0" Grid.ColumnSpan="2">
    <Image Source="{Binding Image, Mode=OneWay}" Stretch="UniformToFill" />
  </Border>
  <TextBlock Text="Name:" Style="{StaticResource TitleTextStyle}" Margin="20"
    VerticalAlignment="Center" HorizontalAlignment="Right" />
  <TextBox Grid.Column="1" Text="{Binding Title, Mode=TwoWay}" Margin="20"
    VerticalAlignment="Center" />
  <TextBlock Grid.Row="1" Text="Description:"
    Style="{StaticResource TitleTextStyle}" Margin="20"
    VerticalAlignment="Center"
    HorizontalAlignment="Right" />
  <TextBox Grid.Row="1" Grid.Column="1"
    Text="{Binding Description, Mode=TwoWay}"
    Margin="20" MaxHeight="100" VerticalAlignment="Center" />
  <Button HorizontalAlignment="Center" VerticalAlignment="Center"
    Visibility="{Binding ImageUploaded,
    Converter={StaticResource visibilityConverter}}" Content="Upload Image"
    Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
    Style="{StaticResource TextButtonStyle}" Click="OnUploadImage"
    Padding="10" Margin="20" />
</Grid>
```

在代码文件中，Item 属性被赋予 navigationHelper_LoadState 方法中 AddMenuCardInfo 类型的对象(它包含在 XAML 代码中绑定的属性) (代码文件 AddMenuCardPage.xaml.cs)：

```
private AddMenuCardInfo info = new AddMenuCardInfo();
private void navigationHelper_LoadState(object sender,
  LoadStateEventArgs e)
{
  this.DefaultViewModel["Item"] = info;
}
```

3. 菜单项页面

应用程序的第三个页面是MenuItemsPage，如图38-9所示。这个页面显示了一个菜单卡中的菜单项，并允许修改数据。

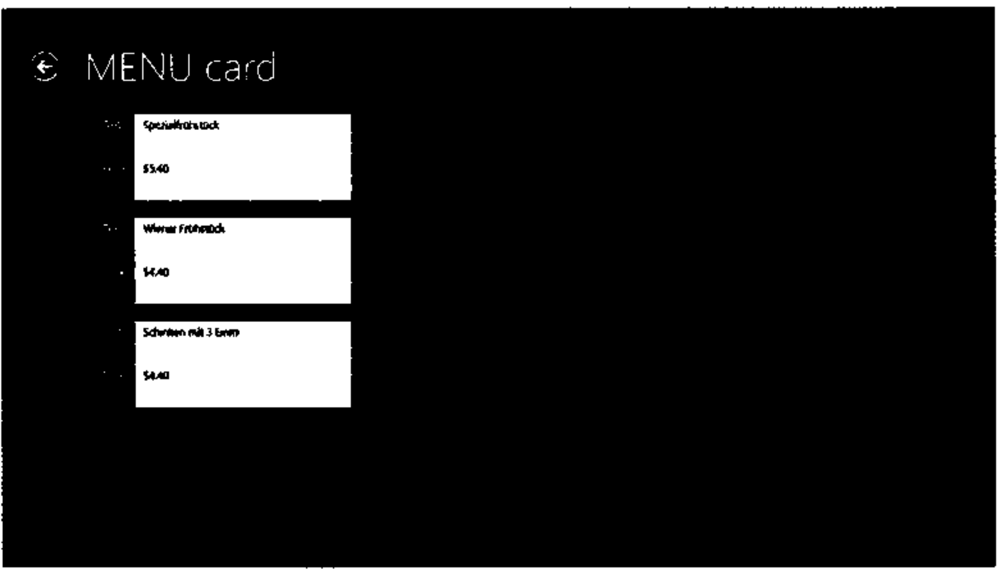


图38-9

这个页面也基于Items Page模板，在navigationHelper_LoadState方法中绑定到一组菜单项上(代码文件MenuItemsPage.xaml.cs)：

```
private void navigationHelper_LoadState(object sender,
  LoadStateEventArgs e)
{
  card = navigationParameter as MenuCard;
  if (card != null)
  {
    this.DefaultViewModel["Items"] = card.MenuItems;
  }
}
```

有了3个页面后，就准备导航Windows Store应用程序。

38.4 应用程序工具栏

尽管在chrome之前放置内容是Windows Store应用程序的一个重要设计方面，但显然用户需要一种方式与UI交互操作。现在这由新的应用程序工具栏(App Bar)提供。Windows以前的版本默认

显示命令，而现在用户可以选择何时显示应用程序命令。

通过触摸，轻击屏幕的底边界或顶边界时，应用程序工具栏就会显示出来。使用鼠标时，单击鼠标右键，会调用应用程序工具栏。使用键盘时，用户可以单击上下文菜单按钮。

可以在页面的 BottomAppBar 和 TopAppBar 属性中定义应用程序工具栏。顶部的应用程序工具栏用于导航，底部的应用程序工具栏用于操作。这两个应用程序工具栏可以用相同的操作同时显示出来。方法是单击鼠标右键，在触摸设备上，可以轻击屏幕的底边界或顶边界。

在页面的 TopAppBar 属性中，可以使用 AppBar 元素。Windows Runtime 2.0 新增了 CommandBar。这个元素可以与 BottomAppBar 一起使用，且已经定义了布局，所以不需要定义包含列的 Grid ，因为这已经由 CommandBar 完成了。

下面的代码段(代码文件 MainPage.xaml)在页面的 BottomAppBar 属性中定义了一个 CommandBar 元素。在 CommandBar 中，可以使用某些易用的应用程序工具栏元素，例如 AppBarButton、AppBarSeparator 和 AppBarToggleButton。在这个示例中，添加了两个 AppBarButton(按钮)控件，它们使用预定义的样式，Command 属性绑定到 AddCommand 属性上，以使用 RelayCommand 类型：

```
<Page.BottomAppBar>
  <CommandBar>
    <AppBarButton Tag="Add" Icon="Add" Label="Add"
      Command="{Binding Commands.AddCommand}" />
    <AppBarButton Tag="Delete" Icon="Delete" Label="Delete"
      Command="{Binding Commands.DeleteCommand}" />
  </CommandBar>
</Page.BottomAppBar>
```

要定义用于 AppBarButton 的符号，有不同的选项。AppBarButton 的 Icon 属性是 IconElement 类型。这里可以指定 Symbol 枚举的一个命名常量(该枚举定义了几个常量，如 Add、Cancel、Accept、Setting)、SymbolIcon (使用 Segoe UI Symbol 字体中的字形)、FontIcon (可以使用任意字体)或 PathIcon (使用 Path 形状)。要使用 Path 元素创建形状，请参见第 35 章。

要把命令绑定到操作上，应把视图模型 Commands 设置为页面的实例(代码文件 MenuCard/MainPage.xaml.cs)：

```
private async void navigationHelper_LoadState(object sender,
  LoadStateEventArgs e)
{
  this.DefaultViewModel["Commands"] = this;

  //...
}
```

命令使用返回 RelayCommand 的属性定义。RelayCommand 通过 AddCommand 引用 OnAdd 方法(代码文件 MenuCard/MainPage.xaml.cs)：

```
private RelayCommand addCommand;
public RelayCommand AddCommand
{
  get
```

```
    {
      return addCommand ?? (addCommand = new RelayCommand(OnAdd));
    }
  }
```

OnAdd 方法的实现代码导航到 AddMenuCardPage 页面上：

```
private void OnAdd()
{
  Frame.Navigate(typeof(AddMenuCardPage));
}
```

图 38-10 显示了带有应用程序工具栏的应用程序。

图 38-10

CommandBar 也允许使用次级命令。主命令显示在右边，次级命令显示在左边。如果屏幕尺寸太小，没有足够的空间显示所有命令，就只显示主命令。次级命令可以通过指定 SecondaryCommands 属性来定义(代码文件 MenuCard/MenuItemsPage.xaml)：

```
<Page.BottomAppBar>
  <CommandBar>
    <AppBarButton Tag="Add" Icon="Add" Label="Add"
      Command="{Binding Commands.AddCommand}" />
    <AppBarButton Tag="Delete" Icon="Delete" Label="Delete"
      Command="{Binding Commands.DeleteCommand}" />
    <CommandBar.SecondaryCommands>
      <AppBarButton Tag="Save" Icon="Save" Label="Save"
        Command="{Binding Commands.SaveCommand}" />
      <AppBarButton Tag="Download" Icon="Download" Label="Download"
        Command="{Binding Commands.DownloadCommand}" />
    </CommandBar.SecondaryCommands>
  </CommandBar>
</Page.BottomAppBar>
```

38.5　启动与导航

为了在页面之间导航，首先显示 MainPage。MainPage 在 App 类的 OnLaunched 方法中激活(代码文件 App.xaml.cs)：

```
protected override async void OnLaunched(LaunchActivatedEventArgs args)
{
  Frame rootFrame = Window.Current.Content as Frame;

  // Do not repeat app initialization when the window already has content,
  // just ensure that the window is active
  if (rootFrame == null)
  {
    rootFrame = new Frame();

    rootFrame.Language = ApplicationLanguages.Languages[0];

    rootFrame.NavigationFailed += OnNavigationFailed;

    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
    }

    Window.Current.Content = rootFrame;
  }

  if (rootFrame.Content == null)
  {
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
  }
  Window.Current.Activate();
}
```

OnLaunched 在应用程序启动的不同时间调用。例如，应用程序之前可能挂起了，或者可能调用它，以共享另一个应用程序中的数据。LaunchActivatedEventArgs 参数提供了启动的原因细节和应用程序以前的状态。LaunchActivatedEventArgs 定义了一个 ActivationKind 类型的 Kind 属性，通过它可以读取启动的原因。这里可以使用的枚举值有 Launch(单击磁贴时应用程序正常启动)、Search(使用 Windows 中的搜索功能来启动应用程序)、File、FileOpenPicker 和 FileSavePicker(选择一个文件来启动应用程序)。启动应用程序的磁贴可以用 LaunchActivatedEventArgs 的 TileId 属性来读取。应用程序可以提供多个磁贴，作为提供了不同信息或行为时的启动动作。例如，天气应用程序使用这个功能，允许用户添加不同的磁贴，显示不同城市的天气。使用 LaunchActivatedEventArgs 参数，应用程序可以在主页中显示指定城市的天气。以前的执行状态用 ApplicationExecutionState 类型的 PreviousExecutionState 属性读取，其可能的值是 NotRunning、Running、Suspended、Terminated 和 ClosedByUser。

如何处理应用程序挂起参见第 31 章。

在 OnLaunched 方法中，创建一个新的 Frame，作为导航上下文。

```
rootFrame = new Frame();
//...
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

导航通过 Frame 类和 NavigationHelper 类来完成。

MainPage.xaml 包含一个 Go Back 按钮，它使用 NavigationHelper 类回退：

```
<Button x:Name="backButton" Command=
  "{Binding NavigationHelper.GoBackCommand, ElementName=pageRoot}"
  Style="{StaticResource NavigationBackButtonNormalStyle}"/>
```

每个页面都有一个关联的 NavigationHelper 对象，该对象在页面的构造函数中实例化。这里，还把 LoadState 事件赋予 navigationHelper_LoadState 方法。这个事件在实例化页面后触发，允许给页面传递一个初始值：

```
public MainPage()
{
  this.InitializeComponent();
  this.navigationHelper = new NavigationHelper(this);
  this.navigationHelper.LoadState += navigationHelper_LoadState;
}
```

NavigationHelper 提供的 GoBack 和 GoForward 方法用于在框架历史中导航，CanGoBack 和 CanGoForward 方法用于在可以进行对应的导航操作时提供信息。绑定到 Back 按钮上的 GoBackCommand 仅调用 GoBack 方法，使用 CanGoBack 定义命令是否可用。

调用 Navigate 方法，就会开始导航。Navigate 方法的第一个参数定义了导航应停止在什么类型的页面上。第二个参数向导航的页面发送一些数据。在示例代码(代码文件 MenuCard/App.xaml.cs)中，没有给 MainPage 发送数据，MainPage 在它自己的请求中获得菜单卡数据。

从主页中退出的一种方法是单击应用程序工具栏上的 Add 按钮。这个按钮的 Click 事件关联了 OnAddMenuCard 处理程序方法(代码文件 MainPage.xaml.cs)，在该方法中，只是导航到 AddMenuCardPage：

```
private void OnAddMenuCard(object sender, RoutedEventArgs e)
{
  this.Frame.Navigate(typeof(AddMenuCardPage));
}
```

从 MainPage 中退出的另一种方法是单击 GridView 控件中的一项。这里，ItemClick 事件被赋予处理方法 OnMenuCardClick，该方法如下所示(代码文件 MainPage.xaml.cs)。导航被发送到 MenuItemsPage。数据用 Navigate 方法的第二个参数传递，e.ClickedItem 表示一个绑定到 GridView 的 MenuCard 实例：

```
private void OnMenuCardClick(object sender, ItemClickEventArgs e)
{
  this.Frame.Navigate(typeof(MenuItemsPage), e.ClickedItem);
}
```

38.6 布局的变化

Windows Store 应用程序必须支持不同的尺寸和布局。应用程序可以占据整个屏幕，水平或垂直显示；也可以只占据屏幕的一部分。Windows 8.1 还支持在屏幕上同时显示多个应用程序，为此，应用程序必须支持至少 320 像素或 500 像素。这个尺寸需要用包清单来设置。还必须定义对横向和纵向显示的支持。

使用 Visual Studio 模拟器，很容易用不同的分辨率检查应用程序的外观。图 38-11 在分辨率设置为 2560×1440 像素的 27 英寸屏幕上显示了应用程序。这里网格切换为两行，而不是一行。模拟器还允许将屏幕分辨率设置为 1024×768 至 2560×1440，屏幕尺寸从 7 英寸到 27 英寸。根据分辨率和屏幕尺寸，可能出现缩放。例如对于分辨率为 2560×1440 的 27 英寸设备，比例设置为 100%。分辨率相同但屏幕尺寸为 10 英寸的设备，使用 180%的比例允许用户在小设备上阅读信息。

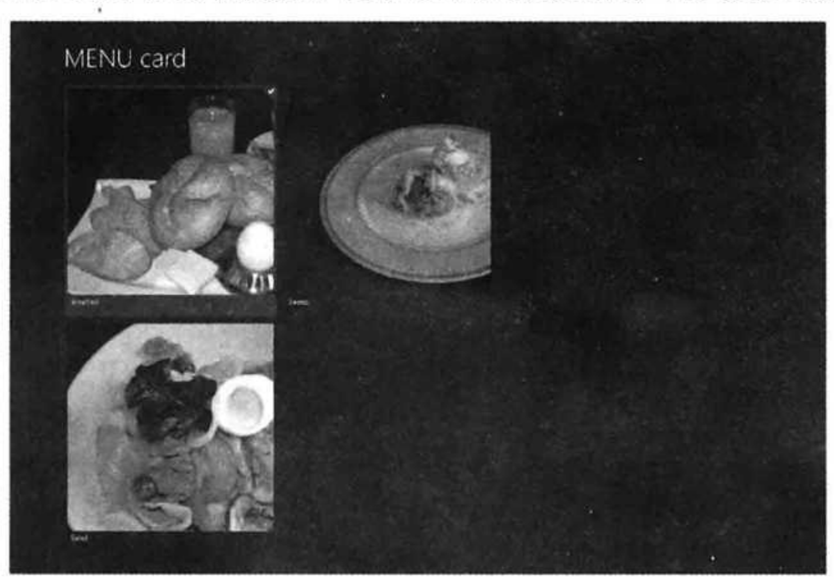


图 38-11

使用 ApplicationView 类可以读取当前布局。ApplicationView.GetForCurrentView 返回一个 ApplicationView 实例。这是 Windows 8.1 中的一个重要变化，而 Windows 8 使用一个静态的 Value 属性返回 ApplicationView。在这个新版本中，现代 UI 支持多个屏幕，它们也可以有不同的分辨率和比例。

ApplicationView 的 Orientation 属性返回 ApplicationViewOrientation，其值可以是 Landscape 或 Portrait。检查 AdjacentToLeftDisplayEdge 和 AdjacentToRightDisplayEdge 属性可以确定应用程序是显示在左边还是右边。注意，用户可以在一个屏幕上显示多个应用程序(根据分辨率和比例因素，至多显示 4 个应用程序，应用程序可能既不显示在左边，也不显示在右边。

所有这些信息都可以用于动态改变控件的更改行为。改变控件行为的一种好方法是使用 VisualStateManager。这个控件的用法参见第 35 章。其用法与 WPF 非常类似。

应用程序数据

对于要在 UI 中使用的数据，应用程序在 DataModel 子目录中定义了几个类型。

用作基类的一个类型实现了INotifiyPropertyChanged接口，用于将更改通知传递到UI上，IsDirty属性用于在项有变化时保存它们(代码文件MenuCard/Extensions/BindableBase.cs)。SetProperty方法由带属性设置器的派生类型调用：

```
public class BindableBase : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void OnPropertyChanged(
    [CallerMemberName] string propertyName = null)
  {
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
  }

  protected virtual bool SetProperty<T>(ref T property, T value,
    [CallerMemberName] string propertyName = null)
  {
    if (!EqualityComparer<T>.Default.Equals(property, value))
    {
      property = value;
      IsDirty = true;
      OnPropertyChanged(propertyName);
      return true;
    }
    return false;
  }
  public bool IsDirty { get; private set; }

  public void ClearDirty()
  {
    IsDirty = false;
  }
}
```

类MenuCard(代码文件MenuCard/DataModel/MenuCard.cs)表示包含应用程序主要数据的菜单卡。这个类定义了用于显示的属性Title、Description和Image。与用于数据绑定的所有类一样，MenuCard也派生自基类BindingBase。

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using Windows.UI.Xaml.Media;
using Wrox.ProCSharp.Extensions;
namespace Wrox.ProCSharp.Model
{
  public class MenuCard : BindableBase
  {
    private string title;
    public string Title
    {
```

```
      get { return title; }
      set { SetProperty(ref title, value); }
    }
    private string description;
    public string Description
    {
      get { return description; }
      set { SetProperty(ref description, value); }
    }
    private ImageSource image;
    public ImageSource Image
    {
      get { return image; }
      set { SetProperty(ref image, value); }
    }
    private string imagePath;
    public string ImagePath
    {
      get { return imagePath; }
      set { imagePath = value; }
    }
    private readonly ICollection<MenuItem> menuItems =
    new ObservableCollection<MenuItem>();
    public ICollection<MenuItem> MenuItems
    {
      get { return menuItems; }
    }
    public void RestoreReferences()
    {
      foreach (var menuItem in MenuItems)
      {
        menuItem.MenuCard = this;
      }
    }
    public override string ToString()
    {
      return Title;
    }
  }
}
```

包含在 MenuCard 中的类 MenuItem(代码文件 MenuCard/DataModel/MenuItem.cs)还定义了带有更改通知的简单属性：

```
using Wrox.ProCSharp.Common;
namespace Wrox.ProCSharp.Model
{
  public class MenuItem : BindableBase
  {
    private string text;
    public string Text
    {
      get { return text; }
      set { SetProperty(ref text, value); }
    }
```

```
    private double price;
    public double Price
    {
      get { return price; }
      set { SetProperty(ref price, value); }
    }
    public MenuCard MenuCard { get; set; }
  }
}
```

类 AddMenuCardInfo(代码文件 MenuCard/DataModel/AddMenuCardInfo.cs)用于创建新菜单卡。这个类也是用于数据绑定的简单类型:

```
using Windows.UI.Xaml.Media;
using Wrox.ProCSharp.Extensions;
namespace Wrox.ProCSharp.Model
{
  public class AddMenuCardInfo : BindableBase
  {
    private string title;
    public string Title
    {
      get { return title; }
      set { SetProperty(ref title, value); }
    }
    private string description;
    public string Description
    {
      get { return description; }
      set { SetProperty(ref description, value); }
    }
    private ImageSource image;
    public ImageSource Image
    {
      get { return image; }
      set { SetProperty(ref image, value); }
    }
    private string imageFileName;
    public string ImageFileName
    {
      get { return imageFileName; }
      set { SetProperty(ref imageFileName, value); }
    }
  }
}
```

类 MenuCardFactory(代码文件 MenuCard/DataModel/MenuCardFactory.cs)独立使用，可返回一个菜单卡列表。方法 InitMenuCards 用于初始化该集合，并把 ObservableCollection<MenuCard>赋予 cards 变量:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
namespace Wrox.ProCSharp.Model
```

```
{
  public class MenuCardRepository
  {
    private ICollection<MenuCard> cards;
    public ICollection<MenuCard> Cards
    {
      get
      {
        return cards;
      }
    }
    public void InitMenuCards(IEnumerable<MenuCard> menuCards)
    {
      cards = new ObservableCollection<MenuCard>(menuCards);
    }
    private MenuCardRepository() { }
    public static MenuCardRepository Instance = new MenuCardRepository();
  }
}
```

类 ObservableCollection<T>用于把集合绑定到 UI 上，因为它实现了 INotify-CollectionChanged 接口。这个类在第 10 章介绍。

这个应用程序用于创建菜单卡，但在第一次启动时，最好向用户显示一些初始的菜单卡。要创建示例数据，可使用 MenuCardDataFactory 类中的 GetSampleMenuCards 方法返回一组填充了一些菜单的菜单卡。示例菜单卡的图像存储在 Assets 文件夹中，从该文件夹中引用，例如 Breakfast.jpg。ms-appx URI 前缀是用于在应用程序包中引用文件的模式：

```
public static ObservableCollection<MenuCard> GetSampleMenuCards()
{
  Uri baseUri = new Uri("ms-appx:///");
  var cards = new ObservableCollection<MenuCard>();
  var card1 = new MenuCard
  {
    Title = "Breakfast"
  };
  card1.MenuItems.Add(new MenuItem
  {
    Text = "Spezialfrühstück",
    Price = 5.4,
    MenuCard = card1
  });
  card1.MenuItems.Add(new MenuItem
  {
    Text = "Wiener Frühstück",
    Price = 4.4,
    MenuCard = card1
  });
  card1.MenuItems.Add(new MenuItem
  {
```

```
    Text = "Schinken mit 3 Eiern",
    Price = 4.4,
    MenuCard = card1
  });
  card1.ImagePath = string.Format("{0}{1}", baseUri, "Assets/Breakfast.jpg");
  cards.Add(card1);
  //... more menu cards in the code download
```

38.7　存储

下面讨论存储 API 来读写文件。典型的 Windows Store 应用程序不通过用户交互操作，一般无法访问完整的文件系统。但是，有一些特殊的目录，应用程序可以在其中读写数据，还有一个选项要求用户提供文件。让用户提供文件是使用选择器实现的，将在 38.8 节介绍。本节首先介绍文件系统的概念以及如何以编程方式使用它。

示例应用程序需要读写菜单卡，包括数据项和图像。给菜单存储文本信息和存储图像的功能是分开的，允许以后在 Windows Azure Storage 中用博客存储图像，在 SQL Azure 或 Windows Azure Table Storage 中存储文本信息。但是这里只使用本地存储。使用云解决方案，可以把本地存储用于缓存。

38.7.1　定义数据协定

应存储的数据在 Storage 文件夹的单独类中定义。这样就可以定义串行化需要的特性，独立于把它们绑定到 UI 元素的类。为了串行化对象，XML 串行化器和数据协定串行化器必须可用于 Windows Store 应用程序。在本章中，串行化使用数据协定完成，所以 DataContract 和 DataMember 特性应用于 MenuCardData 类(代码文件 MenuCard/Storage/MenuCardData.cs)。在这个类中，只为串行化定义了简单的属性 Title、Description、ImagePath 和一个 MenuItemData 集合。该类还实现一个接受 MenuCard 作为参数的构造函数，以及返回 MenuCard 的 ToMenuCard 方法，以便转换为 MenuCard 类，或从 MenuCard 类转换。

```
using System.Collections.Generic;
using System.Runtime.Serialization;
using Wrox.Win8.DataModel;
namespace Wrox.Win8.Storage
{
  [DataContract(Name="MenuCard")]
  public class MenuCardData
  {
    public MenuCardData() { }
    public MenuCardData(MenuCard menuCard)
    {
      this.Title = menuCard.Title;
      this.Description = menuCard.Description;
      this.ImagePath = menuCard.ImagePath;
      MenuItems = new List<MenuItemData>();
      foreach (var item in menuCard.MenuItems)
      {
        MenuItems.Add(new MenuItemData(item));
      }
    }
```

```
    public MenuCard ToMenuCard()
    {
      var menuCard = new MenuCard
      {
        Title = this.Title,
        Description = this.Description,
        ImagePath = this.ImagePath
      };
      foreach (MenuItemData item in this.MenuItems)
      {
        menuCard.MenuItems.Add(item.ToMenuItem());
      }
      menuCard.ClearDirty();
      return menuCard;
    }
    [DataMember]
    public string Title { get; set; }
    [DataMember]
    public string Description { get; set; }
    [DataMember]
    public string ImagePath { get; set; }
    [DataMember]
    public List<MenuItemData> MenuItems { get; set; }
  }
}
```

MenuItemData 类(代码文件 MenuCard/Storage/MenuItemData.cs)表示菜单卡中的菜单项，也需要数据协定特性：

```
using System.Runtime.Serialization;
using Wrox.ProCSharp.Model;
namespace Wrox.ProCSharp.Storage
{
  [DataContract(Name="MenuItem")]
  public class MenuItemData
  {
    public MenuItemData() { }
    public MenuItemData(MenuItem item)
    {
      if (item != null)
      {
        this.Text = item.Text;
        this.Price = item.Price;
      }
    }
    public MenuItem ToMenuItem()
    {
      return new MenuItem
      {
        Text = this.Text,
        Price = this.Price
      };
    }
    [DataMember]
    public string Text { get; set; }
```

```
    [DataMember]
    public double Price { get; set; }
  }
}
```

38.7.2 写入移动数据

现在可以创建一个写入 MenuCard 对象的方法了。应用程序可以在一些预定义的文件夹中写入数据。这些文件夹可以用 ApplicationData 类访问。ApplicationData.Current 返回 ApplicationData 单态对象的一个实例。之后就可以访问 LocalFolder 和 RoamingFolder。LocalFolder 属性会返回针对应用程序的文件夹，该文件夹仅在本地系统上可用；RoamingFolder 返回一个文件夹，在本地写入数据后，该文件夹中的数据写入一个云服务中，用户在每个使用相同活动账户的系统上都可以访问这些数据。

示例应用程序使用移动文件夹，这样用户就可以在所有的 Windows 8.1 系统及更新版本上使用这些数据了。WriteMenuCardAsync 方法(代码文件 Storage\MenuCardStorage)通过参数 menuCard 接收一个 MenuCard，访问第一行上的移动文件夹。接着确定 MenuCard 对象自从上一次写入以来是否有变化。每次属性改变时，MenuCard 的改动(dirty)标记都会变化。在 StorageFolder 上调用 CreateFileAsync 方法，创建一个文件，文件名包含菜单卡的标题。CreateFileAsync 方法的第二个参数可以指定，如果文件存在，会发生什么。可能的选项有抛出异常或打开已有文件。这里是仅覆盖已有文件。接着把打开的文件和菜单卡传递给方法 WriteMenuCardToFileAsync：

```
public async Task WriteMenuCardAsync(MenuCard menuCard)
{
  StorageFolder folder = ApplicationData.Current.RoamingFolder;
  if (menuCard.IsDirty)
  {
    StorageFile storageFile = await folder.CreateFileAsync(
      string.Format("MenuCards{0}.xml", menuCard.Title),
      CreationCollisionOption.ReplaceExisting);
    await WriteMenuCardToFileAsync(menuCard, storageFile);
    menuCard.ClearDirty();
  }
}
```

WriteMenuCardToFileAsync 方法最终在数据协定串行化器的帮助下写入数据。StorageFile 类提供的几个方法会返回数据流，以读写数据——例如，OpenAsync 返回一个 IRandomAccessStream，OpenTransactedWriteAsync 返回一个 StorageStreamTransaction。这些数据流都是 Windows Runtime 数据流。对于数据协定串行化，需要一个.NET 数据流。WindowsRuntimeStorageExtensions 类中定义的扩展方法 OpenStreamForWriteAsync 直接返回一个.NET 数据流。从这个方法返回的数据流获取 MemoryStream 的一个副本，该副本在前面用 DataContractSerializer 填充了：

```
public async Task WriteMenuCardToFileAsync(MenuCard menuCard,
  StorageFile storageFile)
{
  var menuCardData = new MenuCardData(menuCard);
  var knownTypes = new Type[]
      {
        typeof(MenuItemData)
```

```
    };
  var cardStream = new MemoryStream();
  var serializer = new DataContractSerializer(typeof(MenuCardData), knownTypes);
  serializer.WriteObject(cardStream, menuCardData);
  using (Stream fileStream = await storageFile.OpenStreamForWriteAsync())
  {
    cardStream.Seek(0, SeekOrigin.Begin);
    await cardStream.CopyToAsync(fileStream);
    await fileStream.FlushAsync();
  }
}
```

用 Windows Store 应用程序读写.NET 数据流与其他应用程序没有区别。.NET 文件和数据流参见第 24 章。

现在，只需要连接代码，以使用 UI 保存菜单卡。保存操作的一个时间点是在 AddMenuCardPage 中创建新菜单卡时。退出页面(例如单击后退按钮)时，会调用 navigationHelper_SaveState 方法(代码文件 MenuCard/AddMenuCardPage.xaml.cs)。在触发 NavigationHelper 的 SaveState 事件时，调用 navigationHelper_SaveState。在这里的实现代码中，从绑定到 UI 的信息中创建一个新的 MenuCard，再调用 WriteMenuCardsAsync，将所有改动了的菜单卡写入移动存储器：

```
protected async void navigationHelper_SaveState(object sender,
SaveStateEventArgs e)
{
  var mc = new MenuCard
  {
    Title = info.Title,
    Description = info.Description,
    Image = info.Image,
    ImagePath = info.ImageFileName
  };
  MenuCardRepository.Instance.Cards.Add(mc);
  var storage = new MenuCardStorage();
  await storage.WriteMenuCardsAsync(
  MenuCardRepository.Instance.Cards.ToList());
}
```

把数据写入存储后，下一步是再次读取它。

38.7.3　读取数据

要读取菜单卡，使用 ReadMenuCardsAsync 方法(代码文件 MenuCard/Storage/MenuCardsStorage.cs)从应用程序的移动文件夹中读取所有菜单卡文件，填充 MenuCard 对象，再返回一个列表。在读取所有文件之前，先从移动文件夹中创建一组 XML 文件。StorageFolder 类中的 CreateFileQuery 方法可以定义一个查询，来搜索文件。这里定义的查询指定不使用索引器，只是读取这个目录，不读取子目录，来搜索 XML 文件。通过 QueryOptions 类也可以使用关键字和属性，利用 Advanced Query Syntax(AQS，高级查询语法)搜索文件。从查询中返回的文件使用一个从扩展方法 OpenStreamForReadAsync 中返回的.NET 数据流来读取。接着，使用数据协定串行化器执行反串行化：

```
public async Task<IEnumerable<MenuCard>> ReadMenuCardsAsync()
{
  var menuCards = new List<MenuCard>();
  StorageFolder folder = ApplicationData.Current.RoamingFolder;
  StorageFileQueryResult result = folder.CreateFileQuery();
  var queryOptions = new QueryOptions();
  queryOptions.IndexerOption = IndexerOption.DoNotUseIndexer;
  queryOptions.FolderDepth = FolderDepth.Shallow;
  queryOptions.FileTypeFilter.Add(".xml");
  result.ApplyNewQueryOptions(queryOptions);
  IReadOnlyList<StorageFile> files = await result.GetFilesAsync();
  foreach (var file in files)
  {
    using (Stream stream = await file.OpenStreamForReadAsync())
    {
      try
      {
        var serializer = new DataContractSerializer(typeof(MenuCardData));
        object data = await Task<object>.Run(() => serializer.ReadObject(stream));
        MenuCard menuCard = (data as MenuCardData).ToMenuCard();
        menuCard.RestoreReferences();
        menuCards.Add(menuCard);
      }
      catch (Exception )
      {
        // log exception
      }
    }
  }
  return menuCards;
}
```

StoreState方法用于写入应用程序状态，而LoadState方法用于读取应用程序状态。下面的代码展示了MainPage中的LoadState方法。这个方法在导航到页面时调用。这里调用ReadMenuCardsAsync方法，以获得MenuCard对象集合。这个集合放在ObservableCollection中，再放在视图模型中，将菜单卡用于与UI绑定的数据：

```
protected override async void LoadState(Object navigationParameter,
  Dictionary<String, Object> pageState)
{
  var storage = new MenuCardStorage();
  MenuCardFactory.Instance.InitMenuCards(
    new ObservableCollection<MenuCard>(
      await storage.ReadMenuCardsAsync()));
  this.DefaultViewModel["Items"] = MenuCardFactory.Instance.Cards;
}
```

现在实现了菜单卡的读写，但图像还没有保存。

38.7.4 写入图像

写入图像需要一些特殊的处理。在示例应用程序中，用户可以上传用于菜单卡的图像。对于图像和视频，需要注意它们的尺寸。用户可以上传像素密度很高的图像，但在显示在屏幕上时，可能

并不需要这么高的像素密度。这里的问题是所使用的移动文件夹可以有相关的限额，对于存储在云中的数据，根据所存储的数据量来收取费用，在网络上传输较大的图像会花很多时间。于是，应只存储需要的图像大小。

在 Windows Runtime 中，重置图像的大小已经是框架的一部分。BitmapDecoder 类可以处理图像大小的重置，如下一个示例所示(在 WriteImageAsync 方法中，代码文件 Storage\MenuCard-ImageStorage.cs)。图像用 IRandomAccessStream 参数接收。BitmapDecoder 访问接收到的图像流，这里可以读取像素的高度和宽度。在重置图像的大小时，要执行一些计算，以维持纵横比，新的宽度和高度传递给 BitmapTransform 对象，接着在 StorageFile 对象的帮助下保存图像：

```
public async Task WriteImageAsync(IRandomAccessStream sourceStream,
  string filename)
{
  BitmapDecoder decoder = await BitmapDecoder.CreateAsync(sourceStream);
  uint scaledWidth = 0;
  uint scaledHeight = 0;
  if (decoder.PixelWidth > decoder.PixelHeight)
  {
    scaledWidth = 600;
    double relation = (double)decoder.PixelHeight / decoder.PixelWidth;
    scaledHeight = Convert.ToUInt32(relation * scaledWidth);
  }
  else
  {
    scaledHeight = 600;
    double relation = decoder.PixelWidth / decoder.PixelHeight;
    scaledWidth = Convert.ToUInt32(relation * scaledHeight);
  }
  var transform = new BitmapTransform()
  { ScaledWidth = scaledWidth, ScaledHeight = scaledHeight };
    PixelDataProvider pixelData = await decoder.GetPixelDataAsync(
    BitmapPixelFormat.Rgba8,
    BitmapAlphaMode.Straight,
    transform,
    ExifOrientationMode.RespectExifOrientation,
    ColorManagementMode.DoNotColorManage);
    var folder = ApplicationData.Current.RoamingFolder;
    StorageFile destinationFile = await folder.CreateFileAsync(filename);
    using (var destinationStream = await destinationFile.OpenAsync(
    FileAccessMode.ReadWrite))
    {
      BitmapEncoder encoder = await BitmapEncoder.CreateAsync(
        BitmapEncoder.PngEncoderId, destinationStream);
      encoder.SetPixelData(BitmapPixelFormat.Rgba8,
        BitmapAlphaMode.Premultiplied,
        scaledWidth, scaledHeight, 96, 96, pixelData.DetachPixelData());
      await encoder.FlushAsync();
    }
  }
```

如本章前面所述，应用程序包含一些示例数据，这为用户提供了一个很好的起点，且有助于演示应用程序的用法。菜单预定义的文本内容使用代码创建，并填充到菜单卡。图像和应用程序需要

的徽标存储在Assets文件夹中。第一次启动应用程序时，最好把菜单卡和图像写入移动文件夹，以便以相同的方式处理它们和用户创建的内容。

应用程序第一次启动的实现是在App类的InitSampleDataAsync方法中定义的(代码文件App.xaml.cs)。该方法本身在同一个类的OnLaunched处理方法中调用。该方法首先验证移动文件夹是否为空。如果不为空，就表示已经写入了菜单卡。GetSampleMenuCards方法返回用示例数据填充的菜单卡，包括到Assets文件夹中图像的链接。这些图像使用RandomAccessStreamReference检索，创建这样一个对象可以用CreateFromUri方法完成。CreateFromFile和CreateFromStream是创建RandomAccessStreamReference对象的另外两个选项。在RandomAccessStreamReference对象中，调用OpenReadAsync方法，获得Windows Runtime数据流。这是用WriteImageAsync方法写入图像所需的数据流类型：

```
private static async Task InitSampleDataAsync()
{
  var storage = new MenuCardStorage();
  var imageStorage = new MenuCardImageStorage();
  if (await storage.IsRoamingFolderEmpty())
  {
    List<MenuCard> menuCards = MenuCardFactory.GetSampleMenuCards().ToList();
    foreach (var card in menuCards)
    {
      RandomAccessStreamReference streamRef =
      RandomAccessStreamReference.CreateFromUri(new Uri(card.ImagePath));
      using (IRandomAccessStreamWithContentType stream =
      await streamRef.OpenReadAsync())
      {
        card.ImagePath = string.Format("{0}.png", Guid.NewGuid());
        await imageStorage.WriteImageAsync(stream, card.ImagePath);
      }
    }
    await storage.WriteMenuCardsAsync(menuCards);
  }
}
```

38.7.5　读取图像

与写入图像相比，读取图像比较简单。在ReadImageAsync方法中(代码文件MenuCard/Storage/MenuCardImageStorage.cs)，首先打开一个文件，来创建IRandomAccessStreamWithContentType，这个数据流会传递给BitmapImage。该图像在读取完成之前返回，因为这些都是异步进行的。要获得成功或失败信息，可以给ImageOpened和ImageFailed事件添加事件处理程序。如果图像的路径不正确或是出现另一个故障，这两个事件将非常有帮助：

```
public async Task<ImageSource> ReadImageAsync(string filename)
{
  StorageFolder folder = ApplicationData.Current.RoamingFolder;
  StorageFile file = await folder.CreateFileAsync(filename,
    CreationCollisionOption.OpenIfExists);
  var image = new BitmapImage();
  image.SetSource(await file.OpenReadAsync());
  image.ImageOpened += (sender1, e1) =>
  {
```

```
  };
  image.ImageFailed += (sender1, e1) =>
  {
  };
  return image;
}
```

38.8 选择器

出于安全原因，如果不与用户交互，Windows Store 应用程序就不能在任何位置上执行读写操作。对于这类任务，可以使用选择器。在存储器中，可以使用 FileOpenPicker 打开一个或多个文件，使用 FileSavePicker 选择文件名、文件夹和文件扩展名来保存文件，使用 FolderPicker 选择文件夹。

AddMenuCardPage 类中的 OnUploadImage 方法(代码文件 MenuCard/AddMenuCardPage.xaml.cs)使用 FileOpenPicker，让用户选择要上传的文件。PickSingleFileAsync 方法返回单个文件。如果用户应选择多个文件，就可以使用 PickMultipleFilesAync。在配置选择器时，应定义开始位置(这里是图片库)和可以选择的文件扩展名。然后读取从选择器中返回的 StorageFile，将图像写入菜单卡。前面讨论图像的写入时，学习了如何使用 BitmapDecoder 重置图像的大小。也可以使用 BitmapImage 定义解码方式(这个方法更简单)，如实现为 lambda 表达式的 ImageOpened 事件处理程序所示：

```
private async void OnUploadImage(object sender, RoutedEventArgs e)
{
  var filePicker = new FileOpenPicker();
  filePicker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
  filePicker.FileTypeFilter.Add(".jpg");
  filePicker.FileTypeFilter.Add(".png");
  StorageFile file = await filePicker.PickSingleFileAsync();
  if (file == null) return;
  var stream = await file.OpenAsync(FileAccessMode.Read);
  var image = new BitmapImage();
  image.SetSource(stream);
  image.ImageOpened += async (sender1, e1) =>
  {
    if (image.PixelHeight > image.PixelWidth)
    {
      image.DecodePixelHeight = 900;
    }
    else
    {
      image.DecodePixelWidth = 900;
    }
    stream.Seek(0);
    MenuCardImageStorage imageStorage = new MenuCardImageStorage();
    MenuCardStorage storage = new MenuCardStorage();
    info.ImageFileName = string.Format("{0}.jpg", Guid.NewGuid().ToString());
    await imageStorage.WriteImageAsync(stream, info.ImageFileName);
  };
  image.ImageFailed += (sender1, e1) =>
  {
    // log error
  };
```

```
    info.Image = image;
  }
```

FileOpenPicker 如图 38-12 所示。这个选择器为所选文件夹中的图像提供了预览。

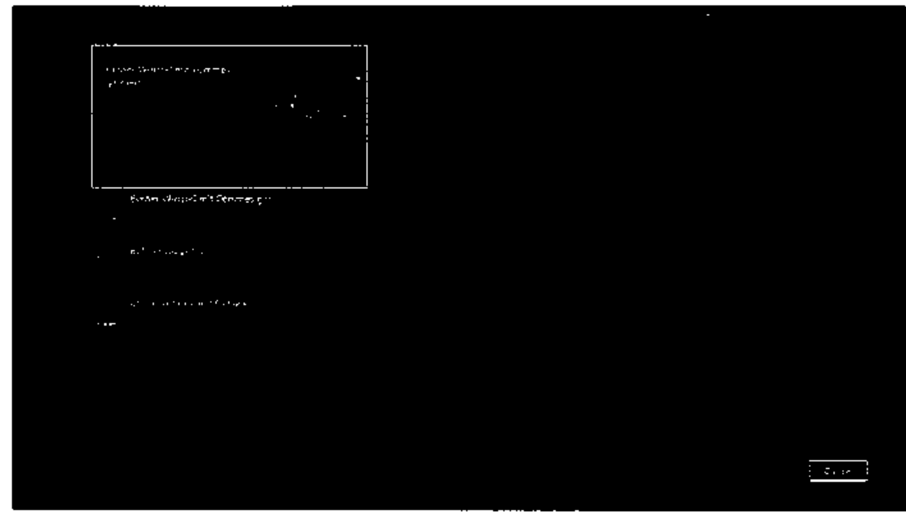


图 38-12

38.9 活动的磁贴

因为磁贴是应用程序的第一个入口点，所以它们应能吸引用户的注意力。使用清单设计器(如图 38-13 所示)，可以为徽标磁贴指定几个尺寸：一个小徽标(70×70 像素，它没有活动的磁贴变体)；两个方形徽标(150×150 像素和 310×310 像素)；一个宽徽标 310×150 像素。磁贴可以显示名称和图像。如果名称应显示出来，就应为普通图像和宽图像单独配置。也可以指定前景色和背景色。

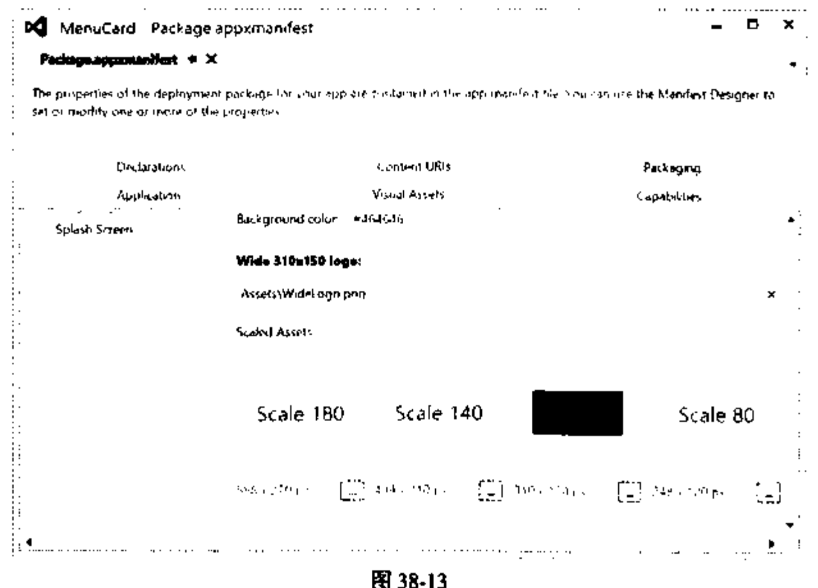


图 38-13

为了支持所有不同的设备分辨率和比例，最好提供不同大小的磁贴。根据图像的不同，自动设置的比例可能不合适。清单设计器显示了所有不同比例需要的像素数。为了让同一个图像可用于不

同的比例，文件要遵循一个命名约定：在文件名的中间是 scale-xxx，之后是文件扩展名。例如，WideLogo.scale-100.png 文件名是 WideLogo.png 的 100%图像，WideLogo.scale-180.png 是 180%的图像。

磁贴还可以在应用程序中动态改变。下面的代码段显示了 UpdateTile 方法中更新磁贴的示例代码(代码文件 Notification/Tile Update.cs)。要更新磁贴，需要创建一个 XML 定义，来定义磁贴和一个可选的通知，该通知定义了磁贴的终止时间，接着就使用 TileUpdater 类更新磁贴：

```
public static void UpdateTile()
{
  TileTemplateType tileTemplate = TileTemplateType.TileWide310x150ImageAndText01;
  XmlDocument tileXml = TileUpdateManager.GetTemplateContent(tileTemplate);
  XmlNodeList tileImageAttributes = tileXml.GetElementsByTagName("image");
  ((XmlElement)tileImageAttributes[0]).SetAttribute("src",
  "ms-appx:///Assets/breakfast400.jpg");
  ((XmlElement)tileImageAttributes[0]).SetAttribute("alt", "Breakfast");
  var textElements = tileXml.GetElementsByTagName("text");
  ((XmlElement)textElements[0]).InnerText = "MENU card";
  TileNotification notification = new TileNotification(tileXml);
  notification.ExpirationTime = DateTimeOffset.Now.AddMinutes(60);
  TileUpdater tileUpdater = TileUpdateManager.CreateTileUpdaterForApplication();
  tileUpdater.Update(notification);
}
```

下面详细分析这段代码。注意不同应用程序的许多磁贴看起来非常相似，这有一个很好的理由。TileTemplateType 定义了磁贴的几个可定制模板。示例使用的模板是 TileWide310×150ImageAndText01。顾名思义，这个模板显示了带有图像和文本的宽磁贴。其他模板只能用于文本或图像、方块或图像集。TileWide310×150ImageAndText01 包含一个图像和文本，其中文本可以换行，而 TileWide310×150ImageAndText02 包含图像和文本，但文本不能换行。本例使用 Windows.Data.Xml.Dom 命名空间中的 XML 类，修改 XML 内容，以添加 Assets 文件夹中的一个图像和一些文本。磁贴的最终 XML 代码如下所示：

```
<tile>
  <visual>
    <binding template="TileWideImageAndText01">
      <image id="1" src="ms-appx:///Assets/breakfast400.jpg" alt="Breakfast"/>
      <text id="1">MENU card</text>
    </binding>
  </visual>
</tile>
```

在 XML 内容中，创建了一个 TileNotification。ExpirationTime 指定磁贴在 60 分钟后重置。最后在 TileUpdater 中更新。TileUpdater 在 CreateTileUpdaterForApplication 方法中创建。因为一个应用程序可以有多个磁贴，所以其他磁贴可以用 CreateTileUpdaterForSecondaryTile 更新。

更新不只能进行一次，还可以指定定期更新。StartPeriodicUpdate 方法允许指定服务器的 URL，按指定的时间间隔重复调用。时间间隔用一个枚举指定，该枚举定义了从每半小时到一天的时间间隔值。服务器必须返回磁贴的 XML 代码。定期更新可以在不激活应用程序的情况下运行。启动 TileUpdater，就告诉 Windows 执行磁贴更新。

Windows 8.1 提供了一个简单的选项，使用服务器中的一个链接来更新磁贴。使用清单编辑器，

可以在 Tile Update 设置中添加一个 URI Template 链接。重复发生的设置为从每半小时到一天的时间间隔值，它指定了 Windows 请求服务器更新磁贴的频率。服务器只需给磁贴返回需要的 XML 格式。这种实现方法可以通过第 44 章介绍的 ASP.NET Web API 来完成。以这种方式更新磁贴，应用程序就不需要运行。Windows 本身会请求 URL 更新，来修改磁贴的内容。

38.10 小结

本章介绍了编写 Windows Store 应用程序的许多不同方面。XAML 与前几章编写 WPF 应用程序非常相似。数据绑定、内容控件和项控件一起使用。利用 Visual State Manager 处理不同的布局变化。Windows Runtime 访问存储器，以读写数据和图像，使用移动存储器。使用 FileOpenPicker，通过与用户交互来上传文件。最后介绍了磁贴，这是应用程序第一个重要的入口点。

当然，设计 Windows Store 应用程序还有许多内容。还有更多的选择器(如联系人选择器)，为应用程序提供文件打开选择器的协定；改进的搜索功能(允许应用程序使用 toast 给用户提供信息)等。可惜，本章篇幅有限，不可能全面介绍这些主题。尽管如此，读者现在已有足够的知识开始编写 Windows Store 应用程序了。

下一章介绍 Windows Store 应用程序的更多内容，包括协定和设备。

第39章

Windows Store 应用程序：协定和设备

本章要点

- 搜索
- 共享
- 相机
- 定位
- 感应器

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章的代码只包含一个大示例，它展示了本章的各个方面：

- 菜单卡(Menu Card)
- 相机示例(Camera Sample)
- 定位示例(Geolocation Sample)
- 感应器示例(Sensor Sample)
- 旋转的钢珠(Rolling Marble)

39.1 概述

上一章介绍了 Windows Store 应用程序的 UI 元素，以及使用 Windows 运行库的 API 读写数据。本章继续介绍如何添加搜索功能，如何使用共享协定允许集成不同的应用程序，如何使用许多不同的设备，例如用相机拍照、录制视频、获得用户的位置信息、使用几个感应器(例如加速计和倾斜计)获得用户如何移动设备的信息。

下面先介绍搜索，它在 Windows 8 和 Windows 8.1 之间的区别很大。

39.2 搜索

在 Windows 8 中，搜索 Windows Store 应用程序通常是通过搜索协定完成的。搜索应用程序时，首先打开 Charms 栏，其问题是即使应用程序支持搜索，用户也很难找到。Windows 8.1 改进了这一点，支持搜索的应用程序在 UI 上使用一个新的 SearchBox 控件。

上一章的示例应用程序 Menu Card 将用搜索功能加以扩展。在主页上，用户可以搜索菜单项。如果找到了一个菜单项，就打开包含该菜单项的菜单卡页面。

首先在主页上添加 SearchBox 控件(代码文件 MenuCard/MainPage.xaml)：

```
<SearchBox
  QuerySubmitted="OnSearchQuery"
  SuggestionsRequested="OnSuggestionRequested" />
```

上面添加了两个事件处理程序。只要用户完成查询输入，就触发事件 QuerySubmitted。在此之前，可以给用户提供建议。用户不需要完成查询的输入，就可以选择其中一个建议。要添加建议，必须添加 SuggestionsRequested 事件的事件处理程序。

要列出用户可以搜索的一组单词，应使用 LINQ 查询创建映射到菜单卡的一个单词查找列表。LINQ 查询迭代菜单卡中的所有菜单项，把单词从菜单项的文本中分离出来，去除填充词，再创建一个查找表，其中包含映射到 MenuCard 对象的单词(代码文件 MenuCard/MainPage.xaml.cs)：

```
private ILookup<string, MenuCard> GetSearchWords()
{
  string[] fillWords = { "der", "die", "mit", "und", "im", "auf" };

  return cards.SelectMany(card => card.MenuItems).
    SelectMany(mi => mi.Text.Split(),
               (mi, word) => new { MenuItem = mi, Word = word }).
    Where(item => !fillWords.Contains(item.Word)).
    ToLookup(item => item.Word, item => item.MenuItem.MenuCard);
}
```

包含复合 from 语句(或 SelectMany 方法)的 LINQ 参见第 11 章。

属性 WordsLookup 和 Keys 使用 GetSearchWords 方法，很容易访问建议：

```
private ILookup<string, MenuCard> wordsLookup;
public ILookup<string, MenuCard> WordsLookup
{
  get
  {
    return wordsLookup ?? (wordsLookup = GetSearchWords());
  }
}
```

```
public IEnumerable<string> Keys
{
  get
  {
    return WordsLookup.Select(w => w.Key);
  }
}
```

只要使用 SearchBox，就会触发 SuggestionsRequested 事件。在这个事件的处理程序 OnSuggestionRequested 中，SearchBoxSuggestionsRequestedEventArgs 保存了用户输入的查询信息(属性 QueryText)，这里还可以传送建议(属性 SearchSuggestionCollection)。

万一查询文本仍为空(SuggestionsRequested 事件在用户输入第一个字符前触发)，事件处理程序就会返回。输入文本后，处理程序会检查输入的文本是否能在单词查找表的关键字中找到。如果找到了，就添加建议，用户很容易选择其中一个建议，而不是继续写入文本：

```
private void OnSuggestionRequested(SearchBox sender,
  SearchBoxSuggestionsRequestedEventArgs args)
{
  if (string.IsNullOrEmpty(args.QueryText))
    return;
  string query = args.QueryText;
  var suggestions = this.Keys.Where(k => k.StartsWith(query)).ToList();
  args.Request.SearchSuggestionCollection.AppendQuerySuggestions(
    suggestions);
}
```

用户单击搜索按钮或选择一个建议时，会触发 SearchQuery 事件。该事件的处理程序接收 SearchBoxQuerySubmittedEventArgs 来读取 QueryText。之后，导航到 MenuItemsPage，打开找到的 MenuCard，其中包含搜索词：

```
private async void OnSearchQuery(SearchBox sender,
  SearchBoxQuerySubmittedEventArgs args)
{
  string query = args.QueryText;
  var cards = WordsLookup[query];
  MenuCard card = cards.FirstOrDefault();
  if (card != null)
  {
    this.Frame.Navigate(typeof(MenuItemsPage), card);
  }
  else
  {
    MessageDialog dlg = new MessageDialog("Word not found");
    await dlg.ShowAsync();
  }
}
```

图 39-1 显示的 Menu Card 应用程序在搜索框中显示了建议。

图 39-1

39.3 共享协定

如果应用程序提供与其他应用程序的交互，就会更有用。不需要像桌面应用程序那样复制粘贴，Menu Card 应用程序可以通过电子邮件提供直接使用的数据，或者通过应用程序把信息传递给网站。该应用程序也可以接收其他应用程序的信息，如用于菜单卡的图片。

这种与 Windows Store 应用程序的通信是使用协定实现的。下面先将示例应用程序变成一个共享源，再变成一个共享目标。

39.3.1 共享源

关于共享，首先要考虑的是确定哪些数据以何种格式共享。可以共享简单文本、富文本、HTML 和图像，也可以共享自定义类型。当然，其他应用程序(即共享目标)必须知道且能使用所有这些类型。对于自定义类型，只有知道该类型且是该类型的共享目标的应用程序才能共享它。示例应用程序仅为菜单卡提供了 HTML 代码。

菜单卡的 HTML 代码在 MenuCardExtensions 类的 ToHtml 方法中创建为 MenuCard 类型的一个扩展方法(代码文件 MenuCard/Extensions/MenuCardExtensions.cs)。其中，使用 LINQ to XML 从菜单卡包含的 MenuItem 对象的 Text 和 Price 属性中创建 HTML 内容：

```
static class MenuCardExtensions
{
  public static string ToHtml(this MenuCard card)
  {
    return
      new XElement("table",
        new XElement("thead",
        new XElement("td", "Text"),
        new XElement("td", "Price"),
        card.MenuItems.Select(mi =>
          new XElement("tr",
```

```
        new XElement("td", mi.Text),
        new XElement("td", mi.Price.ToString("C")))))).ToString();
  }
}
```

对 LINQ to XML 的介绍参见第 34 章。

打开 Menu 项页面时，从 Menu Card 中提供共享——共享一个菜单卡。共享数据的核心是 DataTransferManager。用户打开 Charms 栏，请求共享时，就触发 DataRequested 事件。在 Menu 项页面中(代码文件 MenuCard/MenuItemsPage.xaml.cs)，事件处理程序在 InNavigatedTo 方法中注册，在 OnNavigatedFrom 方法中注销：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  DataTransferManager.GetForCurrentView().DataRequested +=
    OnShareDataRequested;

  navigationHelper.OnNavigatedTo(e);
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
  navigationHelper.OnNavigatedFrom(e);

  DataTransferManager.GetForCurrentView().DataRequested -=
  OnShareDataRequested;
}
```

该事件触发时，会调用 OnShareDataRequested 方法。这个方法把 DataTransferManager 作为第一个参数，把 DataRequestedEventArgs 作为第二个参数接收。在共享数据时，需要填充 args.Request.Data 引用的 DataPackage。Title、Description 和 Thumbnail 属性可以用于给用户界面提供信息。应共享的数据必须用一个 SetXXX 方法来传递。示例代码共享 HTML 代码，所以使用 SetHtmlFormat 方法。HtmlFormatHelper 类帮助创建共享所需的其余 HTML 代码。菜单卡的 HTML 代码用前面介绍的 ToHtml 扩展方法创建：

```
private void OnShareDataRequested(DataTransferManager sender,
  DataRequestedEventArgs args)
{
  Uri baseUri = new Uri("ms-appx:///");
  DataPackage package = args.Request.Data;
  package.Properties.Title = string.Format("MENU card {0}", card.Title);
  if (card.Description != null)
  {
    package.Properties.Description = card.Description;
  }
  package.Properties.Thumbnail = RandomAccessStreamReference.CreateFromUri(
    new Uri(baseUri, "Assets/Logo.png"));
  package.SetHtmlFormat(HtmlFormatHelper.CreateHtmlFormat(
    card.ToHtml()));
}
```

除了提供HTML代码之外，其他方法(如SetBitmap、SetRtf和SetUri)可以提供其他数据格式。

如果需要共享操作何时完成的信息，例如，为了从源应用程序中删除数据，DataPackage类会触发OperationCompleted和Destroyed事件。

图39-2显示了从Charms工具栏中激活共享的过程。其中，Mail应用程序是接受HTML内容的唯一共享目标。

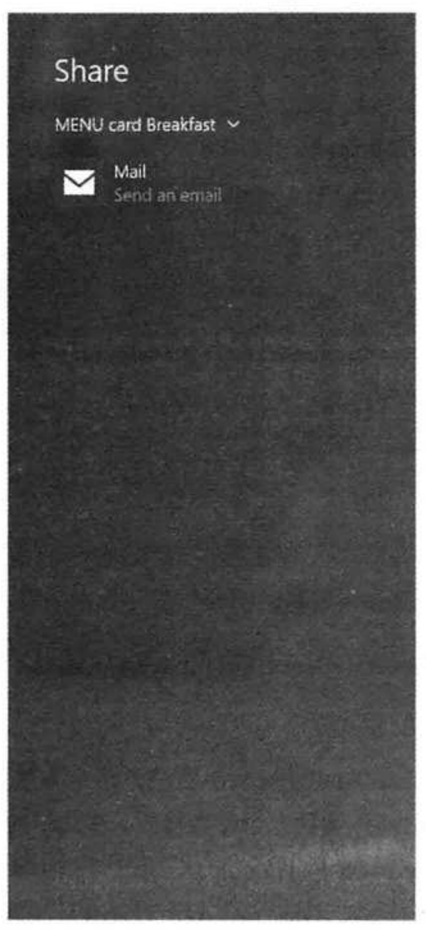


图39-2

选择Mail应用程序，触发DataRequested事件，Menu Card应用程序就把菜单卡信息传递给DataPackage，而DataPackage又是从Mail应用程序中接收的。图39-3显示了Mail应用程序格式化接收到的HTML内容，在这里可以用电子邮件直接发送数据。

如果用户选择在没有提供共享功能的页面中与应用程序共享，就会显示一个错误，指出应用程序不支持共享。要为用户获得实现共享所需的更多信息，最好在其他页面上也实现共享功能，调用DataRequest对象上的FailWithDisplayText方法，显示一条消息(代码文件MenuCard/MainPage.xaml.cs)：

```
private void OnShareDataRequested(DataTransferManager sender,
  DataRequestedEventArgs args)
{
  args.Request.FailWithDisplayText("Open a menu card beforesharing");
}
```

图 39-3

39.3.2　共享目标

现在看看共享内容的接收者。如果应用程序应从共享源中接收信息，就需要将其声明为共享目标。图 39-4 显示了清单设计器在 Visual Studio 中的 Declarations 页面，在其中可以定义共享目标。在这里添加 Share Target 声明，它至少要包含一种数据格式。可能的数据格式是 Text、URI、Bitmap、HTML、StorageItems 或 RTF。还可以添加文件扩展名，以指定应支持哪些文件类型。

图 39-4

在注册应用程序时，要使用软件包清单中的信息。这告诉 Windows，哪些应用程序可用作共享目标。用户可以在 PC 设置页面的 Share 对话框中配置这些信息，如图 39-5 所示。

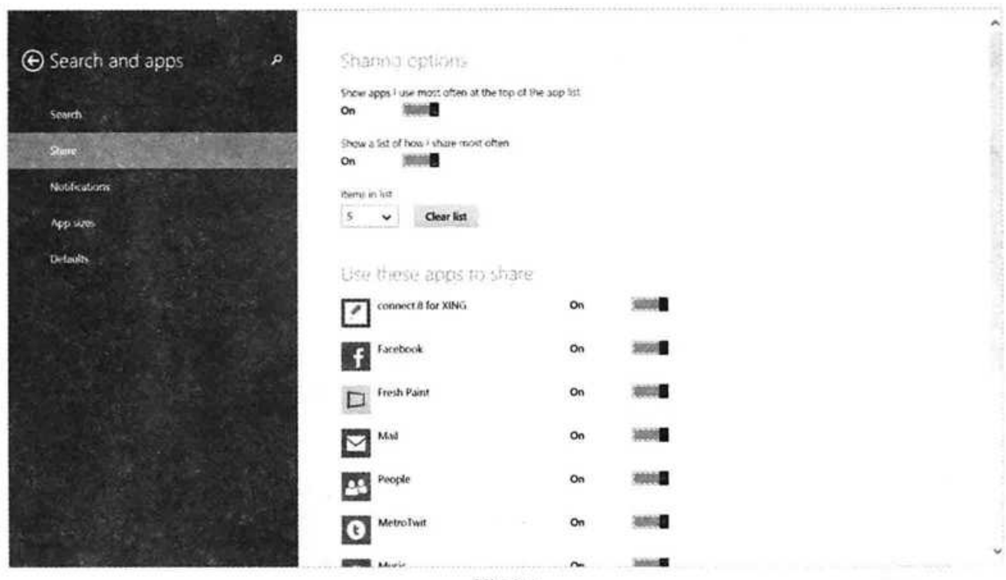


图 39-5

使用 Visual Studio 项模板 Share Target Contract 可以创建共享目标所需要的基础代码。共享源提供要共享的数据，且用户选择了可用的共享目标后，就调用 App 类中的 OnShareTargetActivated 方法。在激活共享目标时，并没有调用以前使用的 OnLaunched 方法，而调用了 OnShareTargetActivated 方法。在其实现代码中，激活了要用于共享请求的页面。这个页面由 Charms 工具栏显示，以用于共享。

```
protected override void OnShareTargetActivated(
  ShareTargetActivatedEventArgs e)
{
  var shareTargetPage = new ShareTargetApp.ShareTargetPage();
  shareTargetPage.Activate(e);
}
```

ShareTargetPage1 类中的 Activate 方法使用 ShareTargetActivatedEventArgs 对象接收激活信息。ShareOperation 属性返回一个包含 DataPackageView 的 ShareOperation 对象。该视图提供了可用数据的信息。可用数据可以用 AvailableFormats 属性检索，根据格式的不同，使用对应的方法(如 GetTextAsync 和 GetBitmapAsync)访问这些数据：

```
public async void Activate(ShareTargetActivatedEventArgs args)
{
  this._shareOperation = args.ShareOperation;

  var shareProperties = this._shareOperation.Data.Properties;
  if (sharedProperties == null) return;
  var thumbnailImage = new BitmapImage();
  this.DefaultViewModel["Title"] = shareProperties.Title;
  this.DefaultViewModel["Description"] = shareProperties.Description;
  this.DefaultViewModel["Image"] = thumbnailImage;
  this.DefaultViewModel["Sharing"] = false;
  this.DefaultViewModel["ShowImage"] = false;
  this.DefaultViewModel["Comment"] = String.Empty;
  this.DefaultViewModel["SupportsComment"] = true;
```

```
    Window.Current.Content = this;
    Window.Current.Activate();

    // Update the shared content's thumbnail image in the background
    if (shareProperties.Thumbnail != null)
    {
      var stream = await shareProperties.Thumbnail.OpenReadAsync();
      thumbnailImage.SetSource(stream);
      this.DefaultViewModel["ShowImage"] = true;
    }
  }
```

检索数据后，就必须用 ReportStarted 和 ReportCompleted 方法来通信：

```
private void ShareButton_Click(object sender, RoutedEventArgs e)
{
  this.DefaultViewModel["Sharing"] = true;
  this._shareOperation.ReportStarted();
  // TODO: Perform work appropriate to your sharing scenario using
  //       this._shareOperation.Data, typically with additional information
  //       captured through custom user interface elements added to this page
  //       such as this.DefaultViewModel["Comment"]
  this._shareOperation.ReportCompleted();
}
```

要测试是否可以用希望支持的所有格式来共享，最佳方式是使用示例应用程序 Sharing content source 和 Sharing content target。这两个应用程序示例都可以从 http://code.msdn.microsoft.com/windowsapps/上获得。如果把一个应用程序作为共享源，就使用示例目标应用程序，反之亦然。

39.4　相机

应用程序的可视化越来越强，越来越多的设备提供了一两个相机内置功能，所以使用这个功能就越来越成为应用程序的一个重要方面——这很容易通过 Windows 运行库实现。

使用相机需要在清单编辑器中配置 Webcam 功能。要录制视频，还需要配置 Microphone 功能。

照片和视频可以用 CameraCaptureUI 类(在名称空间 Windows.Media.Capture 中)捕获。首先，照片和视频设置需要配置为使用下面的 CaptureFileAsync 方法。第一个代码段(代码文件 Camera Sample/MainPage.xaml.cs)捕获照片。在实例化 CameraCaptureUI 类后，就应用 PhotoSettings。可能的照片格式有 JPG、JPGXR 和 PNG。也可以定义剪辑，相机捕获功能的 UI 直接要求用户，根据剪辑大小从完整的图片中选择一个剪辑。对于剪辑，可以用 CroppedSizeInPixels 属性定义像素大小，或用

CroppedAspectRatio 定义一个比例。拍照后，示例代码会使用 CaptureFileAsync 方法返回的 StorageFile，把它存储为一个文件，通过 FolderPicker 放在用户选择的文件夹中。

```
private async void OnTakePhoto(object sender, RoutedEventArgs e)
{
  var cam = new CameraCaptureUI();
  cam.PhotoSettings.AllowCropping = true;
  cam.PhotoSettings.Format = CameraCaptureUIPhotoFormat.Png;
  cam.PhotoSettings.CroppedSizeInPixels = new Size(300, 300);
  StorageFile file = await cam.CaptureFileAsync(CameraCaptureUIMode.Photo);
  if (file != null)
  {
    var picker = new FolderPicker();
    picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
    picker.FileTypeFilter.Add(".png");
    StorageFolder folder = await picker.PickSingleFolderAsync();
    await file.CopyAsync(folder);
  }
}
```

第二段代码用于录制视频。与前面类似，首先需要进行配置。除了 PhotoSettings 属性之外，CameraCaptureUI 类还定义了 VideoSettings 属性。可以根据最大分辨率和最大持续时间限制所录制的视频(使用枚举值 CameraCaptureUIMaxVideoResolution.HighestAvailable 允许用户选择任何可用的分辨率)。可能的视频格式有 WMV 和 MP4：

```
private async void OnRecordVideo(object sender, RoutedEventArgs e)
{
  var cam = new CameraCaptureUI();
  cam.VideoSettings.AllowTrimming = true;
  cam.VideoSettings.MaxResolution =
    CameraCaptureUIMaxVideoResolution.StandardDefinition;
  cam.VideoSettings.Format = CameraCaptureUIVideoFormat.Wmv;
  cam.VideoSettings.MaxDurationInSeconds = 5;
  StorageFile file = await cam.CaptureFileAsync(
    CameraCaptureUIMode.Video);

  if (file != null)
  {
    var picker = new FolderPicker();
    picker.SuggestedStartLocation = PickerLocationId.VideosLibrary;
    picker.FileTypeFilter.Add(".wmv");
    StorageFolder folder = await picker.PickSingleFolderAsync();
    await file.CopyAsync(folder);
  }
}
```

如果用户要捕获视频或图片，就可以把 CameraCaptureUIMode.PhotoOrVideo 参数传送给 CaptureFileAsync 方法。

39.5 定位

知道用户的位置是应用程序的一个重要方面。应用程序可能要显示地图，还有许多其他场景需

要知道用户的位置。例如应用程序要显示用户所在区域的天气情况，或者需要确定用户的数据应保存到哪个最近的云中心。

有了 Geolocator(在名称空间 Windows.Devices.Geolocation 中)，很容易确定用户的位置。示例应用只包含一个按钮(单击它时请求位置)和一个 TextBlock 元素(绑定到位置请求的结果，代码文件 GeolocationSample/MainPage.xaml)：

```
<TextBlock Grid.Row="0" Grid.Column="1" Margin="20"
  Text="{Binding GeoResult, Mode=OneWay}"
  Style="{StaticResource BodyTextBlockStyle}" />
```

单击按钮，就调用事件处理方法 OnGeolocation(代码文件 GeolocationSample/MainPage.xaml.cs)。在实现代码中，实例化一个 Geolocator 对象，调用 GetGeopositionAsync 方法。在定位器中，设置属性 DesiredAccuracy，可以配置需要的精度。Default 和 High 是 PositionAccuracy 枚举的两个可能值。也可以设置属性 DesiredAccuracyInMeters，以“米”为单位指定需要的精度。如果应用程序不需要精度设置，就不应把精度设置为 High(或较小的米数)。设备可以使用不同的功能获得位置，这可能会增加电池的消耗。例如，如果设备支持 GPS，在需要高精度时就会打开它：

```
private async void OnGetLocation(object sender, RoutedEventArgs e)
{
  bool hasError = false;
  try
  {
    var locator = new Geolocator();
    locator.DesiredAccuracy = PositionAccuracy.High;
    Geoposition position = await locator.GetGeopositionAsync();

    DefaultViewModel["GeoResult"] = GetGeoInfo(position);
  }
  catch (UnauthorizedAccessException)
  {
    hasError = true;
  }
  if (hasError)
  {
    var dlg = new MessageDialog("Geolocation permission required");
    await dlg.ShowAsync();
  }
}
```

从 GetGeopositionAsync 方法返回 Geoposition 对象，GetGeoInfo 方法会分析该对象返回的结果，并写入一个结果字符串。写入的信息是纬度、经度和高度。并不是每个设备都返回高度值，此时应使用值 0。PositionSource 属性返回位置信息的来源。根据设备的支持情况，以及精度是设置为 Default 还是 High，可能会得到不同的源信息。位置的可能来源有设备的 IP 地址、无线网络、手机网络数据或行星信息(和 GPS)。Accuracy 属性返回结果的精确程度，其精度单位是米：

```
private string GetGeoInfo(Geoposition position)
{
  StringBuilder result = new StringBuilder();
  result.AppendFormat("latitude: {0}\n",
    position.Coordinate.Point.Position.Latitude);
```

```
  result.AppendFormat("longitude: {0}\n",
    position.Coordinate.Point.Position.Longitude);
  result.AppendFormat("altitude: {0}\n",
    position.Coordinate.Point.Position.Altitude);
  result.AppendFormat("source: {0}\n", position.Coordinate.PositionSource);
  result.AppendFormat("accuracy: {0} m\n", position.Coordinate.Accuracy);
  return result.ToString();
}
```

Geoposition对象有一些比前面更有趣的属性。例如，它包含CivicAddress和属性City、Postalcode、State、Country。但是，只填充了Country属性，其值不取自位置坐标，而取自用户设置。填充CivicAddress需要城市地址提供程序。但是，编写这种提供程序似乎没有什么乐趣，所以要获得城市地址，就需要其他库，例如Bing Map API。

除了一次获得位置之外，位置还可以根据时间段或用户的移动来检索。使用Geolocator，可以把ReportInterval属性设置为位置更新的最小时间段(单位为毫秒)。例如，另一个应用程序需要较短时间段的位置信息，更新就可能比较频繁。除了时间段之外，也可以指定用户的移动来触发位置信息的获得。属性MovementThreshold指定了这个移动(以米为单位)。

设置了时间段或移动阈值后，PositionChanged事件会在每次更新位置时触发：

```
private GeoLocator locator;
private void OnGetContinuousLocation(object sender, RoutedEventArgs e)
{
  locator = new Geolocator();
  locator.DesiredAccuracy = PositionAccuracy.High;
  // locator.ReportInterval = 1000;
  locator.MovementThreshold = 10;
  locator.PositionChanged += (sender1, e1) =>
    {
      DefaultViewModel["GeoResult"] = GetGeoInfo(e1.Position);
    };
  locator.StatusChanged += (sender1, e1) =>
    {
      DefaultViewModel["StatusChanged"] = e1.Status;
    };
}
```

图39-6中的示例应用程序显示了纬度、经度、来源和精度。

用位置的变化来调试应用程序时，并不需要用户现在就钻进一辆汽车，边开车边调试应用程序。模拟器是一个很有帮助的工具。

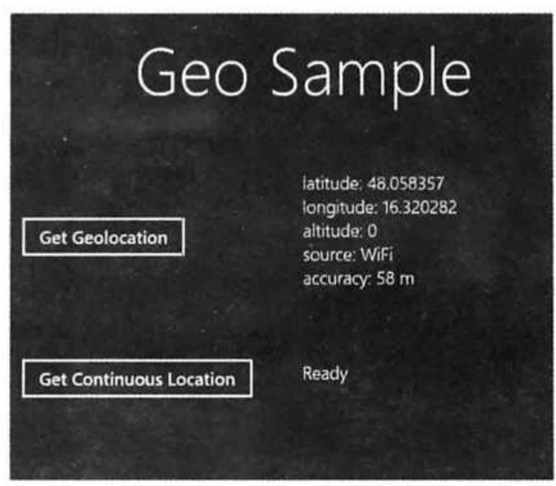


图 39-6

39.6 感应器

Windows 运行库可以直接访问许多感应器。名称空间 Windows.Devices.Sensors 包含了用于几个感应器的类，这些感应器可以通过不同的设备使用。

在介绍代码之前，先在表 39-1 中概述不同的感应器及其用途。一些感应器的功能非常明确，但其他感应器需要一些解释。

表 39-1

感 应 器	功 能
光线	光线感应器返回单位为勒克斯的光线。这个信息由 Windows 本身用于设置屏幕亮度
罗盘	罗盘提供了用磁力计测量出的设备偏离北方的角度。这个感应器区分磁力北方和地理北方
加速计	加速计测量 x、y 和 z 设备轴上的重力值。应用程序可以使用这个感应器显示在屏幕上滚过的钢珠
陀螺仪	陀螺仪测量沿着 x、y 和 z 设备轴上的角速度。如果应用程序关注设备的旋转，就可以使用这个感应器。但是，移动设备也会影响陀螺仪的值。可能需要用加速计的值来补偿陀螺仪的值，以去除设备的移动，只处理实际的角速度
倾斜计	倾斜计给出了设备绕 x 轴(倾斜)、y 轴(滚动)和 z 轴(偏航)的角度值。应用程序显示匹配倾斜、滚动和偏航的飞机时，可以使用这个感应器
方向	方向使用来自加速计、陀螺仪和磁力计的数据，以四元数和旋转矩阵来提供数值

感应器数据的一个重要方面是感应器根据使用设备方向的坐标来返回信息。这与显示方向时使用的坐标相反。例如，Surface Pro 默认用底部的 Windows 按钮水平定位，其 x 轴指向右，y 轴指向上，z 轴从用户指向外部。

使用感应器的示例应用程序显示单槽(cone-shot)的所有感应器的结果，且继续使用事件。使用这个应用程序可以看出，哪些感应器数据可以用于设备，移动设备时会返回哪些数据。

39.6.1 光线

知道如何使用一种感应器后，其他感应器的用法是非常相似的。下面先看看 LightSensor。首先，调用静态方法 GetDefault 访问一个对象。调用 GetCurrentReading 方法可以获得感应器的实际值。对于 LightSensor，GetCurrentReading 返回 LightSensorReading 对象。这个读数对象定义了 IlluminanceInLux 属性，它返回单位为勒克斯的照明度(代码文件 SensorSample/MainPage.xaml.cs)：

```
private void OnGetLight(object sender, RoutedEventArgs e)
{
  LightSensor sensor = LightSensor.GetDefault();
  if (sensor != null)
  {
    LightSensorReading reading = sensor.GetCurrentReading();
    this.DefaultViewModel["LightResult"] =
      string.Format("Illuminance: {0} Lux", reading.IlluminanceInLux);
  }
  else
  {
    this.DefaultViewModel["LightResult"] = "Light sensor not found";
  }
}
```

要获得连续更新的值，应触发 ReadingChanged 事件。指定 ReportInterval 就指定了用于触发事件的时间段。它不能低于 MinimumReportInterval。对于该事件，第二个参数 e1 的类型是 LightSensorReadingChangedEventArgs，用 Reading 属性指定 LightSensorReading：

```
private LightSensor sensor;
private void OnGetLight2(object sender, RoutedEventArgs e)
{
  sensor = LightSensor.GetDefault();
  sensor.ReportInterval = sensor.MinimumReportInterval;
  sensor.ReadingChanged += (sender1, e1) =>
    {
      this.DefaultViewModel["LightResult"] = string.Format(
        "{0:T}\tIlluminance: {1} Lux",
        e1.Reading.Timestamp, e1.Reading.IlluminanceInLux);
    };
}
```

39.6.2 罗盘

罗盘的用法非常类似。GetDefault 方法返回 Compass 对象，GetCurrentReading 检索表示罗盘当前值的 CompassReading。CompassReading 定义了属性 HeadingAccuracy、HeadingMagneticNorth 和 HeadingTrueNorth。

如果 HeadingAccuracy 返回 MagnometerAccuracy.Unknown 或 Unreliable，罗盘就需要校正：

```
private void OnGetCompass(object sender, RoutedEventArgs e)
{
  Compass compass = Compass.GetDefault();
  CompassReading reading = compass.GetCurrentReading();

  this.DefaultViewModel["CompassResult"] = GetCompassResult(reading);
```

```
}

private string GetCompassResult(CompassReading reading)
{
  var sb = new StringBuilder();
  sb.AppendFormat("heading accuracy: {0}\n", reading.HeadingAccuracy);
  sb.AppendFormat("magnetic north: {0}\n", reading.HeadingMagneticNorth);
  sb.AppendFormat("true north: {0}\n", reading.HeadingTrueNorth);
  return sb.ToString();
}
```

罗盘也可以持续更新：

```
private Compass compass;
private void OnGetCompass2(object sender, RoutedEventArgs e)
{
  compass = Compass.GetDefault();
  compass.ReportInterval = compass.MinimumReportInterval;
  compass.ReadingChanged += (sender1, e1) =>
    {
      this.DefaultViewModel["CompassResult"] =
      GetCompassResult(e1.Reading);
    };
}
```

39.6.3 加速计

加速计给出了 x、y 和 z 设备轴上的重力值。对于景观设备，x 轴是水平的，y 轴是垂直的，z 轴从用户指向外部。如果设备底部的 Windows 按钮面对桌面，x 的值就是-1。如果旋转设备，使 Windows 按钮在顶部，x 的值就是+1。

与前面介绍的感应器类似，GetDefault 静态方法返回 Accelerometer，GetCurrentReading 通过 AccelerometerReading 对象给出了加速计的实际值。AccelerationX、AccelerationY 和 AccelerationZ 是可以读取的值：

```
private void OnGetAccelerometer(object sender, RoutedEventArgs e)
{
  Accelerometer accelerometer = Accelerometer.GetDefault();
  AccelerometerReading reading = accelerometer.GetCurrentReading();
  this.DefaultViewModel["AccelerometerResult"] =
  GetAccelerometerResult(reading);
}
private string GetAccelerometerResult(AccelerometerReading reading)
{
  var sb = new StringBuilder();
  sb.AppendFormat("x: {0}\n", reading.AccelerationX);
  sb.AppendFormat("y: {0}\n", reading.AccelerationY);
  sb.AppendFormat("z: {0}\n", reading.AccelerationZ);
  return sb.ToString();
}
```

与其他感应器类似，给 ReadingChanged 事件指定处理程序，就可以获得加速计的连续更新值。这与前面介绍的感应器完全相同，这里不再列出其代码。但使用本章的下载代码可以获得该功能。可以测试设备，不断地移动它，读取加速计的值。

39.6.4 倾斜计

倾斜计用于高级方向，它给出了相对于重力的偏航、倾斜和滚动值(角度)。得到的值用PitchDegrees、RollDegrees和YawDegrees属性指定：

```
private void OnGetInclinometer(object sender, RoutedEventArgs e)
{
  Inclinometer inclinometer = Inclinometer.GetDefault();
  InclinometerReading reading = inclinometer.GetCurrentReading();
  this.DefaultViewModel["InclinometerResult"] =
    GetInclinometerResult(reading);
}

private string GetInclinometerResult(InclinometerReading reading)
{
  var sb = new StringBuilder();
    sb.AppendFormat("pitch {0} degrees\n", reading.PitchDegrees);
  sb.AppendFormat("roll {0} degrees\n", reading.RollDegrees);
  sb.AppendFormat("yaw accuracy {0}\n", reading.YawAccuracy);
  sb.AppendFormat("yaw {0} degrees\n", reading.YawDegrees);
  return sb.ToString();
}
```

39.6.5 陀螺仪

Gyrometer给出了x、y和z设备轴的角速度值：

```
private void OnGetGyrometer(object sender, RoutedEventArgs e)
{
  Gyrometer gyrometer = Gyrometer.GetDefault();
  GyrometerReading reading = gyrometer.GetCurrentReading();
  this.DefaultViewModel["GyrometerResult"] = GetGyrometerResult(reading);
}

private string GetGyrometerResult(GyrometerReading reading)
{
  var sb = new StringBuilder();
  sb.AppendFormat("x {0}\n", reading.AngularVelocityX);
  sb.AppendFormat("y {0}\n", reading.AngularVelocityY);
  sb.AppendFormat("z {0}\n", reading.AngularVelocityZ);
  return sb.ToString();
}
```

39.6.6 方向

OrientationSensor是最复杂的，它从加速计、陀螺仪和磁力计中获取值。所有这些值放在一个四元数中，用Quaternion属性表示，或用旋转矩阵表示(RotationMatrix属性)。

试一试示例应用程序，看看这些值以及如何移动设备：

```
private void OnGetOrientation(object sender, RoutedEventArgs e)
{
  OrientationSensor orientation = OrientationSensor.GetDefault();
  OrientationSensorReading reading = orientation.GetCurrentReading();
```

```
    this.DefaultViewModel["OrientationSensorResult"] =
      GetOrientationSensorResult(reading);
  }

  private string GetOrientationSensorResult(OrientationSensorReading reading)
  {
    var sb = new StringBuilder();
    sb.AppendFormat("quaternion w: {0}, x: {1}, y: {2}, z: {3}\n",
      reading.Quaternion.W, reading.Quaternion.X,
      reading.Quaternion.Y, reading.Quaternion.Z);
    sb.AppendFormat("{0,10:0.000}{1,10:0.000}{2,10:0.000}\n" +
      "{3,10:0.000}{4,10:0.000}{5,10:0.000}\n" +
      "{6,10:0.000}{7,10:0.000}{8,10:0.000}\n",
      reading.RotationMatrix.M11, reading.RotationMatrix.M12,
      reading.RotationMatrix.M13, reading.RotationMatrix.M21,
      reading.RotationMatrix.M22, reading.RotationMatrix.M23,
      reading.RotationMatrix.M31, reading.RotationMatrix.M32,
      reading.RotationMatrix.M33);
    sb.AppendFormat("yaw accuracy: {0}\n", reading.YawAccuracy);
    return sb.ToString();
  }
```

39.6.7 Rolling Marble 示例

为了查看感应器的值，而不仅仅是查看在 TextBlock 元素中显示的结果值，使用 Accelerometer 建立一个简单的示例应用程序，它在屏幕上滚动一个钢珠。

钢珠用一个红色的椭圆表示(代码文件 RollingMarble/MainPage.xaml)。将一个 Ellipse 元素定位在 Canvas 元素内部，就可以用一个附加的属性移动 Ellipse：

```
<Canvas Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Ellipse Fill="Red" Width="100" Height="100" Canvas.Left="550"
    Canvas.Top="400" x:Name="ell1" />
</Canvas>
```

附加的属性参见第 29 章，Canvas 元素参见第 35 章。

MainPage 的构造函数初始化 Accelerometer，请求最小时间间隔的连续读数。为了确定窗口的边界，在页面的 LayoutUpdated 事件中，把 MaxX 和 MaxY 设置为窗口的宽度和高度(减去椭圆的尺寸)：

```
public sealed partial class MainPage : Page
{
  private Accelerometer accelerometer;
  private double MinX = 0;
  private double MinY = 0;
  private double MaxX = 1000;
  private double MaxY = 600;
  private double currentX = 0;
  private double currentY = 0;
```

```
public MainPage()
{
  this.InitializeComponent();
  accelerometer = Accelerometer.GetDefault();
  accelerometer.ReportInterval = accelerometer.MinimumReportInterval;
  accelerometer.ReadingChanged += OnAccelerometerReading;
  this.DataContext = this;
  this.LayoutUpdated += (sender, e) =>
  {
    MaxX = this.ActualWidth - 100;
    MaxY = this.ActualHeight - 100;
  };
}
```

从加速计中获得了每个值后，OnAccelerometerReading 事件处理方法使椭圆在 Canvas 元素内部移动。在设置值之前，根据窗口的边界检查它：

```
private async void OnAccelerometerReading(Accelerometer sender,
  AccelerometerReadingChangedEventArgs args)
{
  currentX += args.Reading.AccelerationX * 80;
  if (currentX < MinX) currentX = MinX;
  if (currentX > MaxX) currentX = MaxX;

  currentY += -args.Reading.AccelerationY * 80;
  if (currentY < MinY) currentY = MinY;
  if (currentY > MaxY) currentY = MaxY;

  await this.Dispatcher.RunAsync(CoreDispatcherPriority.High, () =>
  {
    Canvas.SetLeft(ell1, currentX);
    Canvas.SetTop(ell1, currentY);
  });
}
```

现在运行应用程序，移动设备，获得钢珠滚动的效果，如图 39-7 所示。

图 39-7

39.7 小结

本章介绍了编写 Windows Store 应用程序的更多内容，讨论了如何使用 SearchBox 方便地集成搜索功能。

与其他应用程序的交互是使用共享协定实现的。DataTransferManager 用于给其他应用程序提供 HTML 数据。实现“共享目标”协定，就可以接收其他应用程序的数据。

本章的另一个主要部分是几个设备，讨论了拍照和录制视频的相机、获取用户的位置，以及使用不同的感应器获取设备的移动方式。

接下来的三章要介绍如何编写 Web 应用程序，提供使用 ASP.NET 的基础知识，下一章是其中的第一章。

第40章

核心 ASP.NET

本章要点

- ASP.NET 技术简介
- 创建处理程序和模块
- 配置应用程序
- 状态管理
- 成员和角色

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 处理程序和模块
- 状态管理
- 成员和角色

40.1 用于 Web 应用程序的.NET Framework

ASP.NET 是.NET Framework 的一部分，在通过 HTTP 请求文档时，这种技术可以在 Web 服务器上动态创建它们。WPF 需要在客户端安装.NET Framework，而 ASP.NET 客户端只需一个浏览器。这里，.NET 代码在服务器运行，因此需要在服务器上安装.NET Framework。客户端只需要支持 HTML 和 JavaScript。

在.NET Framework 和 Visual Studio 2013 中，创建 Web 应用程序有不同的架构。ASP.NET Web Forms 是这些技术的旧版本，ASP.NET MVC 是新版本。这些技术有自己的用途，且各有优缺点。

本章详细介绍 ASP.NET 的基础，包括其工作原理、功能，以及 ASP.NET Web Forms 和 ASP.NET MVC 的共同点。

ASP.NET 为创建 Web 应用程序提供了不同的架构：ASP.NET Web Forms、ASP.NET Web Pages 和 ASP.NET MVC。ASP.NET Web Forms 是这些技术中最古老的，从.NET 1.0 开始就有了。其他技术都比较新，基于较新的概念。下面几节介绍将 HTML 返回给客户端的这些选项。

40.1.1　ASP.NET Web Forms

ASP.NET Web Forms 自从 2002 年引入.NET 以来就存在，现在可用于 4.5 版本。ASP.NET Web Forms 的目标是方便 Windows Forms 开发人员使用。这个架构提供了服务器端控件，其属性和方法非常类似于 Windows Forms 控件。使用该架构的开发人员不需要了解 HTML 和 JavaScript，因为控件本身会创建要返回给客户端的 HTML 和 JavaScript。

即使不了解 HTML、JavaScript 或通过网络发送的 HTML 请求，也很容易使用该架构。但是，了解这些技术总是有用的，否则就可能在视图状态中通过网络发送不必要的数据。视图状态由服务器端控件用于处理服务器端事件。第 41 章将详细介绍视图状态，因为它用于 ASP.NET 服务器端控件。有时生成的 HTML 代码并不是需要的代码。服务器端控件常常提供选项，用自定义模板定义 HTML 代码。

对于小网站，ASP.NET Web Forms 非常容易使用，能很快得到结果。对于大型复杂的网站，就一定要注意从客户端发送给服务器的回发，以及通过网络发送的视图状态；否则应用程序很可能变慢。ASP.NET Web Forms 提供了很多选项来改进它，使之快速流畅，但这抵消了使用 Web Forms 的优点，此时使用其他架构可能会得到更好的结果。使 Web Forms 快速流畅，意味着不使用一些可用的控件，而是编写自定义代码。所以使用 Web Forms 不编写自定义代码的优点就丧失了。

40.1.2　ASP.NET Web Pages

对于Microsoft .NET的新手而言，ASP.NET Web Pages是一项新技术。该技术更容易控制HTML和JavaScript。实际上，使用这个技术开发时，必须编写HTML和JavaScript。.NET代码可以添加到HTML代码所在的页面上。显示代码和功能混合在同一个文件中。在编写单元测试程序时，这的确是一个很大的缺点，但它为HTML和JavaScript开发人员提供了一种开始使用.NET的更简单方式。

ASP.NET Web Pages 提供的辅助类允许通过几行代码使用指定的功能，例如从数据库中读取数据，如下所示：

```
@{
  var db = Database.OpenConnectionString(
    "server=(local)\sqlexpress;database=Formula1;trusted_connection=true");
}
//...
  @foreach (row in db.Query("SELECT * FROM Racers") {
  //...
```

要创建 ASP.NET Web Pages，可以使用免费工具 WebMatrix，它可以从 Microsoft 网站上下载 http://www.microsoft.com/web/webmatrix/。该工具为预定义的 Web 页面提供了几个模板，还为使用 ASP.NET Web Pages 编写 Web 应用程序提供了许多功能。

Database 类在 WebMatrix 程序集中。它可以使用几行代码查询数据库。有了它，数据库代码和 UI 代码就混合在同一个文件中。为了便于管理代码，最好不要这么做，但这是开始编写简单网站的一种好方法。

开始使用 ASP.NET Web Pages 后，用户很容易迁移到 ASP.NET MVC 上。使用 ASP.NET MVC 和 ASP.NET Web Pages 很容易创建网站，也很容易把页面中的代码迁移到 ASP.NET MVC 使用的控制器中。

使用 ASP.NET Web Pages 时，代码很难重用和测试。本书适用于专业程序员，所以本章不讨论 ASP.NET Web Pages，而主要讨论 ASP.NET Web Forms 和 ASP.NET MVC。

40.1.3 ASP.NET MVC

ASP.NET MVC 基于 MVC 模式：Model-View-Controller(模型-视图-控制器)。如图 40-1 所示，该标准模式(在 GOF 的 *Design Patterns* 一书中提到的模式)定义了一个模型、一个视图和一个控制器。其中，模型实现了数据实体和数据访问；视图表示显示给用户的信息；控制器使用模型，把数据发送给视图。控制器接收来自浏览器的请求，返回一个响应。要建立该响应，控制器可以使用模型提供一些数据，使用视图定义返回的 HTML。

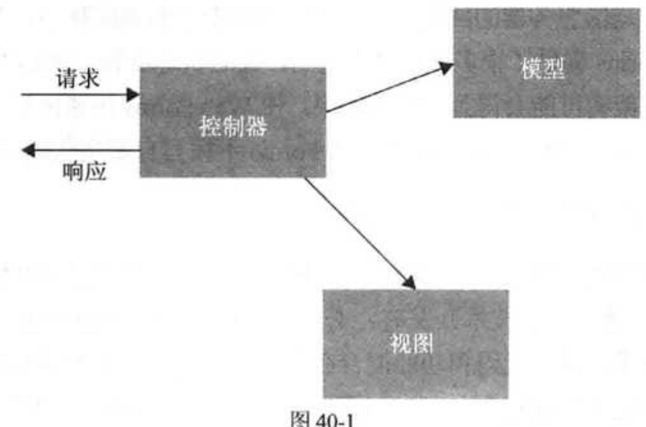
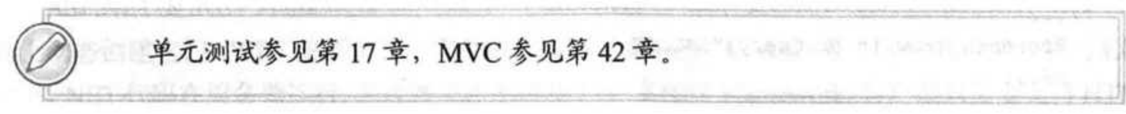


图 40-1

使用 ASP.NET MVC，控制器和模型一般用 C#和.NET 代码创建，这些代码都在服务器端运行。视图是带有 JavaScript 的 HTML 代码，只有少量 C#代码来访问服务器端信息。

MVC 模式这种分离的主要优点是，单元测试很容易测试功能。控制器只包含带参数和返回值的方法，该方法很容易用单元测试来覆盖。



单元测试参见第 17 章，MVC 参见第 42 章。

40.2 Web 技术

在介绍 ASP.NET 的基础知识之前，本节讨论创建 Web 应用程序时必须了解的核心 Web 技术：HTML、CSS、JavaScript 和 jQuery。

40.2.1 HTML

HTML 是由 Web 浏览器解释的标记语言。它定义的元素显示各种标题、表格、列表和输入元素，如文本框和组合框。

HTML4.01 的 W3C 规范在 1999 年 12 月发布。HTML5 在编写本书时(2013 年)仍在开发，却已在使用。HTML5 的有限子集可以用于旧浏览器。它的使用会越来越多，因为有了它的几个新功能，就不再需要使用 Flash 和 Silverlight，所以也不再需要浏览器插件了。一些浏览器，例如 Windows 8 中的 Internet Explorer 或 iPad 中的 Safari，不支持插件。

HTML5 添加的新语义元素可以由搜索引擎使用，更好地分析站点。canvas 元素可以动态使用 2D 图形和图像，video 和 audio 元素使 object 元素过时了。

HTML5 还为拖放操作、存储器、Web 套接字等定义了 API。

在 ASP.NET Web Forms 中，服务器端控件生成了 HTML。在 ASP.NET MVC 中，程序员编写 HTML 代码的责任更大。

40.2.2 CSS

HTML 定义了 Web 页面的内容，CSS 定义了其外观。例如，在 HTML 的早期，列表项标记<i>定义列表元素在显示时是否应带有圆、圆盘或方框。目前，这些信息已从 HTML 中完全删除，而放在层叠样式表(CSS)中。

在 CSS 样式中，HTML 元素可以使用灵活的选择器来选择，还可以为这些元素定义样式。元素可以通过其 id 或名称来选择，也可以定义 CSS 类，从 HTML 代码中引用。在 CSS 的新版本中，可以定义相当复杂的规则，来选择特定的 HTML 元素。

在 Visual Studio 2013 中，Web 项目模板使用 Twitter Bootstrap，这是 CSS 和 HTML 约定的集合，很容易采用不同的外观，下载易用的模板。文档和基本模板可参阅 www.getbootstrap.com。

40.2.3 JavaScript 和 jQuery

并不是所有的平台和浏览器都能使用.NET 代码，但几乎所有的浏览器都能理解 JavaScript。对 JavaScript 的一个常见误解是它与 Java 相关。实际上，它们只是名称相似，因为它们使用某些相同的命名约定，Java 和 JavaScript 有相同的根(C 编程语言)，C#也是这样。JavaScript 是一种函数编程语言，不是面向对象的，但它添加了面向对象功能。

JavaScript 允许从 HTML 页面访问 DOM，因此可以在客户端动态改变元素。除了 JavaScript 之外，Internet Explorer 允许 VBScript 的用户访问 DOM。但是，因为其他浏览器不支持 VBScript，所以 JavaScript 是编写可在任何地方运行的客户端代码的唯一选项。

在不同的浏览器上支持带 JavaScript 的 Web 页面仍是一个噩梦，因为不同浏览器供应商都以不同的方式处理许多实现方案，而且一个供应商也使用不同的浏览器版本。一个解决方案是 JavaScript 库，如 jQuery(http://www.jquery.org)。jQuery 只使用几行代码，就可以完成需要很多 JavaScript 代码完成的任务，它负责处理不同的浏览器引擎，把这个工作抽离了 JavaScript 程序员。

ASP.NET Web Projects 包含 jQuery 库，Visual Studio 2013 也支持 IntelliSense 和 JavaScript 代码的调试。

本书不讨论如何设置 Web 应用程序的样式和编写 JavaScript 代码。HTML、HTML 中的样式和 CSS 的内容可参阅 John Ducket 编写的 *Design and Build Websites*(Wiley, 2011)，阅读 Nicholas C. Zakas 编写的 *Professional JavaScript for Web Developers*(Wrox, 2005)可以加速对 JavaScript 的掌握。

40.3 托管和配置

Web 应用程序需要宿主才能运行。通常，Internet Information Services (IIS)是用于生产站点的宿主。在开发系统上，不一定要安装 IIS。Visual Studio 2013 包含 IIS Express，IIS Express 的使用非常类似完整的 IIS，只缺少管理功能。

在项目设置的 Web 选项卡上给 Web 项目配置服务器，如图 40-2 所示。

Visual Studio 2013 还提供了运行 web 应用程序的外部宿主。这个新宿主模型基于 OWIN (Open Web Interface for .NET) 架构。使用这个技术，很容易创建自定义的宿主。要使用 Microsoft 中的 OwinHost，需要添加 NuGet 包 OwinHost，接着就可以在项目设置的 Web 选项卡上配置 OwinHost 服务器了，如图 40-3 所示。

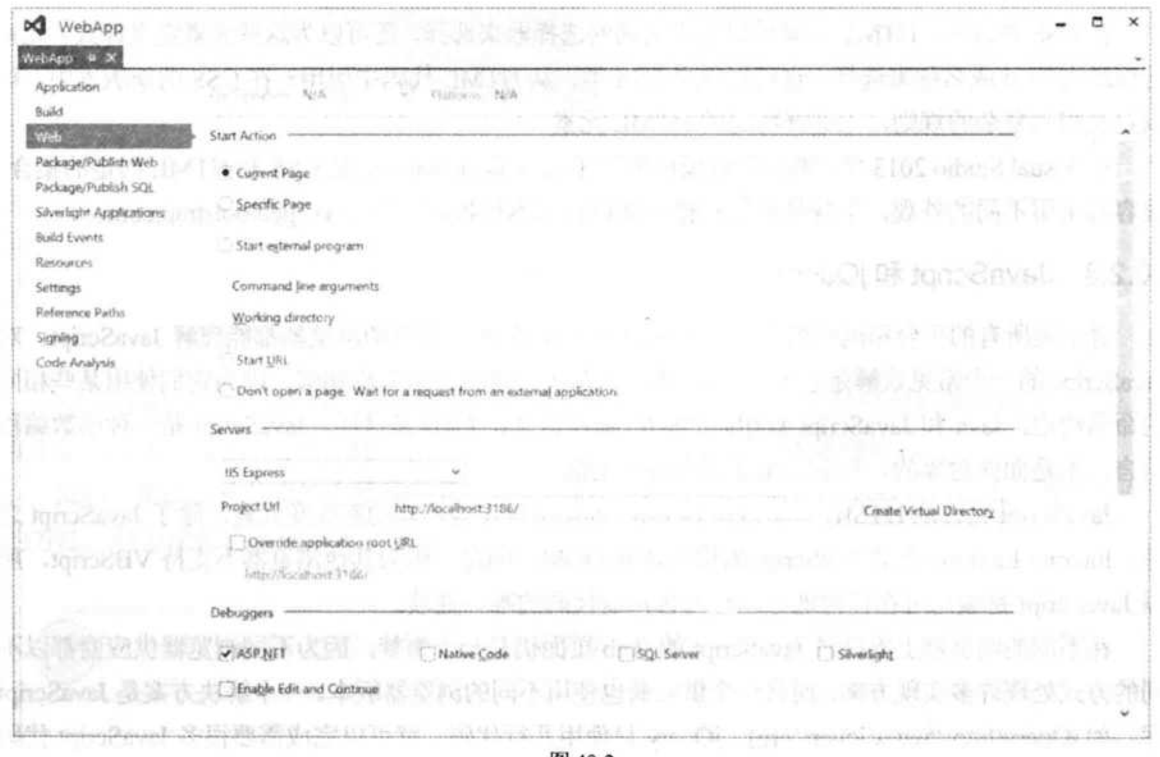


图 40-2

要配置 Web 应用程序，应使用应用程序配置文件。所有.NET 应用程序(不仅仅是 Web 应用程序)使用的第一个配置文件是 machine.config，它在<*windir*>\Microsoft.NET\Framework\v4.0.30319 目录下。对于 Web 应用程序，需要配置成员和角色提供程序。这些提供程序也可以用于其他.NET 应用程序，所以可以将这个配置放在 machine.config 中。

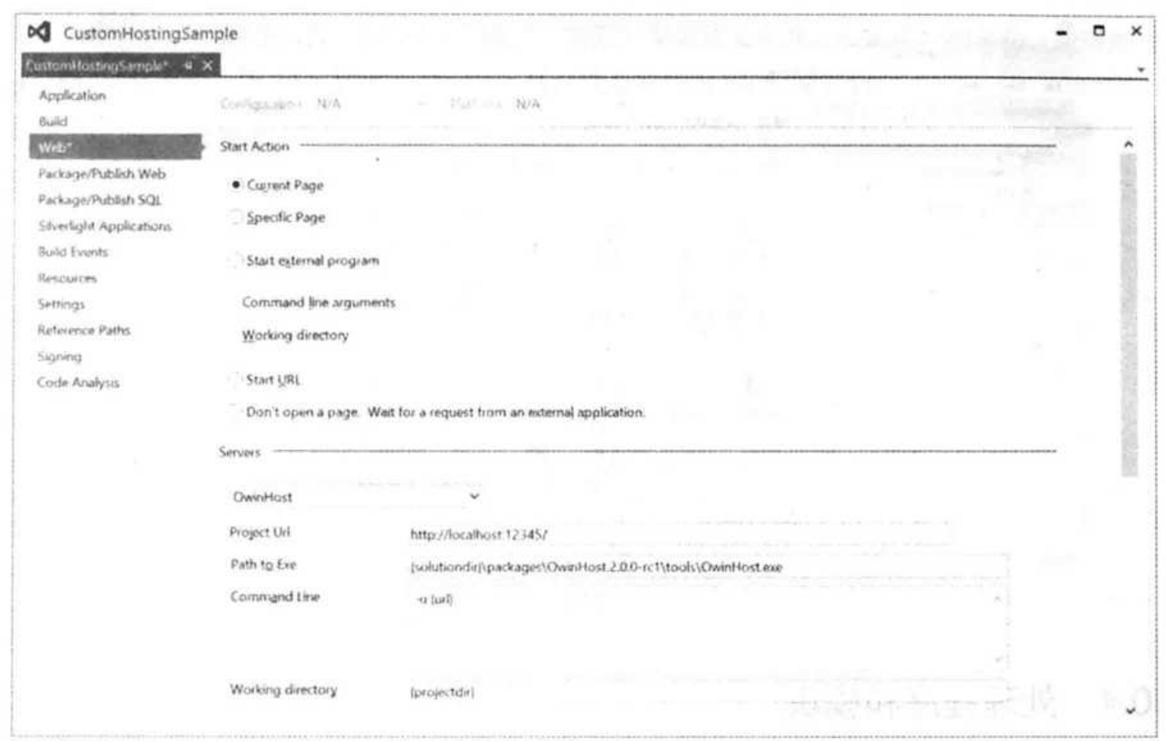


图 40-3

与 machine.config 位于同一个目录下的 web.config 文件用于 ASP.NET 特定的配置。这里的配置专用于 Web 应用程序。其中包括如下默认设置：信任级别和完全信任的程序集(程序集的权限信息参见第 22 章)、用于在第一次使用网站时编译 C#代码的编译器配置、引用的程序集、健康检测、事件日志和配置文件提供程序、HTTP 处理程序和模块、WCF 的配置协议、站点地图和 Web Part(Web 部件)配置。

定义浏览器专用功能的其他配置文件在 Browsers 子目录下。这里有文件 Default.browser、ie.browser、opera.browser、iphone.browser 和 firefox.browser。它们定义了特定浏览器的所有功能。根据调用者的能力，这些功能可以在服务器端控件中使用，影响返回的 HTML 和 JavaScript 代码。

浏览器具有哪些功能是基于从浏览器发送来的浏览器标识符字符串确定的。浏览器可能撒谎，发送错误的标识字符串，例如 Opera 浏览器发送 Internet Explorer 作为它的标识字符串。一些浏览器允许用户定义应使用的标识符字符串。因此，许多 Web 应用程序目前使用 JavaScript 验证某功能是否真的可用。Modernizr 是检查浏览器功能的一个 JavaScript 库，它可以使用 NuGet 软件包安装。

使用 IIS(Internet Information Services)运行 Web 应用程序时，如果全局 Web.config 文件中的设置被覆盖，另一个配置文件位于 inetpub\wwwroot 目录下。每个 Web 应用程序中甚至子目录会创建其他 Web.config 文件，覆盖其父目录的设置。使用 Internet Information Services (IIS) Manager 工具，可以通过图形化 UI 修改配置，如图 40-4 所示。

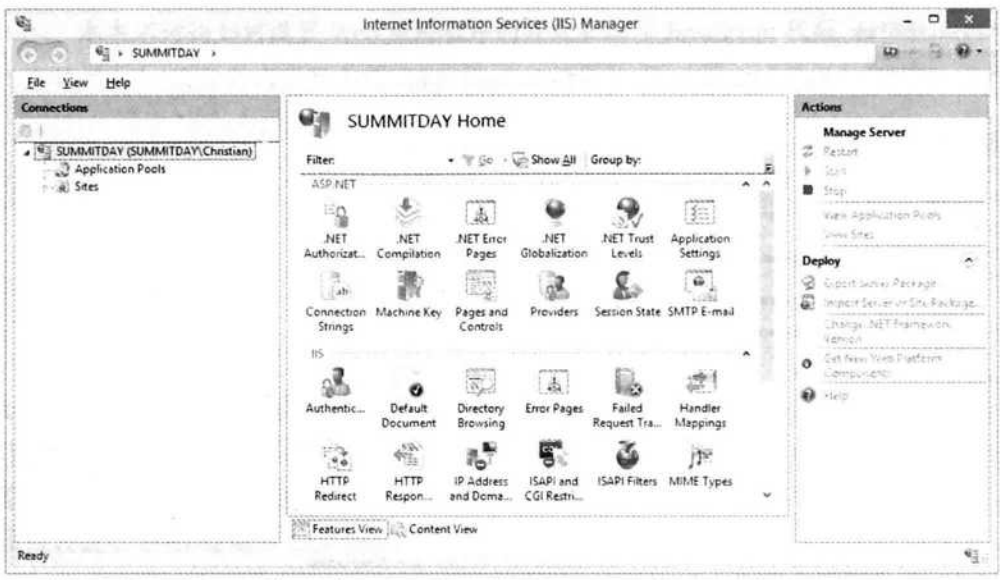


图 40-4

40.4 处理程序和模块

本节介绍客户端向 Web 服务器发出请求时，会发生什么。首先，Web 服务器会尝试查找适合请求类型的处理程序。IIS 包含大量处理程序，如图 40-5 所示。例如，.aspx 文件的处理程序会通过 PageHandlerFactory 实例化页面类，.svc 文件的处理程序由 WCF 使用。

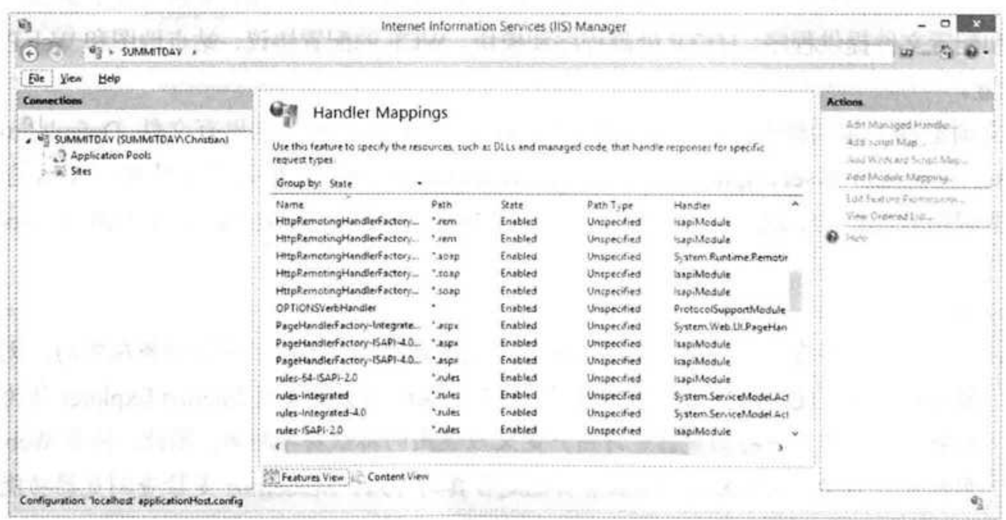


图 40-5

WCF 参见第 43 章。

调用每个处理程序时，应使用几个模块。它们分别处理安全性、验证用户的身份、处理授权、创建会话状态等。图 40-6 显示了 Modules 页，以及如何用 IIS 配置它们。

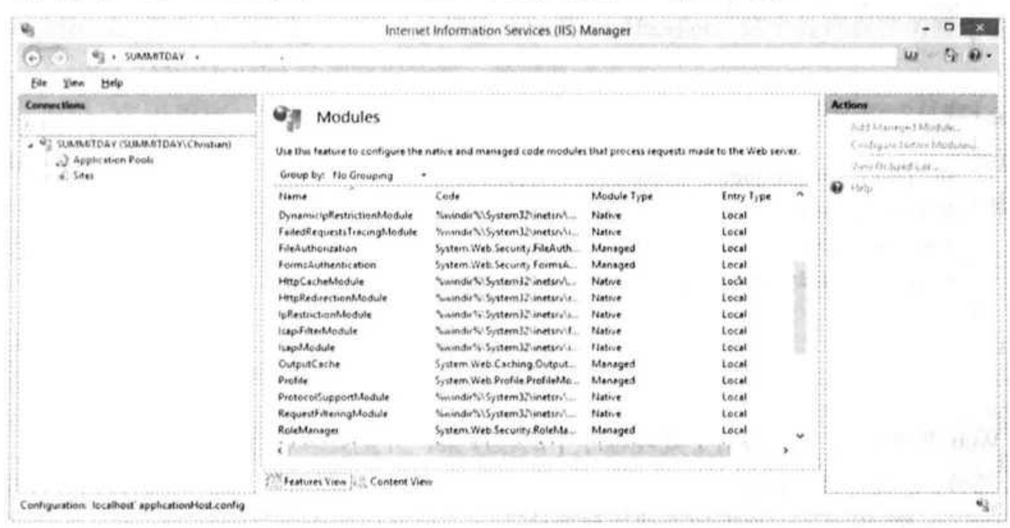


图 40-6

40.4.1　创建自定义处理程序

要创建自定义处理程序，可以创建一个实现了 IHttpHandler 接口的类。下面的示例(代码文件 HandlerSample/SampleHandler.cs)创建了一个库，该库引用 System.Web 程序集，定义 SampleHandler 类，该类实现了 IHttpHandler 接口。这个接口定义了 IsReusable 属性和 ProcessRequest 方法。如果处理程序实例可以在不同的请求中重用，IsReusable 就返回 true。ProcessRequest 方法接收带参数的 HttpContext。HttpContext 允许接收来自调用者的请求信息，并发回一个响应。示例代码定义了要返回的 HTML 字符串。在 HttpRequest 对象中，使用 UserAgent 属性把该属性返回的结果和响应一起发送回去：

```
using System.Web;

namespace Wrox.ProCSharp.ASPNETCore
{
  public class SampleHandler : IHttpHandler
  {
    private string responseString = @"
<!DOCTYPE HTML>
<html>
<head>
  <meta charset=""UTF-8"">
  <title>Sample Handler</title>
</head>
<body>
  <h1>Hello from the custom handler</h1>
  <div>{0}</div>
</body>
</html>";
```

```
        public bool IsReusable
        {
          get { return true; }
        }

        public void ProcessRequest(HttpContext context)
        {
          HttpRequest request = context.Request;
          HttpResponse response = context.Response;
          response.ContentType = "text/html";
          response.Write(string.Format(responseString, request.UserAgent));
        }
      }
    }
```

在 Web 应用程序中，引用了处理程序中的程序集，并把处理程序添加到 Web.config 文件的 handlers 部分。定义处理程序的方法是，指定一个可通过编程引用它的 name，指定 HTTP 方法(GET、POST、HEAD 等)的 verb，指定用户所用链接的 path，以及标识实现了 IHttpHandler 的类的 type。路径也允许指定文件扩展名，例如*.aspx，对 aspx 文件的每个请求都调用该处理程序。

```
<system.webServer>
  <handlers>
    <add name="SampleHandler" verb="*" path="CallSampleHandler"
      type="Wrox.ProCSharp.ASPNETCore.SampleHandler, HandlerSample" />
  </handlers>
</system.webServer>
```

请求链接/CallSampleHandler，就调用该处理程序，从客户端返回用户代理字符串。IE11 中的用户代理信息如图 40-7 所示。

图 40-7

40.4.2 ASP.NET 处理程序

对于 ASP.NET Web Forms 应用程序，给扩展名为 aspx 的文件配置 PageHandlerFactory 处理程序。负责该处理程序的类型是 System.Web.UI.PageHandlerFactory。该类型实现了接口 IHttpHandlerFactory，该接口是 IHttpHandler 对象的工厂。这个接口定义了方法 GetHandler 和 ReleaseHandler，分别用于返回和释放 Web Form 页面。Web Form 基类 Page 实现了 IHttpHandler 接口，用作处理程序。

对于用户请求不应该看到的文件(如扩展名为.cshtml 的文件)，HttpForbiddenHandler 类型会用 HTTP 403 错误响应请求，拒绝访问。

对于 ASP.NET MVC，给路径*配置 ExtensionlessUrlHandler。处理这些请求的类型是 System.Web.Handlers.TransferRequestHandler。要使用路由，就像 ASP.NET MVC 使用它们那样，UrlRoutingModule 类会执行操作，把请求传递给 MvcRouteHandler。这个处理程序为特定的路由创建了 MvcHandler。MvcHandler 会搜索控制器，来提取请求。

对于 Web 应用程序，可以创建泛型处理程序。泛型处理程序的文件扩展名是 ashx，从 SimpleHandlerFactory 类型中间接调用。泛型处理程序实现了 IHttpHandler 接口，其方式与前面相同，但不一定要配置它们。因为文件扩展名是 ashx，所以调用 SimpleHandlerFactory，它会搜索被请求的文件，以传递处理程序请求。

40.4.3 创建自定义模块

要创建自定义模块，类需要实现 IHttpModule 接口。这个接口定义了 Init 和 Dispose 方法。

下面的代码块(代码文件 WebApp/ModuleSample/SampleModule.cs)演示了一个模块，它验证请求是否来自 IP 地址的预定义列表，如果不是，就拒绝访问。Init 方法在启动 Web 应用程序时调用，其参数的类型是 HttpContext。但是，这个方法没有做什么工作，因为还没有填充 HttpContext 参数，该方法在第一个请求创建之前就被调用了。可以给事件添加事件处理程序，如 BeginRequest、EndRequest、AuthorizeRequest、AuthenticateRequest、PreRequestHandlerExecute 等。示例代码向 BeginRequest 和 PreRequestHandlerExecute 事件添加了事件处理程序。在 BeginRequest 方法中，把文件加载到列表集合中，该集合包含所有允许的 IP 地址。PreRequestExecute 方法使用 HttpRequest 对象的 UserHostAddress 属性，验证调用者的 IP 地址是否在允许的 IP 地址列表中。如果不在，就抛出一个 HttpException 类型的异常，HTTP 错误代码为 403：

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

namespace Wrox.ProCSharp.ASPNETCore
{
  public class SampleModule : IHttpModule
  {
    private const string allowedAddressesFile = "AllowedAddresses.txt";
    private List<string> allowedAddresses;

    public void Dispose()
    {
    }

    public void Init(HttpApplication context)
    {
      context.LogRequest += new EventHandler(OnLogRequest);
      context.BeginRequest += BeginRequest;
      context.PreRequestHandlerExecute += PreRequestHandlerExecute;
    }
```

```
        private void BeginRequest(object sender, EventArgs e)
        {
          LoadAddresses((sender as HttpApplication).Context);
        }

        private void LoadAddresses(HttpContext context)
        {
          if (allowedAddresses == null)
          {
            string path = context.Server.MapPath(allowedAddressesFile);
            allowedAddresses = File.ReadAllLines(path).ToList();
          }
        }

        private void PreRequestHandlerExecute(object sender, EventArgs e)
        {
          HttpApplication app = sender as HttpApplication;
          HttpRequest req = app.Context.Request;
          if (!allowedAddresses.Contains(req.UserHostAddress))
          {
            throw new HttpException(403, "IP address denied");
          }
        }

        public void OnLogRequest(Object source, EventArgs e)
        {
          //custom logging logic can go here
        }
      }
    }
```

AllowedAddresses.txt 文件包含一组允许的 IP 地址。如果使用 IPv6，还应添加 IPv6 地址，以允许在客户端和服务器之间通信，如下所示：

```
127.0.0.1
10.0.0.22
::1
```

模块在 Web.config 文件的 system.webServer 部分配置。该配置类似于处理程序，只是需要放在 modules 中：

```
<system.webServer>
  <modules>
    <add name="SampleModule"
      type="Wrox.ProCSharp.ASPNETCore.SampleModule, ModuleSample" />
  </modules>
</system.webServer>
```

40.4.4 通用模块

对于每个请求，都要调用几个模块。下面的代码段来自 HandlerSample 项目的 InfoHandler.cs 文

件。它显示了加载的模块。HttpContext.ApplicationInstance 返回 HttpApplication，该类型定义了 Modules 属性，用于返回所有已加载模块的集合：

```
public void ProcessRequest(HttpContext context)
{
  var sb = new StringBuilder();
  sb.Append("<ul>");
  foreach (var module in context.ApplicationInstance.Modules)
  {
    sb.AppendFormat("<li>{0}</li>", module);
  }
  sb.Append("</ul>");
  context.Response.ContentType = "text/html";
  context.Response.Write(string.Format(responseString, sb.ToString()));
}
```

图 40-8 显示了已配置的处理程序的结果。通用模块包括缓存响应的 OutputCache、保存客户端内存状态的 Session，各种验证和授权模块(如 WindowsAuthentication、FormsAuthentication、FileAuthorization 和 UrlAuthorization)，基于用户长期存储信息的 Profile，以及用于 WCF 的 ServiceModule。

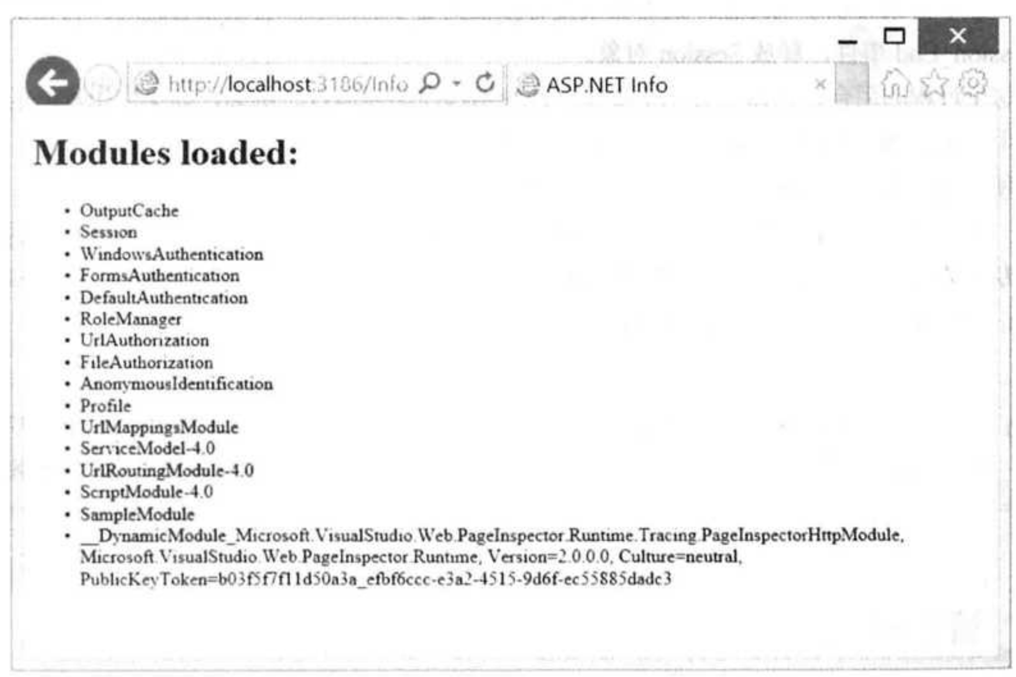


图 40-8

除了使用处理程序和模块全局地处理通用功能之外，另一种方式是使用全局的应用程序类，如下所述。

40.5　全局的应用程序类

全局的应用程序类可以在 Web 应用程序中全局使用，以独立于页面处理事件。在全局应用程序

类中，可以为 Web 应用程序添加初始化代码，以及在收到每个请求时调用的代码。第一次启动应用程序时，也就是第一次收到 HTTP 请求时，会创建 Application 对象。此时，也会触发 Application_Start 事件，并创建一个 HttpApplication 实例池。每个入站的请求都会接收这些实例中的一个，来处理请求。注意这意味着 HttpApplication 对象不需要处理并发访问，这与全局 Application 对象不同。所有的 HttpApplication 实例完成其工作后，就触发 Application_End 事件，应用程序终止，释放 Application 对象。

前面提及的事件处理程序(以及本章前面讨论的所有其他事件处理程序)都可以在 global.asax 文件中定义，该文件可以添加到任何 Web 站点项目中(在 Web 应用程序中添加新项时，它在所出现的模板中显示为 Global Application Class)。生成的文件包含一些空白，供用户填充代码，如下所示：

```
void Application_Start(Object sender, EventArgs e)
{
    // Code that runs on application startup
}
```

单个用户访问 Web 应用程序时，会启动一个会话。与应用程序类似，这涉及创建一个用户专用的 Session 对象，并触发 Session_Start 事件。在会话中，单个请求会触发 Application_BeginRequest 和 Application_EndRequest 事件。在会话作为应用程序中的不同资源来访问时，这些事件可能触发若干次。单个会话可以手动终止，或者如果没有接收到更多的请求，会话就会超时。会话的终止会触发 Session_End 事件，释放 Session 对象。

在这个过程的后台，可以执行几个操作，使应用程序顺畅运行。例如，如果应用程序的所有实例都使用一个资源集中的对象，就可以考虑在应用程序级别上实例化它。这可以提高性能，减少多个用户使用的内存量，因为在大多数请求中，不需要这样的实例化。

另一个可以使用的技术是存储会话级别的信息，以便由单个用户跨多个请求来使用。这可能包含用户专用的信息，这些信息是在用户第一次连接时从数据存储中提取的(在 Session_Start 事件处理程序中)，且一直可用，直到会话终止为止(通过超时或用户请求来终止)。

注意 HttpContext 不能用于 Session_End 和 Application_End 事件。也不能确定 Application_End 一定会被调用。在工作进程需要立即触发时，这个事件可能不会触发。

40.6 请求和响应

在前面的处理程序示例中，介绍了如何响应客户端的请求。请求中的信息可以直接使用 HttpRequest 对象访问，定义返回什么内容则由 HttpResponse 打包。下面介绍这些对象。

40.6.1 使用 HttpRequest 对象

HttpRequest 对象可以使用类的 Request 属性或 HttpContext 来访问。HttpRequest 的一个功能是接收包括浏览器功能的浏览器信息。HttpRequest 的 Browser 属性返回一个 HttpBrowserCapabilities 对象，该对象提供访问浏览器的功能。有了这个对象，就可以检查 JavaScript 版本，浏览器是否支

持 cookie 和框架等。

下面的代码段使用 HttpRequest 对象的 Browser 属性获得浏览器功能的信息。可以使用强类型化访问功能来检查各个特性，例如使用 CanInitiateVoiceCall 检查是否进行语音调用，使用 CanSendMail 检查能否发送电子邮件。下面的代码段使用 Capabilities 属性直接访问一个目录：

```
HttpBrowserCapabilities browserCapabilities = Request.Browser;
Response.Write("<ul>");
foreach (var key in browserCapabilities.Capabilities.Keys)
{
  Response.Write("<li>");
  Response.Write(string.Format("{0}: {1}", key,
    browserCapabilities.Capabilities[key]));
  Response.Write("</li>");
}
Response.Write("</ul>");
```

图 40-9 显示了从 IE11 返回的功能。

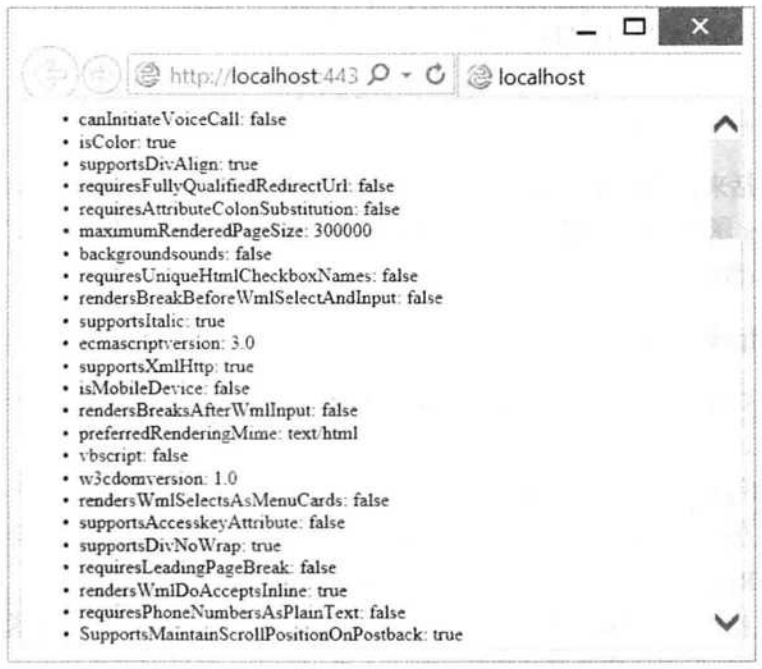


图 40-9

所有的功能信息都从浏览器配置文件中获得，如前面的配置部分所述。HttpBrowserCapabilities 对象的 Browsers 属性提供了如何检索功能的信息。IE11 用这个属性返回 default、mozilla、ie、ie6plus 和 ie10plus，它精确指定功能是从哪些配置中创建的。

HttpRequest 对象的 Headers 属性返回所有的 HTTP 标题信息。下面的代码段从浏览器中获得了所有的标题信息，结果如图 40-10 所示。

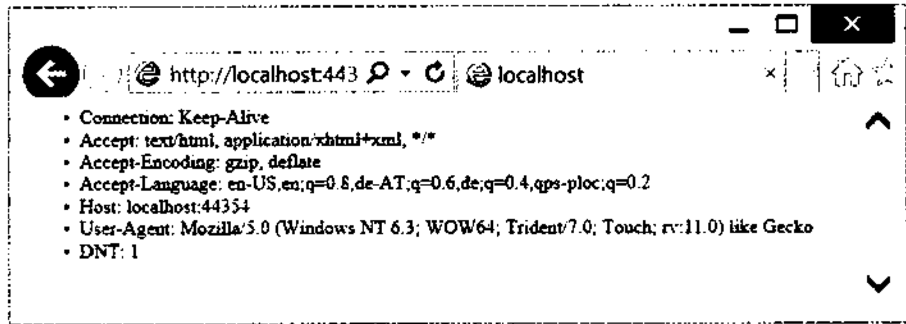


图 40-10

```
NameValueCollection headers = Request.Headers;
Response.Write("<ul>");
foreach (var key in headers.Keys)
{
  foreach (var value in headers.GetValues(key.ToString()))
  {
    Response.Write("<li>");
    Response.Write(string.Format("{0}: {1}", key, value));
    Response.Write("</li>");
  }
}
Response.Write("</ul>");
```

HTTP 标题包括来自客户端的 cookie。但是，要检索 cookie，比使用 Headers 属性更简单的访问方式是使用 cookies 属性，参见 40.8.2 小节。也可以使用请求对象来访问 HTML 窗体中从客户端发送来的用户信息和数据。

40.6.2 使用 HttpResponse 对象

HttpResponse 对象允许把数据发送回客户端。Page 和 HttpContext 的 Response 属性返回当前的 HttpResponse 对象。

前面介绍了如何使用 HttpResponse 对象把数据返回给客户端(使用 Write 方法)。只是 HTTP 标题和 cookie 信息可以使用 HttpRequest 对象访问，同样，返回客户端的标题和 cookie 会受到 HttpResponse 对象的影响。HttpResponse 还定义了 Headers 和 Cookies 属性。

响应对象不仅把内容发送给客户端，还可以发送重定向请求。Redirect 会发送 HTTP 302 状态码和应使用另一个 URL 发送给客户端的信息。RedirectPermanent 发送 HTTP 301 状态码，告诉调用者永久地使用新的 URL。RedirectToRoute 使用路由表查找匹配的路由，来建立到客户端的重定向请求。

40.7 状态管理

HTTP 协议是无状态的。每个新页面请求都可以是一个新连接。但是，常常需要保存用户信息。状态可以在客户端或服务器上保存。本节讨论保存状态的不同选项，以及如何对它们编程。示例使用 ASP.NET Web Forms 页面以及简单的 TextBox(文本框)、Label(标签)和 Button(按钮)控件，并使用同样简单的事件处理程序，来演示不同的状态功能。Page 类的属性用于访问状态管理功能，例如，HttpSessionState 对象可以在 Page 类的 Session 属性中直接访问。在 Page 类的外部，使用 HttpContext

也可以获得相同的结果。HttpContext.Current 返回活动的 HttpContext 对象，这个类的 Session 属性也返回 HttpSessionState。换言之，所有的状态管理功能都很容易从 ASP.NET Web Forms 和 ASP.NET MVC 中实现。

要在客户端保存状态，ASP.NET 提供了不同的选项：视图状态、cookie 和参数。因为安全问题，使用这些状态就有一些限制。在服务器端可使用会话对象、全局应用程序状态、缓存和用户配置文件来保存状态。所有这些不同的信息都在下面几小节中介绍。

40.7.1 视图状态

视图状态仅在页面内部可用。只要用户位于同一个页面中，就可以使用视图状态。视图状态在页面中创建隐藏的 HTML 字段，该字段会发送给服务器，因为它位于<form>标记中。

视图状态可以使用 Page 的 ViewState 属性来访问。ViewState 属性返回一个 StateBag 对象。把一个键值传递给索引器，就可以使用视图状态来读写数据。下面的示例使用 state1 键从视图状态中读取数据，把 TextBox1 的 Text 属性值写入同一个视图状态对象中：

```
protected void Button1_Click(object sender, EventArgs e)
{
  Label1.Text = string.Format("TextBox1.Text: {0}", TextBox1.Text);
  Label2.Text = string.Format("ViewState[\"state1\"] {0}",
    ViewState["state1"]);
  ViewState["state1"] = TextBox1.Text;
}
```

第一次打开示例页面 ViewState1.aspx 时，还没有调用 Button1_Click 方法，因此两个标签都显示初始值 Label。

如果把 one 写入文本框控件并单击按钮，就会给服务器发送一个回发，且第一次调用 Button1_Click 方法。这里 TextBox1.Text 属性返回输入的数据，因此第一个标签的 Text 属性用这个数据填充。第二个标签仍显示消息的第一部分，ViewState["state1"]返回空。在方法的最后一行，视图状态初始化为文本框中的值。

把 two 写入文本框，第二次单击按钮，则给服务器发送另一个回发，ViewState["state1"]现在返回以前输入的数据 one，TextBox1.Text 返回新字符串 two。

视图状态用页面中的隐藏字段来存储：

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value=
  "XSLM3n7Gl3EVtL9CN4jAYfe3T5x/Lr26ORPT4+MEsapcFdvlu0Ooc9uiyOGku2IKOyEgv3WyR0
  0iUNUKM0kBaVN1nMVm/W8c8Ilx2cyeHO+zVbzAfZCYVUPDlgIIlup2ZLW9fpfYZ+d8S+uBM/Vg
  WbCmsYBHW5RGaINY2QsSxep2kMfeoueD4YHND36J29XcRMV9K86Bzw4/OcX9uc7WwA==" />
```

使用隐藏字段的优点是没有超时。只要页面处于打开状态，就存在隐藏字段，并在下一次单击时发送给服务器。如果用户关闭了页面，状态就消失了。如果用户切换到另一个页面上，状态也会消失。视图状态的两个缺点是状态必须在字符串中表示，所有的视图状态数据总是通过网络发送。这可能涉及传输大量数据，降低了性能。

ASP.NET 服务器端控件使用视图状态。服务器端事件模型基于该状态模型。把窗体内容发送给服务器时，窗体会在视图状态中包含文本框的以前值，在文本框中包含当前值。这样，事件机制就可以确定是否应触发变动事件，并调用响应的处理方法。

40.7.2 cookie

cookie 存储在客户端的浏览器内存(会话 cookie)或磁盘上(永久 cookie)。它们是 HTTP 协议的一部分，在 HTTP 头发送。每次用户访问 Web 站点时，这个站点的 cookie 就发送给服务器。如果 cookie 设置了路径，它们就只在路径相同时发送。

在 ASP.NET 中，cookie 可以用 HttpResponse 类(Page 的 Response 属性)发送给客户端。HttpResponse 的 SetCookie 方法(代码文件 StateSample/CookieWrite.aspx.cs)传送一个 HttpCookie。HttpCookie 的构造函数允许设置 cookie 的名称和值。不仅如此，HttpCookie 类还定义了 Domain 和 Path 属性，以便仅在请求指定目录中的文件时，才把 cookie 关联并发送给服务器。示例代码通过名为 cookieState 的 cookie 发送一个值。单个 cookie 还可以包含一组可使用 Values 属性赋的值。如果 Expires 属性设置为日期，cookie 就是永久性的；否则它就只是临时 cookie，只要浏览器关闭，该 cookie 就会丢失。下面的代码在选中复选框时创建一个永久的 cookie。把 cookie 的 Secure 属性设置为 true，就只在使用 HTTPS 协议时发送 cookie：

```
protected void Button1_Click(object sender, EventArgs e)
{
  var cookie = new HttpCookie("cookieState", TextBox1.Text);
  if (CheckBox1.Checked)
  {
    cookie.Expires = DateTime.Now.AddYears(1);
  }
  Response.SetCookie(cookie);
}
```

浏览器把 cookie 发送给服务器。当然，浏览器没有把所有的 cookie 发送给服务器，只发送了域名和路径(假定指定了路径)相同的 cookie。检索 cookie 是由 HttpRequest 对象的 Cookies 属性完成的。只需把名称传递给索引器，并检索 HttpCookie 即可(代码文件 StateSample/CookieRead.aspx.cs)：

```
protected void Page_Load(object sender, EventArgs e)
{
  HttpCookie cookie = Request.Cookies["cookieState"];
  if (cookie != null)
  {
    Label1.Text = cookie.Value;
  }
}
```

使用 IE 中的开发工具(按 F12 键)，在完成网络配置后，就很容易看到所发送的 cookie。图 40-11 显示同一个页面 CookieWrite.aspx 的 cookie 信息。

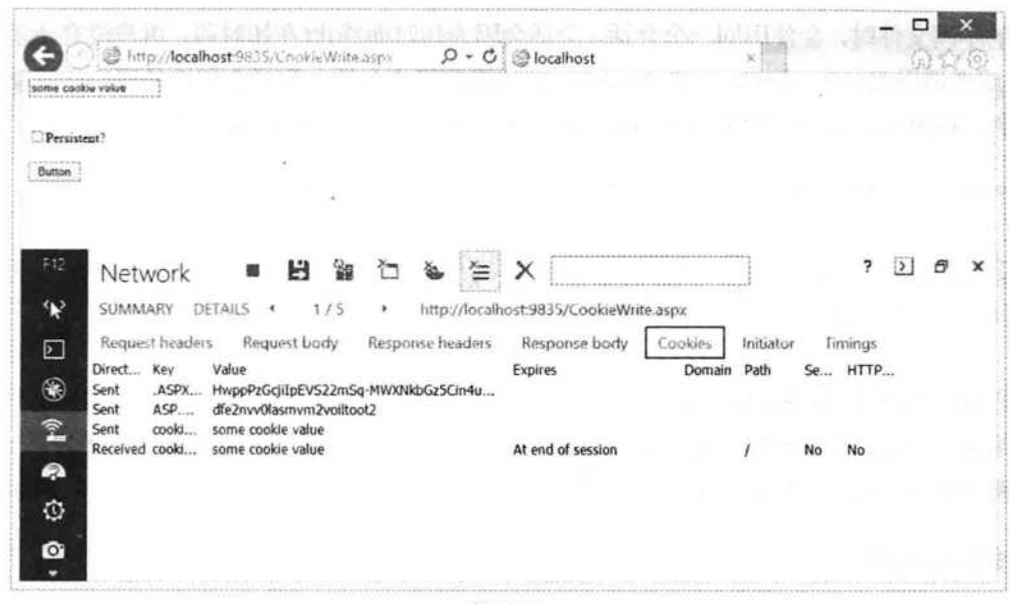


图 40-11

cookie 有一些限制。用户可以通过浏览器设置关闭永久性 cookie，所以根本无法保证它们能正常工作。应总是假设 cookie 可能在客户端上删除了。一些用户常常会删除 cookie。浏览器本身也可能删除 cookie。还有一些限制：cookie 的大小不能超过 4096 字节，浏览器在每个域中存储的 cookie 数不能超过 50，浏览器存储的总 cookie 数不能超过 3000。如果达到了这些限制，浏览器就可以删除 cookie，而不需要与用户交互操作。浏览器至少要支持指定的 cookie 数，但也可以使用更大的限制值。由于有这些限制，cookie 通常仅用于在客户端发送一些标识符，用于把它映射到服务器上真实的用户数据上。

40.7.3　会话

使用从 Page 的 Session 属性返回的 HttpSessionState 对象，可以存储用户会话的状态。很容易使用键把对象写入服务器上保存的会话(代码文件 StateSample/SessionWrite.aspx.cs)。任何串行化对象都可以添加到会话状态中：

```
protected void Button1_Click(object sender, EventArgs e)
{
  Session["state1"] = TextBox1.Text;
}
```

要读取数据，可以使用索引器的获取访问器。应总是检查索引器是否返回了值：

```
protected void Page_Load(object sender, EventArgs e)
{
  object state1 = Session["state1"];
  if (state1 != null)
  {
    Label1.Text = state1.ToString();
  }
}
```

用户在服务器上打开一个页面时，就会启动会话，此前会话是不存在的。用户浏览同一个 Web

站点上的不同文件时，会使用同一个会话。会话会因为超时而终止(在超时前，用户没有请求另一个页面)，或者调用 HttpSessionState 的 Abandon 方法提前终止会话。为了全局地处理会话启动和会话终止事件，Global.asax.cs 文件定义了 Session_Start 和 Session_End 事件处理程序：

```
protected void Session_Start(object sender, EventArgs e)
{
}
protected void Session_End(object sender, EventArgs e)
{
}
```

用户可以通过 IE 的多个窗口来使用同一个会话。在 IE 中选择 File | New Window 命令，会打开一个使用同一个会话的新窗口。选择 File | New Session 命令会创建一个新会话。这允许同一个 Web 站点的两个会话在两个不同的窗口中。

1. 会话标识符

服务器上有了会话状态，客户端就需要用某种方式来标识，以便把会话映射到客户端。这默认使用名为 ASP.NET_SessionId 的临时 cookie 来实现，如图 40-12 所示。

图 40-12

会话状态可以用各种方式配置。如果不使用 cookie，也可以使用 URL 标识会话。为此，可以在 system.web 配置中定义 sessionState，并把 cookieless 特性设置为 UseUri：

```
<sessionState cookieless="UseUri" />
```

图 40-13 显示了会话标识符和用 UseUri 配置创建的 URL 字符串。在链接中使用 URL 和标识符，不如省略它们，但会话在工作时不带 cookie 是有好处的。

图 40-13

cookieless 特性允许启用 UseCookies(这是默认的)、UseUri、UseDeviceProfile 和 AutoDetect 的设置。使用 UseDeviceProfile，可以从配置文件中提取浏览器的功能，如果浏览器支持 cookie，就使用

cookie。这个设置不会检测用户是否用浏览器关闭了 cookie。设置 AutoDetect 会启动自动检测功能，自动检测功能会把检测 cookie 发送给客户端，客户端给会话使用返回的任何检测过的 cookie，否则就使用 URI。

2. 会话的存储

默认情况下，会话信息仅存储在 ASP.NET 进程内部的内存中。用户在不同的服务器上处理不同的请求时，这在 Web 场中是不切实际的。当 ASP.NET 进程回收时，会话状态就会丢失。在不同的地方存储会话状态，而不只在进程中存储，只是配置问题。

使用 StateServer 模式，会话就可以存储在单独的进程中。ASP.NET State Service(状态服务)安装在运行 ASP.NET 的每个系统上。它只需用本地服务来启动。在会话配置中，mode 需要设置为 StateServer，stateConnectionString 设置为状态服务的服务器名和端口，默认为端口 42424：

```
<sessionState mode="StateServer"
  stateConnectionString="tcpip=127.0.0.1:42424"
  cookieless="UseCookies"
  timeout="20" />
```

使用状态服务器对 ASP.NET 进程的回收非常有帮助，但它无助于 Web 场的情形。如果 Web 场用于可靠性问题，即使 Web 场中的一个系统崩溃了，也会给客户端提供回应，状态服务不能服务时，整个 Web 场的可靠性就没有什么帮助。这里，如果会话状态存储在 SQL Server 数据库群集中，它是有帮助的。Mode 需要设置为 SQLServer：

```
<sessionState mode="SQLServer"
  sqlConnectionString="Integrated Security=SSPI;database=StateServer;" />
```

状态服务的数据库可以用 aspnet_regsql 工具配置。

会话状态也可以存储在自定义状态提供程序中。自定义会话状态提供程序需要从基类 SessionStateProviderBase 派生，并相应实现抽象方法。自定义状态提供程序的配置在下面列出，以使用 Distributed CacheSessionStateStoreProvider，而 DistributedCacheSessionStateStoreProvider 通过 Windows Azure 使用分布式内存：

```
<sessionState mode="Custom" customProvider="DistributedSessionProvider">
  <providers>
    <add name="DistributedSessionProvider"
      type=
  "Microsoft.Web.DistributedCache.DistributedCacheSessionStateStoreProvider,
  Microsoft.Web.DistributedCache" cacheName="default"
     applicationName="AzureSampleApp"
     useBlobMode="true" />
  </providers>
</sessionState>
```

对于所有不同的提供程序而言，会话状态的编程总是相同的。只需注意使用进程内会话状态时，任何对象都可以传递给会话。对于其他提供程序，放入会话中的类型必须是可串行化的。最好确保所有放入会话中的对象都是可串行化的，即使仅使用进程内会话状态，也是如此。

40.7.4 应用程序状态

会话状态是每个用户的服务器端状态。应用程序状态是全局的服务器端状态，在所有用户间共享。

在下面的代码段(代码文件 StateSample/Global.asax.cs)中，Application 属性返回一个 HttpApplicationState 对象，它的用法与 HttpSessionState 对象非常类似。但是，因为应用程序状态在所有用户间共享，所以在修改值之前需要锁定它。使用 Application.Lock 和 Application.UnLock，在进行解锁时要特别小心。为了安全起见，应使用 try/finally。锁定和解锁之间的时间间隔应非常短——在这个时间段应只使用内存访问，长时间的锁定会降低性能，因为只有一个线程可以拥有该锁定，其他所有线程都必须等待该锁定用 Lock 方法来释放：

```
{
  Application["UserCount"] = 0;
}

protected void Session_Start(object sender, EventArgs e)
{
  try
  {
    Application.Lock();
    int userCount = (int)Application["UserCount"];
    Application["UserCount"] = ++userCount;
  }
  finally
  {
   Application.UnLock();
  }
}
```

在代码文件 StateSample/ApplicationStateRead.aspx.cs 中，读取应用程序的状态：

```
protected void Page_Load(object sender, EventArgs e)
{
  int userCount = (int)Application["UserCount"];
  Label1.Text = userCount.ToString();
}
```

40.7.5 缓存

缓存非常类似于应用程序状态，它也在多个用户间共享，但可以对缓存对象的生命周期进行更多的控制。下面的代码段(代码文件 StateSample/CacheWrite.aspx.cs)显示了如何把对象添加到缓存中。Cache 是 Page 类的一个属性，它从 System.Web.Caching 命名空间中返回一个 Cache 对象：

```
protected void Button1_Click(object sender, EventArgs e)
{
  Cache.Add(key: "cache1", value: TextBox1.Text, dependencies: null,
  absoluteExpiration: Cache.NoAbsoluteExpiration,
  slidingExpiration: TimeSpan.FromMinutes(30),
  priority: CacheItemPriority.Normal, onRemoveCallback: null);
}
```

Cache 类的 Add 方法允许灵活地控制所添加的缓存对象。第一和第二个参数定义对象的键和值。

1. 缓存的依赖关系

第三个参数是类型 CacheDependency。依赖可以定义缓存对象何时失效。除了传递 Cache Dependency 之外，还可以添加派生自 CacheDependency 类的任何类型，例如 SqlCacheDependency 和 AggregateCacheDependency。有了这个依赖，就可以把文件的内容加载到缓存中，创建对该对象的依赖——该文件变化时，缓存对象失效。

2. 时间

第四和第五个参数定义了缓存应何时失效。使用 absoluteExpiration 参数，可以指定一个 DateTime，指明缓存应失效的绝对时间。slidingExpiration 参数允许指定 TimeSpan，它只能设置为这两个值中的一个。如果使用了 sliding Expiration，绝对时间就必须设置为 Cache.NoAbsoluteExpiration；相反，如果使用了 absoluteExpiration，TimeSpan 就必须设置为 Cache.NoSlidingExpiration。

3. 优先级

另一个参数可以指定对象的优先级。Web 应用程序没有足够的内存时，ASP.NET 运行库就会删除缓存对象。优先级低的缓存对象比优先级高的缓存对象先删除。优先级用 CacheItemPriority 类型的枚举来定义，其值是 Low、BelowNormal、Normal、AboveNormal、High 和 NotRemovable。

4. 回调方法

在最后一个参数中，可以定义 CacheItemRemovedCallback 类型的回调方法。这个方法在缓存项删除时调用。删除缓存项的原因放在 CachItemRemovedReason 枚举中，其值是 DependencyChanged、Expired、Removed 和 Underused。在回调处理程序中，可以确定是否缓存加载的文件内容；依赖关系改变时，是否重载缓存。当然，如果删除缓存项的原因是内存量很低，缓存项就不应立即重新创建。

40.7.6　配置文件

如果读者曾经在网上购物，就很熟悉 Web 站点的购物车功能无法在会话之间保存已选购的商品。这使用户感觉很糟糕。当用户在网上购物期间小憩时，若超时了，他们不应丢失已选购的商品。否则，用户就必须重复以前的步骤，再次填充购物车。用这种方式设计站点的公司会影响销售量。相反，购物车中的商品应放在一个数据库中。ASP.NET 配置文件设置使之很容易实现。

1. Profile 提供程序

Profile API 基于一个提供程序模型。提供程序用于 ASP.NET 的许多功能，例如前面介绍的会话状态的提供程序。提供程序派生自 System.Configuration.Provider 名称空间中的 ProviderBase 基类。配置文件提供程序派生自 ProfileProvider 基类，ProfileProvider 则派生自 SettingsProvider。.NET Framework 中包含的配置文件提供程序是 SqlProfileProvider。这个提供程序把配置文件信息存储在 SQL Server 数据库中。默认提供程序用 machine.config 配置，使用通过连接字符串 LocalSqlServer 定义的 SQL Server 数据库，如下所示：

```
<profile>
  <providers>
    <add name="AspNetSqlProfileProvider"
      connectionStringName="LocalSqlServer" applicationName="/"
      type="System.Web.Profile.SqlProfileProvider, System.Web,
      Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
  </providers>
</profile>
```

2. 创建数据库

数据库可以随时创建。第一次使用 Profile API(或后面讨论的成员 API)时，会创建一个新的数据库。这是因为 machine.config 文件包含 LocalSqlServer 连接字符串，该字符串引用了 DataDirectory (App_data)中的 aspnetdb.mdf 数据库文件。连接字符串 LocalSqlServer 默认由配置文件提供程序使用：

```
<connectionStrings>
  <add name="LocalSqlServer" connectionString="data source=.\SQLEXPRESS;
    Integrated Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;
    User Instance=true" providerName="System.Data.SqlClient" />
</connectionStrings>
```

在 Visual Studio 2013 中，连接字符串可以改为使用 LocalDb，如下面的 Web.config 文件所示：

```
<connectionStrings>
  <clear/>
  <add name="LocalSqlServer" connectionString="data source=(localdb)\v11.0;
    Integrated Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

示例应用程序不是在第一次请求时自动创建数据库，而是提前创建了一个数据库。使用 aspnet_regsql 工具(位于.NET Runtime 的目录下)可以创建一个 ASP.NET SQL Server 数据库，它包含为不同 ASP.NET 服务创建的所有表。不带任何选项启动 aspnet_regsql，会启动 ASP.NET SQL Server Setup Wizard，如图 40-14 所示。

配置数据库时，可以定义数据库名或使用默认的名称(aspnetdb)，如图 40-15 所示。

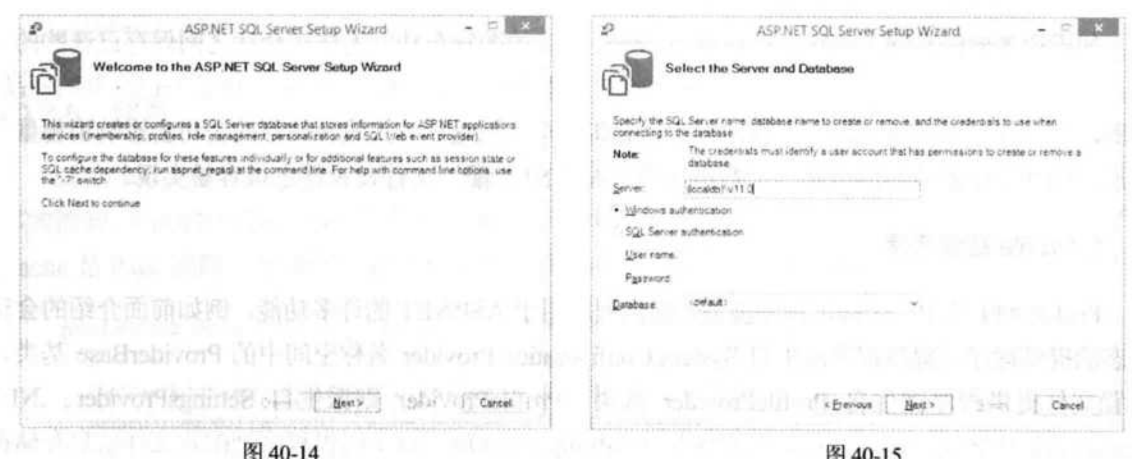


图 40-14　　　　图 40-15

运行向导，会创建一个数据库，其中包含用于配置文件、成员、角色、个性化等的所有表。如

果只需要 ASP.NET 功能的一个子集，并希望数据库包含较少的表，可以使用 aspnet_reqsql 工具的命令行版本，用选中的功能创建表。

Web.config 文件现在引用新创建的数据库：

```
<connectionStrings>
  <clear/>
  <add name="LocalSqlServer" connectionString=
    "data source=(local);Database=aspnetdb;Integrated Security=SSPI;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

3. 配置文件设置

有了默认的配置文件提供程序，配置文件信息就可以用 system.web 元素中的 Web.config 定义。可以为登录到系统中的用户和匿名用户保存配置文件信息。如果用户没有登录，且启动了匿名标识，就创建一个匿名用户 ID。为了在后面的会话中把用户映射到匿名用户上，使用永久性 cookie。这样，设置就总是映射到同一个匿名用户上。应为匿名用户存储的所有属性必须用 allowAnonymous 特性标记。配置文件属性在 profile/properties 中定义。要添加属性，可以使用 add 元素。配置文件属性用 name 和 type 描述。type 用于保存属性的值。类型在数据库中的串行化方式用 serializeAs 特性定义。串行化可以处理为一个字符串，使用二进制或 XML 串行化器处理，或者使用处理串行化过程的自定义类来处理。为了组合配置文件状态信息，可以把属性放在 group 元素中：

```
<anonymousIdentification enabled="true" />
<profile>
  <properties>
    <add allowAnonymous="true" name="Color" type="System.String"
      serializeAs="Xml" />
    <add allowAnonymous="true" name="ShoppingCart"
      type="StateSample.ShoppingCart" serializeAs="Binary" />
    <group name="UserInfo">
      <add name="Name" type="String" serializeAs="Binary" />
    </group>
  </properties>
</profile>
```

4. 使用自定义类型

示例配置文件使用了 ShoppingCart 类型，它定义为二进制串行化。这个类型(代码文件 StateSample/ShoppingCart.cs)包含一组已串行化的项：

```
[Serializable]
public class ShoppingCart
{
  private List<Item> items = new List<Item>();
  public IList<Item> Items
  {
    get
    {
      return items;
    }
  }
```

```
    public decimal TotalCost
    {
      get
      {
        return items.Sum(item => item.Cost);
      }
    }
  }

  [Serializable]
  public class Item
  {
    public string Description { get; set; }
    public decimal Cost { get; set; }
  }
```

5. 写入配置文件数据

有了这个设置，就很容易写入用户配置文件数据了。HttpContext 定义了返回 ProfileBase 的 Profile 属性。有了 ProfileBase，就可以使用索引器读写配置文件属性了。前面完成的配置文件定义了配置属性 Color，它在这里用于索引器：

```
this.Context.Profile["Color"] = "Blue";
this.Context.Profile.Save();
```

如果使用 Visual Studio 站点而不是 Web 项目，Page 类定义 Profile 属性，用于返回动态创建的 ProfileCommon 类。ProfileCommon 派生自 ProfileBase 基类，它提供的属性用配置类型定义为用于强类型化访问的属性。对于 Web 项目，Page 的 Profile 属性是不可用的。ProfileCommon 类以类似的方式创建。使用 dynamic 关键字，编程代码看起来就比使用索引器好一些(代码文件 StateSample/ProfileWrite.aspx.cs)。Save 方法把属性名和值写入数据库：

```
dynamic p = this.Context.Profile;
p.Color = "Red";
p.UserInfo.Name = "Christian";

var cart = new ShoppingCart();
cart.Items.Add(new Item { Description = "Sample1", Cost = 20.30M });
cart.Items.Add(new Item { Description = "Sample2", Cost = 14.30M });

p.ShoppingCart = cart;
p.Save();
```

dynamic 关键字参见第 12 章。

保存配置文件后，就可以在 aspnet_Profile 表中看到一个新行，其属性名和值如图 40-16 所示。如果用户是匿名的，就创建带唯一用户 ID 的新用户。

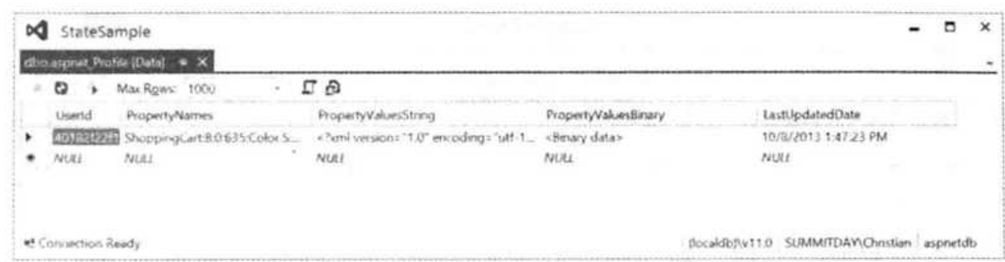


图 40-16

6. 读取配置文件数据

读取配置文件数据与从 HttpContext 访问 Profile 属性的方式类似。下面的示例代码访问 Color 配置文件属性、在 UserInfo 组中定义的配置文件属性，并定制类型 ShoppingCart(代码文件 StateSample/ ProfileRead.aspx.cs)：

```
dynamic profile = Context.Profile;
Response.Write(string.Format("Color: {0}", profile.Color));
Response.Write("<br />");
Response.Write(string.Format("Name: {0}", profile.UserInfo.Name));
Response.Write("<br />");
ShoppingCart shoppingCart = profile.ShoppingCart;
foreach (var item in shoppingCart.Items)
{
  Response.Write(string.Format("{0} {1}", item.Description, item.Cost));
  Response.Write("<br />");
}
Response.Write(shoppingCart.TotalCost);
Response.Write("<br />");
```

7. 配置文件管理器

配置文件状态用于匿名用户时，“残片”会随着时间的推移而累积。如果匿名用户删除了其 cookie，下次就会创建带新匿名用户 ID 的另一个用户。老用户 ID 就不能再映射最初的用户了。

有一个 API 可以管理所有的配置文件。System.Web.Profile 名称空间中的 ProfileManager 提供了简单的方法，来检索所有的配置文件(GetAllProfiles)，包括不活动的配置文件(GetAllInactiveProfiles)。下面的代码段返回从自去年开始就不活动的匿名用户的所有配置文件：

```
var inactiveProfiles = ProfileManager.GetAllInactiveProfiles(
   ProfileAuthenticationOption.Anonymous, DateTime.Now.AddYears(-1));
```

也可以直接使用 DeleteInactiveProfiles 方法来删除不活动的配置文件。

40.8 ASP.NET 身份系统

身份验证和授权是 Web 应用程序的重要部分。如果 Web 站点或其中的一部分不应公开，就必须给用户授权。对于用户的身份验证，创建 ASP.NET Web 应用程序时可以使用不同的选项：不验证身份、单个的用户账户、组织账户和 Windows 身份验证，如图 40-17 所示。

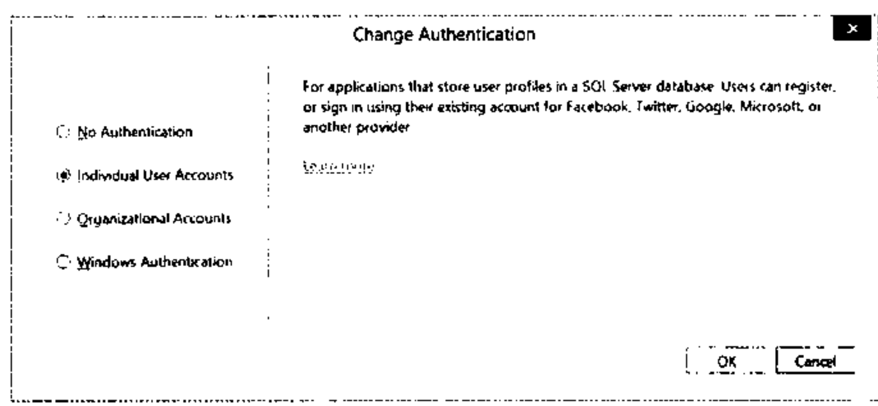


图 40-17

选择 Windows 身份验证，会把身份验证模式定义为 Windows，并拒绝未授权的用户访问：

```
<authentication mode="Windows" />
<authorization>
  <deny users="?" />
</authorization>
```

这种身份验证模式是由 IIS 完成的，验证用户是否是 Windows 用户。使用身份验证部分，可以添加一组允许访问和拒绝访问的用户。? users 的 deny 项指定不允许匿名用户访问，所有用户都必须验证身份。

Organizational Accounts 选项使用 Active Directory，它位于基础系统、Windows Azure 或 Office 365 中。

ASP.NET 现在提供一个新的身份系统：ASP.NET 身份系统。ASP.NET 身份系统可以与 Web Forms、ASP.NET MVC 和 ASP.NET Web API 一起使用。这个系统基于 OWIN 体系结构，可以很容易自定义驻留方式。通过 OWIN 体系结构，建立一个管道，身份验证是管道中的一个组件。

可以用 Individual User Accounts 选项选择 ASP.NET 身份系统。使用这个设置，Web 应用程序可以在定制数据库中存储用户，也可以使用 OAuth 提供程序中的用户，例如 Facebook、Twitter 或 Microsoft 账户。下面详细介绍这个主题。

示例应用程序 AuthenticationSample 是一个用 Individual User Accounts 选项创建的 Web Forms 项目。这个模板添加了 NuGet 包 Microsoft ASP.NET Identity Core。因此，如果从空白的 Web 应用程序开始，就需要这个 NuGet 包。

40.8.1 基础知识

与 ASP.NET 身份系统一起使用的、最重要的接口和类如表 40-1 所示。

表 40-1

类　型	说　明
IUser	这个接口为用户定义唯一的密码和用户名
IdentityUser	这个类型实现了接口 IUser，为登录、声明和角色定义了接口集合的成员
IdentityDbContext	这个类型是一个数据上下文，它把用户和角色信息映射到数据库
UserManager<TUser>	这是一个泛型类型，它会创建、更新、删除和查找用户，添加、改变密码和声明，从而帮助管理用户

(续表)

类 型	说 明
IAuthenticationManager	这个接口允许注册和注销用户。它由 OWIN 管道中的身份验证中间件实现，可以从 HTTP 上下文中访问

40.8.2 存储和检索用户信息

对于用户管理，用户信息需要添加到库中。类型 IdentityUser(在名称空间 Microsoft.AspNet.Identity.EntityFramework 中)定义了一个名称，列出了角色、注册和声明。通过 Visual Studio 模板创建一个 ApplicationUser 类，它派生自基类 IdentityUser。ApplicationUser 类默认为空，但可以从用户那里获得需要的信息(代码文件 AuthenticationSample/Models/IdentityModels.cs)：

```
public class ApplicationUser : IdentityUser
{
}
```

本章前面 40.7.6 节介绍的配置文件提供程序不支持 ASP.NET 身份系统，它们基于成员提供程序。使用 ASP.NET 身份系统可以给 ApplicationUser 类添加用户的额外信息。

与数据库的连接是通过 IdentityDbContext<TUser>类型建立的。这是一个泛型类，派生自基类 DbContext，因此使用 Entity Framework Code First 编程模型。IdentityDbContext<TUser>类型定义了 IDbSet<TEntity>类型的属性 Roles 和 Users，IDbSet<T>类型定义了到数据库表的映射。为了方便起见，创建 ApplicationDbContext 类型，把 ApplicationUser 类型定义为 IdentityDbContext 类的泛型类型：

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
  public ApplicationDbContext()
    : base("DefaultConnection")
  {
  }
}
```

Entity Framework Code First 编程模型参见第 33 章。

除了实体类型和数据上下文类型之外，还创建了辅助类。UserManager 类派生自名称空间 Microsoft.AspNet.Identity 中的泛型类 UserManager<TUser>。这个辅助类提供的虚拟方法可以创建(因此可以通过数据上下文把它写入数据库)、更新、删除和查找用户，处理用户的角色：

```
public class UserManager : UserManager<ApplicationUser>
{
  public UserManager()
```

```
    : base(new UserStore<ApplicationUser>(new ApplicationDbContext()))
  {
  }
}
```

40.8.3 安全启动

身份验证提供程序在 ConfigureAuth 方法中配置为启动(代码文件 AuthenticationSample/App_Start/Startup.Auth.cs)。UseCookieAuthentication 是 IAppBuilder 接口的一个扩展方法。这个方法配置为在用户没有获得身份验证 Cookie 时，就把用户重定向到/Account/Login：

```
public partial class Startup
{
  public void ConfigureAuth(IAppBuilder app)
  {
    app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
      AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
      LoginPath = new PathString("/Account/Login")
    });
    app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);
  }
}
```

模板生成的方法也包含添加了注释符号的代码，用于验证带有 Microsoft、Twitter、Facebook 或 Google 账户的用户。只需取消对应代码行的注释，提供客户 ID 和密码：

```
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//    consumerKey: "",
//    consumerSecret: "");
```

40.8.4 使用注册和身份验证

用户在 Register.aspx.cs 文件中用 UserManager 类的 Create 方法注册。注册后，用户就登录，调用 IdentityHelper 的 SignIn 方法：

```
var manager = new UserManager();
var user = new ApplicationUser() { UserName = UserName.Text };
IdentityResult result = manager.Create(user, Password.Text);
if (result.Succeeded)
{
  IdentityHelper.SignIn(manager, user, isPersistent: false);
  IdentityHelper.RedirectToReturnUrl(
    Request.QueryString["ReturnUrl"], Response);
}
else
{
  ErrorMessage.Text = result.Errors.FirstOrDefault();
}
```

IdentityHelper 是一个简单的辅助类，便于使用管道的身份验证中间件中的 IAuthentication Manager。通过 HTTP 上下文检索身份验证管理器后，IdentityHelper 的 SignIn 方法调用 IAuthenticationManager 的 SignIn 方法：

```
public static void SignIn(UserManager manager, ApplicationUser user,
   bool isPersistent)
{
  IAuthenticationManager authenticationManager =
     HttpContext.Current.GetOwinContext().Authentication;
  authenticationManager.SignOut(
    DefaultAuthenticationTypes.ExternalCookie);
  var identity = manager.CreateIdentity(user,
    DefaultAuthenticationTypes.ApplicationCookie);
  authenticationManager.SignIn(new AuthenticationProperties()
    { IsPersistent = isPersistent }, identity);
}
```

如果用户已经注册，且用户登录到本地数据库(不是外部的提供程序)，就在 Login.aspx.cs 文件的 LogIn 事件处理程序中登录。用户信息使用 UserManager 从数据库中检索，给该方法传递用户名和密码，下一个登录用 IdentityHelper 完成：

```
protected void LogIn(object sender, EventArgs e)
{
  if (IsValid)
  {
    // Validate the user password
    var manager = new UserManager();
    ApplicationUser user = manager.Find(UserName.Text, Password.Text);
    if (user != null)
    {
      IdentityHelper.SignIn(manager, user, RememberMe.Checked);
      IdentityHelper.RedirectToReturnUrl(
        Request.QueryString["ReturnUrl"], Response);
    }
    else
    {
      FailureText.Text = "Invalid username or password.";
      ErrorMessage.Visible = true;
    }
  }
}
```

ASP.NET 身份系统和身份验证部分，以及 Web Forms 和 ASP.NET MVC 之间的区别参见第 41 和第 42 章。

40.9 小结

本章提供了 ASP.NET 的核心信息，包括 ASP.NET Web Forms 和 ASP.NET MVC 共享的 ASP.NET 部分。我们了解了处理程序和模块的后台情况，这也可以用于创建为每个请求调用的特定功能。本章还介绍了 Web 应用程序的请求和响应、状态管理和一些提供程序，如成员和角色。

我们论述了如何创建处理程序和模块，以及 ASP.NET Web Forms 和 ASP.NET MVC 的基础知识。即使使用非.NET 技术创建 Web 应用程序，也可以创建自定义处理程序和模块。

本章探讨了状态管理的不同选项，如何在客户端和服务器上处理状态。在客户端，视图状态绑定到单个页面上，只要浏览器会话是活动的，Cookie 就可以是临时的。Cookie 也可以是永久的。在服务器端状态，有不同的选项：会话、应用程序、缓存和配置。会话状态基于用户会话，用户会话可以配置为存储在进程内部、状态服务器上或数据库中。应用程序状态在所有用户间共享，还需要注意锁定问题。缓存仅存储在内存中，主要用于引用数据，以不会对每个请求协商数据库。配置文件用作保存用户数据的永久存储器。

第 41 章介绍 ASP.NET Web Forms，这是自从.NET 1.0 以来就有的 ASP.NET 技术。

第41章

ASP.NET Web Forms

本章要点

- 服务器端控件
- 母版页
- 站点导航
- 验证用户输入
- 数据访问
- 安全性
- ASP.NET AJAX

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章的代码分为以下几个主要的示例文件：

- ProCSharpSample
- ProCSharpAjaxSample

41.1 概述

本章将学习 ASP.NET Web Forms 提供的一些可以增强 Web 应用程序的技术。使用这些技术时，创建 Web 站点和应用程序将更加简单，而且还可以添加高级功能，改善用户体验。

首先讨论 ASP.NET Web Forms 的页面模型，了解页面事件和回送，然后详细介绍用于编写代码、绑定数据和使用表达式的<%语法。

本章还介绍了母版页，这是一项为 Web 站点提供模板的技术。使用母版页，可以通过大量可重用的代码在整个网站中实现复杂的 Web 页面布局。还将介绍如何结合使用导航 Web 服务器控件和母版页在整个 Web 站点中提供一致的导航。

站点导航可以针对用户，例如只让特定的用户(如站点的注册用户或站点的管理员等)访问站点

的某个部分。还将讨论站点的安全性，以及如何登录Web应用程序——通过Web服务器登录控件，这是很容易实现的。

最后，将讨论ASP.NET AJAX，这是一组强大的技术，用来增强用户体验。它使Web站点和Web应用程序能够独立更新页面的各个部分，从而提高它们的响应速度，简化添加客户端功能的过程。

41.2 ASPX页面模型

当客户端向Web Forms应用程序发出HTTP请求时，会实例化一个页面并创建一个响应。为了了解这个页面及其模型，创建一个ASP.NET Empty Web Application，命名为ProCSharpSample，选择Empty模板，给文件夹添加Web表单和核心引用选项。创建这个项目后，在其中添加一个名为ShowMeetingRooms.aspx的Web表单。

ASPX页面的第一行包含一个Page指令，如下面的代码片段所示(代码文件ProCSharpWeb/PageModel/ShowMeetingRooms.aspx)。这个指令定义了ASP.NET页面解析器、编译器和Visual Studio的特性。编译器在运行期间使用Language特性来编译ASPX页面内的语句。ASPX语句包含在<% %>内。AutoEventWireup特性设为true，意味着页面事件的事件处理程序将被自动关联。只需要在定义方法时，提供正确的名称和签名，以激活页面的事件处理程序。运行期间不会使用CodeBehind特性，这告诉Visual Studio，文件ShowMeetingRooms.aspx.cs属于ShowMeetingRooms.aspx页面，因此在Solution Explorer中显示时，能看到它们之间的关系。对于ASPX引擎来说，重要的是Inherits特性。在ASPX页面中，从Inherits特性定义的基类创建了一个派生类：

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="ShowMeetingRooms.aspx.cs"
  Inherits="ProCSharpSample.ShowMeetingRooms" %>
```

文件ShowMeetingRooms.aspx.cs是代码隐藏文件。默认情况下，只实现了Page的Load事件的处理程序方法Page_Load。由于AutoEventWireup特性，完成了到这个处理程序的映射：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace ProCSharpSample
{
  public partial class ShowMeetingRooms : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {

    }
  }
}
```

41.2.1　添加控件

把控件从工具箱拖放到编辑器的设计器或源代码视图，即可把它们添加到页面。应用程序的第一个页面显示了一个下拉列表，用于选择一个会议室。为此，页面中包含了一个 DropDownList、一个 Label 和一个 Button 控件。在设计器视图中，单击 DropDownList 控件的智能标记将打开 DropDownList 任务，其中包含菜单项 Edit Items。选择此菜单项可打开如图 41-1 所示的 ListItem Collection Editor 对话框。

图 41-1

使用此编辑器可向 DropDownList 控件中添加一些会议室的名称。ShowMeetingRooms.aspx 的代码如下所示：

```
<body>
  <form id="form1" runat="server">
  <div>
    <asp:DropDownList ID="DropDownListMeetingRooms" runat="server" Width="165px">
      <asp:ListItem>Sacher</asp:ListItem>
      <asp:ListItem>Hawelka</asp:ListItem>
      <asp:ListItem>Hummel</asp:ListItem>
      <asp:ListItem>Prückel</asp:ListItem>
      <asp:ListItem>Landtmann</asp:ListItem>
      <asp:ListItem>Sperl</asp:ListItem>
      <asp:ListItem>Alt Wien</asp:ListItem>
      <asp:ListItem>Eiles</asp:ListItem>
    </asp:DropDownList>
    <br />
    <br />
    <asp:Label ID="LabelSelectedRoom" runat="server" Text=""></asp:Label>
    <br />
    <br />
    <asp:Button ID="Button1" runat="server" Text="Submit" />
  </div>
  </form>
</body>
```

从 runat="server"特性可以看出，DropDownList、Label 和 Button 都是服务器端控件。这些控件可用服务器端 C#代码编程，并把 HTML 和 JavaScript 代码返回到客户端。代码隐藏文件是一个部分类。设计器为相同类型的类创建了另一个部分类文件，其中只包含 ASPX 文件中命名的服务器控件

的成员变量。这样，就可以从代码隐藏文件访问所有控件，因为代码隐藏文件与该部分类文件的类相同。

41.2.2 使用事件

现在向控件添加事件处理程序。在 ASPX 中，可以直接在 ASPX 代码编辑器中添加事件处理程序，也可以通过属性(Properties)窗口添加。对于示例页面中的 DropDownList 控件，把 OnRoomSelection 方法赋给 OnSelectedIndexChanged 事件。如果用户改变了选择，服务器端就会进行相应的处理：

```
protected void OnRoomSelection(object sender, EventArgs e)
{
  this.LabelSelectedRoom.Text = DropDownListMeetingRooms.SelectedItem.Value;
}
```

41.2.3 使用回送

试着在 Visual Studio 中启动该页面。下拉列表中包含了所有的会议室，生成的 HTML 代码如下所示。这段代码不包含任何服务器端代码。runat="server"特性也被移除：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
</title></head>
<body>
  <form method="post" action="ShowMeetingRooms.aspx" id="form1">
    <div class="aspNetHidden">
      <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
        value="u1PYwDLRsU6bWhjWCNAUuO+9ETPHK9DCp2yJxKTHikrAh/ghb3nUb81ZP06x2
        sDPdBHpJ4ObOMGKB8reZ2yNJqg42ep+xM6cgmks2irc7+ZrY5bnMtGj22CfjGOW5otD" />
    </div>
    <div class="aspNetHidden">
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
        value="AIZNlXgDjsO7fINvnNT9WSFZZfkci1pv28cSbIIvwKCAGjBtX9ZDzL+NL4+S
        LcTF2t7XXvjezChHhEHzRI08UHIfPpQ1AfR1c81+s3If019+FPdZg4d8ByuVtUu9nIL
        0mZEaiwn3Ab8KrkuYaHm6KaXqksh4/BJrp4SV5BjetsYgC/F5+JFdFi70Uy/yORSlzr
        8XJGOmHEmjxXf3XILwf1MEkBifAF9KAc/05a9h7Ih5HSFh6/8nODcbHCsywcvpWnoW1
        kCCe3DeAD74aIoert/JOR+9cjwqBcvS+uRE7Vs=" />
    </div>
    <div>
      <select name="DropDownListMeetingRooms" id="DropDownListMeetingRooms"
        style="width:165px;">
        <option selected="selected" value="Sacher">Sacher</option>
        <option value="Hawelka">Hawelka</option>
        <option value="Hummel">Hummel</option>
        <option value="Prückel">Prückel</option>
        <option value="Landtmann">Landtmann</option>
        <option value="Sperl">Sperl</option>
        <option value="Alt&#32;Wien">Alt Wien</option>
        <option value="Eiles">Eiles</option>
      </select>
      <br />
      <br />
      <span id="LabelSelectedRoom"></span>
```

```
        <br />
        <br />
        <input type="submit" name="Button1" value="Submit" id="Button1" />
      </div>
    </form>
  </body>
  </html>
```

但是，这里有一个问题：选择发生变化时，并不会调用事件处理程序。只有当单击 Submit 按钮时，才会在服务器端调用事件处理程序。ASPX 页面模型是基于回送的。因为编译过的 C#代码在服务器上运行，所以客户端需要向服务器发送请求来调用事件处理程序。单击 Submit 按钮就发送了这样的一个请求：使用 HTTP POST 请求，将 form 元素内的控件的所有状态信息发送给服务器。其中也包含视图状态信息。ASPX 控件使用视图状态来管理事件处理功能。把页面发送给客户端时，视图状态包含了关于控件实际状态的信息，例如在 DropDownList 控件中选择了什么。用户在 DropDownList 控件中改变一个值时，这些状态信息仍不会改变。把数据发送到服务器时，视图状态仍然包含原始信息，与视图内的 DropDownList 一起传递的状态也包含了当前信息。在服务器端，现在可以检测到原始状态到当前状态之间发生的变化，所以将触发事件 OnSelectedIndexChanged，并调用相应的事件处理程序。

41.2.4　使用自动回送

有时，必须在 DropDownList 控件发生变化后，立即向服务器发送回送。在做出选择后，可能必须要修改页面的其他一些部分。通过设置 DropDownList 控件的 AutoPostBack="true"属性，可以实现这种操作。在 HTML 中，不使用 JavaScript 完不成这种修改，但 DropDownList 控件会自动创建 JavaScript 代码，在 select 元素的 onchange 事件后进行表单回送。

41.2.5　回送到其他页面

前面介绍的回送总是在请求相同的页面。如果在完成回送后需要把不同的页面返回给客户端，有几种不同的方法。通过调用 Response.Redirect 方法，客户端会接收 HTTP 重定向请求，以请求另一个页面。这个方法需要多请求一次服务器。通过调用 Server.Transfer 方法，会在服务器端调用另一个页面。这不会再次经过服务器，但是客户端看到的 URL 是原始页面(而非新页面)的 URL。ASPX 还支持另外一种方式：跨页面回送。

现在创建另外一个页面 MeetingRoomInformation.aspx。这个页面包含一个 Label 控件，用于显示选中的会议室。

在 ShowMeetingRooms.aspx 页面中，修改 Submit 按钮，将其 PostBackUrl 属性设为新页面。有了这些信息后，生成的 HTML 代码包含到 Button 控件的 JavaScript 事件 onclick，它会将表单回送到新页面：

```
<asp:Button ID="Button1" runat="server" Text="Submit"
  PostBackUrl="~/MeetingRoomInformation.aspx"/>
```

上述代码中 URL 使用的~映射到虚拟应用程序根目录上。

MeetingRoomInformation.aspx 页面的 Page_Load 事件处理程序可以访问前一个页面的页面值。当发生跨页面回送时，PreviousPage 属性包含了前一个页面的信息。为了处理这种行为，前一个页面的 IsCrossPagePostback 属性被设为 true。因此，可以访问前一个页面的所有已设置的控件及其状态，以及从客户端传递的值。使用页面的 FindControl 方法，并传递控件的名称，可以访问该控件。在下面的代码段中，把从 DropDownList 控件选择的值赋值给当前页面内的 Label 的 Text 属性：

```
protected void Page_Load(object sender, EventArgs e)
{
  if (this.PreviousPage != null)
  {
    DropDownList meetingRoomSelection = this.PreviousPage.FindControl(
      "DropDownListMeetingRooms") as DropDownList;
    if (meetingRoomSelection != null)
    {
      this.Label1.Text = meetingRoomSelection.SelectedItem.Value;
    }
  }
}
```

为进行测试，在浏览器中打开 ShowMeetingRooms.aspx 页面，然后单击 Submit 按钮。这会产生跨页面回送，并打开第二个页面。

41.2.6 定义强类型化的跨页面回送

ASPX 允许对前一个页面进行强类型化访问。为了利用这一功能，向 MeetingRooms 类添加只读属性 SelectedMeetingRoom。这个属性用于访问在 DropDownList 控件中选择的值：

```
public string SelectedMeetingRoom
{
  get
  {
    return DropDownListMeetingRooms.SelectedItem.Value;
  }
}
```

在 MeetingRoomInformation.aspx 文件中，把 PreviousPageType 指令添加到 Page 指令的后面。前一个页面用 VirtualPath 特性指定：

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="MeetingRoomInformation.aspx.cs"
  Inherits="Meetingroom.MeetingRoomInformation" %>
<%@ PreviousPageType VirtualPath="~/ShowMeetingRooms.aspx" %>
```

在示例代码中，前一个页面始终是 ShowMeetingRooms.aspx。当需要把一个以上的页面作为前一个页面时，可以创建一个派生自 Page 类的类，这个类本身是其前面所有 Page 类型的父类。然后，就可以在 PreviousPageType 指令中指定 TypeName 特性，而不是指定 VirtualPath 特性。

现在可以简化 Page_Load 方法的实现，以直接访问 SelectedMeetingRoom 属性：

```
protected void Page_Load(object sender, EventArgs e)
{
  if (this.PreviousPage != null)
  {
    this.Label1.Text = this.PreviousPage.SelectedMeetingRoom;
  }
}
```

41.2.7　使用页面事件

在 ShowMeetingRooms.aspx 页面中，已经加载和改变了页面的事件。页面事件还有很多。在呈现页面前，会在第一次请求时触发这些页面事件：PreInit、Init、InitComplete、PreLoad、Load、LoadComplete、PreRender、Render 和 RenderComplete。这些事件多数用 3 个步骤定义：PreXXX、XX 和 XXComplete。在回送到页面时，会触发一些额外的事件来处理视图状态和 HTTP POST 数据，以填充控件的值，以及触发变化和动作事件。

页面生命周期的第一步是初始化 Page。初始化之前，会触发 PreInit 事件。此时可以修改母版页和主题。该事件之后就不能再次设置这些属性了。也可以在这个事件中动态创建控件。当页面的所有控件均被初始化后，将触发 Init 事件。使用 Page_Init 事件处理程序可以设置页面中控件的初始化值。当页面及其控件完成初始化后，将会触发 InitComplete 事件。

初始化阶段过后，是加载阶段。当加载了页面及其控件的视图状态，并把表单的回送数据赋值给页面的控件时，就会触发 PreLoad 事件。触发 Load 事件时，页面已被恢复，所有控件都被设为之前的状态(基于视图状态)。可以在这里完成验证，以及动态创建新控件，这些控件不基于回送的视图状态初始化。变化和动作事件将在这个状态触发，例如 DropDownList 控件的 OnSelectedIndex-Changed 事件，以及 Button 控件的 OnClick 事件。变化事件在动作事件之前触发。LoadComplete 事件标志着加载阶段完成。

在加载阶段后，是呈现阶段。在 PreRender 事件的处理程序中，可以完成对页面及其控件的最终修改。这个事件在保存任何视图状态前触发。这是信息进入视图状态前，最后一次修改属性值的机会。保存视图状态的时候，会触发 SaveStateComplete 事件。然后就可以准备为客户端呈现内容，并生成 HTML 和 JavaScript。Render 方法就在这里调用。在完成呈现后，页面就被卸载，UnLoad 事件就被触发。这里可以完成清理，以释放构建页面所需要的所有资源。

为了分析页面事件，以及查看什么时候发生了什么，最好打开 ASP.NET Web 跟踪。通过 web.config 文件的 trace 元素，可以启用跟踪。这个元素被定义为 system.web 的子元素。下面的代码段中的示例配置启用了跟踪，并定义了一些跟踪配置。跟踪信息不应该显示在被请求的页面本身(pageOutput)中，而是应该使用 trace.axd URL。requestLimit 指定了只应该记忆最近的 10 个请求，更早的请求将被丢弃。将 mostRecent 设为 true，指定对请求的限制只适用于最近的页面，而不是自应用程序启动后第一批请求的页面。localOnly 特性指定了跟踪信息只在与服务器位于相同系统的客户端显示。进行这种限制是为了确保安全。traceMode 被设为按时间排序跟踪。另一个选项是按类别排序跟踪：

```
<trace enabled="true" pageOutput="false" mostRecent="true"
```

```
requestLimit="10" localOnly="true" traceMode="SortByTime"/>
```

在启用跟踪后启动应用程序时，可以填充跟踪，并通过在浏览器中打开 trace.axd 查看。输出如图 41-2 所示。在这个应用程序跟踪中，用 GET 请求打开了 ShowMeetingRooms.aspx，然后在 DropDownList 中选择一项，导致向同一个页面发送一个 POST 请求，然后单击 Submit 按钮，向 MeetingRoomInformation.aspx 页面发送一个 POST 请求。

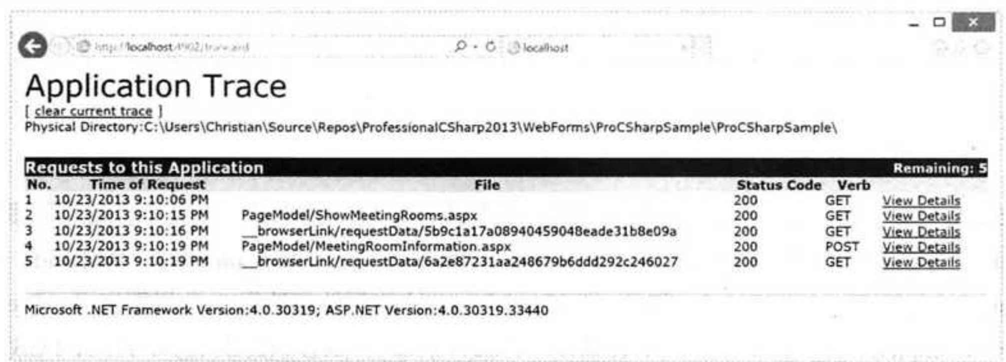


图 41-2

单击页面上的 View Details 链接将提供大量额外的信息。图 41-3 显示了会话标识符(Session ID)的请求详情，以及页面上发生的事件的跟踪信息，包括一些时间信息。从这些信息可知事件处理程序需要的时间，从而判断需要调整哪些代码来获得更好的性能。通过调用 TraceContext 类的 Write 和 Warn 方法，可以在跟踪信息中编写自定义消息。使用页面的 Trace 属性，或者 HttpContext.Current 的 Trace 属性，可以访问 TraceContext 类。

图 41-3

41.2.8 ASPX 代码

在代码隐藏文件中，使用相应变量的 ID，可以访问服务器端控件。如果代码文件中有一个默认隐藏的部分类，那么该代码隐藏文件默认包含 ASPX 文件定义的所有页面成员。下面将讨论在 ASPX 文件内访问代码的不同选项。

1. 写入响应流

使用<% %>语法可在 ASPX 页面内定义代码块。在这个代码块中可以使用多条语句，每条语句必须以分号结束。下面的代码段执行一些计算，并使用 Response.Write 把结果写入响应：

```
<div>
  <%
    int a = 3;
    int b = 4;
    int c = a + b;
    Response.Write(c);
  %>
</div>
```

2. 获得结果

为了把方法或属性的结果直接写入响应流中，可以使用<%=语法。下面的代码段调用了页面的 GetText1 方法，并把结果直接写入 div 元素的内容：

```
<div>
  <%= GetText1() %>
</div>
```

GetText1 方法只是返回一个从 ASPX 页面调用的简单字符串：

```
public string GetText1()
{
  return "Hello from the Page";
}
```

3. 编码

使用<%=存在潜在的风险，特别是如果写入 UI 的内容来自用户，并且不对用户输入进行验证。为了理解这种潜在的危险，向 GetText2 方法的返回字符串添加一个简单的脚本。如果按前面所示调用此方法，将在浏览器中调用该脚本。alert 函数只是在客户端打开消息框，但是有可能访问完整的 DOM 模型，并把用户重定向到其他页面：

```
public string GetText2()
{
  return @"<script>alert(""Hello"");</script>";
}
```

为了避免这类脚本攻击，应该对输出进行编码。Server.HtmlEncode 编码输入字符串，并返回一个 HTML 编码的字符串，使浏览器能够把脚本显示为文本：

```
<div>
  <%= Server.HtmlEncode(GetText2()) %>
</div>
```

因为编码的用途很广，所以针对 Server.HtmlEncode 和<%=，存在一种简写表示法：

```
<div>
  <%: GetText2() %>
</div>
```

4. 数据绑定

<%=可用于绑定方法或属性的结果。下面的代码段将 GetText1 方法的结果绑定到 Button 控件的 Text 属性：

```
<asp:Button ID="Button1" runat="server" Text="<%# GetText1() %>"/>
```

简单地定义绑定并不会发生什么。必须调用 DataBind 方法。下面在 Page_Load 事件处理程序中调用了 DataBind 方法：

```
protected void Page_Load(object sender, EventArgs e)
{
  this.Button1.DataBind();
}
```

除了在控件上调用 DataBind 方法，还可以在 Page 上调用 DataBind 方法。这会在与该 Page 关联的所有控件上调用 DataBind。

5. 表达式

<%$是使用表达式生成器的语法。下面的代码段使用了资源表达式来访问来自资源 SampleResources 的结果，其中使用了键 Message1：

```
<asp:Button ID="Button2" runat="server"
  Text="<%$ Resources:SampleResources, Message1 %>" />
```

资源表达式以<%$ Resources:开头。第 28 章详细讨论了资源。

ASP.NET 提供了几种表达式类型。AppSettings 表达式从配置文件中读取应用程序的配置值，ConnectionString 表达式从配置文件中读取连接字符串，RouteUrl 和 RouteValue 表达式使用 URL 链接从路由中获取值。

为编辑各种类型的表达式，在设计编辑器中选择 Properties，然后打开 Expressions 编辑器，如图 41-4 所示。

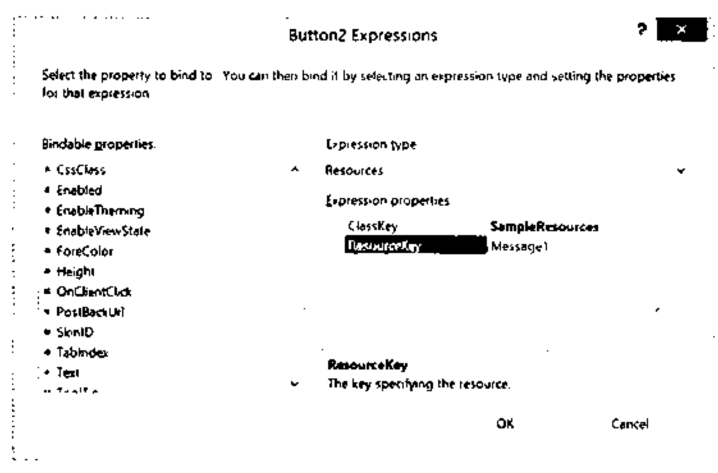


图 41-4

41.2.9 服务器端控件

到目前为止，见到了两种 Web 服务器控件：DropDownList 和 Button。当然，还有其他很多 Web

服务器控件，它们按照类别分组在工具栏中。本章将讨论到其中大多数类别。

工具栏的 Standard 类别不只包含简单的控件，如 Button、Label、DropDownList、CheckBox 和 ListBox，它们很容易以 HTML 表示，还包含需要用复杂的 HTML 表示的控件，如 Table、Calendar、MultiView 和 Wizard。MultiView 支持定义不同的视图，以根据不同的选项选择不同的视图(例如，根据用户是否登录显示不同的视图)，Wizard 则允许用户逐步完成一个过程。

Data 类别包含用于显示数据的控件(Repeater、FormView、GridView)，以及用于访问数据的控件(EntityDataSource、ObjectDataSource、SiteMapDataSource)。

Validation类别包含验证器控件，可用于为客户端和服务器端代码检查用户输入。RequiredFieldValidator要求有用户输入，RangeValidator则检查用户信息是否在指定的值范围内，CompareValidator 不只检查新输入的密码是否匹配在另一个字段中输入的密码，还会验证用户输入是否符合日期或货币值的要求。使用RegularExpressionValidator，则可以通过指定正则表达式来验证用户输入。

Navigation 类别包含 Menu、SiteMapPath 和 Tree 控件，用于为用户创建导航结构。

Login类别包含与安全性任务相关的控件。Login控件允许输入用户名和密码，它使用了Membership API。用户可以使用CreateUserWizard控件注册，使用ChangePassword和PasswordRecovery 修改和找回密码，使用LoginName、LoginStatus和LoginView控件显示身份验证状态。

WebParts 类别包含用于生成动态 Web 应用程序的控件。使用此类别中的控件时，用户可以选择要显示哪部分 Web 部件，并在不同部分之间移动 Web 部件。

借助于 ASP.NET Web Forms 的服务器端控件模型，AJAX Extensions 类别包含的控件使得使用 AJAX 功能变得非常简单，不再需要编写 JavaScript 代码。这个类别中包含 ScriptManager、UpdatePanel 和 Timer 控件。

最后，HTML 类别默认情况下没有服务器端功能的控件。这个类别包含简单的 HTML 控件，如 Input(Button)、Input(Text)和 Textarea。通过应用 runat="server"特性，也可以在服务器端代码中使用这些控件。由于可以具有服务器端功能，这些控件类型定义在 System.Web.UI.HtmlControls 名称空间中。与 Web 服务器控件(包含在 System.Web.UI.WebControls 名称空间中)不同，这些控件的服务器端属性和名称与访问 HTML DOM 的客户端脚本代码中的函数对应。

41.3 母版页

许多 Web 应用程序包含在所有页面之间共享的一些部分。ASP.NET Web Forms 中使用母版页(master pages)来处理这种行为。使用母版页时，内容页不会返回完整的 HTML 代码。内容页只定义应该放在母版页中的页面部分。母版页使用内容占位符定义内容页应该出现在什么位置。使用这些内容占位符，母版页还可以定义当内容页没有提供内容时显示的默认内容。

41.3.1 创建母版页

本小节在示例 Web 应用程序中添加了一个 Web 表单母版页 Company.Master。Visual Studio 2013 模板创建了一个母版页，它的 head 部分和 body 部分各包含一个 ContentPlaceHolder 控件。head 部分的内容占位符允许内容页向页面添加脚本代码或样式表。在下面的代码段中，向 body 部分添加了多个 ContentPlaceHolder 控件，以及一些 HTML 代码。

查看示例中母版页的代码，注意母版页以 Master 指令开头。这与在 Web 页面中使用 Page 指令

类似。但是，客户端从不会请求带.Master文件扩展名的母版页。实际上，如果请求母版页文件，处理程序映射会定义一个禁用的处理程序。因此，客户端请求ASPX页面，ASPX页面处理程序使用母版页来为客户端生成HTML代码。<html>根元素(包含head、body和form)在母版页中定义，而不是在使用母版页的内容页中定义。向母版页添加HTML和Web服务器控件的方式与向Web页面添加它们的方式相同。

母版页内容的特殊之处在于ContentPlaceHolder控件。示例代码定义了ID为topContent、leftContent和mainContent的ContentPlaceHolder控件。它们将被内容页替换掉。母版页也可以提供默认的内容，如本例中名为leftContent的ContentPlaceHolder控件所示。在示例中，用一个导航项列表定义了nav元素。当内容页没有为这个内容占位符添加自己的内容时，就在内容页中显示这个列表：

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Company.master.cs"
  Inherits="Meetingroom.Company" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <link rel="stylesheet" type="text/css" href="Company.css" />
  <title></title>
  <asp:ContentPlaceHolder ID="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
  <div class="top">
    <h1>Professional C# 2012 Demo Web Application</h1>
  </div>
  <div class="top2">
    <asp:ContentPlaceHolder ID="topContent" runat="server">
    </asp:ContentPlaceHolder>
  </div>
  <div class="left">
    <!-- Navigation Controls -->
    <asp:ContentPlaceHolder ID="leftContent" runat="server">
      <nav>
        <ul>
          <li>Home
          <ul>
            <li>Reserve Room</li>
            <li>Show Rooms</li>
          </ul>
          </li>
          <li>About</li>
        </ul>
      </nav>
    </asp:ContentPlaceHolder>
  </div>
  <div class="main">
    <asp:ContentPlaceHolder ID="mainContent" runat="server">
    </asp:ContentPlaceHolder>
  </div>
  <div class="bottom">
    <div>CN innovation</div>
    <div>http://www.cninnovation.com</div>
```

```
    </div>
  </form>
</body>
</html>
```

如图 41-5 所示，Visual Studio 的设计视图显示了页面的外观，以及应用到母版页的 CSS 文件。

图 41-5

41.3.2 使用母版页

为了使用母版页，Visual Studio 2013 包含了一个模板 Web Form Using Master Page。使用这个项模板，可以在创建页面时选择母版页。所生成页面的 Page 指令只是定义额外的 MasterPageFile 特性，用于引用所选择的母版页。除了 Page 指令，页面中只包含 Content 控件。Content 控件定义了替换母版页中的什么内容。Content 控件通过 ContentPlaceHolderId 属性引用 ContentPlaceHolder 控件，以定义要替换的内容：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Company.Master"
  AutoEventWireup="true" CodeBehind="ReserveRoom.aspx.cs"
  Inherits="Meetingroom.ReserveRoom" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="Content2" runat="server" contentplaceholderid="topContent">
</asp:Content>
<asp:Content ID="Content3" runat="server" contentplaceholderid="mainContent">
</asp:Content>
```

示例页面通过在 Page 指令中应用 MasterPageFile 特性来使用母版页。引用母版页的其他选项包括在配置文件中全局定义母版页，或者在页面事件中通过编程引用母版页。在 web.config 配置文件中，page 元素使用 MasterPageFile 特性指出在什么地方全局设置母版页。在页面内使用 Page 指令或代码隐藏文件，都可以覆盖该设置。使用代码隐藏文件时，可在 Page_PreLoad 事件处理程序中设置母版页(这是能够设置 MasterPageFile 属性的最后一个位置)。以编程方式应用此设置时，可以根据不同的需求(例如不同的合作商和不同的合同)使用不同的母版页文件，或者为移动设备提供不同的母版页。

默认情况下，对于母版页没有为其创建默认内容的所有占位符，将创建 Content 控件。为替换默认内容，可以访问 Web 页面的设计视图，使用控件的智能标记替换内容。

在示例 Web 页面中，mainContent 的内容被前一个示例中用过的 DropDownList 替换：

```
<asp:Content ID="Content3" runat="server" contentplaceholderid="mainContent">
  <div>
    <asp:DropDownList ID="DropDownListMeetingRooms" runat="server"
      OnSelectedIndexChanged="OnRoomSelection" Width="165px" AutoPostBack="True">
      <asp:ListItem>Sacher</asp:ListItem>
      <asp:ListItem>Hawelka</asp:ListItem>
      <asp:ListItem>Hummel</asp:ListItem>
      <asp:ListItem>Prückel</asp:ListItem>
      <asp:ListItem>Landtmann</asp:ListItem>
      <asp:ListItem>Sperl</asp:ListItem>
      <asp:ListItem>Alt Wien</asp:ListItem>
      <asp:ListItem>Eiles</asp:ListItem>
    </asp:DropDownList>
    <br />
    <br />
    <asp:Label ID="LabelSelectedRoom" runat="server" Text=""></asp:Label>
    <br />
    <br />
    <asp:Button ID="Button1" runat="server" Text="Submit" />
  </div>
</asp:Content>
```

运行应用程序，HTML 代码与母版页和内容页混合，如图 41-6 所示。

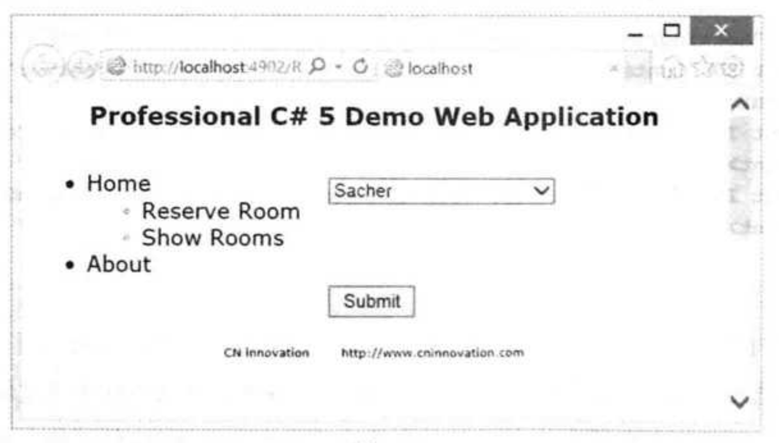


图 41-6

41.3.3 在内容页中定义母版页内容

使用占位符和内容控件来定义页面内容是一个很强大的选项。但是，有时候只需要替换母版页上的少量占位符，或者修改一些内容。在内容页中，使用 page 对象的 Master 属性即可访问母版页。

下面的代码段使用 Page 的 Master 属性(它返回一个 MasterPage 对象)，访问母版页中的 Label

控件 LabelBottom，再使用 FindControl 方法确定在什么地方设置内容：

```
protected void Page_Load(object sender, EventArgs e)
{
  Label label = Master.FindControl("LabelBottom") as Label;
  if (label != null)
  {
    label.Text = "Hello from the content page";
  }
}
```

就像可以对跨页面回送进行强类型访问，也可以对母版页进行强类型访问。接下来，在母版页的代码隐藏文件 Company.Master.cs 中，定义了属性 LabelBottomText，以允许读写 LabelBottom 控件的 Text 属性：

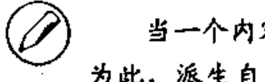
```
public string LabelBottomText
{
  get
  {
    return LabelBottom.Text;
  }
  set
  {
    LabelBottom.Text = value;
  }
}
```

为了定义强类型母版页，可以在内容页中应用 MasterType 指令：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Company.Master"
  AutoEventWireup="true" CodeBehind="ReserveRoom.aspx.cs"
  Inherits="Meetingroom.ReserveRoom" %>
<%@ MasterType VirtualPath="~/Company.Master" %>
```

现在，访问母版页的代码可以简化为直接使用所定义的属性：

```
protected void Page_Load(object sender, EventArgs e)
{
  Master.LabelBottomText = "Hello from the content page";
}
```



当一个内容页中要使用多个母版页时，以强类型方式访问母版页也是可行的。为此，派生自 MasterPage 类的自定义基类可以定义内容页需要的属性。在使用 MasterType 指令引用母版页以进行强类型访问时，可以使用 TypeName 特性代替 VirtualPath。

41.4 导航

现在添加一些导航内容，以便在 Web 应用程序的不同页面间导航。ASP.NET 提供了站点地图，可以与 Menu 和 MenuPath 控件一起使用。

41.4.1 站点地图

Web 应用程序的结构可以在一个站点地图文件中描述。为了创建站点地图，Visual Studio 2013 提供了 Site Map 项模板，它会创建一个名为 Web.sitemap 的文件。下面的代码段显示了一个示例站点地图文件。siteMap 是根元素，它可以包含 siteMapNode 元素的一个层次结构。siteMapNode 定义了所访问的 url，显示在菜单中的 title，以及显示在工具提示中的 description：

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="~/Default.aspx" title="Home"
    description="Professional C# 2012 Demo Web Application">
    <siteMapNode url="MeetingRooms.aspx" title="Meeting Rooms" description="" >
      <siteMapNode url="~/ReserveRoom.aspx" title="Reserve a Room" />
      <siteMapNode url="~/CancelRoom.aspx" title="Cancel a Room" />
    </siteMapNode>
    <siteMapNode url="~/Accounts.aspx" title="Accounts">
      <siteMapNode url="~/RegisterUser.aspx" title="Register User"
        description="" />
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

除了示例中使用的特性外，siteMapNode 还提供了本地化以及对角色的支持。对于本地化，可使用一个 resourceKey 来指定用于菜单标题和工具提示的资源。而使用 roles 特性时，只有指定角色的用户才可以进行访问。

第 28 章讨论过用于本地化的资源。

ASP.NET 站点地图与爬取程序用来寻找 Web 站点链接的 XML 站点地图不同。ASP.NET 站点地图是导航控件使用的数据源，非常适合 ASP.NET 的一些功能，如数据绑定。客户端不能直接访问这个站点地图。与之相反，XML 站点地图可以从客户端访问，并且它为爬取程序使用了不同的语法，详细介绍请参考 http://www.sitemaps.org。

41.4.2 Menu 控件

Web 站点地图由 Menu 控件使用。为 Menu 控件填充内容，只需添加一个 SiteMapDataSource。

默认情况下，SiteMapDataSource 使用名为 Web.sitemap 的站点地图文件。Menu 控件使用 DataSourceID 属性引用数据源(代码文件 Company.Master)：

```
<div class="left">
  <asp:ContentPlaceHolder ID="leftContent" runat="server">
    <asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1">
    </asp:Menu>
    <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
  </asp:ContentPlaceHolder>
</div>
```

站点地图是基于提供程序的，这与 Membership、Roles 和 Profiles 类似。可以创建自定义提供程序，使其派生自基类 SiteMapProvider。XmlSiteMapProvider 是.NET Framework 中唯一一个具体的站点地图提供程序。XmlSiteMapProvider 派生自基类 StaticSiteMapProvider，后者又派生自 SiteMapProvider。第 39 章详细介绍了 ASP.NET 的基于提供程序的模型。

图 41-7 显示了 Menu 控件。默认情况下，Menu 控件使用 ul、li 和 a 元素创建 HTML 列表内容。其结果是完全可定制的。把 RenderingMode 属性设为 List 或 Table，可分别创建列表或表格元素；菜单项可静态或动态显示；使用 Orientation 属性可以配置横向或纵向显示菜单；还可以设置颜色、样式和 CSS 类。

Home
Meeting Rooms
Accounts

图 41-7

41.4.3 菜单路径

SiteMapPath 控件是一个痕迹导航控件，不只指出了当前位置，还说明了如何到达这个位置：

```
<div class="top2">
  <asp:ContentPlaceHolder ID="topContent" runat="server">
    <asp:SiteMapPath ID="SiteMapPath1" runat="server"></asp:SiteMapPath>
  </asp:ContentPlaceHolder>
</div>
```

图 41-8 显示了 Web 页面 Reserve a Room 上的 SiteMapPath 控件。

Home > Meeting Rooms > Reserve a Room

图 41-8

41.5 验证用户输入

ASP.NET Web Forms 提供了几个验证控件，可以在客户端和服务器端实现验证功能。在客户端进行验证是为了方便用户，由于不需要向服务器发送数据，他们可以更快地知道验证结果。但是，永远不能信任客户端的输入，所以需要在服务器端再次验证数据。

41.5.1 使用验证控件

为了演示验证，本小节将创建一个ValidationDemo.aspx页面。最开始，这个页面包含两个Label控件、两个TextBox控件(名为textName和textEmail)以及一个Button控件。用户必须输入他们的姓名和电子邮件地址，并验证电子邮件地址。可用于这种需求的验证控件包括RequiredFieldValidator和RegularExpressionValidator。

所有验证控件均派生自基类BaseValidator，因此都有属性ControlToValidate，这里需要把它设置为要进行验证的TextBox控件。另外，验证控件还有ErrorMessage属性，用于定义在输入无效时显示的文本。下面的代码段显示了连接到TextBox控件的验证控件，以及设置的ErrorMessage属性。名为TextEmail的TextBox控件有两个验证控件：RequiredFieldValidator和关联的RegularExpression-Validator。对于RegularExpressionValidator，设置了ValidationExpression属性，以验证电子邮件地址是否有效。如图41-9所示的Regular Expression Editor包含一个预定义正则表达式列表，其中有一个就是Internet电子邮件地址。当然，也可以添加自定义的正则表达式。

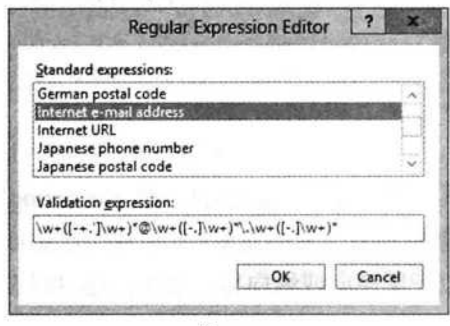


图41-9

```
<table class="auto-style1">
  <tr>
    <td>Name:</td>
    <td>
      <asp:TextBox ID="TextName" runat="server"></asp:TextBox>
      <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
        runat="server" ControlToValidate="TextName"
        ErrorMessage="Name required"></asp:RequiredFieldValidator>
    </td>
  </tr>
  <tr>
    <td>Email:</td>
    <td>
      <asp:TextBox ID="TextEmail" runat="server"></asp:TextBox>
      <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
        runat="server" ControlToValidate="TextEmail" Display="Dynamic"
        ErrorMessage="Email required"></asp:RequiredFieldValidator>
      <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
        runat="server" ControlToValidate="TextEmail" Display="Dynamic"
        ErrorMessage="Please enter an email"
        ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*">
      </asp:RegularExpressionValidator>
    </td>
  </tr>
  <tr>
```

```
      <td>
        <asp:Button ID="Button1" runat="server" Text="Register" />
      </td>
      <td> </td>
    </tr>
  </table>
```

在图 41-10 中，没有输入姓名，电子邮件地址是无效的。因此，应用两个验证控件：TextName 控件的 RequiredFieldValidator 和 TextEmail 控件的 RegularExpressionValidator。来自这两个验证控件的错误消息都显示了出来。有了连接到 TextEmail 控件的 RequiredFieldValidator，Display 属性被设为 Dynamic。只有这样，才可以在 RequiredFieldValidator 的位置显示 RequiredFieldValidator 的错误消息。否则该位置为空。

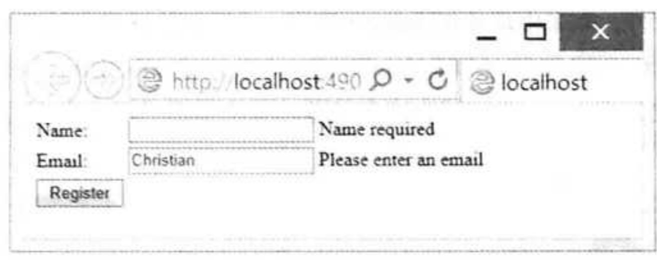


图 41-10

自从 4.5 版本以来，ASP.NET 默认使用隐含的 JavaScript 验证功能。该功能将页面的行为(JavaScript)和(HTML)内容分隔开。ASP.NET Web Forms 通过验证控件使用 jQuery。如果 jQuery 库还没有添加到项目中，就可以通过一个 NuGet 包来添加它。添加了 jQuery 库后，需要给 ScriptManager 添加一个 jquery 定义，这可以在 global.asax.cs 中完成：

```
protected void Application_Start(object sender, EventArgs e)
{
  ScriptManager.ScriptResourceMapping.AddDefinition("jquery",
      new ScriptResourceDefinition
  {
    Path = "~/scripts/jquery-2.0.3.min.js",
    DebugPath = "~/scripts/jquery-2.0.3.js",
    CdnPath = "http://ajax.microsoft.com/ajax/jQuery/jquery-2.0.3.min.js",
    CdnDebugPath = "http://ajax.microsoft.com/ajax/jQuery/jquery-2.0.3.js"
  });
}
```

除了添加 jQuery 之外，还可以在 web.config 文件中添加如下设置，关闭隐含的 JavaScript 验证功能：

```
<add key="ValidationSettings:UnobtrusiveValidationMode" value="None" />
```

41.5.2 使用验证摘要

如果应该将错误消息显示在页面的摘要信息部分，而不是显示在输入控件的旁边，就需要添加一个 ValidationSummary 控件。使用验证控件，Text 属性应设为在出错时显示的字符串。对于示例页面，Text 属性设为*，ErrorMessage 属性将保持前一个代码段中的值。

结果如图 41-11 所示。这里应用了两个 RequiredFieldValidator 验证控件，因此 ErrorMessage 属

性的值显示在 ValidationSummary 控件中。在每个没有通过验证的控件中，都将显示 Text 属性的值。

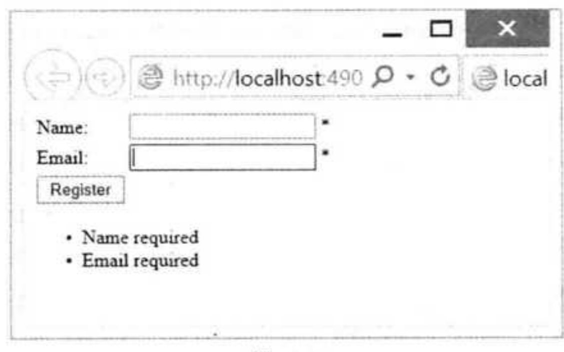


图 41-11

41.5.3 验证组

如果页面包含多个提交类型的按钮，并带有不同的验证区域，就可以指定验证组。如果不使用验证组，那么在每次回送时都会验证所有控件是否正确。在图 41-12 所示的页面中，如果用户单击 Register 按钮，只应该验证属于该部分的控件；如果用户单击 Submit 按钮，只应该验证属于这部分的控件。通过将第一组验证控件以及 Button 控件的 ValidationGroup 属性设为 Register，将第二组验证控件的 ValidationGroup 属性设为 Event，即可解决这个问题。单击属于“Event”验证组的按钮，只会验证同一组的验证控件。

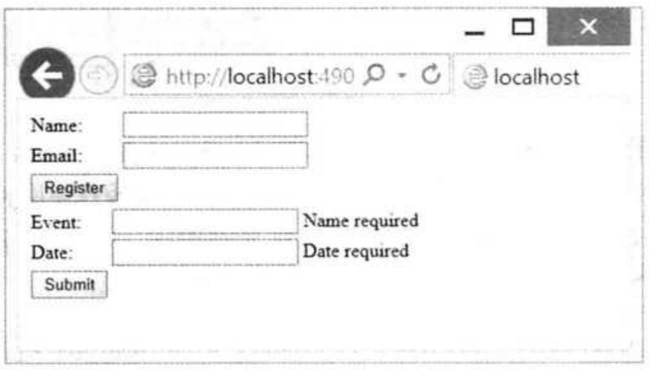


图 41-12

41.6 访问数据

使用 ASP.NET 中的数据控件时，访问数据是十分容易的。不编写任何 C#代码也可以完成许多工作。Data 类别中的控件属于两个不同的组：数据源和 UI 控件。数据源用于访问数据，例如通过使用 ADO.NET Entity Framework 或 DataSet 访问数据。UI 控件则定义不同的外观，如 GridView 和 DetailsView。

表 41-1 描述了数据源控件。

表 41-1

数据源控件	描 述
EntityDataSource	使用 ADO.NET Entity Framework 读写数据。这个控件基于旧的 Entity Framework ObjectContext，而不是 DbContext
LinqDataSource	用于访问 LINQ to SQL。这个映射工具基本上被 ADO.NET Entity Framework 取代
ObjectDataSource	可用于访问带静态或实例成员的自定义对象，以返回对象列表。它提供了添加和更新对象的方法

(续表)

数据源控件	描 述
SiteMapDataSource	读取 Web.sitemap 文件，用于导航。本章前面介绍过这个数据源
SqlDataSource	使用 DataReader 或 DataSet。第 32 章解释了这些类型
XmlDataSource	用于访问文件或 XML 对象中的 XML 数据。第 34 章讨论了 XML 处理

表 41-2 包含了关于 UI 控件的信息。

表 41-2

Web 服务器数据控件	描 述
ListView	一个以网格形式显示数据的强大控件。允许选择、排序、删除、编辑和插入记录。其 UI 可用模板定制
DataList	这个控件可以绑定到数据源，以网格形式显示数据。其 UI 可用模板定制。这个控件也支持选择和编辑数据
DataPager	可以与另一个控件(如 ListView)结合使用，以添加分页功能
GridView	使用表来显示绑定的数据。也可选择、排序和编辑数据
Repeater	与其他 UI 控件不同，Repeater 需要使用模板定制的布局才能得到有用的输出。其他许多控件可以使用模板，但是没有模板也会有输出
DetailsView	显示单条记录，并允许编辑、删除和插入记录
FormView	与 DetailsView 类似，FormView 用来显示单条记录。FormView 要求有用户定义的模板，并且定制程度比 DetailsView 更好

显然，许多控件都可以用来显示网格。多年来，ASP.NET Web Forms 不断添加功能越来越丰富的网格控件。通常，使用 ListView 是一个很好的选择。该控件只在.NET 3.5 后可用，提供了比其他控件更多的功能。GridView 自.NET 2.0 后可用，DataList 自.NET 1.1 后可用。知道有哪些控件可用有助于确定使用哪个控件，以及哪个控件提供了更“现代”的功能。

41.6.1 使用 Object Framework

第 33 章详细讨论过 ADO.NET Entity Framework，不过本小节将介绍如何使用它来定义访问代码。

示例数据库包含两个表：Reservations 和 MeetingRooms。这两个表映射到如图 41-13 所示的 MeetingRoom 和 Reservation 类型。通过添加 ADO.NET Entity Data Model 类型的一个新项，并选择现有的数据库，很容易创建这个模型。数据库可与示例代码一起下载。

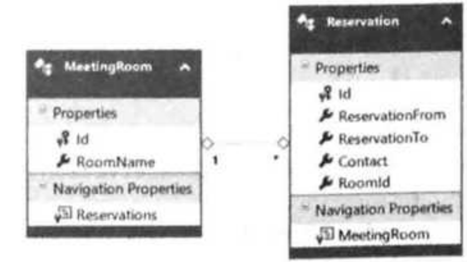


图 41-13

设计器创建了实体类型 MeetingRoom 和 Reservation，它们包含了数据库模式中定义的全部属性。除了实体对象，还创建了对象上下文 RoomReservationEntities。这个上下文定义了每个表的属性，以

返回 ObjectSet<MeetingRoom>和 ObjectSet<Reservation>对象。该上下文类管理到数据库的连接。

41.6.2 创建库

要使用 ObjectDataSource，应创建一个库。RoomReservationRepository 类定义的方法返回所有会议室和客房预定，还定义了添加、更新和删除客房预定的方法：

```
public class RoomReservationRepository
{
  public IEnumerable<MeetingRoom> GetMeetingRooms()
  {
    using (var data = new RoomReservationEntities())
    {
      return data.MeetingRooms.AsNoTracking().ToList();
    }
  }

  public IEnumerable<Reservation> GetReservationsByRoom(int roomId)
  {
    using (var data = new RoomReservationEntities())
    {
      return data.Reservations.AsNoTracking().Where(
        r => r.RoomId == roomId).ToList();
    }
  }

  public void AddReservation(Reservation reservation)
  {
    using (var data = new RoomReservationEntities())
    {
      data.Reservations.Add(reservation);
      data.SaveChanges();
    }
  }

  public void UpdateReservation(Reservation reservation)
  {
    using (var data = new RoomReservationEntities())
    {
      data.Reservations.Attach(reservation);
      data.Entry(reservation).State = EntityState.Modified;
      data.SaveChanges();
    }
  }

  public void DeleteReservation(Reservation reservation)
  {
    using (var data = new RoomReservationEntities())
    {
      data.Reservations.Remove(reservation);
      data.SaveChanges();
    }
  }
}
```

41.6.3 使用 Object Data Source

在第一页中，会议室应该显示在 DropDownList 中，房间的预定情况显示在 GridView 中。

页面ShowReservations.aspx上的前两个控件是DropDownList和Object DataSource控件。使用智能标记，可以把数据源配置为使用类型RoomReservationRepository，该类型访问数据模型。图 41-14 显示了带有Object Data Source的RoomReservationRepository类型的选项，单击Next按钮，会打开如图 41-15 所示的对话框。在这里，单击对话框顶部的选项卡，选择对应的方法，可以选择、插入、更新、删除项。房间只能显示，不能编辑，所以这里只应用了SELECT操作，只选择了GetMeetingRooms方法。

图 41-14

图 41-15

配置数据源后，可以配置 UI 控件来引用数据源控件。为此，可以使用 Data Source Configuration Wizard，如图 41-16 所示。为了在 DropDownList 中显示，使用了 RoomName 属性。为了标识数据字段，使用了 Id 属性。

图 41-16

打开页面后，会显示数据库中的会议室。下一步是根据选择显示会议室的预定情况。为此，在页面中添加一个 GridView 和另一个 ObjectDataSource 控件。第二个数据源控件被配置为访问 RoomReservationRepository，但这次映射了 GetReservationsByRoom、AddReservation、UpdateReservation 和 DeleteReservation 方法。

GetReservationsByRoom 方法需要一个参数 roomId。参数值直接取自 DropDownList 的 SelectedValue 属性，如图 41-17 所示。所选值可以直接在 Expression Editor 中获得。

图 41-17

在示例代码中，参数来自 DropDownList 控件。在 Expression Editor 中，还可以看到参数来源的其他类型：Cookie、Form、Profile、QueryString、Session 和 RouteData。仅配置一些属性，就可以实现许多功能。对于 GridView 控件，只需要把 DataSourceId 设为第二个 EntityDataSource 控件。

到目前为止，代码隐藏文件还是空的，DropDownList、EntityDataSource 和 GridView 控件的配置如下：

```
<asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack="True"
  DataSourceID="ObjectDataSource1" DataTextField="RoomName"
  DataValueField="Id" Height="22px" Width="150px">
</asp:DropDownList>
<br />
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
  SelectMethod="GetMeetingRooms"
  TypeName="ProCSharpSample.DataAccess.RoomReservationRepository"
</asp:ObjectDataSource>
<br />
<asp:GridView ID="GridView1" runat="server"
  AutoGenerateColumns="False"
  DataSourceID="ObjectDataSource2">
  <Columns>
    <asp:BoundField DataField="Id" HeaderText="Id"
      SortExpression="Id" />
    <asp:BoundField DataField="ReservationFrom"
      HeaderText="ReservationFrom" SortExpression="ReservationFrom" />
    <asp:BoundField DataField="ReservationTo"
      HeaderText="ReservationTo" SortExpression="ReservationTo" />
    <asp:BoundField DataField="Contact" HeaderText="Contact"
      SortExpression="Contact" />
    <asp:BoundField DataField="RoomId" HeaderText="RoomId"
      SortExpression="RoomId" />
  </Columns>
</asp:GridView>
<br />
<br />
<asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
  DataObjectTypeName="ProCSharp.Models.Reservation"
  TypeName="ProCSharpSample.DataAccess.RoomReservationRepository"
  SelectMethod="GetReservationsByRoom"
  InsertMethod="AddReservation"
  DeleteMethod="DeleteReservation"
  UpdateMethod="UpdateReservation">
  <SelectParameters>
    <asp:ControlParameter ControlID="DropDownList1" Name="roomId"
      PropertyName="SelectedValue" Type="Int32" />
  </SelectParameters>
</asp:ObjectDataSource>
```

现在可在浏览器中打开该页面，以选择房间预定情况，如图 41-18 所示。

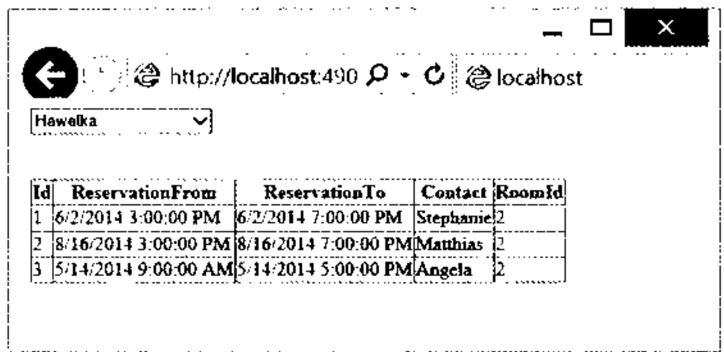


图 41-18

41.6.4 编辑

GridView 本身支持分页、编辑、删除和选择操作，但这取决于数据源的功能。例如，如果给 ObjectDataSource 提供了插入、更新和删除操作，就可以通过 GridView 使用对应的命令。选择该数据源后，就可以在 GridView 中执行编辑和删除操作，如图 41-19 所示。

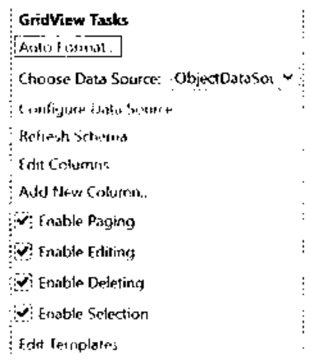


图 41-19

使用 GridView 设置这些选项将设置 AllowPaging 属性，并添加一个带有 Delete、Edit 和 Select 按钮的 CommandField 列：

```
<asp:GridView ID="GridView1" runat="server"
  AutoGenerateColumns="False"
  AllowPaging="True"
  DataSourceID="ObjectDataSource2">
  <Columns>
    <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
      ShowSelectButton="True" />
    <asp:BoundField DataField="Id" HeaderText="Id"
      SortExpression="Id" />
    <asp:BoundField DataField="ReservationFrom"
      HeaderText="ReservationFrom" SortExpression="ReservationFrom" />
    <asp:BoundField DataField="ReservationTo"
      HeaderText="ReservationTo" SortExpression="ReservationTo" />
    <asp:BoundField DataField="Contact" HeaderText="Contact"
      SortExpression="Contact" />
    <asp:BoundField DataField="RoomId" HeaderText="RoomId"
      SortExpression="RoomId" />
  </Columns>
</asp:GridView>
```

GridView 在列标题中使用属性的名称(Id、ReservedFrom、ReservedTo、Contact)，在读模式中使用 Label 控件，在编辑模式中使用 TextBox 控件，如图 41-20 所示。单击 Edit 按钮后，行将进入编辑模式。在编辑模式下，TextBox 控件与数据绑定在一起。修改该控件中的值也将修改 Entity Framework 对象上下文中的实体。单击 Update 将调用对象上下文的 SaveChanges 方法，并把更改写入数据库。

	Id	ReservationFrom	ReservationTo	Contact	RoomId
Edit Delete Select	1	6/2/2014 3:00:00 PM	6/2/2014 7:00:00 PM	Stephanie	2
Update Cancel	2	8/16/2014 3:00:00 PM	8/16/2014 7:00:00 PM	Matthias	2
Edit Delete Select	3	5/14/2014 9:00:00 AM	5/14/2014 5:00:00 PM	Angela	2

图 41-20

41.6.5 定制列

默认情况下设计器将创建所有属性的绑定字段，可以修改所有这些字段。至少应该修改列标题，在编辑模式下 Id 列应保持只读。在如图 41-21 所示的 Fields 编辑器中很容易完成这种定制。

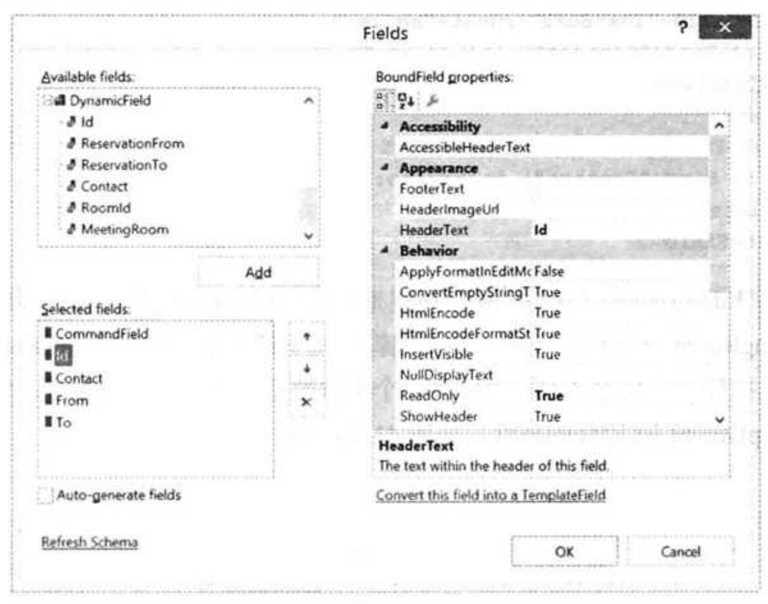


图 41-21

可以选择应该显示的字段。可以配置这些字段的许多方面，包括标题信息和标题图像、页脚文本、null 值的默认显示、格式字符串，以及控件、页眉、页脚和项元素的样式。GridView 的内容现在变为 BoundField 对象，如下面的代码段所示：

```
<asp:GridView ID="GridView1" runat="server" DataSourceID="ObjectDataSource2"
  AllowPaging="True" AutoGenerateColumns="False">
  <Columns>
    <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
      ShowSelectButton="True" />
    <asp:BoundField DataField="Id" HeaderText="Id" ReadOnly="True"
      SortExpression="Id" />
    <asp:BoundField DataField="Contact" HeaderText="Contact"
      SortExpression="Contact" />
    <asp:BoundField DataField="ReservedFrom" HeaderText="From"
      SortExpression="ReservedFrom" />
```

```
    <asp:BoundField DataField="ReservedTo" HeaderText="To"
      SortExpression="ReservedTo" />
  </Columns>
</asp:GridView>
```

41.6.6　在网格中使用模板

为了进一步定制，可以使用模板。在 Fields 编辑器中，可以把列转换为模板。在将 Contact 字段改为模板时，使用了 TemplateField 而不是 BoundField，如下面的代码段所示。使用模板字段时，可以创建不同的用户界面，例如 ItemTemplate、EditItemTemplate、AlternatingItemTemplate、HeaderTemplate、FooterTemplate 等。

默认的 ItemTemplate 使用 Label，EditItemTemplate 使用 TextBox。在这两个模板中，Text 属性绑定到被绑定对象的 Contact 属性：

```
<asp:TemplateField HeaderText="Contact" SortExpression="Contact">
  <EditItemTemplate>
    <asp:TextBox ID="TextBox2" runat="server"
      Text='<%# Bind("Contact") %>'></asp:TextBox>
  </EditItemTemplate>
  <ItemTemplate>
    <asp:Label ID="Label2" runat="server"
      Text='<%# Bind("Contact") %>'></asp:Label>
  </ItemTemplate>
</asp:TemplateField>
```

使用 Template Editor(如图 41-22 所示)，可以把任意 ASP.NET 控件添加到不同的模板模式。这里的示例为 MeetingRoom 列添加了模板，并在编辑模式下把 TextBox 替换为 DropDownList 控件。在 DropDownList 控件中，将返回会议室的第一个实体数据源控件选作数据源。使用 DataBindings 编辑器可绑定 DropDownList 控件的属性，如图 41-23 所示。

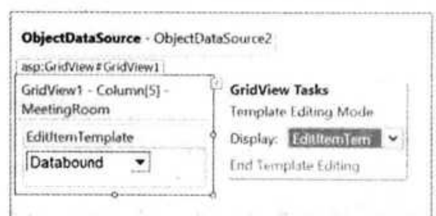


图 41-22

图 41-23

得到的 ASPX 代码现在在 EditItemTemplate 中使用 DropDownList，在 ItemTemplate 中使用 Label：

```
<asp:TemplateField HeaderText="MeetingRoom"
  ConvertEmptyStringToNull="False" SortExpression="MeetingRoom">
  <EditItemTemplate>
    <asp:DropDownList ID="DropDownList2" runat="server"
      DataTextField="RoomName" DataValueField="Id"
      Width="146px" Height="16px"
      SelectedValue='<%# Bind("RoomId") %>'
      DataSourceID="ObjectDataSource1">
    </asp:DropDownList>
  </EditItemTemplate>
  <ItemTemplate>
    <asp:Label ID="Label1" runat="server" Text='<%#
      Bind("MeetingRoom.RoomName") %>'></asp:Label>
  </ItemTemplate>
</asp:TemplateField>
```

图 41-24 显示了打开的页面，其中 DropDownList 控件处于编辑模式。会议室从第一个数据源检索，而表格本身则绑定到第二个数据源。

图 41-24

41.7 安全性

ASP.NET Web 应用程序的另一个重要方面是使用身份验证和授权。在.NET 4.5.1 中，ASP.NET 使用身份验证和授权的一个新架构。以前使用的是成员 API(Membership API)，但成员 API 不用于基于声明的身份验证。新的 ASP.NET 身份架构很灵活，还支持通过 Twitter、Facebook 和 Microsoft 账户进行身份验证。

第 41 章介绍了 ASP.NET 身份。本节介绍如何在空 Web Forms 项目中添加身份验证和授权。

41.7.1 建立 ASP.NET 身份

应用程序示例使用的空模板不允许选择身份验证选项。但以后很容易改变它。为了让用户自己存储用户数据，只需添加 NuGet 包 Microsoft ASP.NET Identity EntityFramework。这个包也安装了

其依赖包 Microsoft ASP.NET Identity Core。

使用 ASP.NET 身份架构创建用户时，会自动创建一个数据库。要定义数据库名，需要把下面的连接字符串添加到 web.config 文件中：

```
<connectionStrings>
  <add name="DefaultConnection" connectionString=
    "Data Source=(LocalDb)\v11.0;AttachDbFilename=
    |DataDirectory|\Authentication.mdf;Initial Catalog=
    Authentication;Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

41.7.2 用户注册

为了允许用户注册，创建一个 Web Form，要求用户提供用户名和密码。Register 按钮定义了提交表单时调用的 OnRegister 处理方法：

```
<form id="form1" runat="server">
  <div>
    <asp:Label Text="Username:" AssociatedControlID="textUsername"
      runat="server" />
    <asp:TextBox ID="textUsername" runat="server" /><br />
    <asp:Label Text="Password:" AssociatedControlID="textPassword"
      runat="server" />
    <asp:TextBox TextMode="Password" ID="textPassword" runat="server" />
    <br />
    <asp:Button Text="Register" OnClick="OnRegister" runat="server" />
    <br />
  </div>
  <asp:Label ID="StatusText" runat="server"></asp:Label>
</form>
```

OnRegister 方法首先用泛型参数 IdentityUser 创建一个新对象 UserStore。UserStore 类封装了一个 DbContext，用于创建和连接数据库。所存储的数据由 IdentityUser 类型定义。UserManager 类可以用于创建和更新用户信息。Create 方法创建一个新用户。示例代码传递用户名和密码：

```
protected void OnRegister(object sender, EventArgs e)
{
  string username = this.textUsername.Text;
  string password = this.textPassword.Text;

  var userStore = new UserStore<IdentityUser>();
  var manager = new UserManager<IdentityUser>(userStore);

  var user = new IdentityUser() { UserName = username };
  IdentityResult result = manager.Create(user, password);

  if (result.Succeeded)
  {
    StatusText.Text = string.Format(
      "User {0} was created successfully!", user.UserName);
  }
  else
```

```
    {
      StatusText.Text = result.Errors.FirstOrDefault();
    }
  }
```

41.7.3 用户的身份验证

前面仅把用户信息添加到数据库中。要验证用户的身份，需要在项目中再添加两个 NuGet 包。新的身份验证代码使用 Open Web Interface for .NET (OWIN)。为了管理和配置 OWIN 身份验证代码，可以通过 Microsoft ASP.NET Identity Owin 包使用扩展类。为了获得 IIS 管道的启动代码，需要 Microsoft.Owin.Host.SystemWeb 包。

给项目安装这些包后，可以使用 Project | Add New Item 选择 OWIN Startup Class，以配置身份验证的启动。接着可以通过 IAppBuilder 调用 UseCookieAuthentication 方法(代码文件 ProCSharpSample/Startup/AuthenticationStartup.cs)：

```
[assembly: OwinStartup(typeof(
  ProCSharpSample.Startup.AuthenticationStartup))]

  public class AuthenticationStartup
  {
    public void Configuration(IAppBuilder app)
    {
      app.UseCookieAuthentication(new CookieAuthenticationOptions
      {
        AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Account/Login.aspx")
      });
    }
  }
```

为了登录，创建另一个 Web 表单 Login.aspx 。这个表单允许输入用户名和密码：

```
<form id="form1" runat="server">
  <div>
    <asp:Label Text="Username:" AssociatedControlID="textUsername"
      runat="server" />
    <asp:TextBox ID="textUsername" runat="server" /><br />
    <asp:Label Text="Password:" AssociatedControlID="textPassword"
      runat="server" />
    <asp:TextBox TextMode="Password" ID="textPassword" runat="server" />
    <br />
    <asp:Button Text="Login" OnClick="OnLogin" runat="server" />
    <br />
  </div>
  <asp:Label ID="StatusText" runat="server"></asp:Label>
</form>
```

在 OnLogin 处理方法中，使用 UserManager 的 Find 方法可以验证用户名和密码。如果用户名和密码都匹配，就返回一个 IdentityUser，否则返回 null。有了这个用户，就创建一个 ClaimsIdentity。用户登录所需的身份验证管理器使用 GetOwinContext 扩展方法从 HTTP 上下文中检索。这个方法在 Microsoft.Owin.Host.SystemWeb 程序集的 System.Web 名称空间中定义。检索了实现

IAuthenticationManager接口的身份验证管理器后，就调用SignIn方法，使用户登录。这个方法为用户设置了身份验证cookie：

```
protected void OnLogin(object sender, EventArgs e)
{
  string username = this.textUsername.Text;
  string password = this.textPassword.Text;

  var userStore = new UserStore<IdentityUser>();
  var userManager = new UserManager<IdentityUser>(userStore);
  IdentityUser user = userManager.Find(username,password);

  if (user != null)
  {
    ClaimsIdentity userIdentity = userManager.CreateIdentity(user,
      DefaultAuthenticationTypes.ApplicationCookie);

    IAuthenticationManager authenticationManager =
      HttpContext.Current.GetOwinContext().Authentication;

    authenticationManager.SignIn(new AuthenticationProperties()
      { IsPersistent = false }, userIdentity);
    Response.Redirect(Request.QueryString["ReturnUrl"]);
  }
  else
  {
    StatusText.Text = "Invalid username or password.";
  }
}
```

41.7.4 用户授权

验证了用户的身份后，就可以使用授权功能定义该用户允许访问什么页面。这可以通过编程指定，也可以使用配置完成。

如果使用web.config文件，在授权部分，可以定义哪些用户或用户角色允许访问。示例配置定义了用户?的deny访问。这个设置拒绝所有未授权用户的访问。除了拒绝访问之外，还可以通过allow元素允许访问，定义一组允许访问的用户：

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>
</configuration>
```

如果在子目录中使用 web.config文件，根 web.config文件的授权部分就会被覆盖。这样，就可以根据目录指定不同的授权规则。也可以根据文件定义授权规则。为此，可以用location元素的path特性定义文件，在对应的位置上定义system.web配置。

以编程方式检查访问规则，可以使用User.Identity.IsAuthenticated属性在HTTP上下文中检查用

户是否通过了身份验证。User.Identity.GetUserName()返回用户名。

41.8 Ajax

Ajax 允许通过异步回发和动态的客户端 Web 页面操作，改进 Web 应用程序的用户界面。术语“Ajax”由 Jesse James Garrett 提出，是 Asynchronous JavaScript and XML 的缩写。

注意，Ajax 不是一个缩写词，因此它不能写作 AJAX。但是，在产品 ASP.NET AJAX 中它是大写，这是 Ajax 的 Microsoft 实现方式，如 41.9 节所述。

根据定义，Ajax 涉及 JavaScript 和 XML。但是，Ajax 编程也需要使用其他技术，如 HTML、CSS 和文档对象模型(DOM)。目前，XML 并不总是与 Ajax 一起用于在客户端和服务器之间传输数据，而使用 JSON (JavaScript Object Notation)来替代，JSON 的系统开销少于 XML。

当然，Ajax 最重要的部分是 XmlHttpRequest。自 Internet Explorer 5 以来，IE 浏览器就把 XmlHttpRequest API 作为一种在客户端和服务器之间执行异步通信的方式。Microsoft 最初引入它，是作为一种技术，在 Outlook Web Access 产品中访问通过 Internet 存储在 Exchange 服务器中的电子邮件。后来它变成在 Web 应用程序中进行异步通信的标准方式，是支持 Ajax 的 Web 应用程序的一个核心技术。这个 API 的 Microsoft 实现方式称为 XMLHTTP，它利用 XMLHTTP 协议来通信。

Ajax 还需要用服务器端代码处理部分页面的回发和完整页面的回发。这包括服务器控件的事件处理程序和 Web 服务。图 41-25 显示了这些技术如何在 Ajax Web 浏览器模型中联合使用，并与传统的 Web 浏览器模型进行比较。

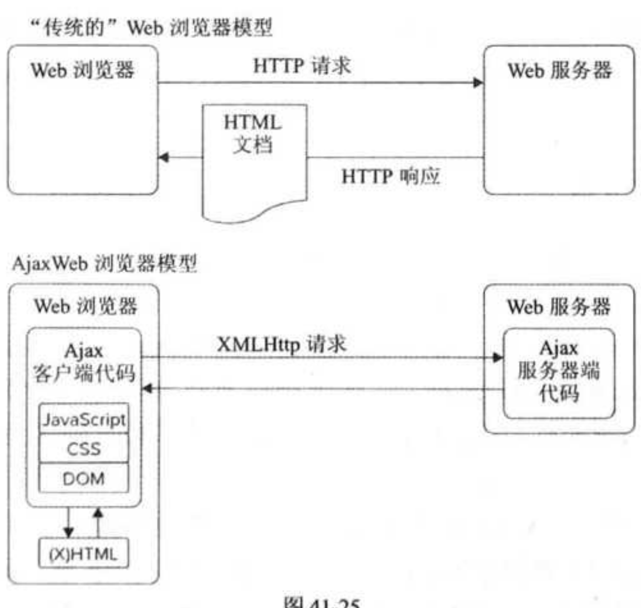


图 41-25

在 Ajax 推出之前，图中列出的前 4 种技术(HTML、CSS、DOM 和 JavaScript)用于创建动态 HTML(DHTML) Web 应用程序。这些应用程序比较有名出自两个原因：它们提供的用户界面更好；

它们一般只能用于一种 Web 浏览器。

自 DHTML 推出以来，标准已有了改进，Web 浏览器遵循的标准级别也提高了。但是，它们仍有区别，Ajax 解决方案必须考虑这些区别。也就是说，大多数开发人员实现 Ajax 解决方案还相当慢。只有开发出更抽象的 Ajax 架构(如 ASP.NET AJAX)，创建支持 Ajax 的 Web 站点才能成为企业级开发的一个可行选项。

41.8.1 ASP.NET AJAX 的概念

ASP.NET AJAX 是 Ajax 架构的 Microsoft 实现，它专门面向 ASP.NET Web Forms 开发人员。它是 ASP.NET Web Forms 核心功能的一部分。专用于 ASP.NET AJAX 的 Web 站点是 http://ajax.asp.net，它包含相关的文档、论坛和示例代码，可以用于所使用的任何 ASP.NET 版本。

ASP.NET AJAX 提供了如下功能：

- 服务器端架构允许 ASP.NET Web 页面响应部分页面的回发操作。
- ASP.NET 服务器控件便于实现 Ajax 功能。
- HTTP 处理程序允许 ASP.NET Web 服务在部分页面的回发操作中，使用 JSON(JavaScript Object Notation，JavaScript 对象标记)串行化功能与客户端代码通信。
- Web 服务支持客户端代码访问 ASP.NET 应用程序服务，包括身份验证和个性化服务。
- Web 站点模板可用于创建支持 ASP.NET AJAX 的 Web 应用程序。
- 客户端的 JavaScript 库对 JavaScript 语法提供了许多改进，还提供了许多代码，来简化 Ajax 功能的实现。

这些服务器控件和服务器端的架构统称为 ASP.NET 扩展(ASP.NET Extension)。ASP.NET AJAX 的客户端部分称为 AJAX 库(AJAX Library)。ASP.NET AJAX Control Toolkit 没有包含在 Visual Studio 2012 安装文件中，这个工具集可以从 NuGet 包中安装。在编写本书时，其最新版本是 4.1.60623，它还带有示例。这个工具集包含许多可用作共享源的服务器控件。

这些下载软件包提供了功能丰富的架构，该架构可以用于在自己的 ASP.NET Web 应用程序中添加 Ajax 功能。下面将介绍 ASP.NET AJAX 的各个组成部分。ASP.NET AJAX 的核心功能分为两部分：AJAX 扩展和 AJAX 库。

1. ASP.NET AJAX 扩展

ASP.NET AJAX 功能包含在 GAC 中安装的两个程序集中：

- System.Web.Extensions.dll——这个程序集包含 ASP.NET AJAX 功能，其中包括 AJAX 扩展和 AJAX 库 JavaScript 文件，它们可以通过 ScriptManager 组件(稍后介绍)来获得。
- System.Web.Extensions.Design.dll——这个程序集包含用于 AJAX 扩展服务器控件的 ASP.NET Designer 组件。它通过 ASP.NET Designer 在 Visual Studio 或 Visual Web Developer 中使用。

ASP.NET AJAX 中的许多 AJAX 扩展组件都涉及支持部分页面的回发和用于 Web 服务的 JSON 串行化。这包括各种 HTTP 处理程序组件和对已有 ASP.NET 架构的扩展。所有这些功能都可以通过站点的 web.config 文件来配置。还有用于其他配置的类和属性。但大多数配置都是透明的，用户很少需要改变默认设置。

与 AJAX 扩展的主要交互操作是使用服务器控件将 Ajax 功能添加到 Web 应用程序中。有几个

服务器控件可以用各种方式增强用户的应用程序。表 41-3 列出了一些服务器端组件。本章后面将介绍它们。

表 41-3

控　件	说　明
ScriptManager	这个控件是 ASP.NET AJAX 功能的核心，使用部分页面回发功能的每个页面都需要它。它的主要作用是管理对 AJAX 库 JavaScript 文件的客户端引用，AJAX 库 JavaScript 文件位于 ASP.NET AJAX 程序集中。AJAX 库主要由 AJAX 扩展服务器控件使用，这些控件会生成自己的客户端代码。这个控件还负责配置要在客户端代码访问的 Web 服务。给 ScriptManager 控件提供 Web 服务信息，就可以生成客户端类和服务器端类，来透明地管理与 Web 服务的异步通信。还可以使用 ScriptManager 控件维护对自己的 JavaScript 文件的引用
UpdatePanel	UpdatePanel 控件非常有用，也许是最常用的 ASP.NET 控件。这个控件与标准的 ASP.NET 占位符类似，可以包含任何其他控件。更重要的是，在部分页面的回发过程中，它还把页面的一部分标记为可以独立于页面的其他部分来更新的区域。 UpdatePanel 控件包含的、引发回送的任意控件(如 Button 控件)，都不会引发整个页面的回发操作，它们只引发部分页面的回发，只更新 UpdatePanel 控件的内容。 在许多情况下，只需要这个控件就可以实现 Ajax 功能。例如，可以把一个 GridView 控件放在 UpdatePanel 控件中，该控件的分页、排序和其他回发功能都在部分页面的回发过程中发挥作用
UpdateProgress	在部分页面的回发过程中，这个控件可以向用户提供反馈。在更新 UpdatePanel 时，可以为要显示的 UpdateProgress 控件提供一个模板。例如，可以使用悬浮的<div>控件显示一条消息如“Updating...”，以便告诉用户应用程序正在忙。注意部分页面的回发不会干扰 Web 页面的其他区域，其他区域仍可以响应
Timer	ASP.NET AJAX 的 Timer 控件是使 UpdatePanel 控件定期更新的一种有效方式。可以配置该控件，以定期触发回发操作。如果 Timer 控件包含在 UpdatePanel 控件中，则每次触发 Timer 控件时，都会更新该 UpdatePanel 控件。Timer 控件也有关联的事件，以便用户可以执行定期的服务器端处理
AsyncPostBackTrigger	这个控件可以在未包含在 UpdatePanel 控件中的控件里触发 UpdatePanel 控件的更新操作。例如，可以在 Web 页面的其他地方放置一个下拉列表，来更新包含 GridView 控件的 UpdatePanel 控件

AJAX 扩展还包含 ExtenderControl 抽象基类，来扩展已有的 ASP.NET 服务器控件。它由 ASP.NET AJAX Control Toolkit 中的各种类使用，如后面所述。

2. AJAX 库

在支持 ASP.NET AJAX 的 Web 应用程序中，AJAX 库包含的 JavaScript 文件由客户端代码使用。在这些 JavaScript 文件中包含许多功能，其中一些是增强 JavaScript 语言的通用代码，一些则专用于 Ajax 功能。AJAX 库包含的功能层彼此互为基础，如表 41-4 所示。

表 41-4

功 能 层	说 明
浏览器兼容性	AJAX 库的最底层代码根据客户机的 Web 浏览器来映射各种 JavaScript 功能。这是必需的，因为 JavaScript 在不同浏览器中的实现方式是有区别的。提供这个功能层，其他层的 JavaScript 代码就不必考虑浏览器的兼容性问题了，也可以编写独立于浏览器、在所有客户机环境中工作的代码
核心服务	这一层包含对 JavaScript 语言的增强，尤其是 OOP 功能。使用这一层的代码，可以使用 JavaScript 脚本文件定义名称空间、类、派生类和接口。C#开发人员对此特别感兴趣，因为它使 JavaScript 代码的编写非常类似于用 C#编写.NET 代码，且鼓励代码的重用
基类库	客户基类库(BCL)包含许多 JavaScript 类，它们为 AJAX 库层次结构中下层的类提供了底层功能。这些类中的大多数都不能直接使用
网络	网络层上的类允许客户端代码异步地调用服务器端代码。这一层包含的基本架构可以调用 URL，响应回调函数的结果。在大多数情况下，这些功能都不能直接使用，而应使用封装了该功能的类。这一层还包含用于 JSON 序列化和反序列化的类，大多数网络类都在客户端的 Sys.Net 名称空间中
用户界面	这一层包含的类提取用户界面元素，如 HTML 元素和 DOM 事件。可以使用这一层的方法和属性编写中性语言的 JavaScript 代码，来操作客户端上的 Web 页面。用户界面类包含在 Sys.UI 名称空间中
控件	AJAX 库的最后一层包含最高级的代码，它们提供 Ajax 行为和服务器控件功能。这包括可以使用的动态生成代码，例如，用于从客户端的 JavaScript 代码中调用 Web 服务

AJAX 库可以用于扩展和定制支持 ASP.NET AJAX 的 Web 应用程序的行为，但注意不一定要这么做。要想在应用程序中不使用任何附加的 JavaScript，还有很长的路要走，只有需要更高级的功能，才需要这么做。如果要编写附加的客户端代码，那么使用 AJAX 库提供的功能会比较容易完成任务。

3. ASP.NET AJAX Control Toolkit

AJAX Control Toolkit 是附加服务器控件的一个集合，包括由 ASP.NET AJAX 社区编写的扩展控件。扩展控件允许在已有的 ASP.NET 服务器控件中添加功能，一般把它关联到一个客户端行为。例如，AJAX Control Toolkit 中的一个扩展程序能在 TextBox 中放置“watermark”文本，以扩展 TextBox 控件，当用户还没有在文本框中添加任何内容时，就会显示该文本。这个扩展控件在服务器控件 TextBoxWatermark 中实现。

使用 AJAX Control Toolkit 可以给站点添加许多功能，它们超出了核心下载包的范围。这些控件可以使浏览操作更有趣，也许能为增强 Web 应用程序提供许多新思路。但是，因为 AJAX Control Toolkit 独立于核心下载包，所以这些控件并没有获得与核心下载包中的控件相同级别的支持。

41.8.2 ASP.NET AJAX 网站示例

前面介绍了 ASP.NET AJAX 的组件部分，下面就开始探讨如何使用它们增强网站。本小节将讨论使用 ASP.NET AJAX 的 Web 应用程序如何工作，以及如何使用 ASP.NET AJAX 中的各种功能。首先仔细研究一个简单的应用程序，然后在后续的章节中添加其他功能。

ASP.NET Web Site 模板包含了 ASP.NET AJAX 的所有核心功能。也可以使用 AJAX Control Toolkit Web Site 模板(在安装它之后)，以包含 AJAX Control Toolkit 中的控件。针对本示例的目的，创建一个新的 ASP.NET Empty Web Application 模板 ProCSharpAjaxSample。

添加一个 Web 窗体 Default.aspx，并修改其代码，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
  Inherits="ProCSharpAjaxSample.Default" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Professional C# ASP.NET AJAX Sample</title>
</head>
<body>
  <form id="form1" runat="server">
  <asp:ScriptManager ID="ScriptManager1" runat="server" />
  <div>
    <h1>Professional C# ASP.NET AJAX Sample</h1>
    This sample obtains a list of primes up to a maximum value.
      <br />
    Maximum:
      <asp:TextBox runat="server" ID="MaxValue" Text="2500" />
    <br />
    Result:
      <asp:UpdatePanel runat="server" ID="ResultPanel">
        <ContentTemplate>
          <asp:Button runat="server" ID="GoButton" Text="Calculate" />
          <br />
          <asp:Label runat="server" ID="ResultLabel" />
          <br />
          <small>Panel render time: <%= DateTime.Now.ToLongTimeString() %>
          </small>
        </ContentTemplate>
      </asp:UpdatePanel>
    <asp:UpdateProgress runat="server" ID="UpdateProgress1">
      <ProgressTemplate>
        <div style="position: absolute; left: 100px; top: 200px;
          padding: 40px 60px 40px 60px; background-color: lightyellow;
          border: black 1px solid; font-weight: bold; font-size: larger;
          filter: alpha(opacity=80);">
          Updating.
        </div>
      </ProgressTemplate>
    </asp:UpdateProgress>
    <small>Page render time: <%= DateTime.Now.ToLongTimeString() %></small>
  </div>
  </form>
</body>
</html>
```

切换到设计视图(注意 ASP.NET AJAX 控件(如 UpdatePanel 和 UpdateProgress)有可视化的设计组件)，双击 Calculate 按钮，添加一个事件处理程序。修改代码，如下所示(代码文件 ProCSharpAjaxSample/Default.aspx.cs)：

```
protected void GoButton_Click(object sender, EventArgs e)
{
  int maxValue = 0;
  var resultText = new StringBuilder();
  if (int.TryParse(MaxValue.Text, out maxValue))
  {
    for (int trial = 2; trial <= maxValue; trial++)
    {
      bool isPrime = true;
      for (int divisor = 2; divisor <= Math.Sqrt(trial); divisor++)
      {
        if (trial % divisor == 0)
        {
          isPrime = false;
          break;
        }
      }
      if (isPrime)
      {
        resultText.AppendFormat("{0} ", trial);
      }
    }
  }
  else
  {
  resultText.Append("Unable to parse maximum value.");
  }
  ResultLabel.Text = resultText.ToString();
 }
}
```

保存修改的内容，按F5键，运行项目。如果有提示，就在web.config文件中启动调试功能。在显示如图41-26所示的Web页面时，注意显示的两个时间相同。

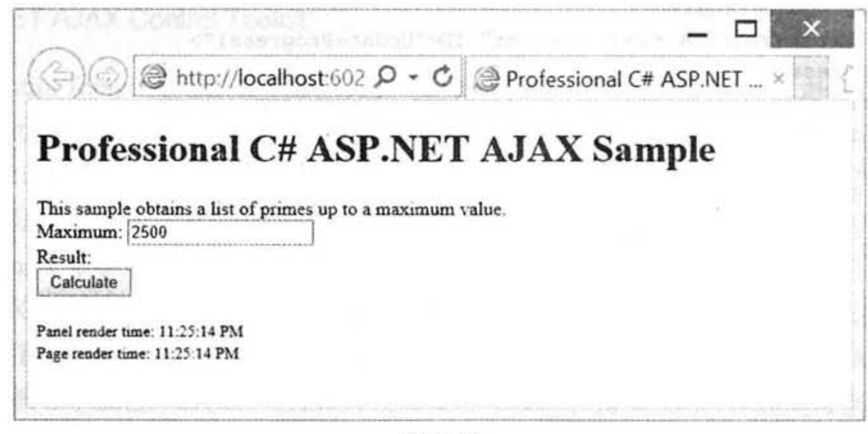


图41-26

单击Calculate按钮，显示小于等于2500的质数。除非在较慢的计算机上运行，否则应立即得到结果。注意显示的时间现在已经不同了，只有UpdatePanel控件中显示的时间改变了。

最后，在最大值中添加一些0，引入一个处理延迟(在较快的PC上添加3个0就足够了)，再次单击Calculate按钮。这次在显示结果之前，注意UpdateProgress控件显示一条部分透明的反馈消息，

如图 41-27 所示。

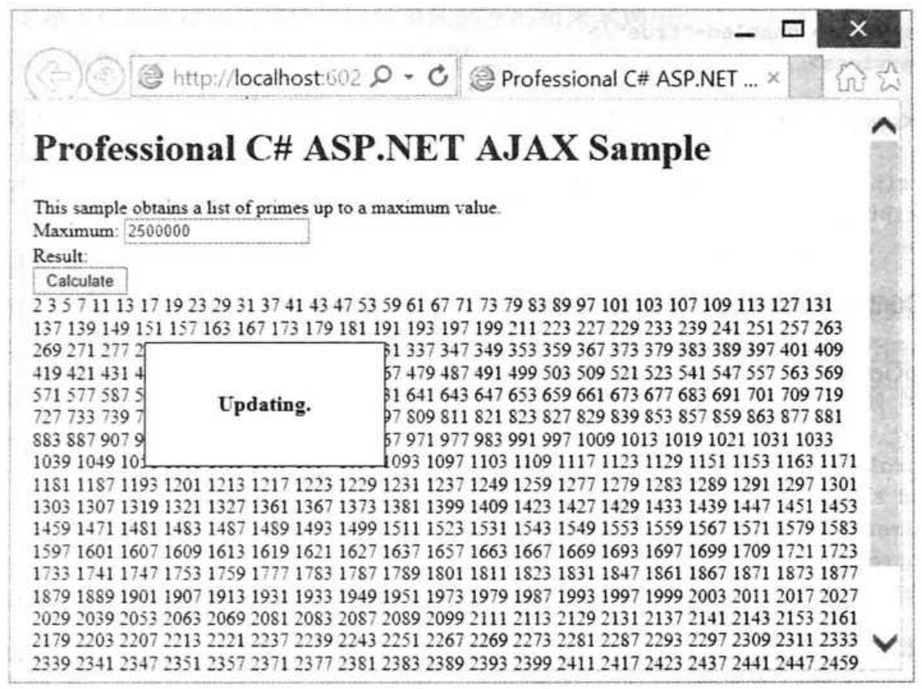


图 41-27

更新应用程序时，页面仍可以响应。例如，可以滚动页面。

更新完成时，把浏览器的滚动位置设置为单击 Calculate 按钮之前的地方。在大多数情况下，部分页面的更新会很快执行完，这非常有利于可用性。

关闭浏览器返回 Visual Studio。

41.8.3 支持 ASP.NET AJAX 的网站配置

ASP.NET AJAX 需要的大多数配置都是默认提供的，如果要修改这些默认值，只需把它们添加到 web.config 文件中。例如，可以添加一个<system.web.extensions>段，来提供额外的配置。可以用这个配置段添加的大多数配置都与 Web 服务相关，并包含在<webServices>元素中，该元素又放在<scripting>元素中。首先，可以添加一段，通过 Web 服务访问 ASP.NET 身份验证服务(这里也可以选择强制 SSL)：

```
<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true" requireSSL="true"/>
```

接下来，通过配置文件 Web 服务，启用和配置对 ASP.NET 个性化功能的访问。

```
<profileService enabled="true"
  readAccessProperties="propertyname1,propertyname2"
  writeAccessProperties="propertyname1,propertyname2" />
```

最后一个与Web服务相关的设置是通过角色Web服务启用和配置对ASP.NET角色功能的访问。

```
  <roleService enabled="true"/>
</webServices>
```

最后，<system.web.extensions>段包含一个元素，该元素允许配置异步通信的压缩和缓存：

```
    <scriptResourceHandler enableCompression="true" enableCaching="true" />
  </scripting>
</system.web.extensions>
```

AJAX Control Toolkit 的其他配置

使用 NuGet 安装 AJAX Control Toolkit，可以在 web.config 文件中添加如下配置：

```
<pages>
  <controls>
    <add tagPrefix="ajaxToolkit" assembly="AjaxControlToolkit"
      namespace="AjaxControlToolkit" />
  </controls>
</pages>
```

这将工具包控件映射到 ajaxToolkit 标记的前缀。这些控件包含在 AjaxControlToolkit.dll 程序集中，该程序集应在 Web 应用程序的/bin 目录下。

还可以使用<%@ Register %>指令单独地在 Web 页面上注册控件。

```
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
  TagPrefix="ajaxToolkit" %>
```

41.8.4 添加 ASP.NET AJAX 功能

在网站上添加 Ajax 功能的第一步是在 Web 页面上添加一个 ScriptManager 控件。之后，添加服务器控件(如 UpdatePanel 控件)，以启用部分页面的呈现功能，再添加 AJAX Control Toolkit 中的动态控件，给应用程序增加可用性和功能。还可以添加客户端代码，使用 AJAX 库有助于进一步定制和增强应用程序的功能。本小节介绍可以使用服务器控件添加的功能。本章后面将讨论客户端技术。

1. ScriptManager 控件

如本章前面所述，ScriptManager 控件必须包含在使用部分页面回发和其他几个 ASP.NET AJAX 功能的所有页面上。

为了确保在 Web 应用程序中的所有页面都包含 ScriptManager 控件，必须将这个控件添加到应用程序使用的母版页中。

除了启用 ASP.NET AJAX 功能之外，还可以使用属性配置这个控件。在这些属性中，最简单的是 EnablePartialRendering 属性，其默认值是 true。如果把这个属性设置为 false，就禁用了所有异步回发处理功能，例如，UpdatePanel 控件提供的页面回发功能。如果要给经理做一个演示，比较支持 AJAX 的网站和传统的网站，就可以这么做。

使用 ScriptManager 控件有几个原因，例如，下面是几种常见情形：

- 确定是否把服务器端代码作为部分页面回发的结果来调用
- 添加对其他客户端 JavaScript 文件的引用
- 引用 Web 服务
- 给客户返回错误消息

下面介绍这些配置选项。

1) 检测部分页面的回发

ScriptManager 控件包含一个布尔属性 IsInAsyncPostBack。可以在服务器端代码中使用这个属性，检测部分页面回发是否在处理过程中。注意页面的 ScriptManager 控件实际上可能在母版页上。除了通过母版页访问这个控件之外，还可以使用静态方法 GetCurrent()，获得对当前 ScriptManager 实例的引用。例如：

```
ScriptManager scriptManager = ScriptManager.GetCurrent(this);
if (scriptManager != null && scriptManager.IsInAsyncPostBack)
{
  // Code to execute for partial-page postbacks.
}
```

必须将对 Page 控件的引用传递给 GetCurrent()方法。例如，如果在 ASP.NET Web 页面的 Page_Load()事件处理程序中使用这个方法，就可以将 this 用作 Page 引用。另外，注意检查 null 引用，以避免异常。

2) 客户端 JavaScript 引用

除了在 HTML 页面的标题或页面的<script>元素中添加代码之外，还可以使用 ScriptManager 类的 Scripts 属性。这可以使脚本引用集中在一起，更便于维护它们。为了以声名方式实现这一点，可以给<UpdatePanel>控件元素添加一个子<Scripts>元素，再给<Scripts>添加<asp:ScriptReference>子控件元素。使用 ScriptReference 控件的 Path 属性引用自定义脚本。

下面的例子说明了如何在 Web 应用程序的根文件夹下，添加对自定义脚本文件 MyScript.js 的引用：

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Scripts>
    <asp:ScriptReference Path="~/MyScript.js" />
  </Scripts>
</asp:ScriptManager>
```

3) Web 服务引用

为了从客户端 JavaScript 代码中访问 Web 服务，ASP.NET AJAX 必须生成一个代理类。要控制这个行为，可以使用 ScriptManager 类的 Services 属性。与 Scripts 属性一样，也可以以声明方式指定这个属性，但这次要使用<Services>元素。给这个元素添加<asp:ServiceReference>控件。对于 Services 属性中的每个 ScriptReference 对象，都需要使用 Path 属性指定到 Web 服务的路径。

ServiceReference 类也有一个 InlineScript 属性，它默认为 false。当这个属性是 false 时，客户端代码通过从服务器请求，会得到一个代理类，来调用 Web 服务。为了增强性能(尤其是在一个页面

上使用大量 Web 服务的情况)，可以将 InlineScript 设置为 true。这会在页面的客户端脚本中定义代理类。

ASP.NET Web 服务的文件扩展名是.asmx。如果不想详细阅读本章，但希望在 Web 应用程序的根文件夹下添加对 Web 服务 MyService.asmx 的引用，应使用下面的代码：

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Services>
    <asp:ServiceReference Path="~/MyService.asmx" />
  </Services>
</asp:ScriptManager>
```

采用这种方式只能添加对本地 Web 服务的引用(即 Web 服务和调用代码在同一个 Web 应用程序中)。可以通过本地 Web 方法间接调用远程 Web 服务。

本章后面将讨论如何从客户端 JavaScript 代码中异步调用 Web 方法，以这种方式使用代理类生成这些方法。

4) 客户端错误消息

如果在部分页面的回发过程中抛出了异常，默认行为就是将异常包含的错误消息放在客户端 JavaScript 警报消息框中。处理 ScriptManager 实例的 AsyncPostBackError 事件，可以定制要显示的消息。在这个事件处理程序中，可以使用 AsyncPostBackErrorEventArgs.Exception 属性访问抛出的异常，使用 ScriptManager.AsyncPostBackErrorMessage 属性设置显示给客户端的消息。这么做可以对用户隐藏异常细节。

如果要重写默认行为，以另一种方式显示消息，就必须使用 JavaScript 处理客户端对象 PageRequestManager 的 endRequest 事件，详见本章后面的内容。

2. 使用 UpdatePanel 控件

UpdatePanel 控件是编写支持 ASP.NET AJAX 的 Web 应用程序时最常用的控件。如本章前面的简单例子所述，这个控件可以封装 Web 页面的一部分，使它能够参与部分页面的回发操作。为此，要在页面中添加一个 UpdatePanel 控件，用它需要包含的控件填充其子元素<ContentTemplate>。

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  <ContentTemplate>
    ...
  </ContentTemplate>
</asp:UpdatePanel>
```

根据 UpdatePanel 控件的 RenderMode 属性值，<ContentTemplate>模板的内容呈现在<div>或<span>元素中。这个属性的默认值是 Block，其结果显示在<div>元素中。要使用<span>元素，应将 RenderMode 属性设置为 Inline。

1) 一个 Web 页面上的多个 UpdatePanel 控件

可以在一个页面上包含任意多个 UpdatePanel 控件。如果回发由包含在页面上的任意 UpdatePanel 控件的<ContentTemplate>模板中的控件引发，就进行部分页面的回发，而不是整个页面的回发。这会使所有 UpdatePanel 控件根据其 UpdateMode 属性值更新。这个属性的默认值是 Always，它表示 UpdatePanel 控件为页面上的部分页面回发操作而更新，即使这个操作在另一个 UpdatePanel

控件中引发，也是如此。如果把这个属性设置为 Conditional，UpdatePanel 控件就仅在它包含的控件引发部分页面回发时更新，或者在激活了已定义的触发器时更新。触发器稍后介绍。

如果把 UpdateMode 属性设置为 Conditional，那么还可以将 ChildrenAsTriggers 属性设置为 false，以禁止 UpdatePanel 控件包含的控件触发 UpdatePanel 控件的更新。但要注意，在这种情况下，这些控件仍会触发一个部分页面更新，它会更新页面上的其他 UpdatePanel 控件。例如，这会更新 UpdateMode 属性值为 Always 的 UpdatePanel 控件，如下面的代码所示：

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <%= DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
  <ContentTemplate>
    <small>Panel 2 render time: <%= DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <%= DateTime.Now.ToLongTimeString() %></small>
```

在这段代码中，把 UpdatePanel2 控件的 UpdateMode 属性值设置为默认的 Always。单击对应按钮时，会引发一个部分页面回发，但只更新 UpdatePanel2 控件。注意，会看到只更新了“Panel2 render time”标签。

2) 服务器端的 UpdatePanel 更新

有时页面上有多个 UpdatePanel 控件，可能不打算更新其中一个控件，除非满足某些条件。在这种情况下，应将 UpdatePanel 控件的 UpdateMode 属性设置为 Conditional，如前所述，再把 ChildrenAsTriggers 属性设置为 false。接着，对于页面上引发部分页面回发更新的一个控件，在服务器端的事件处理程序代码中，(有条件地)调用 UpdatePanel 控件的 Update()方法，例如：

```
protected void Button1_Click(object sender, EventArgs e)
{
  if (TestSomeCondition())
  {
    UpdatePanel1.Update();
  }
}
```

3) UpdatePanel 的触发器

给 Web 页面上其他地方的控件的 Triggers 属性添加触发器，就可以通过该控件更新 UpdatePanel 控件。触发器是 Web 页面上其他地方的控件的事件与 UpdatePanel 控件之间的关联。因为所有控件都有默认事件(如 Button 控件的默认事件是 Click)，所以指定事件名是可选的。有两种触发器可以添加，它们用两个类表示：

- AsyncPostBackTrigger——在指定控件的指定事件发生时，这个类会更新 UpdatePanel 控件。
- PostBackTrigger——在指定控件的指定事件发生时，这个类会更新整个页面。

一般最常使用 AsyncPostBackTrigger 类，但如果希望 UpdatePanel 中的控件触发整个页面的回发，

就可以使用 PostBackTrigger 类。

这两个触发器类都有两个属性 ControlID 和 EventName，ControlID 指定了通过其标识符激活触发器的控件，EventName 指定了控件中链接到触发器的事件名称。

为了扩展前面的例子，考虑下面的代码：

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Button2" />
  </Triggers>
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button2" Text="Click Me" />
    <small>Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
```

新的 Button 控件 Button2 指定为 UpdatePanel1 中的触发器。单击这个按钮时，会更新 UpdatePanel1 和 UpdatePanel2。更新 UpdatePanel1 是因为激活了触发器，更新 UpdatePanel2 是因为它使用了 UpdateMode 的默认值 Always。

3. 使用 UpdateProgress

如前面的例子所示，UpdateProgress 控件允许在部分页面的回发过程中给用户显示进度消息。使用 ProgressTemplate 属性可提供显示进度的 Itemplate，为此，一般使用控件的<ProgressTemplate>子元素。

使用 AssociatedUpdatePanelID 属性将 UpdateProgress 控件与指定的 UpdatePanel 控件关联起来，就可以在页面上放置多个 UpdateProgress 控件。如果没有设置该属性(默认)，无论哪个 UpdatePanel 控件引发了部分页面回发，都显示 UpdateProgress 模板。

发生部分页面回发时，显示 UpdateProgress 模板之前有一个延迟。这个延迟可以通过 DisplayAfter 属性来配置，DisplayAfter 是一个 int 属性，它指定延迟时间(单位是 ms)，默认为 500ms。

最后，可以使用布尔属性 DynamicLayout 指定在显示模板之前，是否为模板分配空间。因为这个属性的默认值是 true，此时页面上的空间是动态分配的，所以为了显示内联的进度模板，需要删除其他控件。如果把这个属性设置为 false，就在显示模板之前，为模板分配空间，这样页面上其他控件的布局不会改变。可以根据显示进度时要达到的效果设置这个属性。对于使用绝对坐标定位的进度模板，如前面的例子所示，应将这个属性设置为默认值。

4. 使用扩展控件

ASP.NET AJAX 的核心软件包包含一个 ExtenderControl 类，它的作用是允许扩展其他 ASP.NET 服务器控件(即增加功能)。它广泛应用于 AJAX Control Toolkit，效果不错。可以使用 ASP.NET AJAX

Server Control Extender 项目模板创建自己的扩展控件。ExtenderControl 控件的工作方式都类似：把它们放在页面上，与目标控件关联起来，添加进一步的配置。接着扩展程序就会发出客户端代码，以添加功能。

为了在一个简单的例子中了解它，添加 NuGet 包 Ajax Control Toolkit，创建一个新的 Web Form，名为 ExtenderDemo.aspx，然后添加如下代码。注意现在使用 ToolkitScriptManager 替代 ScriptManager，ScriptManager 不能给工具集控件加载所有需要的脚本：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ExtenderDemo.aspx.cs"
  Inherits="ProCSharpAjaxSample.ExtenderDemo" %>
<!DOCTYPE html>
<html>
<head runat="server">
<title>Color Selector</title>
</head>
<body>
  <form id="form1" runat="server">
    <ajaxToolkit:ToolkitScriptManager ID="ScriptManager1" runat="server" />
  <div>
    <asp:UpdatePanel runat="server" ID="updatePanel1">
      <ContentTemplate>
        <span style="display: inline-block; padding: 2px;">My favorite color is:
        </span>
        <asp:Label runat="server" ID="favoriteColorLabel" Text="green"
          Style="color: #00dd00; display: inline-block; padding: 2px;
            width: 70px; font-weight: bold;" />
        <ajaxToolkit:DropDownExtender runat="server" ID="dropDownExtender1"
          TargetControlID="favoriteColorLabel"
          DropDownControlID="colDropDown" />
        <asp:Panel ID="colDropDown" runat="server"
          Style="display: none; visibility: hidden; width: 60px;
            padding: 8px; border: double 4px black; background-color: #ffffdd;
          font-weight: bold;">
        <asp:LinkButton runat="server" ID="OptionRed" Text="red"
          OnClick="OnSelect" Style="color: #ff0000;" /><br />
        <asp:LinkButton runat="server" ID="OptionOrange" Text="orange"
          OnClick="OnSelect" Style="color: #dd7700;" /><br />
        <asp:LinkButton runat="server" ID="OptionYellow" Text="yellow"
          OnClick="OnSelect" Style="color: #dddd00;" /><br />
        <asp:LinkButton runat="server" ID="OptionGreen" Text="green"
          OnClick="OnSelect" Style="color: #00dd00;" /><br />
        <asp:LinkButton runat="server" ID="OptionBlue" Text="blue"
          OnClick="OnSelect" Style="color: #0000dd;" /><br />
        <asp:LinkButton runat="server" ID="OptionPurple" Text="purple"
          OnClick="OnSelect" Style="color: #dd00ff;" />
        </asp:Panel>
      </ContentTemplate>
    </asp:UpdatePanel>
  </div>
  </form>
</body>
</html>
```

还需要在这个文件的代码隐藏中添加如下事件处理程序(代码文件 ProCSharpAjaxSample/

ExtenderDemo.aspx.cs)：

```
protected void OnSelect(object sender, EventArgs e)
{
   favoriteColorLabel.Text = ((LinkButton)sender).Text;
   favoriteColorLabel.Style["color"] = ((LinkButton)sender).Style["color"];
}
```

在浏览器中，刚开始并没有显示很多内容，扩展程序似乎没有什么作用，如图 41-28 所示。

但是，把鼠标悬停在文本“green”上时，会动态地显示一个下拉框。如果单击这个下拉框，就会显示一个列表，如图 41-29 所示。

图 41-28

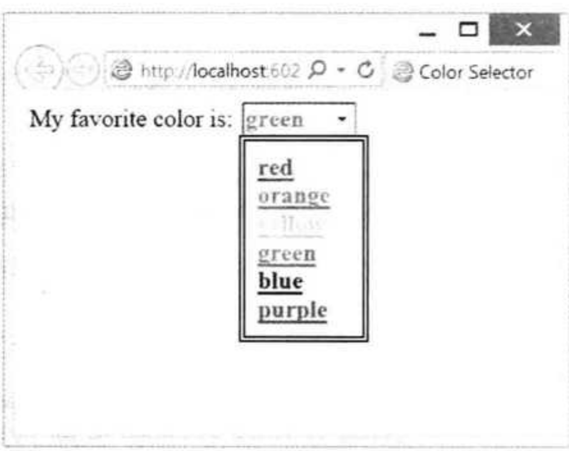


图 41-29

单击下拉列表中的一个链接时，文本会相应地改变(在部分页面回发之后)。

对于这个简单的例子，要注意两个要点：

- 非常容易将扩展程序与目标控件关联起来。
- 下拉列表用自定义代码设置了样式，这表示可以在列表中放置任意内容。这个简单的扩展程序是给 Web 应用程序添加功能的有效方式，使用起来也很简单。

AJAX Control Toolkit 中的扩展程序在不断地增加和更新，如果已经使用 NuGet 安装了它，在新版本可用时，会自动获得通知。

除了 AJAX Control Toolkit 提供的扩展控件之外，还可以创建自己的扩展控件。要创建有效的扩展程序，必须使用 AJAX 库。但是，这个脚本库超出了本书的讨论范围。

41.9　小结

本章介绍了创建 ASP.NET 页面和 Web 应用程序的几种高级技术，例如使用 ASP.NET AJAX 来增强 ASP.NET Web 应用程序。ASP.NET AJAX 包含大量的功能，可以使 Web 站点的响应性和动态性更好，从而提供更好的用户体验。

首先，介绍了 ASP.NET Web Forms 的页面模型和事件模型，以及如何使用跟踪找出事件的详细信息。页面事件对于理解 ASP.NET Web Forms 非常重要。然后，本章讨论了母版页，以及如何为 Web 站点的页面提供一个模板，这是另外一种重用代码和简化开发的方式。

本章还讨论了使用 ASP.NET 验证控件进行的验证，然后简要介绍了安全性，以及如何利用前一章介绍的 API，用最少的努力在 Web 站点上实现基于表单的身份验证。

接下来，介绍了数据控件，以及如何访问 Entity Framework。然后，通过 ObjectDataSource 演示了更多灵活的用法。在本章的最后一部分，学习了 ASP.NET AJAX，这是 Microsoft 实现 Ajax 的方式。

第 42 章将介绍另一种 ASP.NET 框架：ASP.NET MVC。与 Web Forms 不同，在 ASP.NET MVC 中必须处理 HTML 和 JavaScript，以及使用.NET 来实现服务器端功能。因为在这种框架中 UI、功能和数据访问清晰地分离开了，所以进行单元测试更加容易。

第 42 章

ASP.NET MVC

本章要点

- 理解 ASP.NET MVC
- 创建控制器
- 创建视图
- 验证用户输入
- 使用过滤器
- 身份验证和授权
- 使用 ASP.NET Web API

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章的代码分为以下几个主要的示例文件：

- MVC Sample App
- Menu Planner

42.1 ASP.NET MVC 概述

第 40 章介绍了使用 ASP.NET 进行 Web 编程的基础知识，这是 ASP.NET Web Forms 和 ASP.NET MVC 的基础。第 41 章讨论了 ASP.NET Web Forms，这是一个框架，简化了使用服务器端代码创建 Web 应用程序的过程，并且使用的服务器端控件会自己创建 HTML 和 JavaScript 代码。本章则介绍与之相反的内容：HTML 和 JavaScript 在本章要介绍的内容中变得更加重要。为控制器和模型编写服务器端 C#代码，但是对于视图，则应该选择 HTML 和 JavaScript，要编写的 C#代码则很少。在 Visual Studio 2013 中，使用 ASP.NET MVC 5。

本章使用的主要名称空间是 System.Web.Mvc 及其子名称空间，以及 System.Web.Http。

第 40 章介绍了 MVC 模式。现在来看其代码实现。首先创建一个简单的 ASP.NET MVC 项目。

Visual Studio 2013 为 ASP.NET Web 应用程序默认提供了几个使用 MVC 目录和库的模板，例如 MVC、Web API、Single Page Application 和 Facebook。本章的第一个示例应用程序将使用 Empty 模板。对这个模板选择 MVC 选项，如图 42-1 所示。这创建了需要的目录，添加了必要的引用。

创建了这个项目后，看看生成的目录。

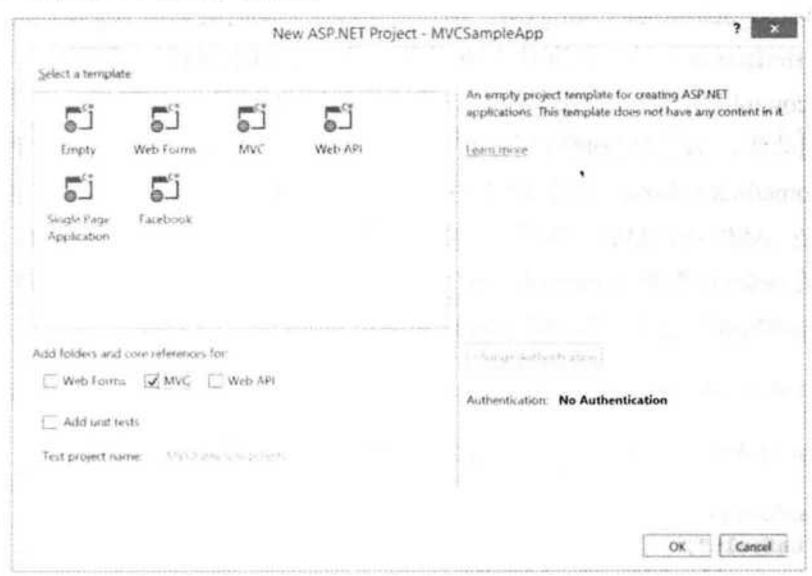


图 42-1

表 42-1

目　录	描　述
App_Data	App_Data 文件夹用于存储数据库文件或其他数据文件，如 XML 文件
App_Start	App_Start 文件夹包含访问控制器和动作的路由定义
Controllers	从名称就可以看出，Controllers 文件夹包含的是控制器。在这个文件夹中应添加响应用户请求的控制器类
Models	Models 文件夹用于数据类，如 ADO.NET Entity Framework
Views	Views 文件夹包含视图。视图通常是 HTML 代码

现在从一个不同的角度看 ASP.NET MVC。图 42-2 显示了在用户发出 HTTP 请求时涉及的 ASP.NET MVC 的各个部分。在服务器端收到请求(Request)时，路由(Routing)定义了应该调用的控制器，以及应该调用的控制器动作。控制器(Controller)负责返回结果。它可以使用一个模型来完成工作，并最终返回一个视图结果(ViewResult)。根据视图结果，选择一个视图引擎(ViewEngine)，它会搜索合适的视图(view)。视图结果放在响应(Response)中返回。

接下来就探讨这些主要步骤，从路由开始。

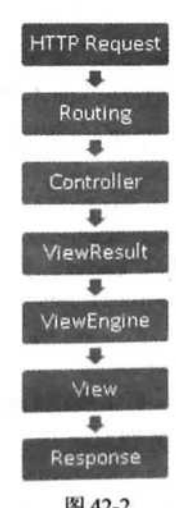


图 42-2

42.2 定义路由

控制器是基于路由选择的。默认的路由在 RegisterRoutes 方法中定义(代码文件 MVCSampleApp/App_Start/RouteConfig.cs)。Web 应用程序启动时会调用 Application_Start 方法，该方法会调用 RegisterRoutes 方法。名为 Default 的路由是 ASP.NET MVC 应用程序的默认路由。默认路由使用 URL {controller}/{action}/{id}定义。该路由映射了 URL 的 3 个段。第一个段映射到控制器，第二个段映射到动作，第三个段映射到参数 id。现在看 ASP.NET MVC 的一个示例 URL，如 http://localhost:Home/Index/demo。在此 URL 中，Controller 的值是 Home，action 的值是 Index，id 的值是 demo。在 ASP.NET MVC 应用程序中，必须有控制器和动作，但是它们可以有默认值。MapRoute 方法的 defaults 参数为 controller 和 action 定义了默认值，并且指定 id 参数是可选的。这样，指定 http://localhost 就定义了控制器 Home 和动作 Index。

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

  routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
);
```

第 40 章解释了应用程序事件，如 Application_Start。

42.2.1 添加路由

添加或修改路由的原因有几种。例如，修改路由以便只使用带链接的动作，而将 Home 定义为默认控制器，向链接添加额外的项，或者使用多个参数。

如果要定义一个路由，让用户通过类似于 http://<server>/About 的链接来使用 Home 控制器中的 About 动作方法，而不传递控制器名称，可以使用如下所示的代码。URL 中省略了控制器。控制器是必须有的，但是可以定义为默认值：

```
routes.MapRoute(
  name: "Default",
  url: "{action}/{id}",
  defaults: new { controller = "Home", action = "Index",
                  id = UrlParameter.Optional }
);
```

下面显示了修改路由的另一种场景。这段代码在路由中添加了一个变量 Language。该变量放在 URL 中服务器名之后、控制器之前，如 http://server/en/Home/About。可以使用这种方法指定语言：

```
routes.MapRoute(
```

```
    name: "Language",
    url: "{language}/{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
        id = UrlParameter.Optional }
);
```

42.2.2 路由约束

在映射路由时，可以指定约束。这样一来，就只能使用约束定义的 URL。下面的约束通过使用正则表达式(en)|(de)，定义了 language 参数只能是 en 或 de。类似于 http://<server>/en/Home/About 或 http://<server>/de/Home/About 的 URL 是合法的：

```
routes.MapRoute(
  name: "Language",
  url: "{language}/{controller}/{action}/{id}",
  defaults: new { controller = "Home", action = "Index",
    id = UrlParameter.Optional },
  constraints: new {language = @"(en)|(de)"}
);
```

如果某个链接只允许使用数字(例如，通过产品代码访问产品)，那么可以使用正则表达式\d+来匹配多个数位构成的数字，但是至少要有一个数位：

```
routes.MapRoute(
    name: "Products",
    url: "{controller}/{action}/{productId}",
    defaults: new { controller = "Home", action = "Index",
        productId = UrlParameter.Optional },
    constraints: new { productId = @"\d+"}
);
```

路由指定了使用的控制器和控制器的动作。因此，接下来就讨论控制器。

42.3 创建控制器

控制器对用户请求做出反应，然后发回一个响应。如稍后所述，视图并不是必要的。

ASP.NET MVC 中存在一些约定。在 ASP.NET MVC 的体系结构中，优先使用约定而不是配置。对于控制器，也是同理。控制器位于目录 Controllers 中，控制器类的名称必须带有 Controller 后缀。

在 Solution Explorer 中选择 Controllers 目录，然后在上下文菜单中选择 Add | Controller 命令，很容易创建控制器。图 42-3 显示了 Add Controller 对话框。在该对话框中，可以创建不同类型的控制器。用于 Web API 的控制器参见第 44 章。这里创建的第一个控制器是空的。本章后面会使用其他控制器模板。选择了空控制器类型后，可以给控制器指定名称。对于所指定的路由，创建 HomeController。

生成的代码中包含了派生自基类 Controller 的 HomeController 类。该类中包含对应于 Index 动作的 Index 方法。请求路由定义的动作时，会调用控制器中的一个方法：

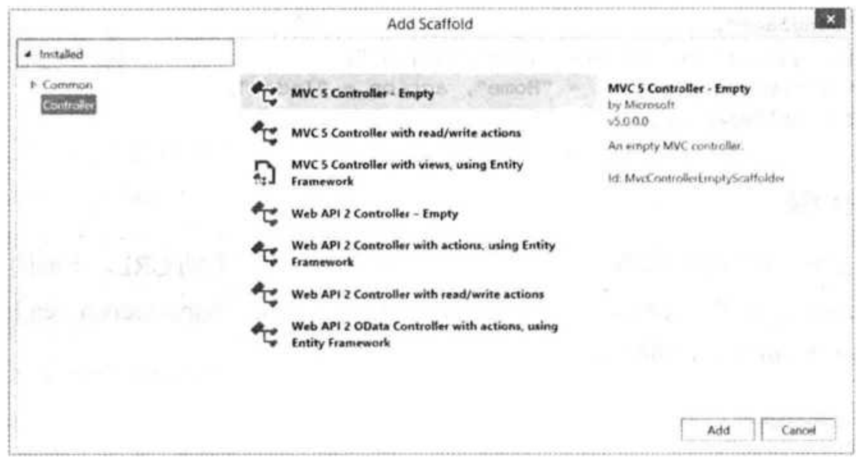


图 42-3

```
public class HomeController : Controller
{
  // GET: /Home/
  public ActionResult Index()
  {
    return View();
  }
}
```

42.3.1 动作方法

控制器中包含动作方法。下面的代码段中的 Hello 方法就是一个简单的动作方法(代码文件 MVCSampleApp/Controllers/HomeController.cs)：

```
public string Hello()
{
  return "Hello, ASP.NET MVC";
}
```

使用链接 http://localhost:42270/Home/Hello 可调用 Home 控制器中的 Hello 动作。当然，端口号取决于自己的设置，可以通过项目设置中的 Web 属性进行配置。在浏览器中打开此链接后，控制器仅仅返回字符串 Hello, ASP.NET MVC。没有 HTML，而只是一个字符串。浏览器显示出了该字符串。

动作可以返回任何东西，例如图像的字节、视频、XML 或 JSON 数据，当然也可以返回 HTML。视图对于返回 HTML 很有帮助。但在介绍视图之前，先看看控制器的一些特性。

42.3.2 参数

如下面的代码段所示，动作方法可以声明为带有参数：

```
public string Greeting(string name)
{
  return HttpUtility.HtmlEncode("Hello, " + name);
}
```

有了此声明，就可以通过请求下面的 URL 来调用 Greeting 动作方法，并在 URL 中为 name 参数传递一个值：http://localhost:42270/Home/Greeting?name=Stephanie。

为了使用更易于记忆的连接，可以使用路由信息来指定参数。在默认的路由配置中，指定了一个带有参数 id 的 Greeting2 动作方法，因为路由指定了此参数值：

```
public string Greeting2(string id)
{
  return HttpUtility.HtmlEncode("Hello, " + id);
}
```

现在可以使用此链接，id 参数包含字符串 Matthias：http://localhost:42270/Home/Greeting2/Matthias。

动作方法也可以声明为带任意数量的参数。例如，可以在 Home 控制器中添加带两个参数的 Add 动作方法：

```
public int Add(int x, int y)
{
  return x + y;
}
```

可以使用如下 URL 来调用此动作，以填充 x 和 y 参数的值：http://localhost:42270/Home/Add?x=及 4&y=5。

使用多个参数时，还可以定义一个路由，以在不同的链接中传递值。下面的代码段显示了路由表中定义的另一个路由，它指定了填充变量 x 和 y 的多个参数(代码文件 MVCSampleApp/Global.asax.cs)：

```
routes.MapRoute(
  name: "MultipleParameters",
  url: "{controller}/{action}/{x}/{y}",
  defaults: new { controller = "Home", action = "Index" }
);
```

现在可以使用如下 URL 调用与之前相同的动作：http://localhost:42270/Home/Add/7/2。

42.3.3　返回数据

到目前为止，只从控制器返回了字符串值。通常，会返回 ActionResult 或者派生自 ActionResult 的类。

下面是 ResultController 类的几个例子(代码文件 MVCSampleApp/Controllers/ResultController.cs)。第一段代码使用 ContentResult 类来返回简单的文本内容。并不需要创建 ContentResult 类的实例并返回该实例，而可以使用基类 Controller 的方法来返回 ActionResult。这里使用 Content 方法来返回文本内容。Content 方法允许指定内容、MIME 类型和编码：

```
public ActionResult ContentDemo()
{
  return Content("Hello World", "text/plain");
}
```

在 JavaScript 方法中，可以返回 JavaScript 代码。在示例代码中，该方法自动将 MIME 类型设为 application/x-javascript：

```
public ActionResult JavaScriptDemo()
```

```
{
  return JavaScript("<script>function foo { alert('foo'); }</script>");
}
```

为了返回 JSON(在 JavaScript 中最好返回这种格式)，可以使用 Jason 方法。在示例代码中，创建 Menu 对象。为了允许客户端发出 HTTP GET 请求，必须用 Json 方法指定 JsonRequestBehavior.AllowGet。使用 JSON 的另一种方法是在服务器端代码的视图中使用它，这时不需要 GET 请求：

```
public ActionResult JsonDemo()
{
  var m = new Menu
  {
    Id = 3,
    Text = "Grilled sausage with sauerkraut und potatoes",
    Price = 12.90,
    Category = "Main"
  };
  return Json(m, JsonRequestBehavior.AllowGet);
}
```

Menu 类定义在 Models 目录中，它定义了一个包含一些属性的简单 POCO 类(代码文件 MVCSampleApp/Models/Menu.cs)：

```
public class Menu
{
  public int Id { get; set; }
  public string Text { get; set; }
  public double Price { get; set; }
  public string Category { get; set; }
}
```

客户端可在响应体内看到这些 JSON 数据，现在它们可轻松地用作 JavaScript 对象：

```
{"Id":3,"Text":"Grilled sausage with sauerkraut und potatoes",
"Price":12.9,"Category":"Main"}
```

通过使用 Controller 类的 Redirect 方法，客户端接收 HTTP 重定向请求。之后，浏览器会请求它收到的链接。Redirect 方法返回一个 RedirectResult(代码文件 MVCSampleApp/Controllers/ResultController.cs)：

```
public ActionResult RedirectDemo()
{
  return Redirect("http://www.cninnovation.com");
}
```

通过指定到另一个控制器和动作的重定向，也可以构建对客户端的重定向请求。RedirectToRoute 返回一个 RedirectToRouteResult，允许指定路由名称、控制器、动作和参数。这会构建一个在收到 HTTP 重定向请求时返回客户端的链接：

```
public ActionResult RedirectRouteDemo()
{
```

```
  return RedirectToRoute(new { controller = "Home", action="Hello" });
}
```

根据方法的不同重载，File 方法可以返回 FilePathResult、FileContentResult 和 FileStreamResult。不同的返回类型取决于使用的参数，例如用于文件路径的字符串，用作流结果的 Stream，用作内容结果的 byte 数组。下面的示例代码创建了一个 Content 目录，其中包含 Images 文件夹和 Stephanie.jpg 图像文件。示例代码返回指定 JPG 文件名的 FilePathResult，contentType 参数也定义了该文件名：

```
public ActionResult FileDemo()
{
  return File("~/Content/Images/Stephanie.jpg", "image/jpg");
}
```

42.4 节演示了如何返回不同的 ViewResult 变体。

42.4 创建视图

返回给客户端的 HTML 代码最好通过视图指定。对于本节的示例，创建了 ViewsDemoController。视图都在 Views 文件夹中定义。ViewsDemo 控制器的视图需要一个 ViewsDemo 子目录，这是视图的约定。

另一个可以搜索视图的地方是 Shared 目录。可以把多个控制器使用的视图(以及多个视图使用的特殊部分视图)放在 Shared 目录中(代码文件 MVCSampleApp/Controllers/ViewDemoController.cs)：

```
public ActionResult Index()
{
  return View();
}
```

在代码编辑器中选择 Index 方法，就可以在上下文菜单中选择 Add | View 命令，创建视图。Add View 对话框如图 42-4 所示。为了开始使用视图，取消选中 Use a layout or master page 复选框，只创建一个简单的视图：

图 42-4

动作方法 Index 使用没有参数的 View 方法，因此视图引擎会在 ViewsDemo 目录中寻找与动作同名的视图文件。View 方法被重载，允许传递不同的视图名称。此时，视图引擎会寻找与在 View 方法中传递的名称对应的文件。

视图包含 HTML 代码，其中混合了一些服务器端代码。下面的的代码段包含默认生成的 HTML 代码(代码文件 MVCSampleApp/Views/ViewsDemo/Index.cshtml)：

```
@{
  Layout = null;
}
    <!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
  </div>
</body>
</html>
```

服务器端代码使用 Razor 语法@sign:编写。42.4.2 小节将讨论这种语法。在那之前，先看看如何从控制器向视图传递数据。

42.4.1 向视图传递数据

控制器和视图运行在同一个进程中。视图直接在控制器内创建，这便于从控制器向视图传递数据。为传递数据，可使用 ViewDataDictionary。该字典以字符串的形式存储键，并允许使用对象值。ViewDataDictionary 可以与 Controller 类的 ViewData 属性一起使用，例如，向键值为 MyData 的字典传递一个字符串：ViewData["MyData"] = "Hello"。更简单的语法是使用 ViewBag 属性。ViewBag 是动态类型，允许指定任何属性名称，以向视图传递数据(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)：

```
public ActionResult PassingData()
{
  ViewBag.MyData = "Hello from the controller";
  return View();
}
```

使用动态类型的优势在于，视图不会直接依赖于控制器。第 12 章详细介绍了动态类型。

为访问控制器传递的数据，可以用类似的方式使用 ViewBag。类似于 Controller 基类，ViewBag 属性在视图的基类 WebViewPage 中定义(代码文件 MVCSampleApp/Views/ViewsDemo/PassingData.cshtml)：

```
<div>
  <div>@ViewBag.MyData</div>
```

```
</div>
```

42.4.2 Razor 语法

前面提到，视图包含 HTML 和服务器端代码。在 ASP.NET MVC 中，可以使用 ASPX 语法或 Razor 语法。Razor 语法更加简单，需要的输入更少。Razor 使用@字符作为迁移字符。@字符之后的代码是 C#代码。

使用 ASPX 语法时，需要使用开始和结束字符来标记代码块的开始和结束，但是在 Razor 语法中则不需要。Razor 可以自动检测到 C#代码的结束位置。

使用 Razor 语法时，需要区分返回值的语句和不返回值的方法。返回的值可以直接使用。例如，ViewBag.MyData 返回一个字符串。该字符串直接放到 HTML 的 div 标记内：

```
<div>@ViewBag.MyData</div>
```

比较一下 Razor 语法与第 41 章使用的 ASPX 语法：用 ASPX 语法表示时，<div>@ViewBag.MyData</div>将变为<div><%:ViewBag.MyData %></div>。Razor 会默认完成 HTML 编码。

如果要调用没有返回值的方法，或者指定其他不返回值的语句，需要使用 Razor 代码块。下面的代码块定义了一个字符串变量：

```
@{
  string name = "Angela";
}
```

现在，使用迁移字符，即可通过简单的语法使用变量：

```
<div>@name</div>
```

使用 Razor 语法时，引擎在找到 HTML 元素时，会自动认为代码结束。在有些情况中，这是无法自动看出来的。此时，可以使用圆括号来标记变量。其后是正常的代码：

```
<div>@(name), Stephanie</div>
```

foreach 块也可以定义 Razor 代码块：

```
@foreach(var item in list)
{
  <li>The item name is @item.</li>
}
```

通常，使用 Razor 可自动检测到文本内容，例如它们以角括号开头，或者使用圆括号包围变量。但在有些情况下是无法自动检测的，此时需要使用@:来显式定义文本的开始位置。

42.4.3 强类型视图

使用 ViewBag 向视图传递数据只是一种方式。另一种方式是向视图传递模型，这样可以创建强类型视图。

现在用动作方法 PassingAModel 扩展 ViewsDemoController。这里创建了 Menu 项的一个新列表，并把该列表传递给基类 Controller 的 View 方法(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)：

```
public ActionResult PassingAModel()
{
  var menus = new List<Menu>
  {
    new Menu { Id=1, Text="Schweinsbraten mit Knödel und Sauerkraut",
               Price=6.9, Category="Main" },
    new Menu { Id=2, Text="Erdäpfelgulasch mit Tofu und Gebäck",
               Price=6.9, Category="Vegetarian" },
    new Menu { Id=3,
               Text="Tiroler Bauerngröst'l mit Spiegelei und Krautsalat",
               Price=6.9, Category="Main" }
  };
  return View(menus);
}
```

动作方法内的信息可以作为模型在视图内使用。如下面的代码段所示，在视图内可用 model 关键字定义模型。此模型的类型是 IEnumerable<Menu>。因为 Menu 类是在 MVCSampleApp.Models 名称空间中定义的，所以使用 using 关键字打开该名称空间。在定义模型后，用抽象基类 WebViewModel<TModel>定义的 Model 属性的类型就是该模型的类型(代码文件 MVCSampleApp/ViewsDemo/PassingAModel.cshtml)：

```
@using MVCSampleApp.Models
@model IEnumerable<Menu>
@{
  Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>PassingAModel</title>
</head>
<body>
  <div>
    <ul>
      @foreach (var item in Model)
      {
        <li>@item.Text</li>
      }
    </ul>
  </div>
</body>
</html>
```

根据视图需要，可以传递任意对象作为模型。例如，编辑单个 Menu 对象时，模型的类型将是 Menu。在显示或编辑列表时，模型的类型可以是 IEnumerable<Menu>。

运行应用程序并显示定义的视图时，浏览器中将显示一个菜单列表。

42.4.4　布局

通常，Web 应用程序的许多页面会显示部分相同的内容，如版权信息、徽标和主导航结构。这就要用到布局页。在第 41 章介绍的 ASP.NET Web Forms 中，母版页完成的功能与 Razor 语法中的布局页相同。

到目前为止，还没有使用布局页，所有的视图都包含完整的 HTML 内容。如果不使用布局页，需要将 Layout 属性设置为 null 来明确指定：

```
@{
  Layout = null;
}
```

1. 使用默认布局页

只要使用 Add View 对话框，选择使用布局页（使布局页的名称文本框为空），就会自动创建几个目录和布局页。这里添加了 NuGet 包 jQuery ，因为生成的布局页要使用 jQuery 。在 Views 目录中还有_ViewStart.cshtml 文件和一个包含_Layout.cshtml 的新建目录 Shared 。

_ViewStart.cshtml 页面包含全部视图的默认配置。默认定义的唯一设置是将 Layout 属性设为共享布局页_Layout.cshtml(代码文件 MVCSampleApp/Views/_ViewStart.cshtml)：

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

布局页包含了所有使用该布局页的页面所共有的 HTML 内容。与视图和控制器的通信可通过 ViewBag 完成。ViewBag.Title 的值可以在内容页中定义，这里在 HTML 的 title 元素中显示它。基类 WebPageBase 的 RenderBody 方法呈现内容页的内容，因而定义了在什么位置放置内容(代码文件 MVCSampleApp/Views/Shared/_Layout.cshtml)：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewBag.Title - My ASP.NET Application</title>
  <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
  <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
  <script src="~/Scripts/modernizr-2.6.2.js"></script>
</head>
<body>
  <script src="~/Scripts/jquery-1.10.2.min.js"></script>
  <script src="~/Scripts/bootstrap.min.js"></script>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
```

```
      <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
  </div>
</body>
</html>
```

现在修改_Layout.cshtml 页面，使其包含页眉和页脚信息，以及一些主要链接的导航结构。Html.ActionLink 是一个 HTML Helper，它创建一个 HTML a 元素来定义链接。42.6 节将讨论 HTML Helper：

```
<body>
  <header>
    <h1>ASP.NET MVC Sample App</h1>
  </header>
  <nav>
    <ul>
      <li>
        @Html.ActionLink("Layout Sample", "LayoutSample")
      </li>
      <li>
        @Html.ActionLink("Layout using Sections", "LayoutUsingSections")
      </li>
    </ul>
  </nav>
  <div>
    @RenderBody()
  </div>
  <footer>Sample code for Professional C#</footer>

  <script src="~/Scripts/jquery-1.10.2.min.js"></script>
  <script src="~/Scripts/bootstrap.min.js"></script>
</body>
```

为动作 LayoutSample 创建视图(代码文件 MVCSampleApp/Views/ViewsDemo/LayoutSample.cshtml)。该视图未设置 Layout 属性，所以会使用默认布局。但是设置了 ViewBag.Title，并在布局的 HTML title 元素中使用它：

```
@{
  ViewBag.Title = "Layout Sample";
}
<h2>LayoutSample</h2>
<p>
  This content is merged with the layout page
</p>
```

现在运行应用程序，布局与视图的内容会合并到一起，如图 42-5 所示。

2. 使用分区

除了呈现页面主体以及使用 ViewBag 在布局和视图之间交换数据，还可以使用分区定义把视图内定义的内容放在什么位置。下面的代码段(代码文件 MVCSampleApp/Views/Shared/_Layout.cshtml)使用了一个名为 PageNavigation 的分区。默认情况下，必须有这类分区，如果没有，加载视图的操

作会失败。如果把 required 参数设为 false，该分区就变为可选：

图 42-5

```
<div>
  @RenderSection("PageNavigation", required: false)
</div>
<div>
  @RenderBody()
</div>
```

在视图内(代码文件 MVCSampleApp/Views/ViewsDemo/LayoutUsingSections.cshtml)，分区由关键字 section 定义。分区的位置与其他内容完全独立。分区 PageNavigation 从布局放置：

```
@{
    ViewBag.Title = "Layout Using Sections";
}
<h2>Layout Using Sections</h2>
Main content here

@section PageNavigation
{
  <div>Navigation defined from the view</div>
  <ul>
    <li>Nav1</li>
    <li>Nav2</li>
  </ul>
}
```

现在运行应用程序，视图与布局的内容将根据布局定义的位置合并到一起，如图 42-6 所示。

分区不只用于在 HTML 页面主体内放置一些内容，还可用于让视图在页面头部放置一些内容，如页面的元数据。

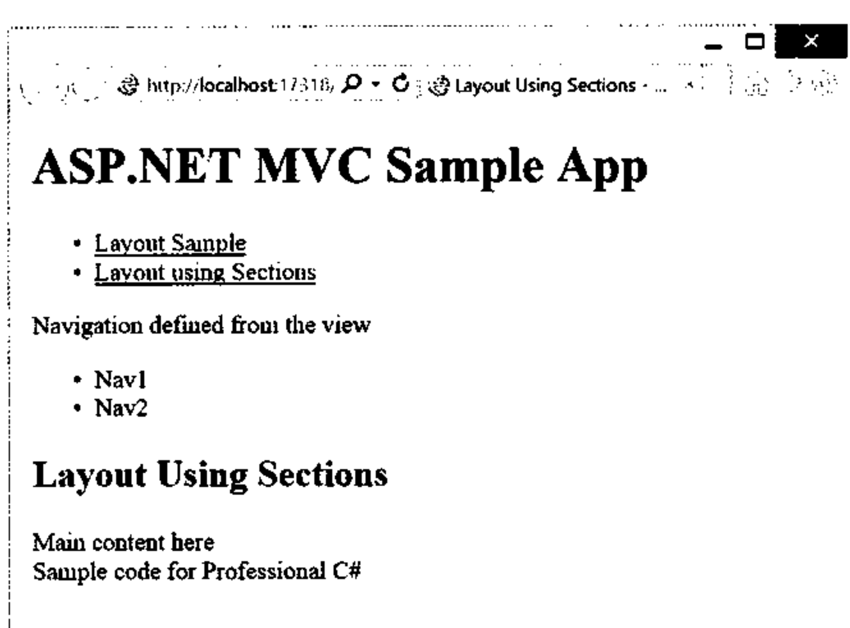


图 42-6

42.4.5　部分视图

布局为 Web 应用程序内的多个页面提供了整体性定义，而部分视图可用于定义视图内的内容。部分视图没有布局。

除此之外，部分视图与标准视图类似。它们也可以有一个模型。部分视图使用与标准视图相同的基类。

下面是部分视图的示例。首先是一个模型，它包含 EventsAndMenus 类定义的独立集合、事件和菜单的属性(代码文件 MVCSampleApp/Models/EventsAndMenus.cs)：

```
public class EventsAndMenus
{
  private IEnumerable<Event> events = null;
  public IEnumerable<Event> Events
  {
    get
    {
      return events ?? (events = new List<Event>()
      {
        new Event { Id=1, Text="Formula 1 G.P. Abu Dhabi, Yas Marina",
            Day=new DateTime(2014, 10, 26) },
        new Event { Id=2, Text="Formula 1 G.P. USA, Austin",
            Day = new DateTime(2014, 11, 9) },
        new Event { Id=3, Text="Formula 1 G.P. Brasil, Sao Paulo",
            Day = new DateTime(2014, 11, 30) }
      });
    }
  }
  private List<Menu> menus = null;
  public IEnumerable<Menu> Menus
  {
```

```
      get
      {
        return menus ?? (menus = new List<Menu>()
        {
          new Menu { Id=1, Text="Baby Back Barbecue Ribs", Price=16.9,
              Category="Main" },
          new Menu { Id=2, Text="Chicken and Brown Rice Piaf", Price=12.9,
              Category="Main" },
          new Menu { Id=3, Text="Chicken Miso Soup with Shiitake Mushrooms",
              Price=6.9, Category="Soup" }
        });
      }
    }
  }
```

下面使用这个模型，介绍从服务器端代码加载的部分视图，然后介绍客户端的 JavaScript 代码请求的部分视图。

1. 使用服务器端代码中的部分视图

在 ViewsDemoController(代码文件 MVCSampleApp/Controllsers/ViewsDemoController.cs)中，动作方法 UseAPartialView 将 EventsAndMenus 的一个实例传递给视图：

```
public ActionResult UseAPartialView1()
{
    return View(new EventsAndMenus());
}
```

这个视图被定义为使用 EventsAndMenus 类型的模型(代码文件 MVCSampleApp/Views/ViewsDemo/UseAPartialView1.cshtml)。使用 HTML Helper 方法 Html.Partial 可以显示部分视图。Html.Partial 返回一个 MvcHtmlString。使用 Razor 语法时，该字符串将被写为 div 元素的内容。Partial 方法的第一个参数接受部分视图的名称。使用第二个参数，则 Partial 允许传递模型。如果没有传递模型，部分视图可以访问与视图相同的模型。这里，视图使用了 EventsAndMenus 类型的模型，部分视图只使用了该模型的一部分，所用模型的类型为 IEnumerable<Event>：

```
@model MVCSampleApp.Models.EventsAndMenus
@{
  ViewBag.Title = "Use a Partial View";
  ViewBag.EventsTitle = "Live Events";
}
<h2>Use a Partial View</h2>
<div>this is the main view</div>
<div>
  @Html.Partial("ShowEvents", Model.Events)
</div>
```

另外一种在视图内呈现部分视图的方法是使用 HTML Helper 方法 Html.RenderPartial，该方法定义为返回 void。该方法将部分视图的内容直接写入响应流。这样一来，就可以在 Razor 代码块中使用 RenderPartial 了。

部分视图的创建方式类似于标准视图。在 Add View 对话框中，有一个用于创建部分视图的复选框。选中该复选框后，就不能分配布局，因为布局要由在其中加载部分视图的视图定义。除此以外，二者的创建方式是相同的。可以访问模型，还可以使用 ViewBag 属性访问字典。部分视图会收到字典的一个副本，以接收可以使用的相同字典数据(代码文件 MVCSampleApp/Views/ViewsDemo/ShowEvents.cshtml)：

```
@using MVCSampleApp.Models
@model IEnumerable<Event>
<h2>
  @ViewBag.EventsTitle
</h2>
<table>
  @foreach (var item in Model)
  {
    <tr>
      <td>@item.Day.ToShortDateString()</td>
      <td>@item.Text</td>
    </tr>
  }
</table>
```

运行应用程序，视图、部分视图和布局都将呈现出来，如图 42-7 所示。

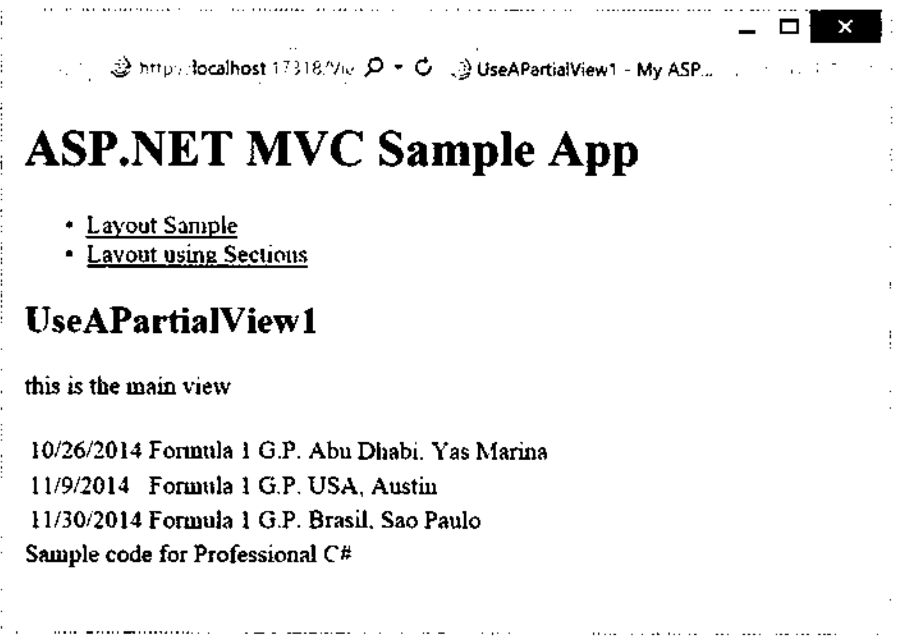


图 42-7

2. 从控制器中返回部分视图

到目前为止，直接加载部分视图，而没有与控制器交互。也可以使用控制器来返回部分视图。

在下面的代码段中(代码文件 MVCSampleApp/Controllers/ViewDemoController.cs)，类 ViewsDemoController 内定义了两个动作方法。第一个动作方法 UsePartialView2 返回一个标准视图，第二个动作方法 ShowEvents 使用控制器方法 PartialView 返回一个部分视图。前面已经创建并使用

过部分视图 ShowEvents，这里再次使用它。PartialView 方法把包含事件列表的模型传递给部分视图：

```
public ActionResult UseAPartialView2()
{
  return View();
}
public ActionResult ShowEvents()
{
  ViewBag.EventsTitle = "Live Events";
  return PartialView(new EventsAndMenus().Events);
}
```

视图 UsePartialView2(代码文件 MVCSampleApp/Views/ViewsDemo/UseAPartialView2.cshtml)通过调用 HTML Helper 方法 Html.Action 来调用控制器。动作名称是 ShowEvents，它使用了与视图相同的控制器。另外，在 Action 方法内也可以为动作方法传递其他控制器和参数：

```
@model MVCSampleApp.Models.EventsAndMenus
@{
  ViewBag.Title = "Use a Partial View";
}
<h2>UseAPartialView</h2>
<div>this is the main view</div>
<div>
  @Html.Action("ShowEvents")
</div>
```

3. 在 jQuery 中调用部分视图

部分视图也可以在客户端代码中直接加载。在下面的代码段(代码文件 MVCSampleApp/Views/ViewsDemo/UseAPartialView3.cshtml)中，事件处理程序被链接到按钮的单击事件。在单击事件处理程序内，向服务器发出了请求/ViewsDemo/ShowEvents 的一个 GET 请求。该请求返回一个部分视图，部分视图的结果放到了名为 events 的 div 元素内：

```
@model MVCSampleApp.Models.EventsAndMenus
@{
  ViewBag.Title = "Use a Partial View";
}
<script>
  $(function () {
    $("#getEvents").click(function () {
      $("#events").load("/ViewsDemo/ShowEvents");
    });
  });
</script>
<h2>Use a Partial View</h2>
<div>this is the main view</div>
<button id="getEvents">Get Events</button>
<div id="events">
</div>
```

42.5 从客户端提交数据

到现在为止，在客户端只是使用 HTTP GET 请求来获取服务器端的 HTML 代码。那么，如何从客户端发送表单数据？

第 40 章深入讨论了 HTTP GET、POST、PUT 的 DELETE 请求。

为提交表单数据，为控制器 SubmitData 创建了视图 CreateMenu。该视图(代码文件 MVCSampleApp/Views/SubmitData/CreateMenu.cshtml)包含一个 HTML 表单元素，它定义了应把什么数据发送给服务器。表单方法声明为 HTTP POST 请求。定义输入字段的 input 元素的名称全部与 Menu 类型的属性对应：

```
@{
  ViewBag.Title = "Create Menu";
}
<h2>Create Menu</h2>
<form action="/SubmitData/CreateMenu" method="post">
<fieldset>
  <legend>Menu</legend>
  <div>Id:</div>
  <input name="id" />
  <div>Text:</div>
  <input name="text" />
  <div>Price:</div>
  <input name="price" />
  <div>Category:</div>
  <input name="category" />
  <div></div>
  <button type="submit">Submit</button>
</fieldset>
</form>
```

图 42-8 显示了在浏览器中打开的页面。

在 SubmitData 控制器内(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)，创建了两个 CreateMenu 动作方法：一个用于 HTTP GET 请求，另一个用于 HTTP POST 请求。C#中存在同名的不同方法，所以这些方法的参数数量或参数类型必须不同。动作方法也存在这种要求。另外，动作方法还需要与 HTTP 请求方法区分开。默认情况下，HTTP 请求方法是 GET，应用 HttpPost 特性后，请求方法是 POST。为读取 HTTP POST 数据，可以使用 Request 对象中的信息(见第 40 章)。但是，定义带参数的 CreateMenu 方法要简单多了。参数的名称与表单字段的名称匹配：

```
public ActionResult CreateMenu()
{
  return View();
}
[HttpPost]
```

```
public ActionResult CreateMenu(int id, string text, double price,
    string category)
{
  var m = new Menu { Id = id, Text = text, Price = price };
  ViewBag.Info = string.Format(
    "menu created: {0}, Price: {1}, category: {2}", m.Text, m.Price,
    m.Category);
  return View("Index");
}
```

http://localhost:173 Create Menu - My ASP.N...
ASP.NET MVC Sample App
- Layout Sample
- Layout using Sections

Create Menu
Menu
Id:
3
Text:
Cheddar Broccoli Casser
Price:
9.7
Category:
Vegetarian
Submit
Sample code for Professional C#

图 42-8

42.5.1 模型绑定器

除了在动作方法中使用多个参数，还可以使用类型，类型的属性与输入的字段名称匹配：

```
[HttpPost]
public ActionResult CreateMenu(Menu m)
{
  ViewBag.Info = string.Format(
      "menu created: {0}, Price: {1}, category: {2}", m.Text, m.Price,
      m.Category);
  return View("Index");
}
```

提交表单数据时，会调用 CreateMenu 方法，它在 Index 视图中显示了提交的菜单数据，如图 42-9 所示。

模型绑定器负责传输 HTTP POST 请求中的数据。模型绑定器实现 IModelBinder 接口。默认情况下，使用 DefaultModelBinder 类将输入字段绑定到模型。这个绑定器支持基本类型、模型类(如 Menu 类型)以及实现了 ICollection<T>、IList<T>和 IDictionary<TKey, TValue>的集合。

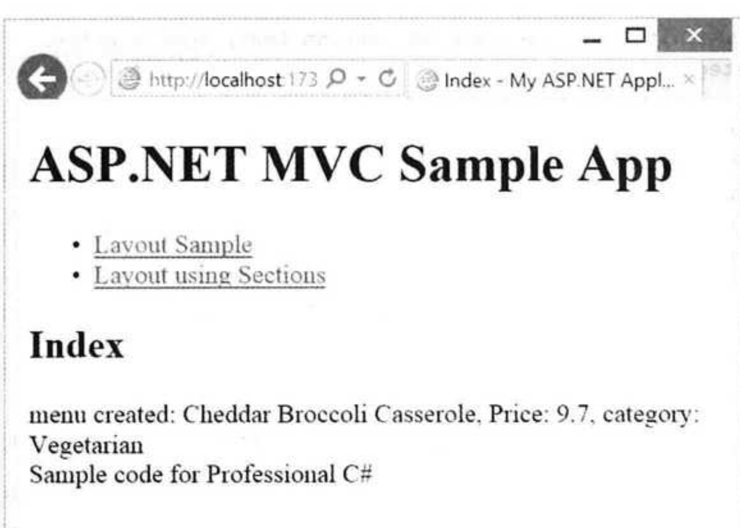


图 42-9

并不是所有参数类型的属性都应从模型绑定器中填充，此时可以使用 Bind 特性。通过这个特性，可以使用 Include 或 Exclude 属性指定一个用逗号分隔开的属性名列表，这些属性应用于绑定，或者不应用于绑定。

还可以使用不带参数的动作方法将输入数据传递给模型，如下面的代码段所示。这段代码创建了 Menu 类的一个新实例，并把这个实例传递给 Controller 类的 UpdateModel 方法:

```
[HttpPost]
public ActionResult CreateMenu2()
{
  var m = new Menu();
  UpdateModel<Menu>(m);
  ViewBag.Info = string.Format(
      "menu created: {0}, Price: {1}, category: {2}", m.Text, m.Price,
      m.Category);
  return View("Index");
}
```

如果在更新后，被更新的模型处于无效状态，UpdateModel 会抛出一个 InvalidOperationException 异常。可以使用 TryUpdateModel 方法来避免这类异常。

如果模型类有一些不应该更新的属性，就不应该使用 UpdateModel 方法。否则恶意用户可以从浏览器修改请求，更新这些属性。使用 TryUpdateModel 方法，可以传递应该更新的属性的白名单，也可以传递不应该更新的属性的黑名单。

42.5.2 注释和验证

可以向模型类型添加一些注释，当更新数据时，会将这些注释用于验证。名称空间 System.ComponentModel.DataAnnotations 中包含的特性可用来为客户端数据指定一些信息，或者用来进行验证。

使用其中的一些特性来修改 Menu 类型(代码文件 MVCSampleApp/Models/Menu.cs)：

```
public class Menu
{
  public int Id { get; set; }

  [Required, StringLength(50)]
  public string Text { get; set; }

  [DisplayName("Price"), DisplayFormat(DataFormatString="{0:C}")]
  public double Price { get; set; }

  [DataType(DataType.Date)]
  public DateTime Date { get; set; }
  [StringLength(10)]
  public string Category { get; set; }
}
```

可用于验证的特性包括：用于比较不同属性的 CompareAttribute，用于验证信用卡号的 CreditCardAttribute，用来验证电子邮件地址的 EmailAddressAttribute，用来比较输入与枚举值的 EnumDataTypeAttribute，以及用来验证电话号码的 PhoneAttribute。

还可以使用其他特性来获得要显示的值，或者用在错误消息中的值，如 DataTypeAttribute 和 DisplayFormatAttribute。

为了使用验证特性，可以在动作方法内使用 ModelState.IsValid 来验证模型的状态，如下所示(代码文件 MVCSampleApp/Controllers/SumitDataController.cs)：

```
[HttpPost]
public ActionResult CreateMenu(Menu m)
{
  if (ModelState.IsValid)
  {
    ViewBag.Info = string.Format(
        "menu created: {0}, Price: {1}, category: {2}", m.Text, m.Price,
        m.Category);
  }
  else
  {
    ViewBag.Info = "not valid";
  }
  return View("Index");
}
```

如果使用由工具生成的模型类，那么很难给属性添加特性。工具生成的类被定义为部分类，可以通过为其添加属性和方法、实现额外的接口或者实现它们使用的部分方法来扩展这些类。对于已有的属性和方法，是不能添加特性的。但是在这种情况中，还是可以利用一些帮助。现在假定 Menu 类是一个工具生成的部分类。可以用一个不同名的新类(如 MenuMetadata)定义与实体类相同的属性，并添加注释：

```
public class MenuMetadata
{
  public int Id { get; set; }
```

```
    [Required, StringLength(25)]
    public string Text { get; set; }

    [DisplayName("Price"), DisplayFormat(DataFormatString="{0:C}")]
    public double Price { get; set; }

    [DataType(DataType.Date)]
    public DateTime Date { get; set; }

    [StringLength(10)]
    public string Category { get; set; }
}
```

MenuMetadata类必须链接到Menu类。对于工具生成的部分类，可以在同一个名称空间中创建另一个部分类型，将MetadataType特性添加到创建连接的该类型定义：

```
[MetadataType(typeof(MenuMetadata))]
public partial class Menu
{
}
```

HTML Helper(如下所示)也可以使用注释来向客户端添加信息。

42.6 HTML Helper

前面介绍了一些HTML Helper方法，如Html.ActionLink和Html.Partial。除了它们，还有很多方法来帮助生成HTML内容。

Html是视图基类WebViewPage的一个属性，它的类型是HtmlHelper。HTML Helper方法实现为扩展方法，用于扩展HtmlHelper类。

类InputExtensions定义了用于创建复选框、密码控件、单选按钮和文本框控件的HTML Helper方法。Helper方法Action和RenderAction由类ChildActionExtensions定义。用于显示的Helper方法由类DisplayExtensions定义。用于HTML表单的Helper方法由类FormExtensions定义。

接下来就看一些使用HTML Helper的例子。

42.6.1 简单的Helper

下面的代码段使用了HTML Helper方法BeginForm、Label和CheckBox。BeginForm开始一个表单元素。还有一个用于结束表单元素的EndForm。示例使用了BeginForm方法返回的MvcForm所实现的IDisposable接口。在释放MvcForm时，会调用EndForm。因此，可以将BeginForm方法放在一个using语句中，在闭花括号处结束表单。DisplayName方法直接返回参数的内容，CheckBox是一个input元素，其type特性设置为checkbox(代码文件MVCSampleApp/Views/HelperMethods/SimpleHelper.cshtml)：

```
@using (Html.BeginForm()) {
  @Html.DisplayName("Check this (or not)")
  @Html.CheckBox("check1")
}
```

得到的 HTML 代码如下所示。CheckBox 方法创建了两个同名的 input 元素，其中一个设为隐藏。其原因是，如果一个复选框的值为 false，那么浏览器不会把与之对应的信息放到表单内容中传递给服务器。只有选中复选框的值才会传递给服务器。这种 HTML 特征在自动绑定到动作方法的参数时会产生问题。简单的解决办法是使用 Helper 方法 CheckBox。该方法会创建一个同名但被隐藏的 input 元素，并将其设为 false。如果没有选中该复选框，则会把隐藏的 input 元素传递给服务器，绑定一个错误值。如果选中了复选框，则同名的两个 input 元素都会传递给服务器。第一个 input 元素设为 true，第二个设为 false。在自动绑定时，只选择第一个 input 元素绑定：

```
<form action="/HelperMethods/Helper1" method="post">
  Check this (or not)
  <input id="check1" name="check1" type="checkbox" value="true" />
  <input name="check1" type="hidden" value="false" />
</form>
```

42.6.2 使用模型数据

Helper 方法可以使用模型数据。下例创建了一个 Menu 对象。本章前面在 Models 目录中声明了此类型。然后，将该 Menu 对象作为模型传递给视图(代码文件 MVCSampleApp/Controllers/HelperMethodsController.cs)：

```
public ActionResult HelperWithMenu()
{
  var menu = new Menu
  {
    Id = 1,
    Text = "Schweinsbraten mit Knödel und Sauerkraut",
    Price = 6.9,
    Date = new DateTime(2012, 10, 5),
    Category = "Main"
  };
  return View(menu);
}
```

视图有一个模型定义为 Menu 类型。与前例一样，HTML Helper 方法 DisplayName 只是返回参数的文本。Display 方法使用一个表达式作为参数，其中以字符串格式传递一个属性名。该方法试图找出具有这个名称的属性，然后使用属性存取器来返回该属性的值(代码文件 MVCSampleApp/Views/HelperMethods/HelperWithMenu.cshtml)：

```
@model MVCSampleApp.Models.Menu
@{
   ViewBag.Title = "HelperWithMenu";
}
<h2>Helper with Menu</h2>
@Html.DisplayName("Text:")
@Html.Display("Text")
<br />
@Html.DisplayName("Category:")
@Html.Display("Category")
```

在得到的HTML代码中，可以从调用DisplayName和Display方法的输出看到这一点：

```
Text:
Schweinsbraten mit Kn&#246;del und Sauerkraut
<br />
Category:
Main
```

Helper方法也提供强类型化方法来访问模型成员，如42.6.5小节所示。

42.6.3　定义HTML特性

大多数HTML Helper方法都有一些可传递任何HTML特性的重载版本。例如，下面的TextBox方法创建一个文本类型的input元素。其第一个参数定义了文本框的名称，第二个参数定义了文本框设置的值。TextBox方法的第三个参数是一个类型对象，允许传递一个匿名类型，在其中将每个属性改为HTML元素的一个特性。在这里，input元素的结果是将required特性设为required，将maxlength特性设为15，将class特性设为CSSDemo。因为class是C#的一个关键字，所以不能直接设为一个属性，而是要加上@作为前缀，以生成用于CSS样式的class特性：

```
@Html.TextBox("text1", "input text here",
  new { required="required", maxlength=15, @class="CSSDemo" });
```

得到的HTML输出如下所示：

```
<input class="Test" id="text1" maxlength="15" name="text1" required="required"
  type="text" value="input text here" />
```

42.6.4　创建列表

为显示列表，需要使用DropDownList和ListBox等Helper方法。这些方法会创建HTML select元素。

在控制器内，首先创建一个包含键和值的字典。然后使用自定义扩展方法ToSelectListItems，将该字典转换为SelectListItem的列表。DropDownList和ListBox方法使用了SelectListItem集合(代码文件MVCSampleApp/Controllers/HelperMethodsController.cs)：

```
public ActionResult HelperList()
{
  var cars = new Dictionary<int, string>();
  cars.Add(1, "Red Bull Racing");
  cars.Add(2, "McLaren");
  cars.Add(3, "Lotus");
  cars.Add(4, "Ferrari");
  return View(cars.ToSelectListItems(4));
}
```

自定义扩展方法ToSelectListItems在扩展了IDictionary<int, string>的SelectListItemsExtensions类中定义，IDictionary<int, string>是cars集合中的类型。在其实现中，只是为字典中的每一项返回

一个新的 SelectListItem 对象：

```
public static class SelectListItemsExtensions
{
  public static IEnumerable<SelectListItem> ToSelectListItems(
      this IDictionary<int, string> dict, int selectedId)
  {
    return dict.Select(item =>
      new SelectListItem
      {
        Selected = item.Key == selectedId,
        Text = item.Value,
        Value = item.Key.ToString()
    });
  }
}
```

在视图中，Helper 方法 DropDownList 直接访问从控制器返回的模型(代码文件 MVCSampleApp/Views/HelperMethods/HelperList.cshtml)：

```
@{
     ViewBag.Title = "Helper List";
}
@model IEnumerable<SelectListItem>
<h2>Helper2</h2>
@Html.DropDownList("carslist", Model)
```

得到的 HTML 创建了一个 select 元素，该元素包含 SelectListItem 创建的一些 option 子元素。这些 HTML 还定义了从控制器中返回的选中项：

```
<select id="carslist" name="carslist">
  <option value="1">Red Bull Racing</option>
  <option value="2">McLaren</option>
  <option value="3">Lotus</option>
  <option selected="selected" value="4">Ferrari</option>
</select>
```

42.6.5 强类型化的 Helper

HTML Helper 方法提供了强类型化的方法来访问从控制器传递的模型。这些方法都带有后缀 For。例如，可以使用 TextBoxFor 代替 TextBox 方法。

下面的示例再次使用返回单个实体的控制器(代码文件 MVCSampleApp/Controllers/HelperMethodsController.cs)：

```
public ActionResult StronglyTypedMenu()
{
  var menu = new Menu
  {
    Id = 1,
    Text = "Schweinsbraten mit Knödel und Sauerkraut",
    Price = 6.9,
    Date = new DateTime(2013, 10, 5),
    Category = "Main"
```

```
  };
  return View(menu);
}
```

视图使用 Menu 类型作为模型，所以可以使用 DisplayNameFor 和 DisplayFor 方法强类型化地访问属性。DisplayNameFor 默认返回属性名(在这里是 Text 属性)，DisplayFor 返回属性值(代码文件 MVCSampleApp/Views/HelperMethods/StronglyTypedMenu.cshtml)：

```
@model MVCSampleApp.Models.Menu

@Html.DisplayNameFor(m => m.Text)
<br />
@Html.DisplayFor(m => m.Text)
```

类似地，可以使用 Html.TextBoxFor(m => m.Text)，它返回一个允许设置模型的 Text 属性的 input 元素。该方法还使用了添加到 Menu 类型的 Text 属性的注释。Text 属性添加了 Required 和 MaxStringLength 特性，所以 TextBoxFor 方法会返回 data-val-length、data-val-length-max 和 data-val-required 特性：

```
<input data-val="true"
  data-val-length="The field Text must be a string with a maximum length of 50."
  data-val-length-max="50" data-val-required="The Text field is required."
  id="Text" name="Text" type="text"
  value="Schweinsbraten mit Knödel und Sauerkraut" />
```

42.6.6 编辑器扩展

除了为每个属性使用至少一个 Helper 方法，EditorExtensions 类中的 Helper 方法还给编辑器提供了类型的所有属性。

使用与前面相同的 Menu 模型，通过方法 Html.EditorFor(m => m)构建一个用于编辑菜单的完整 UI。该方法调用的结果如图 42-10 所示。

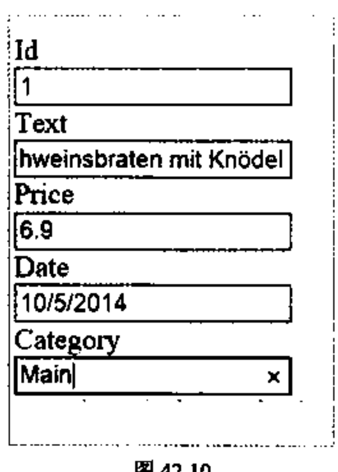


图 42-10

除了使用 Html.EditorFor(m => m)，还可以使用 Html.EditorForModel()。EditorForModel 方法会使用视图的模型，不需要显式指定模型。EditorFor 在使用其他数据源(例如只使用模型提供的属性)方面更加灵活，EditorForModel 需要添加的参数更少。

42.6.7　创建自定义 Helper

Razor 指定了创建自定义 Helper 的语法。一种方法是创建一个扩展了 HtmlHelper 或 HtmlHelper<TModel>类型的扩展方法。采用这种方式，Helper 方法可以像其他所有 HTML Helper 方法一样使用，用 Html 属性返回 HtmlHelper。

创建 Helper 的另一种方法是使用 Razor 的 helper 关键字。这会创建以 Razor 方式实现和使用的方法。在下面的代码段中，Helper 方法 DisplayDay 接收带参数的 DateTime 类型，并且如果传递的日期早于今天，则写入 span 元素。混合 HTML 和代码的方式与之前在 Helper 方法实现中使用 Razor 语法相同。然后，可以在视图中直接使用该方法：

```
@helper DisplayDay(DateTime day)
{
  if (day < DateTime.Today)
  {
    <span>History day</span>
  }
  @String.Format("{0:d}", day);
}
@Html.DisplayFor(m => m.Text)
@Html.DisplayTextFor(m => m.Price)
@Html.TextBoxFor(m => m.Text)
@DisplayDay(Model.Date)
```

42.6.8　模板

使用模板是扩展 HTML Helper 的结果的一种好方法。模板是 HTML Helper 方法隐式或显式使用的一个简单视图，它们存储在特殊的文件夹中。显示模板存储在视图文件夹下的 DisplayTemplates 文件夹中(如 Views/HelperMethods)，或者存储在共享文件夹中(如 Shared/DisplayTemplates)。共享文件夹由全部视图使用，特定的视图文件夹则只有该文件夹中的视图可以使用。对于编辑器模板，则使用 EditorTemplates 文件夹。

现在看一个示例(代码文件 MVCSampleApp/Models/Menu.cs)。在 Menu 类型中，Date 属性有一个注释 DataType，其值为 DataType.Date。指定该特性，DateTime 类型默认并不会显示为日期加时间的形式，而是显示为短日期格式：

```
public class Menu
{
  public int Id { get; set; }

  [Required, StringLength(50)]
  public string Text { get; set; }

  [DisplayName("Price"), DisplayFormat(DataFormatString="{0:c}")]
  public double Price { get; set; }

  [DataType(DataType.Date)]
  public DateTime Date { get; set; }

  [StringLength(10)]
  public string Category { get; set; }
}
```

现在在目录 Views/HelperMethods/DisplayTemplates 中创建了模板 Date.cshtml。这里使用了长日期字符串格式 D 来返回 Model，这个日期字符串格式 D 嵌入在 CSS 类为 markRed 的 div 标记内：

```
<div class="markRed">
  @string.Format("{0:D}", Model)
</div>
```

CSS 类 markRed 在样式表中定义，用于设置红色(代码文件 MVCSampleApp/Content/Site.css)：

```
.markRed {
  color: #f00;
}
```

现在像 DisplayForModel 这样用于显示的 HTML Helper 可以使用已定义的模板。模型的类型是 Menu，所以 DisplayForModel 方法会显示 Menu 类型的所有属性。对于 Date，它找到模板 Date.cshtml，所以会使用该模板以 CSS 样式显示长日期格式的日期(代码文件 MVCSampleApp/Views/HelperMethods/Display.cshtml)：

```
@model MVCSampleApp.Models.Menu
@{
  ViewBag.Title = "Display";
}
<h2>Display</h2>
@Html.DisplayForModel()
```

如果在同一个视图内，某个类型应该有不同的表示，则可以为模板文件使用其他名称。之后就可以使用 UIHint 特性来指定这个模板的名称，或者使用 Helper 方法的模板参数指定模板。

接下来就在数据驱动的应用程序中使用 HTML Helper。

42.7 创建数据驱动的应用程序

在讨论完 ASP.NET MVC 的基础知识后，创建一个使用 ADO.NET Entity Frtamework 的数据驱动的应用程序。该应用程序使用了 ASP.NET MVC 提供的功能和数据访问功能。

第 33 章详细讨论了 ADO.NET Entity Framework。

示例应用程序用于维护数据库中存储的饭店菜单条目。数据库条目的维护只应该由经过身份验证的账户完成。但是，未经身份验证的用户应该能够浏览菜单。

这个项目首先选择 ASP.NET Web Application 模板，再使用 MVC 模板选项。对于身份验证，选择默认选项 Individual User Accounts。这个项目模板给 ASP.NET MVC 添加了几个文件夹，包括 HomeController 、AccountController 和几个脚本库。

42.7.1 定义模型

首先在 Models 目录中定义一个模型。该模型使用 ADO.NET Entity Framework Code-First 创建。MenuCard 类型定义了一些属性和与一组菜单的关系（代码文件 MenuPlanner/Models/MenuCard.cs）：

```
public class MenuCard
{
  public int Id { get; set; }

  [MaxLength(50)]
  public string Name { get; set; }
  public bool Active { get; set; }
  public int Order { get; set; }

  public virtual List<Menu> Menus { get; set; }
}
```

在 MenuCard 中引用的菜单类型由 Menu 类定义（代码文件 MenuPlanner/Models/Menu.cs）：

```
public class Menu
{
  public int Id { get; set; }

  public string Text { get; set; }
  public decimal Price { get; set; }
  public bool Active { get; set; }
  public int Order { get; set; }
  public string Type { get; set; }
  public DateTime Day { get; set; }

  public int MenuCardId { get; set; }
  public virtual MenuCard MenuCard { get; set; }
}
```

数据库连接、Menu 和 MenuCard 类型的设置由 RestaurantEntities 管理。再定义与(localdb)\v11.0 SQL Server 数据库实例的连接，以自动创建数据库。如果没有指定连接字符串，类名就从数据库名中提取（代码文件 MenuPlanner/Models/RestaurantEntities.cs）：

```
public class RestaurantEntities : DbContext
{

  private const string connectionString =
    @"server=(localdb)\v11.0;database=Restaurant;trusted_connection=true";
  public RestaurantEntities()
    : base(connectionString)
  {
  }
  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuCard> MenuCards { get; set; }
}
```

除了创建数据库之外，使用 DatabaseInitializer 类型用最初的菜单卡数据填充数据库。每次启动应用程序时，基类 DropCreateDatabaseAlways<>都会创建一个新的数据库。为了进行演示，这是去

除用户测试数据的一种简单方式。可以把基类改为 DropCreateDatabaseIfNotExists<>，仅在数据库不存在时创建它（代码文件 MenuPlanner/Models/DatabaseInitializer.cs）：

```
public class DatabaseInitializer : DropCreateDatabaseAlways<RestaurantEntities>
{
  protected override void Seed(RestaurantEntities context)
  {
    context.MenuCards.AddOrUpdate(c => c.Name,
      new MenuCard { Name = "Breakfast", Active = true, Order = 1 },
      new MenuCard { Name = "Vegetarian", Active = true, Order = 2 },
      new MenuCard { Name = "Steaks", Active = true, Order = 3 });
      base.Seed(context);
  }
}
```

在 Web.config 中，用 databaseInitializer 上下文配置设置类型，就可以配置 DatabaseInitializer ：

```
<entityFramework>
  <!-- ... -->
  <contexts>
    <context disableDatabaseInitialization="false"
      type="MenuPlanner.Models.RestaurantEntities, MenuPlanner">
      <databaseInitializer
        type="MenuPlanner.Models.DatabaseInitializer, MenuPlanner" />
    </context>
  </contexts>
</entityFramework>
```

有了这些类和配置，就会在每次运行应用程序时创建并初始化数据库。

42.7.2 创建控制器和视图

编译项目后，就可以选择模型中的类来创建控制器和视图。如图 42-11 所示，创建一个新的控制器 MenuAdmin，并选择 Controller with read/write actions and views, using Entity Framework 模板。使用该模板，可以选择模型和数据上下文类。此对话框将基于所做出的选择生成控制器和视图代码。

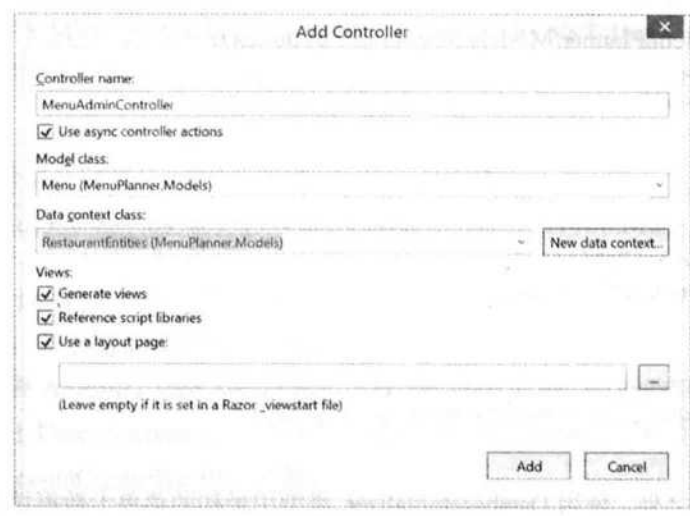


图 42-11

1. 控制器

在创建控制器时，生成的控制器类会通过创建 RestaurantEntities 来使用对象上下文，并提供在数据库中查看、编辑、修改和删除菜单项的动作方法。稍后会看到这些方法。只引用控制器的链接时，默认调用的方法是 Index 方法。这里，创建数据库中的所有 Menu 项，并将它们作为 List<Menu> 传递给视图。

用户创建新菜单时，在收到客户端的 HTTP GET 请求后，会调用第一个 Create 方法。通过 ViewBag 和这个方法的信息创建视图。因为菜单卡片与菜单存在一种关系，所以该 ViewBag 包含关于菜单卡片的信息，用户现在可用新创建的菜单选择一个菜单卡片。当用户填完表单并使用一个 HTTP POST 请求把包含新菜单的表单提交给服务器后，会调用第二个 Create 方法。该方法使用模型绑定把表单数据传递给 Menu 对象，并把 Menu 对象添加到数据上下文中，以把新创建的菜单写入数据库(代码文件 MenuPlanner/Controllers/MenuAdminController.cs)：

```
using MenuPlanner.Models;
using System.Data.Entity;
using System.Net;
using System.Threading.Tasks;
using System.Web.Mvc;

namespace MenuPlanner.Controllers
{
  public class MenuAdminController : Controller
  {
    private RestaurantEntities db = new RestaurantEntities();
    //
    // GET: /MenuAdmin/
    Public async Task<ActionResult> Index()
    {
      var menus = db.Menus.Include(m => m.MenuCard);
      return View(await menus.ToListAsync());
    }
    //
    // GET: /MenuAdmin/Details/5
    public async ActionResult Details(int? id = 0)
    {
      if (id == null)
      {
        Return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
      }
      Menu menu = db.Menus.FindAsync(id);
      if (menu == null)
      {
        return HttpNotFound();
      }
      return View(menu);
    }

    //
    // GET: /MenuAdmin/Create
    public ActionResult Create()
    {
```

```
    ViewBag.MenuCardId = new SelectList(db.MenuCards, "Id", "Name");
    return View();
  }

  //
  // POST: /MenuAdmin/Create
  [HttpPost]
  [ValidateAntiForgeryToken]
  public async Task<ActionResult> Create(
    [Bind(Include="Id,MenuCardId,Text,Price,Active,Order,Type,Day")] Menu menu)
  {
    if (ModelState.IsValid)
    {
      db.Menus.Add(menu);
      await db.SaveChangesAsync();
      return RedirectToAction("Index");
    }
    ViewBag.MenuCardId = new SelectList(db.MenuCards,
        "Id", "Name", menu.MenuCardId);
    return View(menu);
  }

  //
  // GET: /MenuAdmin/Edit/5
  public async Task<ActionResult> Edit(int? id)
  {
    if (id == null)
    {
      return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Menu menu = await db.Menus.FindAsync(id);
    if (menu == null)
    {
      return HttpNotFound();
    }
    ViewBag.MenuCardId = new SelectList(db.MenuCards, "Id",
        "Name", menu.MenuCardId);
    return View(menu);
  }

  //
  // POST: /MenuAdmin/Edit/5
  [HttpPost]
  [ValidateAntiForgeryToken]
  public async Task<ActionResult> Edit(
    [Bind(Include="Id,MenuCardId,Text,Price,Order,Type,Day")] Menu menu)
  {
    if (ModelState.IsValid)
    {
      db.Entry(menu).State = EntityState.Modified;
      await db.SaveChangesAsync();
      return RedirectToAction("Index");
    }
    ViewBag.MenuCardId = new SelectList(db.MenuCards, "Id",
        "Name", menu.MenuCardId);
```

```
      return View(menu);
    }

    //
    // GET: /MenuAdmin/Delete/5
    Public async Task<ActionResult> Delete(int id?)
    {
      if (id == null)
      {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
      }
      Menu menu = await db.Menus.FindAsync(id);
      if (menu == null)
      {
        return HttpNotFound();
      }
      return View(menu);
    }

    //
    // POST: /MenuAdmin/Delete/5
    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> DeleteConfirmed(int id)
    {
      Menu menu = await db.Menus.FindAsync(id);
      db.Menus.Remove(menu);
      await db.SaveChangesAsync();
      return RedirectToAction("Index");
    }

    protected override void Dispose(bool disposing)
    {
      if (disposing)
      {
        db.Dispose();
      }
      base.Dispose(disposing);
    }
  }
}
```

2. 视图

现在看几个设计器生成的视图。Index 视图使用一个 Menu 集合作为模型，并定义了一个 HTML 表。在表的头部元素中，使用 HTML Helper 方法 DisplayNameFor 来访问属性名以进行显示。为了显示菜单项，使用@foreach 迭代菜单集合，使用 DisplayFor 显示每个属性值(代码文件 MenuPlanner/Views/MenuPlanner/Index.cshtml)：

```
@model IEnumerable<MenuPlanner.Models.Menu>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
```

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.MenuCard.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Text)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Active)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Order)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Type)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Day)
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.MenuCard.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Text)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Active)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Order)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Type)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Day)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Details", "Details", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
```

```
        </td>
      </tr>
    }
    </table>
```

这里，MenuAdmin 控制器的第二个视图是 Create 视图。创建一个 HTML 表单，且不向 BeginForm 方法传递参数。这样，在提交表单时，会使用 POST 请求来请求同名(Create)的动作方法。可以看到，表单内容是使用 Helper 方法 DropDownList、ValidationMessageFor 和 EditorFor 构成的(代码文件 MenuPlanner/Views/MenuAdmin/Create.cshtml)：

```
@model MenuPlanner.Models.Menu
@{
  ViewBag.Title = "Create";
}
<h2>Create</h2>
@using (Html.BeginForm())
{
  @Html.AntiForgeryToken()

  <div class="form-horizontal">
    <h4>Menu</h4>
    <hr />
    @Html.ValidationSummary(true)

    <div class="form-group">
      @Html.LabelFor(model => model.MenuCardId, "MenuCardId",
        new { @class = "control-label col-md-2" })
      <div class="col-md-10">
      @Html.DropDownList("MenuCardId", String.Empty)
      @Html.ValidationMessageFor(model => model.MenuCardId)
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.Text, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Text)
      @Html.ValidationMessageFor(model => model.Text)
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.Price, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Price)
      @Html.ValidationMessageFor(model => model.Price)
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.Active, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Active)
      @Html.ValidationMessageFor(model => model.Active)
```

```
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.Order, new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Order)
        @Html.ValidationMessageFor(model => model.Order)
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.Type, new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Type)
        @Html.ValidationMessageFor(model => model.Type)
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.Day, new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Day)
        @Html.ValidationMessageFor(model => model.Day)
      </div>
    </div>

    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Create" class="btn btn-default" />
      </div>
    </div>
  </div>

<div>
  @Html.ActionLink("Back to List", "Index")
  </div>

  @section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

其他视图的创建方式与此类似，所以本书中不再讨论。只需要参考下载代码，或者按照这里的讨论使用 Add Controller 模板创建视图即可。

现在可以使用应用程序给现有的菜单卡添加和编辑菜单了。

42.8 动作过滤器

ASP.NET MVC 在很多方面都可以扩展。可以实现控制器工厂，以搜索和实例化控制器(接口 IControllerFactory)。控制器实现了 IController 接口。使用 IActionInvoker 接口可以找出控制器中的动作方法。使用 ActionMethodSelectorAttribute (...)可以定义允许的 HTTP 方法。通过实现 IModelBinder

接口，可以定制将 HTTP 请求映射到参数的模型绑定器。在 42.5.1 小节中，使用过 DefaultModelBinder 类型。有实现了 IviewEngine 接口的不同视图引擎可供使用。在本章中，使用了 Razor 视图引擎。使用 HTML Helper 也可以实现自定义，前面已详细讨论过。另外，也可以使用动作过滤器实现自定义。大多数可以扩展的地方都不在本书讨论范围内，但是由于很可能需要实现或使用动作过滤器，所以下面就加以讨论。

在动作执行之前和之后，都会调用动作过滤器。使用特性可把它们分配给控制器或控制器的动作方法。通过创建派生自基类 ActionFilterAttribute 的类，可以实现动作过滤器。在这个类中，可以重写基类成员 OnActionExecuting、OnActionExecuted、OnResultExecuting 和 OnResultExecuted。OnActionExecuting 在动作方法调用之前调用，OnActionExecuted 在动作方法完成之后调用。之后，在返回结果前，调用 OnResultExecuting 方法，最后调用 OnResultExecuted 方法。在这些方法内，可以访问 Request 对象来检索调用者信息，根据浏览器决定执行某些操作，访问路由信息，动态修改视图结果等。下面的代码段访问路由信息中的变量 language。为把此变量添加到路由中，可以把路由修改为如 42.2 节所示。用路由信息添加 language 变量后，可以使用 RouteData.Values 访问 URL 中提供的值，如下面的代码段所示。可以根据得到的值，为用户修改区域性：

```
public class LanguageAttribute : ActionFilterAttribute
{
  private string language = null;
  public override void OnActionExecuting(ActionExecutingContext filterContext)
  {
    language = filterContext.RouteData.Values["language"] == null ?
      null : filterContext.RouteData.Values["language"].ToString();
    //...
  }
  public override void OnResultExecuting(ResultExecutingContext filterContext)
  {
  }
}
```

第 28 章讨论了全球化和本地化、区域性设置及其他区域信息。

使用创建的动作过滤器特性类，可以把该特性应用到一个控制器，如代码段所示。对类应用特性后，在调用每个动作方法时，都会调用特性类的成员。另外，也可以把特性应用到一个动作方法，此时只有调用该动作方法时才会调用特性类的成员。

```
[Language]
public class HomeController : Controller
{
```

ASP.NET MVC 包含一些预定义的动作过滤器。可以使用 OutputCacheAttribute 来定义结果的缓存。一些预定义过滤器派生自基类 FilterAttribute(它也是 ActionFilterAttribute 的基类)。使用基类 FilterAttribute 而不是 ActionFilterAttribute 时，只允许在调用动作方法前过滤它们，而不允许在调用后过滤。派生自 FilterAttribute 的类包括 HandleErrorAttribute、AuthorizeAttribute 和 RequireHttpsAttribute。

使用 HandleError 可以处理异常，并定义在发生错误时显示的视图。异常的类型也是可以过滤的，可以根据不同的异常类型指定不同的视图。指定 RequireHttpsAttribute 会检查请求是否通过 HTTPS 发送，如果不是，就拒绝调用动作方法。

42.9 节将讨论 AuthorizeAttribute 的用法。

42.9 身份验证和授权

第 40 章为在 ASP.NET 中使用 ASP.NET 身份系统打下了基础。第 41 章介绍了如何在 ASP.NET Web Forms 中使用它。本章在第 40 章的基础上，介绍如何在 ASP.NET MVC 中使用这些提供程序。

在 MenuPlanner 示例应用程序中，应该使用表单(Form)验证，只允许指定角色的用户修改菜单项。

42.9.1 登录模型

为了允许用户登录，可以创建 LoginModel 控件。该模型定义了 UserName、Password 和 RememberMe 属性——在登录时需要用户确定的所有信息。该模型有一些注释可用于 HTML Helper(代码文件 MVCSampleApp/Models/LoginModel.cs)：

```
public class LoginViewModel
{
  [Required]
  [Display(Name = "User name")]
  public string UserName { get; set; }

  [Required]
  [DataType(DataType.Password)]
  [Display(Name = "Password")]
  public string Password { get; set; }

  [Display(Name = "Remember me?")]
  public bool RememberMe { get; set; }
}
```

42.9.2 登录控制器

用于用户登录的控制器是 AccountController(代码文件 MenuPlanner/Controllers/AccountController.cs)。该控制器定义了可通过 HTTP GET 请求来请求的 Login 动作。Login 动作只是返回 Login 视图，供用户输入用户名和密码。该视图向第二个 Login 动作发出 HTTP POST 请求，使用 LoginModel 作为参数，并将 HTML 表单的值赋给该模型的属性。在实现代码中，用 UserManager 检查用户名和密码。UserManager 使用 ApplicationUser 类型在数据库中访问该类型指定的表中的信息。如果用户有效，就调用 Helper 方法 SignInAsync 完成用户登录。这个 Helper 方法调用 UserManager.CreateIdentityAsync，创建用户身份：

```
using System.Web.Mvc;
using System.Web.Security;
using MenuPlanner.Models;
namespace MenuPlanner.Controllers
```

```
{
  [Authorize]
  public class AccountController : Controller
  {
    public AccountController()
      : this(new UserManager<ApplicationUser>(
        new UserStore<ApplicationUser>(new ApplicationDbContext())))
    {
    }
    public AccountController(UserManager<ApplicationUser> userManager)
    {
      UserManager = userManager;
    }

    public UserManager<ApplicationUser> UserManager { get; private set; }

    //
    // GET: /Account/Login
    [AllowAnonymous]
    public ActionResult Login(string returnUrl)
    {
      ViewBag.ReturnUrl = returnUrl;
      return View();
    }

    //
    // POST: /Account/Login
    [AllowAnonymous]
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Login(LoginModel model, string returnUrl)
    {
      if (ModelState.IsValid)
      {
        var user = await UserManager.FindAsync(model.UserName, modelPassword);
        if (user != null)
        {
          await SignInAsync(user, model.RememberMe);
          return RedirectToLocal(returnUrl);
        }
        else
        {
          ModelState.AddModelError("","Invalid username or password.");
        }
      }
      // If we got this far, something failed, redisplay form
      return View(model);
    }

    //
    // POST: /Account/LogOff
    public ActionResult LogOff()
    {
      AuthenticationManager.SignOut();
      return RedirectToAction("Index", "Home");
```

```
    }

    private IAuthenticationManager AuthenticationManager
    {
      get
      {
        return HttpContext.GetOwinContext().Authentication;
      }
    }
    private async Task SignInAsync(ApplicationUser user, bool isPersistent)
    {
      AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);
      var identity = await UserManager.CreateIdentityAsync(user,
        DefaultAuthenticationTypes.ApplicationCookie);
      AuthenticationManager.SignIn(new AuthenticationProperties()
      { IsPersistent = isPersistent }, identity);
    }
    private ActionResult RedirectToLocal(string returnUrl)
    {
      if (Url.IsLocalUrl(returnUrl))
      {
        return Redirect(returnUrl);
      }
      else
      {
        return RedirectToAction("Index", "Home");
      }
    }
  }
}
```

为指定 Login 动作以及要使用的视图，在 web.config 文件中，将 loginUrl 设为 Account 控制器的 Login 方法(代码文件 MenuPlanner/web.config)：

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" timeout="2880" />
</authentication>
```

42.9.3 登录视图

登录视图定义了一个使用 Account 控制器的表单，并基于模型定义了标签和输入控件。使用 GET 请求 Login 动作时第一次调用该视图，随后它用一个 POST 请求调用 Login 动作，传递模型数据(代码文件 MenuPlanner/Views/Account/Login.cshtml)：

```
@model MenuPlanner.Models.LoginModel
@{
   ViewBag.Title = "Log in";
}
<h2>@ViewBag.Title.</h2>
<div class="row">
  <div class="col-md-8">
    <section id="loginForm">
      @using (Html.BeginForm("Login", "Account",
        new { ReturnUrl = ViewBag.ReturnUrl }, FormMethod.Post,
```

```
        new { @class = "form-horizontal", role = "form" }))
      {
        @Html.AntiForgeryToken()
        <h4>Use a local account to log in.</h4>
        <hr />
        @Html.ValidationSummary(true)
        <div class="form-group">
          @Html.LabelFor(m => m.UserName, new { @class = "col-md-2 control-label" })
          <div class="col-md-10">
            @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.UserName)
          </div>
        </div>
        <div class="form-group">
          @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
          <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.Password)
          </div>
        </div>
        <div class="form-group">
          <div class="col-md-offset-2 col-md-10">
            <div class="checkbox">
              @Html.CheckBoxFor(m => m.RememberMe)
              @Html.LabelFor(m => m.RememberMe)
            </div>
          </div>
        </div>
        <div class="form-group">
          <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Log in" class="btn btn-default" />
          </div>
        </div>
        <p>
          @Html.ActionLink("Register", "Register") if you don't have a local account.
        </p>
      }
    </section>
  </div>
  <div class="col-md-4">
    <section id="socialLoginForm">
      @Html.Partial("_ExternalLoginsListPartial",
        new { Action = "ExternalLogin", ReturnUrl = ViewBag.ReturnUrl })
    </section>
  </div>
</div>
@section Scripts {
  @Scripts.Render("~/bundles/jqueryval")
}
```

现在，只需要确保不是正确角色的用户不能访问方法。为此，可以对 MenuAdminController 类应用 Authorize 特性(代码文件 MenuPlanner/Controllers/MenuAdminController.cs)，并指定允许使用它的角色：

```
[Authorize(Roles="Menu Admins")]
public class MenuAdminController : Controller
{
```

对类应用该特性要求为类的每个动作方法使用角色。如果对不同的动作方法有不同的授权需求，也可以对动作方法应用 Authorize 特性。使用该特性时，可以验证调用者是否已获得了授权(通过检查授权 cookie)。如果调用者还未经授权，则返回一个 401 HTTP 状态代码，并重定向到登录动作(在 Web 配置文件中定义这种行为)。

42.10 小结

本章介绍了一种使用 ASP.NET 的最新 Web 技术：ASP.NET MVC 5 框架。这是一个健壮的框架，非常适合需要恰当地进行单元测试的大型应用程序。通过本章可以看到，使用 ASP.NET MVC 5 时，提供高级功能十分简单，其逻辑结构和功能的分离使代码很容易理解和维护。

本章就结束了编程用户界面的探讨。第 43 章开始介绍通信：WCF。

第Ⅵ部分

通　　信

- 第 43 章　WCF
- 第 44 章　ASP.NET Web API
- 第 45 章　Windows Workflow Foundation
- 第 46 章　对等网络
- 第 47 章　消息队列

第43章

WCF

本章要点

- WCF 概述
- 创建简单的服务和客户端
- 定义服务、操作、数据和消息协定
- 服务的实现
- 给通信使用绑定
- 创建服务的不同宿主
- 通过服务引用和编程方式创建客户端
- 使用双工通信
- 使用路由

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 简单的服务和客户端
- WebSocket
- 双工通信
- 路由

43.1 WCF 概述

Windows Communication Foundation (WCF)是.NET Framework 上灵活的通信技术。在.NET 3.0 推出之前，一个企业解决方案需要几种通信技术。对于独立于平台的通信，使用 ASP.NET Web 服务。对于比较高级的 Web 服务——可靠性、独立于平台的安全性和原子事务等技术——Web Services Enhancements 增加了 ASP.NET Web 服务的复杂性。如果要求通信比较快，客户和服务都是.NET 应

用程序，就应使用.NET Remoting 技术。.NET Enterprise Services 支持自动事务处理，它默认使用 DCOM 协议，比用.NET Remoting 快。DCOM 也是允许传递事务的唯一协议。所有这些技术都有不同的编程模型，这些模型都需要开发人员有许多技巧。

.NET Framework 3.0 引入了一种通信技术 WCF，它包含上述技术的所有功能，把它们合并到一个编程模型中：Windows Communication Foundation(WCF)。

本章介绍的名称空间是 System.ServiceModel。

WCF 合并了 ASP.NET Web 服务、.NET Remoting、消息队列和 Enterprise Services 的功能，WCF 的功能包括：

- **存储组件和服务**——与联合使用自定义主机、.NET Remoting 和 WSE 一样，也可以将 WCF 服务存放在 ASP.NET 运行库、Windows 服务、COM+进程或 Windows 窗体应用程序中，进行对等计算。
- **声明行为**——不要求派生自基类(.NET Remoting 和 Enterprise Services 有这个要求)，而可以使用属性定义服务。这类似于用 ASP.NET 开发的 Web 服务。
- **通信信道**——在改变通信信道方面，.NET Remoting 非常灵活，WCF 也不错，因为它提供了相同的灵活性。WCF 提供了用 HTTP、TCP 和 IPC 信道进行通信的多条信道。也可以创建使用不同传输协议的自定义信道。
- **安全结构**——为了实现独立于平台的 Web 服务，必须使用标准化的安全环境。所提出的标准用 WSE 3.0 实现，这在 WCF 中被继承下来。
- **可扩展性**——.NET Remoting 有丰富的扩展功能。它不仅能创建自定义信道、格式化程序和代理，还能将功能注入客户端和服务器上的消息流。WCF 提供了类似的可扩展性。但是，WCF 的扩展性用 SOAP 标题创建。
- **支持以前的技术**——要使用 WCF，根本不需要完全重写分布式解决方案，因为 WCF 可以与已有的技术集成。WCF 提供的信道使用 DCOM 与服务组件通信。用 ASP.NET 开发的 Web 服务也可以与 WCF 集成。

最终目标是通过进程或不同的系统、通过本地网络或通过 Internet 收发客户和服务之间的消息。如果需要以独立于平台的方式尽快收发消息，就应这么做。在远程视图上，服务提供了一个端点，它用协定、绑定和地址来描述。协定定义了服务提供的操作，绑定给出了协议和编码信息，地址是服务的位置。客户需要一个兼容的端点来访问服务。

图 43-1 显示了参与 WCF 通信的组件。

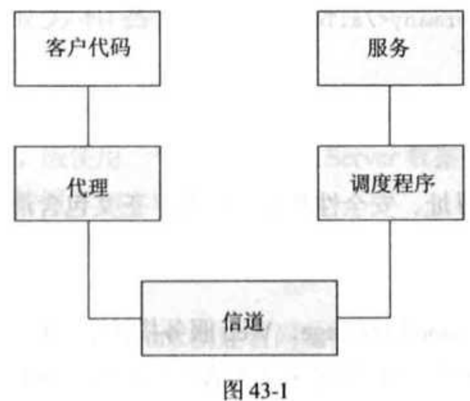


图 43-1

客户调用代理上的一个方法。代理提供了服务定义的方法，但把方法调用转换为一条消息，并把该消息传输到信道上。信道有一个客户端部分和一个服务器端部分，它们通过一个网络协议来通信。在信道上，把消息传递给调度程序，调度程序再把消息转换为用服务调用的方法调用。

WCF 支持几个通信协议。为了进行独立于平台的通信，需要支持 Web 服务标准。要在.NET 应用程序之间通信，可以使用较快的通信协议，其系统开销较小。

下面几节介绍独立于平台的通信所使用的核心服务的功能。

- SOAP(Simple Object Access Protocol，简单对象访问协议)：一个独立于平台的协议，它是几个 Web 服务规范的基础，支持安全性、事务、可靠性。
- WSDL(Web Services Description Language，Web 服务描述语言)：提供描述服务的元数据。
- REST(Representational State Transfer，代表性状态传输)：由支持 REST 的 Web 服务用于在 HTPP 上通信。
- JSON(JavaScript Object Notation，JavaScript 对象标记)：便于在 JavaScript 客户端上使用。

43.1.1 SOAP

为了进行独立于平台的通信，可以使用 SOAP 协议，它得到 WCF 的直接支持。SOAP 最初是 Simple Object Access Protocol 的缩写，但自从 SOAP 1.2 以来，就不再是这样了。SOAP 不再是一个对象访问协议，因为可以发送用 XML 架构定义的消息。

服务从客户中接收 SOAP 消息，并返回一条 SOAP 响应消息。SOAP 消息包含信封，信封包含标题和正文。

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
  </s:Header>
  <s:Body>
    <ReserveRoom xmlns="http://www.cninnovation.com/RoomReservation/2012">
      <roomReservation
          xmlns:a=
            "http://schemas.datacontract.org/2004/07/Wrox.ProCSharp.WCF.Contracts"
          xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:Contact>UEFA</a:Contact>
        <a:EndTime>2012-07-01T22:45:00</a:EndTime>
        <a:Id>0</a:Id>
        <a:RoomName>Kiew</d4p1:RoomName>
        <a:StartTime>2012-07-01T20:45:00</a:StartTime>
        <a:Text>Spain-Germany</a:Text>
      </roomReservation>
    </ReserveRoom>
  </s:Body>
</s:Envelope>
```

标题是可选的，可以包含寻址、安全性和事务信息。正文包含消息数据。

43.1.2 WSDL

WSDL(Web Services Description Language，Web 服务描述语言)文档描述了服务的操作和消息。WSDL 定义了服务的元数据，这些元数据可用于为客户端应用程序创建代理。

WSDL 包含如下信息：

- 消息的类型——用 XML 架构描述。
- 从服务中收发的消息——消息的各部分是用 XML 架构定义的类型。
- 端口类型——映射服务协定，列出了用服务协定定义的操作。操作包含消息，例如，与请求和响应序列一起使用的输入和输出消息。
- 绑定信息——包含用端口类型列出的操作和用 SOAP 变体定义的操作。
- 服务信息——把端口类型映射到端点地址。

在 WCF 中，WSDL 信息由 MEX(Metedata Exchange，元数据交换)端点提供。

43.1.3 REST

WCF 还提供了使用 REST 进行通信的功能。REST 并不是一个协议，但定义了使用服务访问资源的几条规则。支持 REST 的 Web 服务是基于 HTTP 协议和 REST 规则的简单服务。规则按 3 个类别来定义：可以用简单的 URI 访问的服务，支持 MIME 类型，以及使用不同的 HTTP 方法。支持 MIME 类型，就可以从服务中返回不同的数据格式，如普通 XML、JSON 或 AtomPub。HTTP 请求的 GET()方法从服务中返回数据。其他方法有 PUT()、POST()和 DELETE()。PUT()方法用于更新服务端，POST()方法可创建一个新资源，DELETE()方法删除资源。

REST 允许给服务发送的请求比 SOAP 小。如果不需要 SOAP 提供的事务、安全消息(例如，安全通信仍可通过 HTTPS 进行)和可靠性，则利用 REST 构建的服务可以减小系统开销。

使用 REST 体系结构时，服务总是无状态的，服务的响应可以缓存。

43.1.4 JSON

除了发送 SOAP 消息之外，从 JavaScript 中访问服务最好使用 JSON。.NET 包含一个数据协定序列化程序，可以用 JSON 标记创建对象。

JSON 的系统开销比 SOAP 小，因为它不是 XML，而是为 JavaScript 客户端进行了优化。这使之非常适用于从 Ajax 客户端使用。Ajax 详见第 41 章。JSON 没有提供通过 SOAP 标题发送所具备的可靠性、安全性和事务功能，但这些通常是 JavaScript 客户端不需要的功能。

43.2 创建简单的服务和客户端

在详细介绍 WCF 之前，首先看一个简单的服务。该服务用于预订会议室。

要存储会议室预订信息，应使用一个简单的 SQL Server 数据库和 RoomReservations 表。可以从 www.wrox.com 中下载这个数据库和本章的示例代码。

下面是创建服务和客户端的步骤：

(1) 创建服务和数据协定。

(2) 使用 ADO.NET Entity Framework 创建访问数据库的库文件。

(3) 实现服务。

(4) 使用 WCF 服务宿主(Service Host)和 WCF 测试客户端(Test Client)。

(5) 创建定制的服务宿主。

(6) 使用元数据创建客户端应用程序。

(7)使用共享的协定创建客户端应用程序。

(8) 配置诊断设置。

43.2.1 定义服务和数据协定

首先，创建一个新的解决方案 RoomReservation，在其中添加一个新的类库项目，命名为 RoomReservationContracts。创建一个新类 RoomReservation(代码文件 RoomReservation/RoomReservation-Contracts/RoomReservation.cs)。这个类包含属性 Id、RoomName、StartTime、EndTime、Contact 和 Text，来定义数据库中需要的数据，并在网络中传送。要通过 WCF 服务发送数据，应给该类附加 DataContract 和 DataMember 属性。System.ComponentModel.DataAnnotations 名称空间中的 StringLength 属性不仅可用于验证用户输入，还可以在创建数据库表时定义列的模式。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Runtime.CompilerServices;
using System.Runtime.Serialization;

namespace Wrox.ProCSharp.WCF.Contracts
{
  [DataContract]
  public class RoomReservation : INotifyPropertyChanged
  {
    private int id;

    [DataMember]
    public int Id
    {
      get { return id; }
      set { SetProperty(ref id, value); }
    }

    private string roomName;

    [DataMember]
    [StringLength(30)]
    public string RoomName
    {
      get { return roomName; }
      set { SetProperty(ref roomName, value); }
    }

    private DateTime startTime;

    [DataMember]
    public DateTime StartTime
    {
      get { return startTime; }
```

```
      set { SetProperty(ref startTime, value); }
    }

    private DateTime endTime;

    [DataMember]
    public DateTime EndTime
    {
      get { return endTime; }
      set { SetProperty(ref endTime, value); }
    }

    private string contact;

    [DataMember]
    [StringLength(30)]
    public string Contact
    {
      get { return contact; }
      set { SetProperty(ref contact, value); }
    }

    private string text;

    [DataMember]
    [StringLength(50)]
    public string Text
    {
      get { return text; }
      set { SetProperty(ref text, value); }
    }

    protected virtual void OnNotifyPropertyChanged(string propertyName)
    {
      PropertyChangedEventHandler eventHandler = PropertyChanged;
      if (eventHandler != null)
      {
        eventHandler(this, new PropertyChangedEventArgs(propertyName));
      }
    }

    protected virtual void SetProperty<T>(ref T item, T value,
      [CallerMemberName] string propertyName = null)
    {
      if (!EqualityComparer<T>.Default.Equals(item, value))
      {
        item = value;
        OnNotifyPropertyChanged(propertyName);
      }
    }

    public event PropertyChangedEventHandler PropertyChanged;
  }
}
```

接着创建服务协定，服务提供的操作可以通过接口来定义。IRoomService 接口定义了 ReserveRoom()和 GetRoomReservations()方法。服务协定用 ServiceContract 属性定义。由服务定义的操作应用了 OperationContract 属性(代码文件 RoomReservation/RoomReservationContracts/IRoomService.cs)。

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF.Contracts
{
  [ServiceContract(Namespace= "http://www.cninnovation.com/RoomReservation/2012")]
  public interface IRoomService
  {
    [OperationContract]
    bool ReserveRoom(RoomReservation roomReservation);

    [OperationContract]
    RoomReservation[] GetRoomReservations(DateTime fromTime, DateTime toTime);
  }
}
```

43.2.2 数据访问

接着，创建一个库 RoomReservationData，来访问、读写数据库中的预订信息。这次使用 Code First 模型和 ADO.NET Entity Framework，这样就不需要映射信息，所有对象都可以用代码来定义。还可以在运行期间随时创建数据库。定义实体的类已经用 RoomReservationContracts 程序集定义好了，所以需要引用这个程序集。另外还需要 EntityFramework 程序集。现在可以创建 RoomReservationContext 类(代码文件 RoomReservationData/RoomReservationData/RoomReservationContext.cs)。这个类派生于基类 DbContext，用作 ADO.NET Entity Framework 的上下文，还定义了一个属性 RoomReservations，返回 DbSet<RoomReservation>。

```
using System.Data.Entity;
using Wrox.ProCSharp.WCF.Contracts;

namespace Wrox.ProCSharp.WCF.Data
{
  public class RoomReservationContext : DbContext
  {

    public RoomReservationContext()
      : base("name=RoomReservation")
    {

    }
    public DbSet<RoomReservation> RoomReservations { get; set; }
  }
}
```

在类的默认构造函数中，调用了基类构造函数，来传递 SQL 连接字符串名。用这种方式创建的数据库名不会自动映射上下文的名称。如果在启动应用程序前数据库不存在，就会在第一次使用上下文时自动创建它。接着配置需要连接字符串的宿主应用程序，如下所示。连接字符串使用 Microsoft SQL Server

Express LocalDB 数据库，Microsoft SQL Server Express 是 SQL Express 的一个新改进的版本，随 Visual Studio 2013 一起安装。

```
<connectionStrings>
  <add
    name="RoomReservation" providerName="System.Data.SqlClient"
    connectionString="Server=(localdb)\v11.0;Database=RoomReservation;
    Trusted_Connection=true;Integrated Security=True;
    MultipleActiveResultSets=True"/>
</connectionStrings>
```

服务实现代码使用的功能用 RoomReservationData 类定义(代码文件 RoomReservationData/RoomReservationData/RoomReservationData.cs)。ReserveRoom()方法将一条会议室预订记录写入数据库。GetReservations()方法返回指定时间段的 RoomReservation 集合。

```
using System;
using System.Linq;
using Wrox.ProCSharp.WCF.Contracts;

namespace Wrox.ProCSharp.WCF.Data
{
  public class RoomReservationData
  {
    public void ReserveRoom(RoomReservation roomReservation)
    {
      using (var data = new RoomReservationContext())
      {
        data.RoomReservations.Add(roomReservation);
        data.SaveChanges();
      }
    }

    public RoomReservation[] GetReservations(DateTime fromTime, DateTime toTime)
    {
      using (var data = new RoomReservationContext())
      {
        return (from r in data.RoomReservations
                where r.StartTime > fromTime && r.EndTime < toTime
                select r).ToArray();
      }
    }
  }
}
```

ADO.NET Entity Framework 详见第 33 章。

43.2.3　服务的实现

现在开始实现服务。创建一个 WCF 服务库 RoomReservationService。这个库默认包含服务协定和服务实现。如果客户端应用程序只使用元数据信息来创建访问服务的代理，这个模型就是可用的。

但是，如果客户端可能直接使用协定类型，则最好把协定放在一个独立的程序集中，如本例所示。在第一个已完成的客户端中，代理是从元数据中创建的。以后将介绍如何创建客户端，来共享协定程序集。把协定和实现分开，是共享协定的一个准备工作。

RoomReservationService 服务类实现 IRoomService 接口。实现服务时，只需要调用 RoomReservationData 类的相应方法(代码文件 RoomReservation/RoomReservationService/RoomReservationService.cs)。

```
using System;
using System.ServiceModel;
using Wrox.ProCSharp.WCF.Contracts;
using Wrox.ProCSharp.WCF.Data;

namespace Wrox.ProCSharp.WCF.Service
{
  [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
  public class RoomReservationService : IRoomService
  {
    public bool ReserveRoom(RoomReservation roomReservation)
    {
      var data = new RoomReservationData();
      data.ReserveRoom(roomReservation);
      return true;
    }

    public RoomReservation[] GetRoomReservations(DateTime fromTime,
        DateTime toTime)
    {
      var data = new RoomReservationData();
      return data.GetReservations(fromTime, toTime);
    }
  }
}
```

图 43-2 显示了前面创建的程序集及其依赖关系。RoomReservationContracts 程序集由 RoomReservationData 和 RoomReservationService 使用。

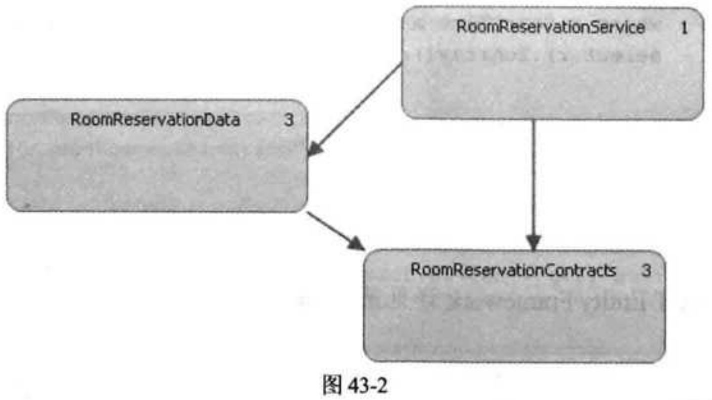


图 43-2

43.2.4 WCF 服务宿主和 WCF 测试客户端

WCF Service Library 项目模板创建了一个应用程序配置文件 App.config，它需要适用于新类名

和新接口名。service 元素引用了包含名称空间的服务类型 RoomReservationService，协定接口需要用 endpoint 元素定义(配置文件 RoomReservation/RoomReservationService/App.config)。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.Service.RoomService">
        <endpoint address="" binding="basicHttpBinding"
            contract="Wrox.ProCSharp.WCF.Service.IRoomService">
          <identity>
            <dns value="localhost" />
          </identity>
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding"
            contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress=
"http://localhost:8733/Design_Time_Addresses/RoomReservationService/Service1/"
            />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="True" httpsGetEnabled="True"/>
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

服务地址 http://localhost:8731/Design_Time_Addresses 有一个关联的访问控制列表(ACL)，它允许交互式用户创建一个侦听端口。默认情况下，非管理员用户不允许在监听模式下打开端口。使用命令行实用程序 netsh http show urlacl 可以查看 ACL，用 netsh http add url=http://+8080/ MyURI user=someUser 添加新项。

从 Visual Studio 2013 中启动这个库，会启动 WCF 服务宿主，它显示为任务栏的注意区域中的一个图标。单击这个图标会打开 WCF 服务宿主窗口，如图 43-3 所示。在其中可以查看服务的状态。WCF 库应用程序的项目属性包含 WCF 选项的选项卡，在其中可以选择运行同一个解决方案中的项目时，是否启动 WCF 服务宿主。默认打开这个选项。另外在项目属性的调试配置中，会发现已定

义了命令行参数/client:"WcfTestClient.exe"。WCF 服务主机使用这个选项，会启动 WCF 测试客户端，如图 43-4 所示，该测试客户端可用于测试应用程序。双击一个操作，输入字段就会显示在应用程序的右边，可以在其中填充要发送给服务的数据。单击 XML 选项卡，可以看到已收发的 SOAP 消息。

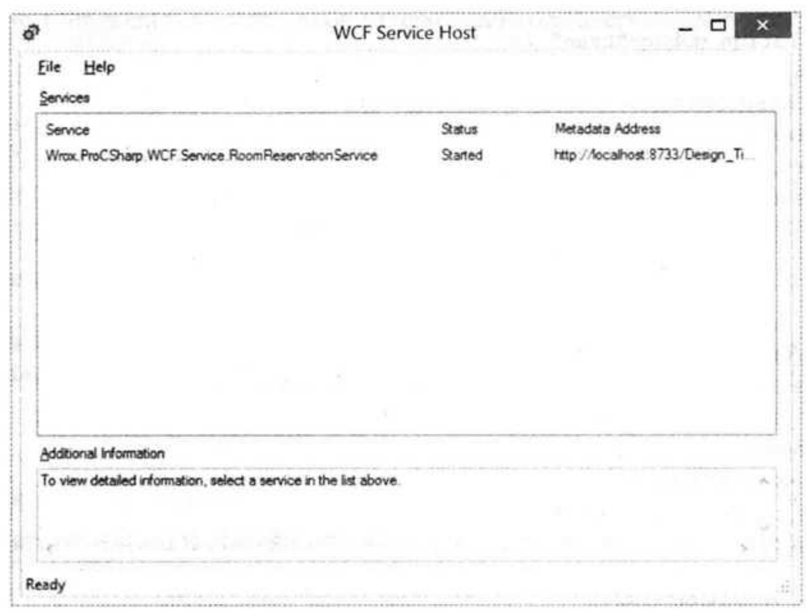


图 43-3

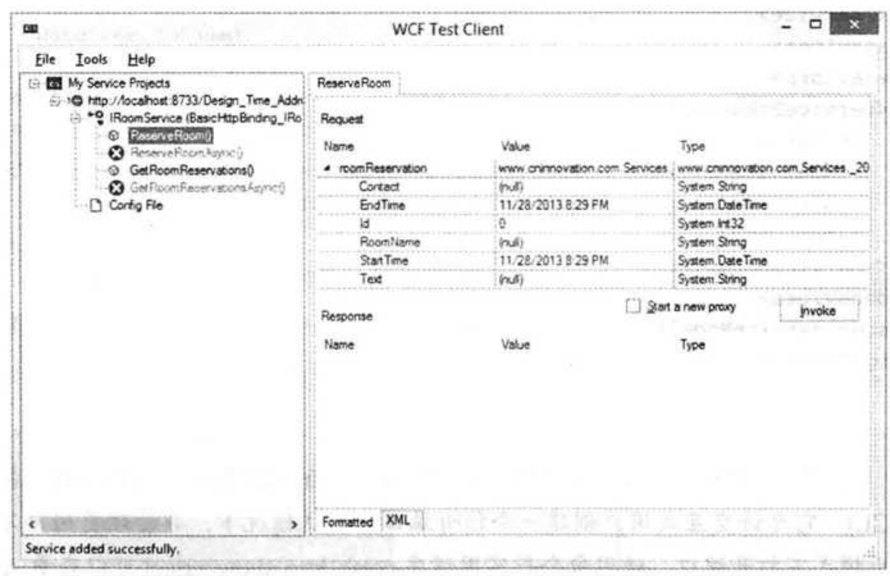


图 43-4

43.2.5　自定义服务宿主

使用 WCF 可以在任意宿主上运行服务。可以为对等服务创建一个 Windows 窗体或 WPF 应用程序，或创建一个 Windows 服务，或用 Windows Activation Services(WAS)或 Internet Information Services(IIS)存放该服务。控制台应用程序也适合于演示简单的主机。

对于服务主机，必须引用 RoomReservationService 库和 System.ServiceModel 程序集。该服务首

先实例化和打开 ServiceHost 类型的对象。这个类在 System.ServiceModel 名称空间中定义。实现该服务的 RoomReservationService 类在构造函数中定义。调用 Open()方法会启动服务的监听器信道，该服务准备用于侦听请求。Close()方法会停止信道。下面的代码段还添加了 ServiceMetadataBehavior 类型的一个操作，添加该操作，就允许使用 WSDL 创建一个客户端应用程序(代码文件 RoomReservation/RoomReservationServiceHost/ Program.cs)：

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using Wrox.ProCSharp.WCF.Service;

namespace Wrox.ProCSharp.WCF.Host
{
  class Program
  {
    internal static ServiceHost myServiceHost = null;

    internal static void StartService()
    {
      try
      {
        myServiceHost = new ServiceHost(typeof(RoomReservationService),
          new Uri("http://localhost:9000/RoomReservation"));
        myServiceHost.Description.Behaviors.Add(new ServiceMetadataBehavior
        { HttpGetEnabled = true });
        myServiceHost.Open();
      }
      catch (AddressAccessDeniedException)
      {
        Console.WriteLine("either start Visual Studio in elevated admin " +
            "mode or register the listener port with netsh.exe");
      }
    }

    internal static void StopService()
    {
      if (myServiceHost != null &&
          myServiceHost.State == CommunicationState.Opened)
      {
        myServiceHost.Close();
      }
    }

    static void Main()
    {
      StartService();

      Console.WriteLine("Server is running. Press return to exit");
      Console.ReadLine();

      StopService();
    }
  }
}
```

对于 WCF 配置，需要把用服务库创建的应用程序配置文件复制到宿主应用程序中。使用 WCF Service Configuration Editor 可以编辑这个配置文件，如图 43-5 所示。

除了使用配置文件之外，还可以通过编程方式配置所有内容，并使用几个默认值。宿主应用程序的示例代码不需要任何配置文件。ServiceHost 构造函数的第二个参数定义了服务的基地址。通过这个基地址的协议，来定义默认绑定。表示 HTTP 的默认值是 BasicHttpBinding。

使用自定义服务宿主，可以在 WCF 库的项目设置中取消用来启动 WCF 服务宿主的 WCF 选项。

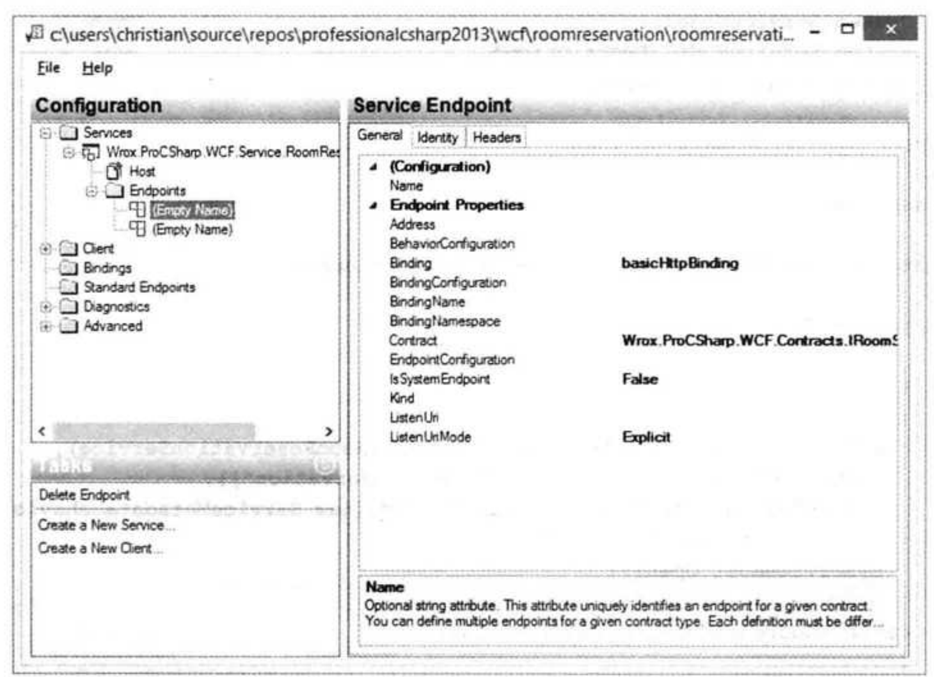


图 43-5

43.2.6 WCF 客户端

对于客户端，WCF 也可以灵活选择所使用的应用程序类型。客户端也可以是一个简单的控制台应用程序。但是，对于预订会议室，应创建一个包含控件的 Windows 窗体应用程序，如图 43-6 所示。

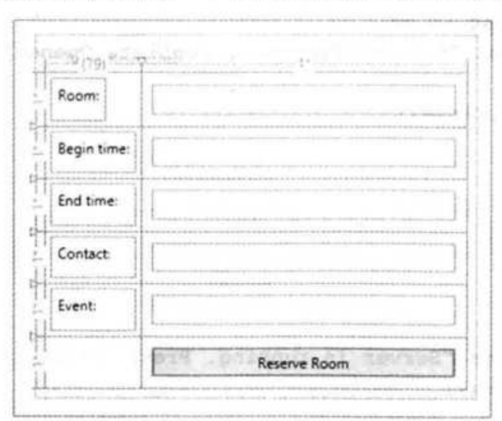


图 43-6

因为服务宿主用 ServiceMetadataBehavior 配置，所以它提供了一个 MEX 端点。启动服务宿主后，就可以在 Visual Studio 中添加一个服务引用。在添加服务引用时，会弹出如图 43-7 所示的对话

框。用 URL http://localhost:9000/ RoomReservation?wsdl 进入服务元数据的连接，把名称空间设置为 RoomReservationService。这将为生成的代理类定义名称空间。

图 43-7

添加服务引用，会在服务中添加对 System.Runtime.Serialization 和 System.ServiceModel 程序集的引用，还会添加一个包含绑定信息和服务端点地址的配置文件。

从数据协定中把 RoomReservation 生成为一个部分类。这个类包含协定的所有[DataMember]元素。RoomServiceClient 类是客户端的代理，该客户端包含由服务协定定义的方法。使用这个客户端，可以将会议室预订信息发送给正在运行的服务。

在代码文件 RoomReservation/RoomReservationClient/MainWindow.xaml.cs 中，通过按钮的 Click 事件调用 ReserveRoomAsync 方法。通过服务代理调用 ReserveRoomAsync。reservation 变量通过数据绑定接收 UI 的数据。

```
public partial class MainWindow : Window
{
  private RoomReservation reservation;
  public MainWindow()
  {
    InitializeComponent();
    reservation = new RoomReservation
    { StartTime = DateTime.Now, EndTime = DateTime.Now.AddHours(1) };
    this.DataContext = reservation;
  }

  private async void OnReserveRoom(object sender, RoutedEventArgs e)
  {
    var client = new RoomServiceClient();
    bool reserved = await client.ReserveRoomAsync(reservation);
```

```
        client.Close();

        if (reserved)
          MessageBox.Show("reservation ok");
      }
    }
```

运行服务和客户端，创建数据库后，就可以将会议室预订信息添加到数据库中。在 RoomReservation 解决方案的设置中，可以配置多个启动项目，在本例中是 RoomReservationClient 和 RoomReservationHost。

43.2.7 诊断

运行客户端和服务应用程序时，知道后台发生了什么很有帮助。为此，WCF 使用一个需要配置的跟踪源。可以使用 Service Configuration Editor，选择 Diagnostics 节点，启用 Tracing and Message Logging 功能来配置跟踪。把跟踪源的跟踪级别设置为 Verbose 会生成非常详细的信息。这个配置更改把跟踪源和监听器添加到应用程序配置文件中，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add
      name="RoomReservation" providerName="System.Data.SqlClient"
      connectionString="Server=(localdb)\v11.0;Database=RoomReservation;
      Trusted_Connection=true;Integrated Security=True;
      MultipleActiveResultSets=True" />
  </connectionStrings>

  <system.diagnostics>
    <sources>
      <source name="System.ServiceModel.MessageLogging"
        switchValue="Verbose,ActivityTracing">
        <listeners>
          <add type="System.Diagnostics.DefaultTraceListener" name="Default">
            <filter type="" />
          </add>
          <add name="ServiceModelMessageLoggingListener">
            <filter type="" />
          </add>
        </listeners>
      </source>
      <source propagateActivity="true" name="System.ServiceModel"
        switchValue="Warning,ActivityTracing">
        <listeners>
          <add type="System.Diagnostics.DefaultTraceListener" name="Default">
            <filter type="" />
          </add>
          <add name="ServiceModelTraceListener">
            <filter type="" />
          </add>
```

```
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add initializeData=
        "c:\code\wcf\roomreservation\roomreservationhost\app_messages.svclog"
        type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
        name="ServiceModelMessageLoggingListener"
        traceOutputOptions="DateTime, Timestamp, ProcessId, ThreadId">
        <filter type="" />
      </add>
      <add initializeData=
        "c:\code\wcf\roomreservation\roomreservationhost\app_tracelog.svclog"
        type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
        name="ServiceModelTraceListener"
        traceOutputOptions="DateTime, Timestamp, ProcessId, ThreadId">
        <filter type="" />
      </add>
    </sharedListeners>
  </system.diagnostics>
  <startup>
   <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <system.serviceModel>
    <diagnostics>
      <messageLogging logEntireMessage="true" logMalformedMessages="true"
       logMessagesAtTransportLevel="true" />
      <endToEndTracing propagateActivity="true" activityTracing="true"
       messageFlowTracing="true" />
    </diagnostics>
  </system.serviceModel>
</configuration>
```

WCF 类的实现代码使用 System.ServiceModel 和 System.ServiceModel.MessageLogging 跟踪源来写入跟踪消息。跟踪和配置跟踪源及监听器的更多内容详见第 20 章。

启动应用程序时，使用 verbose 跟踪设置的跟踪文件会很快变得很大。为了分析 XML 日志文件中的信息，.NET SDK 包含一个 Service Trace Viewer 工具 svctraceviewer.exe。图 43-8 显示了输入一些数据的客户端应用程序，图 43-9 显示了这个工具选择跟踪和消息日志文件后的视图。BasicHttpBinding 用传送来的信息突出显示。如果把配置改为使用 WsHttpBinding，就会看到许多

消息都与安全性相关。根据安全性需求，可以选择其他配置选项。

图 43-8

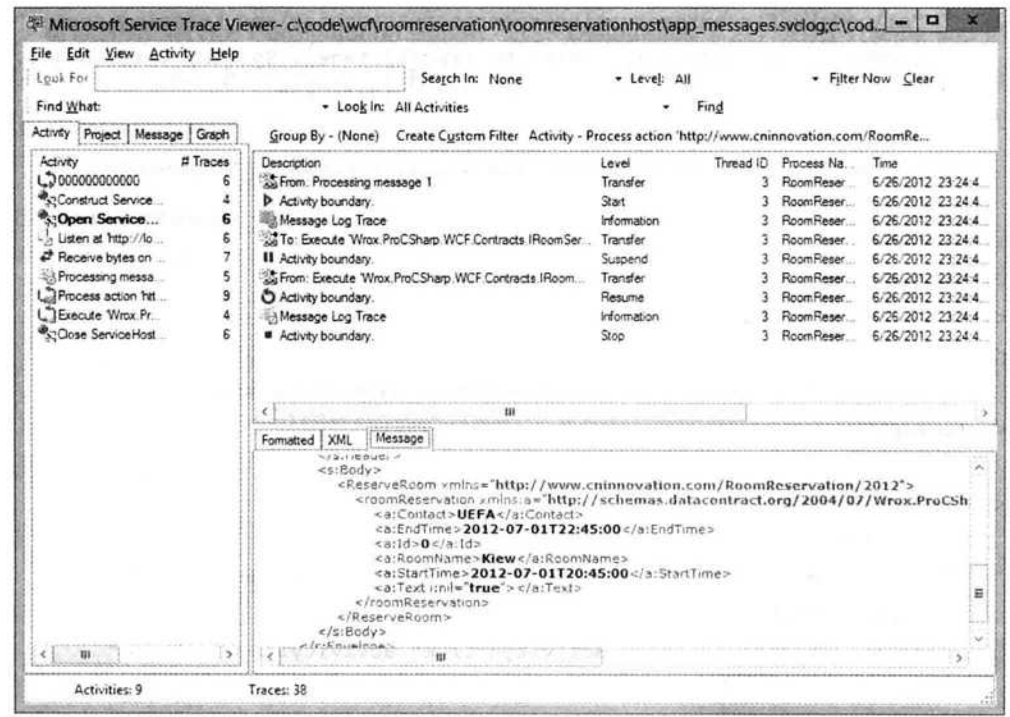


图 43-9

下面详细介绍 WCF 的细节和不同的选项。

43.2.8 与客户端共享协定程序集

在前面的 WPF 客户端应用程序中，使用元数据创建了一个代理类，用 Visual Studio 添加了一个服务引用。客户端也可以用共享的协定程序集来创建，如下所示。使用协定接口和 ChannelFactory<TChannel>来实例化连接到服务上的通道。

类 ChannelFactory<TChannel>的构造函数接受绑定配置和端点地址作为参数。绑定必须与服务宿主定义的绑定兼容，用 EndPointAddress 类定义的地址引用了当前运行的服务的 URI。CreateChannel 方法创建了一个连接到服务上的通道，接着就可以调用服务的方法了。

```
using System;
using System.ServiceModel;
using System.Windows;
using Wrox.ProCSharp.WCF.Contracts;

namespace RoomReservationClientSharedAssembly
{
  /// <summary>
  /// Interaction logic for MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    private RoomReservation roomReservation;
    public MainWindow()
    {
      InitializeComponent();
      roomReservation = new RoomReservation
      {
        StartTime = DateTime.Now,
        EndTime = DateTime.Now.AddHours(1)
      };
      this.DataContext = roomReservation;
    }

    private void OnReserveRoom(object sender, RoutedEventArgs e)
    {
      var binding = new BasicHttpBinding();
      var address = new EndpointAddress("http://localhost:9000/RoomReservation");

      var factory = new ChannelFactory<IRoomService>(binding, address);
      IRoomService channel = factory.CreateChannel();
      if (channel.ReserveRoom(roomReservation))
      {
       MessageBox.Show("success");
      }
    }
  }
}
```

43.3　协定

协定定义了服务提供的功能和客户端可以使用的功能。协定可以完全独立于服务的实现代码。

由 WCF 定义的协定可以分为 4 种不同的类型：数据协定、服务协定、消息协定和错误协定。协定可以用.NET 属性来指定：

- 数据协定——数据协定定义了从服务中接收和返回的数据。用于收发消息的类关联了数据协定属性。
- 服务协定——服务协定用于定义描述了服务的 WSDL。这个协定用接口或类定义。
- 操作协定——操作协定定义了服务的操作，在服务协定中定义。

- 消息协定——如果需要完全控制 SOAP 消息，消息协定就可以指定应放在 SOAP 标题中的数据以及放在 SOAP 正文中的数据。
- 错误协定——错误协定定义了发送给客户端的错误消息。

下面几节将详细探讨这些协定类型，并进一步讨论定义协定时应考虑的版本问题。

43.3.1 数据协定

在数据协定中，把 CLR 类型映射到 XML 架构。数据协定不同于其他.NET 序列化机制。在运行库序列化中，所有字段都会序列化(包括私有字段)，而在 XML 序列化中，只序列化公共字段和属性。数据协定要求用 DataMember 特性显式标记要序列化的字段。无论字段是私有或公共的，还是应用于属性，都可以使用这个特性。

```
[DataContract(Namespace="http://www.cninnovation.com/Services/20102"]
public class RoomReservation
{
    [DataMember] public string Room { get; set; }
    [DataMember] public DateTime StartTime { get; set; }
    [DataMember] public DateTime EndTime { get; set; }
    [DataMember] public string Contact { get; set; }
    [DataMember] public string Text { get; set; }
}
```

为了独立于平台和版本，如果要求用新版本修改数据，且不破坏旧客户端和服务，使用数据协定是指定要发送哪些数据的最佳方式。还可以使用 XML 序列化和运行库序列化。XML 序列化是 ASP.NET Web 服务使用的机制，.NET Remoting 使用运行库序列化。

使用 DataMember 特性，可以指定表 43-1 中的属性。

表 43-1

用 DataMember 指定的属性	说　明
Name	序列化元素的名称默认与应用了[DataMember]特性的字段或属性同名。使用 Name 属性可以修改该名称
Order	Order 属性指定了数据成员的序列化顺序
IsRequired	使用 IsRequired 属性，可以指定元素必须经过序列化，才能接收。这个属性可以用于解决版本问题。如果在已有的协定中添加了成员，协定不会被破坏，因为在默认情况下字段是可选的(IsRequired=false)。将 IsRequired 属性设置为 true，就可以破坏已有的协定
EmitDefaultValue	EmitDefaultValue 属性指定有默认值的成员是否应序列化。如果把 EmitDefaultValue 属性设置为 true，具有该类型的默认值的成员就不序列化

43.3.2 版本问题

创建数据协定的新版本时，注意更改的种类，如果应同时支持新旧客户端和新旧服务，就应执行相应的操作。

在定义协定时，应使用 DataContractAttribute 的 Namespace 属性添加 XML 名称空间信息。如果创建了数据协定的新版本，破坏了兼容性，就应改变这个名称空间。如果只添加了可选的成员，就

没有破坏协定——这就是一个可兼容的改变。旧客户端仍可以给新服务发送消息，因为不需要其他数据。新客户端可以给旧服务发送消息，因为旧服务仅忽略额外的数据。

删除字段或添加需要的字段会破坏协定。此时还应改变 XML 名称空间。名称空间的名称可以包含年份和月份，如 http://thinktecture.com/SampleServices/2012/08。每次做了破坏性的修改时，都要改变名称空间，如把年份和月份改为实际值。

43.3.3 服务协定

服务协定定义了服务可以执行的操作。ServiceContract 特性与接口或类一起使用，来定义服务协定。由服务提供的方法通过 IRoomService 接口应用 OperationContract 特性，如下所示：

```
[ServiceContract]
public interface IRoomService
{
  [OperationContract]
  bool ReserveRoom(RoomReservation roomReservation);
}
```

可能用 ServiceContract 特性设置的属性如表 43-2 所示。

表 43-2

用 ServiceContract 设置的属性	说　明
ConfigurationName	这个属性定义了配置文件中服务配置的名称
CallbackContract	当服务用于双工消息传递时，CallbackContract 属性定义了在客户端中实现的协定
Name	这个 Name 属性定义了 WSDL 中<portType>元素的名称
Namespace	Namespace 属性定义了 WSDL 中<portType>元素的 XML 名称空间
SessionMode	使用 SessionMode 属性，可以定义调用这个协定的操作所需的会话。其值用 SessionMode 枚举定义，包括 Allowed、NotAllowed 和 Required
ProtectionLevel	ProtectionLevel 属性确定了绑定是否必须支持保护通信。其值用 ProtectionLevel 枚举定义，包括 None、Sign、EncryptAndSign

使用 OperationContract 特性可以指定如表 43-3 所示的属性。

表 43-3

用 OperationContract 指定的属性	说　明
Action	WCF 使用 SOAP 请求的 Action 属性，把该请求映射到相应的方法上。Action 属性的默认值是协定 XML 名称空间、协定名和操作名的组合。该消息如果是一条响应消息，就把 Response 添加到 Action 字符串中。指定 Action 属性可以重写 Action 值。如果指定值“*”，服务操作就会处理所有消息
ReplyAction	Action 属性设置了入站 SOAP 请求的 Action 名，而 ReplyAction 属性设置了回应消息的 Action 名
AsyncPattern	如果使用异步模式来实现操作，就把 AsyncPattern 属性设置为 true。异步模式详见第 21 章

(续表)

用 OperationContract 指定的属性	说 明
IsInitiating IsTerminating	如果协定由一系列操作组成，且初始化操作本应把 IsInitiating 属性赋予它，该系列的最后一个操作就需要指定 IsTerminating 属性。初始化操作启动一个新会话，服务器用终止操作来关闭会话
IsOneWay	设置 IsOneWay 属性，客户端就不会等待回应消息。在发送请求消息后，单向操作的调用者无法直接检测失败
Name	操作的默认名称是指定了操作协定的方法名。使用 Name 属性可以修改该操作的名称
ProtectionLevel	使用 ProtectionLevel 属性可以确定消息是应只签名，还是应加密后签名

在服务协定中，也可以用[DeliveryRequirements]特性定义服务的传输要求。RequireOrderedDelivery 属性指定所发送的消息必须以相同的顺序到达。使用 QueuedDeliveryRequirements 属性可以指定，消息以断开连接的模式发送，例如，使用消息队列(参见第 47 章)。

43.3.4 消息协定

如果需要完全控制 SOAP 消息，就可以使用消息协定。在消息协定中，可以指定消息的哪些部分要放在 SOAP 标题中，哪些部分要放在 SOAP 正文中。下面的例子显示了 ProcessPersonRequestMessage 类的一个消息协定。该消息协定用 MessageContract 特性指定。SOAP 消息的标题和正文用 MessageHeader 和 MessageBodyMember 属性指定。指定 Position 属性，可以确定正文中的元素顺序。还可以为标题和正文字段指定保护级别。

```
[MessageContract]
public class ProcessPersonRequestMessage
{
  [MessageHeader]
  public int employeeId;

  [MessageBodyMember(Position=0)]
  public Person person;
}
```

ProcessPersonRequestMessage 类与用 IProcessPerson 接口定义的服务协定一起使用：

```
[ServiceContract]
public interface IProcessPerson
{
  [OperationContract]
  public PersonResponseMessage ProcessPerson(ProcessPersonRequestMessage message);
}
```

与 WCF 服务相关的另一个重要协定是错误协定，这个协定参见 43.4.2 节。

43.3.5 错误协定

默认情况下，在服务中出现的详细异常消息不返回客户端应用程序。其原因是安全性。不应把详细的异常消息提供给使用服务的第三方。而异常应记录到服务上(为此可以使用跟踪和事件日志功

能)，包含有用信息的错误应返回调用者。

可以抛出一个 FaultException 异常，来返回 SOAP 错误。抛出 FaultException 异常会创建一个非类型化的 SOAP 错误。返回错误的首选方式是生成强类型化的 SOAP 错误。

应与强类型化的 SOAP 错误一起传递的信息用数据协定定义，如下面的 StateFault 类所示(代码文件 RoomReservation/RoomReservationContracts/RoomReservationFault.cs)：

```
[DataContract]
public class RoomReservationFault
{
  [DataMember]
  public string Message { get; set; }
}
```

SOAP 错误的类型必须用 FaultContractAttribute 和操作协定定义：

```
[FaultContract(typeof(RoomReservationFault))]
[OperationContract]
bool ReserveRoom(RoomReservation roomReservation);
```

在实现代码中，抛出一个 FaultException<TDetail>异常。在构造函数中，可以指定一个新的 TDetail 对象，在本例中就是 StateFault。另外，FaultReason 中的错误信息可以赋予构造函数。FaultReason 支持多种语言的错误信息。

```
FaultReasonText[] text = new FaultReasonText[2];
text[0] = new FaultReasonText("Sample Error", new CultureInfo("en"));
text[1] = new FaultReasonText("Beispiel Fehler", new CultureInfo("de"));
FaultReason reason = new FaultReason(text);

throw new FaultException<RoomReservationFault>(
  new RoomReservationFault() { Message = m }, reason);
```

在客户端应用程序中，可以捕获 FaultException<RoomReservationFault>类型的异常。出现该异常的原因由 Message 属性定义。RoomReservationFault 用 Detail 属性访问。

```
try
{
  //...
}
catch (FaultException<RoomReservationFault> ex)
{
  Console.WriteLine(ex.Message);
  StateFault detail = ex.Detail;
  Console.WriteLine(detail.Message);
}
```

除了捕获强类型化的 SOAP 错误之外，客户端应用程序还可以捕获 FaultException<Detail>的基类的异常：FaultException 异常和 CommunicationException 异常。捕获 CommunicationException 异常还可以捕获与 WCF 通信相关的其他异常。

在开发过程中，可以把异常返回给客户端。为了传播异常，需要使用 serviceDebug 元素配置一个服务行为配置。serviceDebug 元素的 IncludeExceptionDetailInFaults 特性可以设置为 true，来返回异常信息。

43.4 服务的行为

服务的实现代码用 ServiceBehavior 特性标记，如下面的 RoomReservationService 类所示：

```
[ServiceBehavior]
public class RoomReservationService: IRoomService
{
  public bool ReserveRoom(RoomReservation roomReservation)
  {
    // implementation
  }
}
```

ServiceBehavior 特性用于描述 WCF 服务提供的操作，以截获所需功能的代码，如表 43-4 所示。

表 43-4

用 ServiceBehavior 指定的属性	说　明
TransactionAutoComplete-OnSessionClose	当前会话正确完成时，就自动提交该事务。这类似于 Enterprise Services 中的 AutoComplete 属性
TransactionIsolationLevel	要定义服务中事务的隔离级别，可以把 TransactionIsolationLevel 属性设置为 IsolationLevel 枚举中的一个值。事务信息的隔离级别详见第 25 章
ReleaseServiceInstanceOn-TransactionComplete	完成事务处理后，可回收服务的实例
AutomaticSessionShutdown	如果在客户端关闭连接时没有关闭会话，就可以把 AutomaticSessionShutdown 属性设置为 false。在默认情况下，会关闭会话
InstanceContextMode	使用 InstanceContextMode 属性，可以确定应使用有状态的对象还是无状态的对象。默认设置为 InstanceContextMode.PerCall，为每个方法调用创建一个新对象。其他可能的设置有 PerSession 和 Single。这两个设置都使用有状态的对象。但是，PerSession 会为每个客户端创建一个新对象，而 Single 允许在多个客户端上共享同一个对象
ConcurrencyMode	因为有状态的对象可以由多个客户端(或同一个客户的多个线程)使用，所以必须注意这种对象类型的并发问题。如果把 ConcurrencyMode 属性设置为 Multiple，多个线程就可以访问对象，但必须处理同步问题。如果把该属性设置为 Single，一次就只有一个线程能访问对象，但不必处理同步问题；如果客户端较多，就可能出现可伸缩性问题。Reentrant 值表示只有从调用返回的线程才能访问对象。对于无状态的对象，这个设置没有任何意义，因为每个方法调用都会实例化一个新对象，所以不共享状态

(续表)

用 ServiceBehavior 指定的属性	说　明
UseSynchronizationContext	Windows 窗体和 WPF 控件的成员都只能从创建线程中调用。如果服务位于 Windows 应用程序中，其服务方法调用了控件成员，就把 UseSynchronizationContext 属性设置为 true。这样，服务就运行在 SynchronizationContext 属性定义的线程中
IncludeExceptionDetailInFaults	在.NET 中，错误被看作异常。SOAP 指定，SOAP 错误返回客户端，以防服务器出问题。出于安全考虑，最好不要把服务器端异常的细节返回客户端。因此，异常默认转换为未知错误。要返回特定的错误，可抛出 FaultException 类型的异常。为了便于调试，返回真实的异常信息很有帮助。此时应把 IncludeExceptionDetailInFaults 属性的设置改为 true。这里抛出 FaultException <TDetail>异常，其中原始异常包含详细信息
MaxItemsInObjectGraph	使用 MaxItemsInObjectGraph 属性，可以限制要序列化的对象数。如果序列化一个对象树型结构，则默认的限制过低
ValidateMustUnderstand	把 ValidateMustUnderstand 属性设置为 true，表示必须理解 SOAP 标题(默认)

为了演示服务行为，IStateService 接口定义了一个服务协定，其中的两个操作用于获取和设置状态。有状态的服务协定需要一个会话。这就是把服务协定的 SessionMode 属性设置为 SessionMode.Required 的原因。服务协定还将 IsInitiating 和 IsTerminating 特性应用于操作协定，以定义了启动和关闭会话的方法。

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IStateService
{
  [OperationContract(IsInitiating=true)]
  void Init(int i);

  [OperationContract]
  void SetState(int i);

  [OperationContract]
  int GetState();

  [OperationContract(IsTerminating=true)]
  void Close();
}
```

服务协定由 StateService 类实现。服务的实现代码定义了 InstanceContextMode.PerSession，使状态与实例保持同步。

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class StateService: IStateService
{
  int i = 0;

  public void Init(int i)
  {
    this.i = i;
```

```
    }

    public void SetState(int i)
    {
      this.i = i;
    }

    public int GetState()
    {
      return i;
    }

    public void Close()
    {
    }
  }
```

现在必须定义对地址和协议的绑定。其中，将basicHttpBinding赋予服务的端点：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="StateServiceSample.Service1Behavior"
        name="Wrox.ProCSharp.WCF.StateService">
        <endpoint address="" binding="basicHttpBinding"
            bindingConfiguration=""
            contract="Wrox.ProCSharp.WCF.IStateService">
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding"
             contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8731/Design_Time_Addresses/
                                  StateServiceSample/Service1/" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="StateServiceSample.Service1Behavior">
          <serviceMetadata httpGetEnabled="True"/>
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

如果用定义的配置启动服务宿主，就会抛出一个InvalidOperationException类型的异常。该异常的错误消息是“协定需要会话，但绑定‘BasicHttpBinding’不支持它，或者没有正确配置为支持它。”

并不是所有绑定都支持所有服务。因为服务协定需要用[ServiceContract(ServiceMode=

ServiceMode.Required)]特性指定一个会话，主机会失败，这是因为所配置的绑定不支持会话。

只要修改对绑定的配置，使之支持会话(如 wsHttpBinding)，服务器就会成功启动。

```
<endpoint address="" binding="wsHttpBinding"
  bindingConfiguration=""
  contract="Wrox.ProCSharp.WCF.IStateService">
</endpoint>
```

在服务的实现代码中，可以通过 OperationBehavior 特性将服务方法应用于如表 43-5 所示的属性。

表 43-5

通过 OperationBehavior 应用的属性	说　明
AutoDisposeParameters	默认情况下，所有可释放的参数都自动释放。如果参数不应释放，就可以把 AutoDisposeParameters 属性设置为 false。接着，发送方将负责释放该参数
Impersonation	使用 Impersonation 属性，可以模拟调用者，以调用者的身份运行方法
ReleaseInstanceMode	InstanceContextMode 使用服务行为设置定义对象实例的生命周期。使用操作行为设置，可以根据操作重写设置。ReleaseInstanceMode 用 ReleaseInstanceMode 枚举定义实例发布模式。其 None 值使用 InstanceContextMode 设置。BeforeCall、AfterCall 和 BeforeAndAfterCall 值定义了操作的循环时间
TransactionScopeRequired	使用 TransactionScopeRequired 属性可以指定操作是否需要一个事务。如果需要一个事务，且调用者已经发出一个事务，就使用同一个事务。如果调用者没有发出事务，就创建一个新的事务
TransactionAutoComplete	TransactionAutoComplete 属性指定事务是否自动完成。如果把该属性设置为 true，在抛出异常的情况下就终止事务。如果这是一个根事务，且没有抛出异常，就提交事务

43.5　绑定

绑定描述了服务的通信方式。使用绑定可以指定如下特性：

- 传输协议
- 安全性
- 编码格式
- 事务流
- 可靠性
- 形状变化
- 传输升级

43.5.1　标准的绑定

绑定包含多个绑定元素，它们描述了所有绑定要求。可以创建自定义绑定，也可以使用表 43-6

中的某个预定义绑定：

表43-6

标准绑定	说　明
BasicHttpBinding	BasicHttpBinding 绑定用于最广泛的交互操作的第一代 Web 服务。所使用的传输协议是 HTTP 或 HTTPS，其安全性仅由传输协议保证
WSHttpBinding	WSHttpBinding 绑定用于下一代 Web 服务，用 SOAP 扩展确保安全、可靠性和事务处理的平台。所使用的传输协议是 HTTP 或 HTTPS，为了确保安全，实现了 WS-Security 规范的安全性。使用 WS-Coordination、WS-AtomicTransaction 和 WS-BusinessActivity 规范支持事务，通过 WS-ReliableMessaging 的实现方式支持可靠的消息传送。WS-Profile 也支持用于发送附件的 MTOM(Message Transmission Optimization Protocol，消息传输优化协议)编码。WS-*标准的规范可参见 http://www.oasis-open.org
WS2007HttpBinding	WS2007HttpBinding 派生自基类 WSHttpBinding，支持 OASIS(Organization for the Advancement of Structured Information Standards，结构化信息标准促进组织)定义的安全性、可靠性和事务规范。这个类提供了更新的 SOAP 标准
WSHttpContextBinding	WSHttpContextBinding 派生自基类 WSHttpBinding，开始支持没有使用 cookie 的上下文。这个绑定会添加 ContextBindingElement，交换上下文信息。Workflow Foundation 3.0 需要 ContextBindingElement
WebHttpBinding	这个绑定用于通过 HTTP 请求(而不是 SOAP 请求)提供的服务，它对于脚本客户端很有用，如 ASP.NET AJAX
WSFederationHttpBinding	WSFederationHttpBinding 是一种安全、可交互操作的绑定，支持在多个系统上共享身份，以进行身份验证和授权
WSDualHttpBinding	与 WSHttpBinding 相反，WSDualHttpBinding 绑定支持双工的消息传送
NetTcpBinding	所有用 Net 作为前缀的标准绑定都使用二进制编码在.NET 应用程序之间通信。这个编码比 WSxxx 绑定使用的文本编码快。NetTcpBinding 绑定使用 TCP/IP 协议
NetTcpContextBinding	类似于 WSHttpContextBinding，NetTcpContextBinding 会添加 ContextBindingElement，与 SOAP 标题交换上下文信息
NetHttpBinding	这是.NET 4.5 新增的绑定，支持 WebSocket 传输协议
NetPeerTcpBinding	NetPeerTcpBinding 为对等通信提供绑定
NetNamedPipeBinding	NetNamedPipeBinding 为同一系统上的不同进程之间的通信进行了优化
NetMsmqBinding	NetMsmqBinding 为 WCF 引入了排队通信。这里消息会发送到消息队列中
MsmqIntegrationBinding	MsmqIntegrationBinding 是用于使用消息队列的已有应用程序的绑定。而 NetMsmqBinding 绑定需要位于客户端和服务器上的 WCF 应用程序
CustomBinding	使用 CustomBinding，可以完全定制传输协议和安全要求

43.5.2　标准绑定的特性

不同的绑定支持不同的功能。以 WS 开头的绑定独立于平台，支持 Web 服务规范。以 Net 开头的绑定使用二进制格式，使.NET 应用程序之间的通信有很高的性能。新的 NetHttpBinding 修改了命名约定，因为它不需要.NET 应用程序存在于线的两端，它基于 Web 套接字标准。

其他功能支持会话、可靠的会话、事务和双工通信。表 43-7 列出了支持这些功能的绑定。

表 43-7

功　能	绑　定
会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding
可靠的会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding
事务	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetMsmqBinding、MsmqIntegrationBinding
双工通信	WSDualHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetPeerTcpBinding

除了定义绑定之外，服务还必须定义端点。端点依赖于协定、服务的地址和绑定。在下面的代码示例中，实例化了一个 ServiceHost 对象，将地址 http://localhost:8080/RoomReservation、一个 WSHttpBinding 实例和协定添加到服务的一个端点上。

```
static ServiceHost host;

static void StartService()
{
  var baseAddress = new Uri("http://localhost:8080/RoomReservation");
  host = new ServiceHost(typeof(RoomReservationService));

  var binding1 = new WSHttpBinding();
  host.AddServiceEndpoint(typeof(IRoomService), binding1, baseAddress);
  host.Open();
}
```

除了以编程方式定义绑定之外，还可以在应用程序配置文件中定义它。WCF 的配置放在<system.serviceModel>元素中，<service>元素定义了所提供的服务。同样，如代码所示，服务需要一个端点，该端点包含地址、绑定和协定信息。WSHttpBinding 的默认绑定配置用 XML 属性 bindingConfiguration 修改，该属性引用了绑定配置 WSHttpConfig1。这个绑定配置在<bindings>段中，它用于修改 WSHttpBinding 配置，以启用 reliableSession。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address=" http://localhost:8080/RoomReservation"
            contract="Wrox.ProCSharp.WCF.IRoomService"
            binding="wsHttpBinding" bindingConfiguration="wsHttpBinding" />
```

```
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
        <binding name="wsHttpBinding">
          <reliableSession enabled="true" />
        </binding>
      </wsHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

43.5.3 Web 套接字

WebSocket 是基于 TCP 的一个新通信协议。HTTP 协议是无状态的。服务器利用 HTTP，可以在每次回应请求后关闭连接。如果客户端需要从服务器连续接收信息，使用 HTTP 协议就总是会有一些问题。

因为 HTTP 连接是保存的，所以解决这个问题的一种方式是让一个服务运行在客户端上，服务器连接到该客户端，并发送回应。如果在客户端和服务器之间有防火墙，这种方式通常无效，因为防火墙阻塞了入站的请求。

解决这个问题的另一种方式是使用另一个协议替代 HTTP 协议。这样连接可以保持活跃。使用其他协议的问题是端口需要用防火墙打开。防火墙总是一个问题，但需要防火墙来禁止坏人进入。

这个问题的通常解决方法是每次都实例化来自客户端的请求。客户端向服务器询问，是否有新的信息。这是有效的，但其缺点是要么客户端询问了很多次，都没有得到新信息，因此增加了网络流量，要么客户端获得了旧信息。

新的解决方案是使用 WebSocket 协议。这个协议由 W3C 定义(http://www.w3.org/TR/WebSocket)，开始于一个 HTTP 请求。客户端首先发出一个 HTTP 请求，防火墙通常允许发送该请求。客户端发出一个 GET 请求时，在 HTTP 头中包含 Upgrade: websocket Connection: Upgrade，再加上 WebSocket 版本和安全信息。如果服务器支持 WebSocket 协议，服务器就会用一个升级来回应，并从 HTTP 切换到 WebSocket 协议。

在 WCF 中，.NET 4.5 提供的两个新绑定支持 WebSocket 协议：netHttpBinding 和 netHttpsBinding。

现在创建一个使用 WebSocket 协议的示例。开始一个空白的 Web 应用程序，用于保存服务。

HTTP 协议的默认绑定是前面介绍的 basicHttpBinding。定义 protocolMapping，来指定 netHttpBinding，就可以修改它，如下所示。这样就不需要配置服务元素，来匹配协定、绑定和端点地址了。有了配置，就启用 serviceMetadata，允许客户端使用 Add Service Reference 对话框来引用服务(配置文件 WebSocketsSample/web.config)。

```
<system.serviceModel>
  <protocolMapping>
    <remove scheme="http" />
    <add scheme="http" binding="netHttpBinding" />
    <remove scheme="https" />
    <add scheme="https" binding="netHttpsBinding" />
  </protocolMapping>
  <behaviors>
    <serviceBehaviors>
```

```
        <behavior name="">
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="false" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
       multipleSiteBindingsEnabled="true" />
</system.serviceModel>
```

服务协定由接口 IDemoServices 和 IDemoCallback 定义(代码文件 WebSocketsSample/IDemoServices.cs)。IDemoServices 是定义了方法 StartSendingMessages 的服务接口。客户端调用方法 StartSendingMessages，启动过程，使服务可以给客户端返回消息。所以客户端需要实现 IDemoCallback 接口。这个接口由服务器调用，由客户端实现。

接口的方法定义为返回任务，于是，服务很容易使用异步功能，但这不会影响协定。以异步方式定义方法，独立于所生成的 WSDL：

```
using System.ServiceModel;
using System.Threading.Tasks;

namespace WebSocketsSample
{
  [ServiceContract]
  public interface IDemoCallback
  {
    [OperationContract(IsOneWay = true)]
    Task SendMessage(string message);
  }

  [ServiceContract(CallbackContract = typeof(IDemoCallback))]
  public interface IDemoService
  {
    [OperationContract]
    Task StartSendingMessages();
  }
}
```

服务的实现在 DemoService 类中完成(代码文件 WebSocketsSample/DemoService.cs)。在方法 StartSendingMessages 中，要返回给客户端的回调接口通过 OperationContext.Current.GetCallbackChannel 来检索。客户端调用该方法时，它在第一次调用 SendMessage 方法后立即返回。线程在完成 SendMessage 方法之前不会阻塞。在完成 SendMessage 方法后，使用 await 把一个线程返回给 StartSendingMessages。接着延迟一秒，之后客户端接收另一个消息。While 循环退出时，关闭通信通道。

```
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Threading.Tasks;

namespace WebSocketsSample
{
  public class DemoService : IDemoService
```

```
    {
      public async Task StartSendingMessages()
      {
        IDemoCallback callback =
          OperationContext.Current.GetCallbackChannel<IDemoCallback>();
        int loop = 0;
        while ((callback as IChannel).State == CommunicationState.Opened)
        {
          await callback.SendMessage(string.Format(
            "Hello from the server {0}", loop++));
          await Task.Delay(1000);
        }
      }
    }
}
```

客户端应用程序创建为一个控制台应用程序。因为元数据可以通过服务获得，所以添加服务引用会创建一个代理类，它可以用于调用服务，实现回调接口。添加服务引用不仅会创建代理类，还会把 netHttpBinding 添加到配置文件中：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <system.serviceModel>
    <bindings>
      <netHttpBinding>
        <binding name="NetHttpBinding_IDemoService">
          <webSocketSettings transportUsage="Always" />
        </binding>
      </netHttpBinding>
    </bindings>
    <client>
      <endpoint address="ws://localhost:20839/DemoService.svc"
        binding="netHttpBinding"
        bindingConfiguration="NetHttpBinding_IDemoService"
        contract="DemoService.IDemoService"
        name="NetHttpBinding_IDemoService" />
    </client>
  </system.serviceModel>
</configuration>
```

回调接口的实现代码只把一个消息写入控制台，说明从服务中接收到了信息。要启动所有的过程，应创建一个 DemoServiceClient 实例，它接收一个 InstanceContext 对象。InstanceContext 对象包含 CallbackHandler 的一个实例，这个引用由服务接收，并返回给客户端。

```
using System;
using System.ServiceModel;
using ClientApp.DemoService;

namespace ClientApp
{
  class Program
```

```
  {
    private class CallbackHandler : IDemoServiceCallback
    {
      public void SendMessage(string message)
      {
        Console.WriteLine("message from the server {0}", message);
      }
    }

    static void Main(string[] args)
    {
      Console.WriteLine("client... wait for the server");
      Console.ReadLine();
      StartSendRequest();
      Console.WriteLine("next return to exit");
      Console.ReadLine();
    }

    static async void StartSendRequest()
    {
      var callbackInstance = new InstanceContext(new CallbackHandler());
      var client = new DemoServiceClient(callbackInstance);
      await client.StartSendingMessagesAsync();
    }
  }
}
```

运行应用程序，客户端向服务请求消息，服务就做出与客户端无关的回应：

```
client... wait for the server

next return to exit
message from the server Hello from the server 0
message from the server Hello from the server 1
message from the server Hello from the server 2
message from the server Hello from the server 3
message from the server Hello from the server 4

Press any key to continue . . .
```

43.6 宿主

在选择运行服务的宿主时，WCF 非常灵活。宿主可以是 Windows 服务、COM+应用程序、WAS(Windows Activation Services，Windows 激活服务)或 IIS、Windows 应用程序，或简单的控制台应用程序。在用 Windows 窗体或 WPF 创建自定义主机时，很容易创建对等的解决方案。

43.6.1 自定义宿主

先从自定义宿主开始。下面的示例代码列出了控制台应用程序中的服务宿主。但在其他自定义主机类型中，如 Windows 服务或 Windows 应用程序，可以用相同的方式编写服务。

在 Main()方法中，创建一个 ServiceHost 实例。之后，读取应用程序配置文件，来定义绑定。也

可以通过编程方式定义绑定，如前面所示。接着，调用ServiceHost类的Open()方法，使服务接受客户调用。在控制台应用程序中，必须注意在关闭服务之前，不能关闭主线程。这里实际上在调用Close()方法时，要求用户“按回车键”，以结束(退出)服务。

```
using System;
using System.ServiceModel;

public class Program
{
  public static void Main()
  {
    using (var serviceHost = new ServiceHost())
    {
      serviceHost.Open();

      Console.WriteLine("The service started. Press return to exit");
      Console.ReadLine();

      serviceHost.Close();
    }
  }
}
```

要终止服务宿主，可以调用ServiceHost类的Abort()方法。要获得服务的当前状态，State属性会返回CommunicationState枚举定义的一个值，该枚举的值有Created、Opening、Opened、Closing、Closed和Faulted。

如果从Windows窗体或WPF应用程序中启动服务，该服务的代码调用Windows控件的方法，就必须确保，只有控件的创建线程才能访问该控件的方法和属性。在WCF中，设置[ServiceBehavior]特性的UseSynchronizationContext属性，就可以实现该行为。

43.6.2 WAS宿主

在WAS宿主中，可以使用WAS工作进程中的功能，如自动激活服务、健康监控和进程循环。

要使用WAS宿主，只需要创建一个Web站点和一个.svc文件，其中的ServiceHost声明包含服务类的语言和名称。下面的代码使用Service1类。另外，还必须指定包含服务类的文件。这个类的实现方式与前面定义WCF服务库的方式相同。

```
<%@ServiceHost language="C#" Service="Service1" CodeBehind="Service1.svc.cs" %>
```

如果使用WAS宿主中可用的WCF服务库，就可以创建一个.svc文件，它只包含类的引用：

```
<%@ ServiceHost Service="Wrox.ProCSharp.WCF.Services.RoomReservationService" %>
```

自从引入Windows Vista和Windows Server 2008以来，WAS允许定义.NET TCP和Message Queue绑定。如果使用以前的版本，Windows Server 2003和Windows XP中的IIS 6或IIS 5.1，就只能使用HTTP绑定从.svc文件中激活服务。

43.6.3 预配置的宿主类

为了减少配置的必要性，WCF 还提供了一些带预配置绑定的宿主类。一个例子是 System.ServiceModel.Web 名称空间中的 System.ServiceModel.Web 程序集中的 WebServiceHost 类。如果没有用 WebHttpBinding 配置默认端点，这个类就为 HTTP 和 HTTPS 基址创建一个默认端点。另外，如果没有定义另一个行为，这个类就会添加 WebHttpBehavior。利用这个行为，可以执行简单的 HTTP GET 操作和 POST、PUT、DELETE (使用 WebInvoke 属性)操作，而无需额外的设置(代码文件 RoomReservation/RoomReservationWebServiceHost/Program.cs)。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Web;
using Wrox.ProCSharp.WCF.Service;

namespace RoomReservationWebHost
{
  class Program
  {
    static void Main()
    {
      var baseAddress = new Uri("http://localhost:8000/RoomReservation");
      var host = new WebServiceHost(typeof(RoomReservationService), baseAddress);
      host.Open();

      Console.WriteLine("service running");
      Console.WriteLine("Press return to exit...");
      Console.ReadLine();

      if (host.State == CommunicationState.Opened)
        host.Close();
    }
  }
}
```

要使用简单的 HTTP GET 请求接收预定信息，GetRoomReservation()方法需要一个 WebGet 特性，把方法参数映射到 GET 请求的输入上。在下面的代码中，定义一个 UriTemplate，这需要待添加到基址中的 Reservations 后跟 From 和 To 参数。From 和 To 参数依次映射到 fromDate 和 toDate 变量上(代码段 RoomReservation/RoomReservationService/RoomReservationService.cs)。

```
[WebGet(UriTemplate="Reservations?From={fromTime}&To={toTime}")]
public RoomReservation[] GetRoomReservations(DateTime fromTime,
  DateTime toTime)
{
  var data = new RoomReservationData();
  return data.GetReservations(fromTime, toTime);
}
```

现在可以使用简单的请求来调用服务了，如下所示。返回给定时间段的所有预定信息。

```
http://localhost:8000/RoomReservation/Reservations?From=2012/1/1&To=2012/8/1
```

System.Data.Services.DataServiceHost 是另一个带预配置特性的类。这个类派生自 WebServiceHost。

43.7 客户端

客户端应用程序需要一个代理来访问服务。给客户端创建代理有 3 种方式：

- **Visual Studio 添加服务引用**——这个实用程序会从服务的元数据中创建代理类。
- **ServiceModel 元数据实用工具(Svcutil.exe)**——使用 SvcUtil 实用程序可以创建代理类。该实用程序从服务中读取元数据，以创建代理类。
- **ChannelFactory 类**——这个类由 Svcutil 实用程序生成的代理使用，然而，它也可以用于以编程方式创建代理。

43.7.1 使用元数据

从 Visual Studio 中添加服务引用，需要访问 WSDL 文档。WSDL 文档由 MEX 端点创建，MEX 端点需要用服务配置。在下面的配置中，带相对地址 mex 的端点使用 mexHttpBinding，并实现 ImetadataExchange 协定。为了通过 HTTP GET 请求访问元数据，应把 behaviorConfiguration 配置为 MexServiceBehavior。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration=" MexServiceBehavior "
        name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address="Test" binding="wsHttpBinding"
          contract="Wrox.ProCSharp.WCF.IRoomService" />
        <endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress=
        "http://localhost:8733/Design_Time_Addresses/RoomReservationService/" />
          <baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MexServiceBehavior">
          <! - To avoid disclosing metadata information,
          set the value below to false and remove the metadata endpoint above
          before deployment ->
          <serviceMetadata httpGetEnabled="True"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
```

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

类似于 Visual Studio 中的添加服务引用，Svcutil 实用程序需要元数据来创建代理类。Svcutil 实用程序可以从 MEX 元数据端点、程序集的元数据、WSDL 和 XSD 文档中创建代理。

```
svcutil http://localhost:8080/RoomReservation?wsdl /language:C# /out:proxy.cs
svcutil CourseRegistration.dll
svcutil CourseRegistration.wsdl CourseRegistration.xsd
```

生成代理类后，它需要从客户端代码中实例化，再调用方法，最后必须调用 Close()方法：

```
var client = new RoomServiceClient();
client.RegisterForCourse(roomReservation);
client.Close();
```

43.7.2 共享类型

生成的代理类派生自基类 ClientBase<TChannel>，该基类封装 ChannelFactory<TChannel>类。除了使用生成的代理类之外，还可以直接使用 ChannelFactory<TChannel>类。构造函数需要绑定和端点地址；之后，就可以创建信道，调用服务协定定义的方法。最后，必须关闭该工厂。

```
var binding = new WsHttpBinding();
var address = new EndpointAddress("http://localhost:8080/RoomService");

var factory = new ChannelFactory<IStateService>(binding, address);

IRoomService channel = factory.CreateChannel();
channel.ReserveRoom(roomReservation);

//.
factory.Close();
```

ChannelFactory<TChannel>类有几个属性和方法，如表 43-8 所示。

表 43-8

ChannelFactory 类的成员	说　明
Credentials	Credentials 是一个只读属性，可以访问 ClientCredentials 对象，该对象被赋予信道，对服务进行身份验证。Credentials 可以用端点来设置
Endpoint	Endpoint 是一个只读属性，可以访问与信道相关联的 ServiceEndpoint。端点可以在构造函数中指定
State	State 属性的类型是 CommunicationState，它返回信道的当前状态。CommunicationState 是一个枚举，其值是 Created、Opening、Opened、Closing、Closed 和 Faulted
Open()	该方法用于打开信道
Close()	该方法用于关闭信道
Opening、Opened、Closing、Closed 和 Faulted	可以指定事件处理程序，从而确定信道的状态变化。这些事件分别在信道打开前后、信道关闭前后和出错时触发

43.8 双工通信

下面的示例程序说明了如何在客户端和服务之间直接进行双工通信。客户端会启动到服务的连接。之后，服务就可以回调客户端了。前面的 WebSocket 协议也演示了双工通信。除了使用 WebSocket 协议(只有 Windows 8 和 Windows Server 2012 支持它)之外，双工通信还可以使用 WsHttpBinding 和 NetTcpBinding 来实现。

43.8.1 双工通信的协定

为了进行双工通信，必须指定一个在客户端中实现的协定。这里用于客户端的协定由 ImyMessageCallback 接口定义。由客户端实现的方法是 OnCallback()。操作应用了 IsOneWay=true 操作协定设置。这样，服务就不必等待方法在客户端上成功调用了。在默认情况下，服务实例只能从一个线程中调用(服务行为的 ConcurrencyMode 属性默认设置为 ConcurrencyMode.Single)。

如果服务的实现代码回调客户端，等待获得客户端的结果，则从客户端中获得回应的线程就必须等待，直到锁定服务对象为止。因为服务对象已经由客户端的请求锁定，所以出现了死锁。WCF 检测到这个死锁，就抛出一个异常。为了避免这种情况，可以将 ConcurrencyMode 属性的值改为 Multiple 或 Reentrant。使用 Multiple 设置，多个线程可以同时访问实例。这里必须自己实现锁定。使用 Reentrant 设置，服务实例将只使用一个线程，但允许将回调请求的回应重新输入到上下文中。除了改变并发模式之外，还可以用操作协定指定 IsOneWay 属性。这样，调用者就不会等待回应了。当然，只有不需要返回值，才能使用这个设置。

服务协定由 IMyMessage 接口定义。回调协定用服务协定定义的 CallbackContract 属性映射到服务协定上(代码文件 DuplexCommunication/MessageService/IMyMessage.cs)。

```
public interface IMyMessageCallback
{
  [OperationContract(IsOneWay=true)]
  void OnCallback(string message);
}

[ServiceContract(CallbackContract=typeof(IMyMessageCallback))]
public interface IMyMessage
{
  [OperationContract]
  void MessageToServer(string message);
}
```

43.8.2 双工通信的服务

MessageService 类实现了服务协定 IMyMessage。服务将来自客户端的消息写入控制台。要访问回调协定，可以使用 OperationContext 类。OperationContext.Current 返回与客户端中当前请求关联的 OperationContext。使用 OperationContext 可以访问会话信息、消息标题和属性，在双工通信的情况下还可以访问回调信道。泛型方法 GetCallbackChannel()将信道返回客户端实例。接着调用由回调接口 IMyMessageCallback 定义的 OnCallback()方法，可以使用这个信道将消息发送给客户端。为了演示这些操作，还可以从服务中使用独立于方法的完成的回调信道，创建一个接收回调信道的新线程。

这个新线程再次使用回调信道，将消息发送给客户(代码文件 DuplexCommunication/MessageService/MessageService.cs)。

```
public class MessageService: IMyMessage
{
  public void MessageToServer(string message)
  {
    Console.WriteLine("message from the client: {0}", message);
    IMyMessageCallback callback =
        OperationContext.Current.
           GetCallbackChannel<IMyMessageCallback>();

    callback.OnCallback("message from the server");

    Task.Factory.StartNew(new Action<object>(TaskCallback), callback);
  }

  private async void ThreadCallback(object callback)
  {
    IMyMessageCallback messageCallback = callback as IMyMessageCallback;
    for (int i = 0; i < 10; i++)
    {
      messageCallback.OnCallback("message " + i.ToString());
      await Task.Delay(1000);
    }
  }
}
```

存放服务的方式与前面的例子相同，这里不再赘述。但是对于双工通信，必须配置一个支持双工通信的绑定。支持双工信道的一个绑定是 wsDualHttpBinding，它在应用程序的配置文件中配置。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.MessageService">
        <endpoint contract="Wrox.ProCSharp.WCF.IMyMessage"
            binding="wsDualHttpBinding"/>
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8733/Service1" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

43.8.3 双工通信的客户端应用程序

在客户端应用程序中，必须用 ClientCallback 类实现回调协定，该类实现了 IMyMessageCallback 接口，如下所示(代码文件 DuplexCommunication/MessageClient/Program.cs):

```
class ClientCallback: IMyMessageCallback
```

```
{
  public void OnCallback(string message)
  {
     Console.WriteLine("message from the server: {0}", message);
  }
}
```

在双工信道中，不能像前面那样使用 ChannelFactory 启动与服务的连接。要创建双工信道，可以使用 DuplexChannelFactory 类。这个类有一个构造函数，除了绑定和地址配置之外，它还有一个参数，这个参数指定 InstanceContext，它封装 ClientCallback 类的一个实例。把这个实例传递给工厂，服务就可以通过信道调用对象。客户端只需使连接一直处于打开状态。如果关闭连接，服务就不能通过它发送消息。

```
private async static void DuplexSample()
{
  var binding = new WSDualHttpBinding();
  var address = new EndpointAddress("http://localhost:8733/Service1");

  var clientCallback = new ClientCallback();
  var context = new InstanceContext(clientCallback);

  var factory = new DuplexChannelFactory<IMyMessage>(context, binding,
                    address);

  IMyMessage messageChannel = factory.CreateChannel();

  await Task.Run(() => messageChannel.MessageToServer("From the client"));
}
```

启动服务主机和客户端应用程序，就可以实现双工通信。

43.9 路由

与 HTTP GET 请求和 REST 相比，使用 SOAP 协议有一些优点。SOAP 的一个高级特性是路由。通过路由，客户端不直接寻址服务，而是由客户端和服务器之间的路由器传送请求。

可以在不同的情形下使用这个特性，一种情形是失败(如图 43-10 所示)。如果请求无法到达服务，或者返回了一个错误，路由器就调用另一个宿主上的服务。这是从客户端抽象出来的，客户端只是接收一个结果。

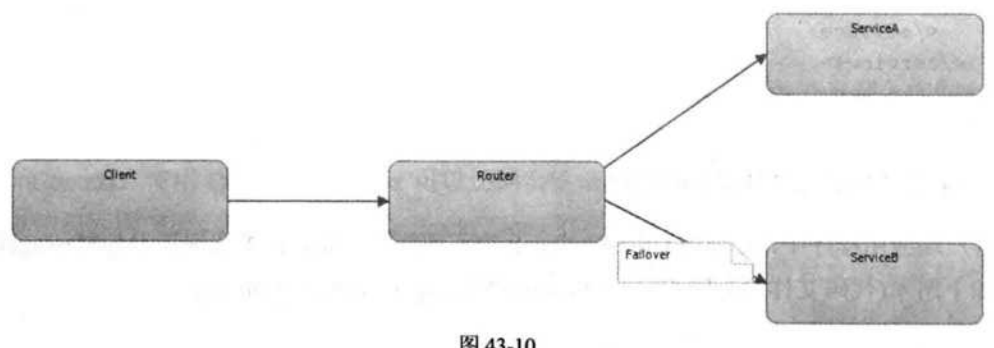


图 43-10

路由也可以用于改变通信协议(如图 43-11 所示)。客户端可以使用 HTTP 协议调用请求，把它发送给路由器。路由器用作带 net.tcp 协议的客户端，调用服务，来发送消息。

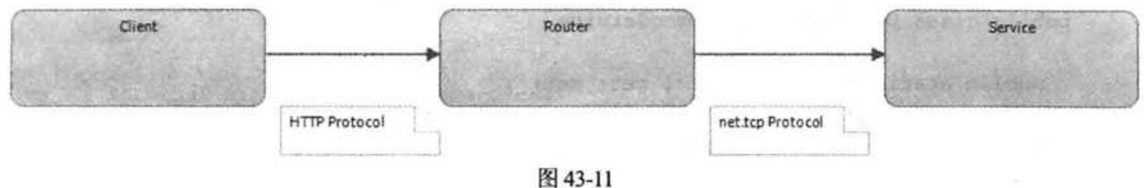


图 43-11

使用路由来实现可伸缩性是另一种情形(如图 43-12 所示)。根据消息标题的一个字段或者来自消息内容的信息，路由器可以确定是把请求发送给一个服务器还是另一个服务器。来自客户的、以 A-F 字母开头的请求会发送给第一个服务器，以 G-N 字母开头的请求会发送给第二个服务器，以 Q-Z 字母开头的请求会发送给第三个服务器。

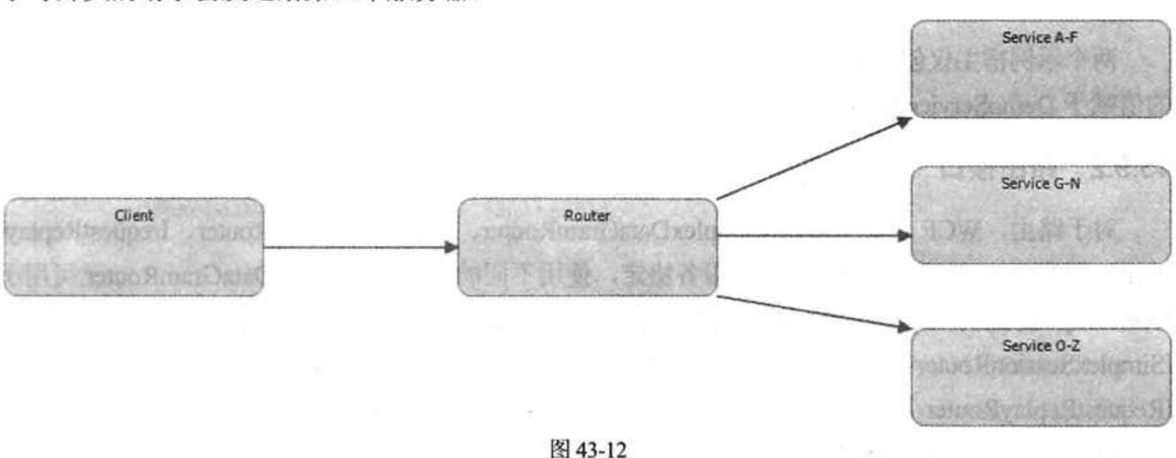


图 43-12

43.9.1　示例应用程序

在路由示例应用程序中，定义一个简单的服务协议，其中调用者可以从 IDemoService 接口中调用 GetData 操作(代码文件 RoutingService/DemoService/IDemoService.cs)：

```
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
  [ServiceContract(Namespace="http://www.cninnovation.com/Services/2012")]
  public interface IDemoService
  {
    [OperationContract]
    string GetData(string value);
  }
}
```

服务的实现代码(代码文件 RoutingService/DemoService/DemoService.cs)仅用 GetData 方法返回一个消息，该消息包含接收到的信息和一个在主机上初始化的服务器端字符串。这样就可以看到主机给客户端返回了调用。

```
using System;
```

```
namespace Wrox.ProCSharp.WCF
{
  public class DemoService : IDemoService
  {
    public static string Server { get; set; }

    public string GetData(string value)
    {
      string message = string.Format("Message from {0}, You entered: {1}",
        Server, value);
      Console.WriteLine(message);
      return message;
    }
  }
}
```

两个示例宿主仅创建了一个 ServiceHost 实例，打开它以启动监听器。每个定义的宿主都把不同的值赋予 DemoService 的 Server 属性。

43.9.2 路由接口

对于路由，WCF 定义了接口 ISimplexDataGramRouter、ISimplexSessionRouter、IrequestReplayRouter 和 IDuplexSessionRouter。根据服务协定，使用不同的接口。ISimplexDataGramRouter 可用于 IsOneWay 设置为 OperationContract 的操作。对于 ISimplexDataGramRouter，会话是可选的。ISimplexSessionRouter 可用于单向消息，例如 ISimplexDataGramRouter，但这里会话是强制的。IRequestReplayRouter 用于最常见的情形：请求和响应消息。接口 IDuplexSessionRouter 用于双工通信(例如前面使用的 WsDualHttpBinding)。

根据所使用的消息模式，定制路由器需要实现对应的路由器接口。

43.9.3 WCF 路由服务

不要创建定制路由器，而可以使用名称空间 System.ServiceModel.Routing 中的 RouterService。这个类实现了所有的路由接口，因此可以用于所有的消息模式。它可以像其他服务那样驻留(代码文件 RoutingService/Router/Program.cs)。在 StartService 方法中，传递了 RoutingService 类型，实例化一个新的 ServiceHost。这类似于前面的其他宿主。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Routing;

namespace Router
{
  class Program
  {
    internal static ServiceHost routerHost = null;

    static void Main()
    {
      StartService();

      Console.WriteLine("Router is running. Press return to exit");
```

```
        Console.ReadLine();

        StopService();
      }

      internal static void StartService()
      {
        try
        {
          routerHost = new ServiceHost(typeof(RoutingService));
          routerHost.Faulted += myServiceHost_Faulted;
          routerHost.Open();
        }
        catch (AddressAccessDeniedException)
        {
          Console.WriteLine("either start Visual Studio in elevated admin " +
            "mode or register the listener port with netsh.exe");
        }
      }

      static void myServiceHost_Faulted(object sender, EventArgs e)
      {
        Console.WriteLine("router faulted");
      }

      internal static void StopService()
      {
        if (routerHost != null && routerHost.State == CommunicationState.Opened)
        {
          routerHost.Close();
        }
      }
    }
  }
```

43.9.4 为失败使用路由器

比宿主代码更有趣的是路由器的配置(配置文件 Router/App.Config)。路由器用作客户端应用程序的服务器，和服务的客户端，所以两个部件都需要配置。如下所示的配置提供了 wsHttpBinding 作为服务器部件，使用 wsHttpBinding 作为客户端来连接服务。服务端点需要指定用于该端点的协定。使用服务提供的请求-回应操作，协定由 IRequestReplyRouter 接口定义。

```
<system.serviceModel>
  <services>
    <service behaviorConfiguration="routingData"
      name="System.ServiceModel.Routing.RoutingService">
      <endpoint address="" binding="wsHttpBinding"
        name="reqReplyEndpoint"
        contract="System.ServiceModel.Routing.IRequestReplyRouter" />
      <endpoint address="mex" binding="mexHttpBinding"
        contract="IMetadataExchange" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8000/RoutingDemo/router" />
        </baseAddresses>
```

```
      </host>
    </service>
  </services>
```

路由器的客户端部件为服务定义了两个端点。为了测试路由服务，可以使用一个系统。当然，这通常是运行在另一个系统上的主机。协定可以设置为*，允许所有的协定传送给这些端点覆盖的服务。

```
<client>
  <endpoint address="http://localhost:9001/RoutingDemo/HostA"
   binding="wsHttpBinding" contract="*" name="RoutingDemoService1" />
  <endpoint address="http://localhost:9001/RoutingDemo/HostB"
   binding="wsHttpBinding" contract="*" name="RoutingDemoService2" />
</client>
```

服务操作的配置对路由很重要。操作配置 routingData 通过前面的服务配置来引用。在路由时，必须用操作设置路由元素，这里使用属性 filterTableName 来引用路由表。

```
<behaviors>
  <serviceBehaviors>
    <behavior name="routingData">
      <serviceMetadata httpGetEnabled="True"/>
      <routing filterTableName="routingTable1" />
      <serviceDebug includeExceptionDetailInFaults="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

过滤表 routingTable1 包含一个 filterType 为 MatchAll 的过滤器。这个过滤器匹配每个请求。现在来自客户端的每个请求都路由到端点 RoutingDemoService1。如果这个服务失败，不能访问，后备列表就很重要。后备列表 failOver1 定义了第二个端点，在第一个端点失败时，就使用第二个端点。

```
<routing>
  <filters>
   <filter name="MatchAllFilter1" filterType="MatchAll" />
  </filters>
  <filterTables>
    <filterTable name="routingTable1">
      <add filterName="MatchAllFilter1" endpointName="RoutingDemoService1"
        backupList="failOver1" />
    </filterTable>
  </filterTables>
  <backupLists>
    <backupList name="failOver1">
      <add endpointName="RoutingDemoService2"/>
    </backupList>
  </backupLists>
</routing>
```

有了路由服务器和路由配置，就可以启动客户端，通过路由器调用服务。如果一切顺利，客户端就会从运行在主机 1 上的服务中获得回应。如果停止主机 1，而客户端发出了另一个请求，主机 2 就负责返回一个回应。

43.9.5　改变协定的桥梁

如果路由器应改变协议，就可以配置主机，来使用 netTcpBinding 替代 wsHttpBinding。对于路由器，客户端配置需要改为引用另一个端点。

```
<endpoint address="net.tcp://localhost:9010/RoutingDemo/HostA"
  binding="netTcpBinding" contract="*" name="RoutingDemoService1" />
```

这就改变了协定。

43.9.6　过滤器的类型

在示例应用程序中，使用了 MatchAll 过滤器。WCF 提供了更多过滤器类型。如表 43-9 所示。

表 43-9

过滤器类型	说　明
Action	Action 过滤器根据消息上的操作来启用过滤功能。参见 OperationContract 的 Action 属性
Address	Address 过滤器对位于 SOAP 标题的 To 字段中的地址启用过滤功能。
AddressPrefix	AddressPrefix 过滤器不匹配完整的地址，而匹配地址的最佳前缀
MatchAll	MatchAll 过滤器会匹配每个请求
XPath	使用 XPath 消息过滤器，可以定义一个 XPath 表达式，来过滤消息标题。可以使用消息协定给 SOAP 标题添加信息
Custom	如果需要根据消息的内容进行路由，就需要 Custom 过滤器类型。使用这个类型，需要创建一个派生于基类 MessageFilter 的类，过滤器的初始化用一个带 string 参数的构造函数来完成。string 参数可以从配置初始值中传递

如果把多个过滤器应用于一个请求，就可以给过滤器使用优先级。但是，最好避免使用优先级，因为这会降低性能。

43.10　小结

本章学习了如何使用 Windows Communication Foundation 在客户端和服务器之间通信。WCF 与 ASP.NET Web 服务一样，也独立于平台，但它提供了与.NET Remoting、Enterprise Services 和消息队列类似的功能。

WCF 主要利用服务协定、数据协定和消息协定，来简化客户端和服务的独立开发，并支持独立的平台。可以使用几个属性定义服务的行为和对应操作。

我们还探讨了如何从服务提供的元数据中创建客户端，如何使用.NET 接口协定来创建客户端。本章介绍了不同绑定选项的功能。WCF 不仅提供了独立于平台的绑定，还提供了在.NET 应用程序之间快速通信的绑定。本章还探讨了如何创建自定义主机和如何使用 WAS 主机。如何定义回调接口，应用服务协定，在客户端应用程序中实现回调协定，实现双工通信。

后面几章继续介绍 WCF 的功能。第 44 章讨论 WCF 数据服务，第 45 章讨论 Windows Workflow Foundation，如何使用 WCF 与工作流实例通信。第 46 章解释如何使用 WCF 服务进行对等通信。第 47 章学习如何联合使用 WCF 绑定和断开连接的消息队列功能。

第44章 ASP.NET Web API

本章要点

- ASP.NET Web API 概述
- 创建服务
- .NET 客户程序
- Web API 路由
- 使用 OData
- 安全性
- 定制宿主

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 图书服务示例
- 方程式 1 数据服务示例
- 保护 Web API
- 自驻留的应用程序

44.1 概述

发布 WCF 时，它是一种通信技术，替代了.NET 栈中的其他几个技术(其中的两个是.NET Remoting 和 ASP.NET Web 服务)。其目标是只用一种非常灵活的通信技术来满足所有需求。但是，WCF 最初基于 SOAP。现在有许多情形都不需要强大的 SOAP 改进功能。对于 HTTP 请求返回 JSON 这样的简单情形，WCF 过于复杂。因此在 2012 年引入了另一种技术：ASP.NET Web API。在 Visual Studio 2013 中，发布了 ASP.NET Web API 的下一个重要版本 2.0。本章介绍这个版本。

ASP.NET Web API 提供了一种基于 REST(Representational State Transfer)的简单通信技术。REST

是基于一些限制的体系结构。下面比较基于 REST 体系结构的服务和使用 SOAP 的服务，以了解这些限制。

REST 服务和使用 SOAP 协议的服务都利用了客户端-服务器技术。SOAP 服务可以是有状态的，也可以是无状态的，REST 服务总是无状态的。SOAP 定义了它自己的消息格式，该格式有标题和正文，可以选择服务的方法。而在 REST 中，使用 HTTP 动词 GET、POST、PUT 和 DELETE。GET 用于检索资源，POST 用于添加新资源，PUT 用于更新资源，DELETE 用于删除资源。

本章介绍 ASP.NET Web API 的各个重要方面——创建服务、使用不同的路由方法、创建客户程序、使用 OData(这是版本 2 的新增功能)、保护服务和使用自定义的宿主。

注意：
SOAP 和 WCF 参见第 43 章。

44.2 创建服务

首先创建服务。ASP.NET Web API 属于 ASP.NET，所以需要先建立一个新的 Web 应用程序项目。之后选择模板 Web API，如图 44-1 所示。这个模板给 MVC 和 Web API 添加文件夹和引用。添加 MVC，是因为使用这个技术会创建服务的帮助页面。

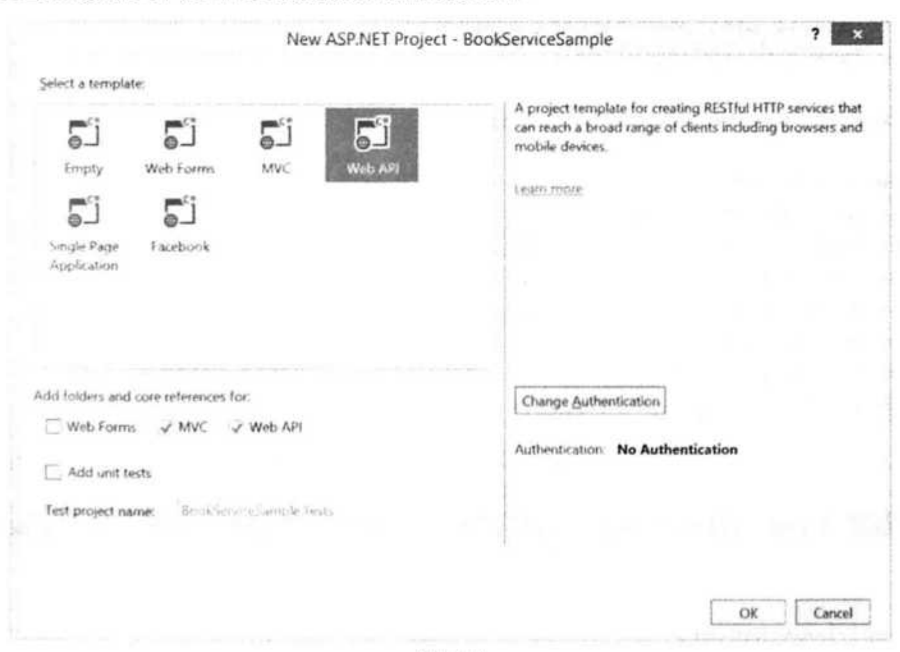


图 44-1

注意：
ASP.NET MVC 参见第 42 章。

用这个模板创建的目录结构包含 ASP.NET MVC 需要的主文件夹。对于 Web API，Controllers 目录很重要，因为它不仅包含 MVC，还包含 Web API 控制器和用于数据模型的 Models 目录。

所创建的服务返回图书的章节列表，并允许动态增删章节。提供该服务的这个项目的名称是 BookServiceSample。

44.2.1 定义模型

首先需要一个类来表示要返回和修改的数据。在 Models 目录中定义的类的名称是 BookChapter，它包含表示一章的简单属性(代码文件 BookServiceSample/Model/BookChapter.cs)：

```
public class BookChapter
{
  public int Number { get; set; }
  public string Title { get; set; }
  public int Pages { get; set; }
}
```

44.2.2 创建控制器

服务的控制器通过 Project | Add Controller | Web API 2 Controller 创建，带有读写动作，应命名为 BookChaptersController。这个类派生自基类 ApiController，它类似于 MVC 控制器 Controller 类型，但不完全相同。与 MVC 控制器相比，区别很大的是，路由的定义是完全不同的，它不在 URL 名中定义，而是在 HTTP 谓词中定义，如后面所述。对于初始数据，定义一个 List<BookChapter>，用静态构造函数初始化(代码文件 BookServiceSample/Controllers/BookChaptersController.cs)：

```
public class BookChaptersController : ApiController
{
  private static List<BookChapter> chapters;
  static BookChaptersController()
  {
    chapters = new List<BookChapter>()
    {
      new BookChapter { Number=1, Title=".NET Architecture", Pages=20},
      new BookChapter { Number=2, Title="Core C#", Pages=42},
      new BookChapter { Number=3, Title="Objects and Types", Pages=24},
      new BookChapter { Number=4, Title="Inheritance", Pages=18},
      new BookChapter { Number=5, Title="Generics", Pages=22},
      new BookChapter { Number=17, Title="Visual Studio 2012", Pages=50},
      new BookChapter { Number=42, Title="ASP.NET Dynamic Data",
        Pages=14}
    };
  }
```

模板中创建的 Get 方法被重命名，并被修改为返回类型为 IEnumerable<BookChapter>的完整集合：

```
// GET api/bookchapters
public IEnumerable<BookChapter> GetBookChapters()
{
  return chapters;
}
```

带一个参数的 Get 方法被重命名为 GetBookChapter，用 LINQ 查询操作符 Where 过滤集合。它使用 SingleOrDefault 返回图书的一章：

```
// GET api/bookchapters/5
public BookChapter GetBookChapter(int id)
```

```
{
  return chapters.Where(c => c.Number == id).SingleOrDefault();
}
```

注意：

LINQ 参见第 11 章。

要添加图书的新章节，应添加 PostBookChapter。这个方法把参数接收的 BookChapter 添加到集合中：

```
// POST api/bookchapters
public void PostBookChapter([FromBody]BookChapter value)
{
  chapters.Add(value);
}
```

更新条目需要基于 HTTP PUT 请求。PutBookChapter 方法从集合中删除一个已有的条目，再添加一个新条目，如下所示。当然，另一种实现方式是在集合中找到已有的条目，用新条目更新其属性。例如，找到一个已有的条目，根据新值修改该条目的属性：

```
// PUT api/bookchapters/5
public void PutBookChapter(int id, [FromBody]BookChapter value)
{
  chapters.Remove(chapters.Where(c => c.Number == id).Single());
  chapters.Add(value);
}
```

对于 HTTP DELETE 请求，删除图书的章节：

```
// DELETE api/bookchapters/5
public void DeleteBookChapter(int id)
{
  chapters.Remove(chapters.Where(c => c.Number == id).Single());
}
```

有了这个控制器，就可以在浏览器上进行第一组测试了。打开链接 http://localhost:11825/api/BookChapters，返回 JSON 或 XML。你的系统可能使用不同的端口号，可以在 Project Properties 的 Web Settings 选项卡上配置它。用 Internet Explorer 打开这个链接，就会返回如下 JSON 文件：


```
[{"Number":1,"Title":".NET Architecture","Pages":20},
 {"Number":2,"Title":"Core C#","Pages":42},
 {"Number":3,"Title":"Objects and Types","Pages":24},
 {"Number":4,"Title":"Inheritance","Pages":18},
 {"Number":5,"Title":"Generics","Pages":22},
 {"Number":6,"Title":"Arrays and Tuples","Pages":22},
 {"Number":7,"Title":"Operators and Casts","Pages":32},
 {"Number":17,"Title":"Visual Studio 2012","Pages":50},
 {"Number":42,"Title":"ASP.NET Dynamic Data","Pages":14}]
```


用 Google Chrome 打开这个链接，会返回 XML 内容：


```
<ArrayOfBookChapter xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://schemas.datacontract.org/2004/07/BookServiceSample.Models">
```

```
  <BookChapter>
    <Number>1</Number>
    <Pages>20</Pages>
    <Title>.NET Architecture</Title>
  </BookChapter>
  <BookChapter>
    <Number>2</Number>
    <Pages>42</Pages>
    <Title>Core C#</Title>
  </BookChapter>
  <!-- ... -->
</ArrayOfBookChapter>
```

Web 浏览器有这个区别，是因为 Google Chrome 会发送 HTTP Accept 标题 application/xml 以获得 XML 内容，而 IE 不会。这个 Accept 标题可用时，ASP.NET Web API 返回 XML 或 JSON；JSON 是返回的默认格式。

44.2.3 错误处理

模板生成的动作方法返回 void。但是，最好根据结果返回专用的 HTML 错误和成功码。现在使用 ASP.NET Web API 2.0 很容易实现这一点。方法不声明为 void，而是声明为返回 IHttpActionResult。这样就可以返回实现这个接口的任意对象了。ApiController 基类已经定义了一些方法，它们返回实现了 IHttpActionResult 的对象；例如 Ok 方法返回 OkResult，BadRequest 方法返回 BadRequestResult。在示例代码中，如果没有找到要更新的引用图书章节，方法 Single 就抛出一个 InvalidOperation-Exception 异常。处理该异常时，要返回一个 BadRequestResult 对象(它返回 HTTP 状态码 400，错误的请求)。如果更新成功，OkResult 就定义 HTTP 状态码 200——OK：

```
public IHttpActionResult PutBookChapter(
  int id, [FromBody]BookChapter value)
{
  try
  {
    chapters.Remove(chapters.Where(c => c.Number == id).Single());
    chapters.Add(value);
    return Ok();
  }
  catch (InvalidOperationException) // chapter does not exist
  {
    return BadRequest();
  }
}
```

返回 HTTP 状态码的其他方法有：Conflict(409)、Created(201)、InternalServerError(500)、NotFound(404)和 Unauthorized(401)。使用 StatusCode 方法还可以返回更多的状态码。使用这个方法可以返回 HttpStatusCode 枚举定义的任意 HTTP 状态码。这个枚举大约定义了 50 个不同的状态码值。

44.3 创建.NET 客户程序

使用浏览器调用服务是一种简单的测试方法。客户程序常常使用 JavaScript(这是 JSON 的优点)和.NET 客户端。本书创建一个控制台应用程序项目来调用服务。

44.3.1　发送 GET 请求

要发送 GET 请求，应使用 HttpClient 类。对于这种类型，需要引用程序集 System.Net.Http，打开名称空间 System.Net.Http。GetStringAsync 会把一个 HTTP GET 请求发送给链接/api/BookChapters，前面在 Web 浏览器中也使用了这个链接。这个链接调用控制器中的 GetBookChapters 方法。返回的字符串使用 JavaScriptSerializer 转换为 BookChapter 数组(代码文件 BookServiceClientApp/Program.cs)。这个序列化器在名称空间 System.Web.Script.Serialization 的 System.Web.Extension 程序集中定义：

```
private static async Task ReadArraySample()
{
  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11825")
  string response = await client.GetStringAsync("/api/BookChapters");
  Console.WriteLine(response);
  var serializer = new JavaScriptSerializer();
  BookChapter[] chapters =
    serializer.Deserialize<BookChapter[]>(response);

  foreach (BookChapter chapter in chapters)
  {
    Console.WriteLine(chapter.Title);
  }
}
```

注意：
HttpClient 类参见第 26 章，要使用这个类，需要引用程序集 System.Net.Http。

ReadArraySample 方法在 Main 方法中调用：

```
static void Main()
{
  Console.WriteLine("Client app, wait for service");
  Console.ReadLine();
  ReadArraySample().Wait();
  Console.ReadLine();
}
```

要接收 XML 而不是 JSON，客户程序需要发送 Accept 标题 application/xml。为此，使用 HttpClient 类，可以在 DefaultRequestHeaders 的 Accept 集合中添加 MediaTypeWithQualityHeaderValue，返回的 XML 字符串使用 XElement 类解析：

```
private static async Task ReadSampleXml()
{

  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11825");
  client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/xml"));

  string response = await client.GetStringAsync("/api/BookChapters/3");

  XElement chapter = XElement.Parse(response);
```

```
    Console.WriteLine("{0}", chapter);
  }
```

与前面的代码段不同，这里只请求一个 BookChapter，给 URL 添加 3。之后调用控制器中的 GetBookChapters 方法。得到的 XML 如下所示：

```
<BookChapter xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://schemas.datacontract.org/2004/07/BookServiceSample.Models">
  <Number>3</Number>
  <Pages>24</Pages>
  <Title>Objects and Types</Title>
</BookChapter>
```

注意：
XElement 类参见第 34 章。

添加 System.Net.Http.Formatting 程序集，就可以使用辅助方法更方便地转换 BookChapter 对象事件。HttpClient.GetAsync 方法返回一个对象 HttpResponseMessage，它带一个 HttpContent 类型的 Content 属性。内容现在可以用 ReadAsStringAsync 方法读作一个字符串(以前在 HttpClient 类上调用 GetStringAsync 方法来实现)。但是，Formatting 程序集现在包含 ReadAsAsync<T>方法，它会立即转换到泛型类型 T(这里是 BookChapter)：

```
private static async Task ReadWithExtensionsSample()
{
  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11825");
  HttpResponseMessage response =
    await client.GetAsync("/api/BookChapters/3");
  BookChapter chapter =
    await response.Content.ReadAsAsync<BookChapter>();

  Console.WriteLine("{0}. {1}", chapter.Number, chapter.Title);
}
```

44.3.2 发送 POST 请求

HTTP POST 请求的工作方式与 GET 请求类似。这个请求会创建一个新的服务器端对象，调用控制器中的 PostBookChapter 方法。

HttpClient. PostAsync 方法发送 POST 请求。这个请求需要把图书章节作为请求的正文。这个正文利用 ObjectContent<T>类型和 JsonMediaTypeFormatter 传输。这两个类都在 System.Net.Http. Formatting 程序集中。

发送 POST 请求后，再次调用 ReadArraySample 读取图书章节，因此要发送一个 GET 请求。这要验证是否在章节列表中添加了该图书章节：

```
private static async Task AddSample()
{
  var newChapter = new BookChapter
  {
    Title = "ASP.NET Web API",
```

```
    Number = 44,
    Pages = 29
  };

  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11825");

  HttpContent content = new ObjectContent<BookChapter>(
    newChapter, new JsonMediaTypeFormatter());

  HttpResponseMessage response =
    await client.PostAsync("/api/BookChapters", content);

  response.EnsureSuccessStatusCode();

  await ReadArraySample();
}
```

44.3.3　发送 PUT 请求

HTTP PUT 请求用于更新记录，使用扩展方法 PutAsJsonAsync 来发送。这个扩展方法在引入 System.Net.Http. Formatting 程序集后可用，在一个方法调用中发送 PUT 请求时，会格式化内容。

```
private static async Task UpdateSample()
{
  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11825");

  var updatedChapter = new BookChapter
  {
    Title = "Visual Studio 2013",
    Number = 17,
    Pages = 50
  };
  await client.PutAsJsonAsync("/api/BookChapters/17", updatedChapter);

  await ReadArraySample();
}
```

44.3.4　发送 DELETE 请求

最后一个请求是 HTTP DELETE 请求。前面使用了 GetAsync、PostAsync 和 PutAsync，显然，发送 DELETE 请求的方法是 DeleteAsync。下面的代码段调用 EnsureSuccessStatusCode 方法，检查返回的状态码是否正确。如果没有成功的状态码，这个方法就显示它捕获的 HttpRequestException 类型的异常：

```
private static async Task DeleteSample()
{
  try
  {
    var client = new HttpClient();
    client.BaseAddress = new Uri("http://localhost:11825");

    HttpResponseMessage response =
```

```
        await client.DeleteAsync("/api/BookChapters/42");

      response.EnsureSuccessStatusCode();

      await ReadArraySample();
    }
    catch (HttpRequestException ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

44.4 Web API 路由和操作

ASP.NET MVC 路由根据 URL 来定义，而在 ASP.NET Web API 中，路由基于 HTTP 方法。GET 请求路由到以 Get 开头的控制器方法，POST 请求路由到以 Post 开头的控制器方法。方法命名为 Get 还是 GetBookChapters 并不重要，这两个方法名都与 GET 请求匹配。

ASP.NET Web API 的路由在 BookServiceSample/App_Start/WebApiConfig.cs 文件中定义。Register 方法在 Global.asax.cs 的应用程序启动代码中调用：

```
public static void Register(HttpConfiguration config)
{
  //...

  config.Routes.MapHttpRoute(
      name: "DefaultApi",
      routeTemplate: "api/{controller}/{id}",
      defaults: new { id = RouteParameter.Optional }
  );

  //...
}
```

这个路由定义非常类似于 ASP.NET MVC 路由定义。在 ASP.NET MVC 中，路由由 MapRoute 方法定义。MapHttpRoute 方法不路由 URL 链接，而是映射 HTTP 请求。

路由模板用 URL 指定 api/{controller}/{id}。api 用作路由中的前缀，以区分 Web API 控制器和 MVC 控制器。控制器指定控制器的名称，本章的第一个示例把 BookChapters 作为控制器名称。id 是参数的可选名称。方法 GetBookChapter、PutBookChapter 和 DeleteBookChapter 都把 id 定义为参数。

44.4.1 给操作添加 HTTP 方法

前面简单的命名约定用于将 HTTP 方法映射到动作方法。HTTP GET 请求映射到方法 GetBookChapter，HTTP POST 请求映射到方法 PostBookChapter。

```
public IEnumerable<BookChapter> GetBookChapters()
{
  return chapters;
}
```

```
public void PostBookChapter([FromBody]BookChapter value)
{
  chapters.Add(value);
}
```

如果动作方法的名称没有定义到 HTTP 方法的映射，也可以使用特性来定义。例如，一个方法的名称是 BookChapters，就可以使用 HttpGet 特性把它映射到 HTTP GET 请求：

```
[HttpGet]
public IEnumerable<BookChapter> BookChapters()
{
  return chapters;
}
```

与 GET 请求类似，HEAD、POST、PUT 和 DELETE 请求可以使用其他特性：HttpHead、HttpPost、HttpPut 和 HttpDelete。另一个选项是使用 AcceptVerb 特性。这个特性允许将多个 HTTP 方法赋予一个动作方法。假定有一个与动作方法不同的 URL 链接，就可以使用 ActionName 特性：

```
[AcceptVerbs("POST")]
[ActionName("bookchapter")]
public void AddBookChapter([FromBody]BookChapter value)
{
  chapters.Add(value);
}
```

为了取消控制器中公共方法与 URL 的映射，可以使用 NonAction 特性。

44.4.2 基于特性的路由

基于约定的路由的优点是，它在一个地方定义(调用 MapHttpRoute 方法)，不需要用控制器指定任何映射。但是它很难用于大量应定义父子关系的控制器。例如，一本书包含许多章节，映射书中 id 为 42 的第 11 章的 URL 应是 books/42/bookchapters/11。ASP.NET Web API 2 包含一个新功能，可以利用特性来路由，其方式与基于约定的路由不同。

要启动基于特性的路由，需要用注册路由调用 MapHttpAttributeRoutes 方法(代码文件 BookServiceSample/App_Start/WebApiConfig.cs)。在 Web API 2 中，这个方法定义为扩展方法，位于程序集 System.Web.Http 的名称空间 System.Web.Http 中：

```
public static void Register(HttpConfiguration config)
{
  config.MapHttpAttributeRoutes();

  config.Routes.MapHttpRoute(
      name: "DefaultApi",
      routeTemplate: "api/{controller}/{id}",
      defaults: new { id = RouteParameter.Optional }
  );
}
```

为了用基于特性的路由显示引用的条目，创建 Book 类(代码文件 BookServiceSample/Models/Book.cs)：

```
public class Book
{
  public Book(int id, string title, params BookChapter[] chapters)
  {
    this.Id = id;
    this.Title = title;
    this.BookChapters = chapters.ToList();

  }
  public int Id { get; private set; }
  public string Title { get; private set; }

  public ICollection<BookChapter> BookChapters { get; private set;}
}
```

利用控制器，创建一个图书列表和章节列表(代码文件 BookServiceSample/Controllers/BookChaptersAttrController.cs)：

```
private static List<BookChapter> chapters;
private static List<Book> books;
static BookChaptersAttrController()
{
  chapters = new List<BookChapter>()
  {
    new BookChapter { Number=1, Title=".NET Architecture", Pages=20},
    new BookChapter { Number=2, Title="Core C#", Pages=42},
    new BookChapter { Number=3, Title="Objects and Types", Pages=24},
    new BookChapter { Number=4, Title="Inheritance", Pages=18},
    new BookChapter { Number=5, Title="Generics", Pages=22},
    new BookChapter { Number=17, Title="Visual Studio 2012", Pages=50},
    new BookChapter { Number=42, Title="ASP.NET Dynamic Data", Pages=14}
  };
  books = new List<Book>()
  {
    new Book(1, "Professional C# 5 and .NET 4.5.1", chapters.ToArray()),
    new Book(2, "Professional ASP.NET MVC 4")
  };
}
```

在基于特性的路由中，Route 特性可以应用于动作方法。如果应用了这个特性，就使用基于特性的路由。对于没有这种特性的动作方法，就使用基于约定的路由。有了 Route 特性，模板可以指定为在花括号中包含方法的参数。下面的路由应用于 URL http://server/books/2，其中 2 传递为 bookId 参数：

```
[Route("books/{bookId}")]
public IEnumerable<BookChapter> GetBookChapters(int bookId)
{
  return books.Where(b => b.Id == bookId).Single().BookChapters;
}
```

使用 Route 特性的模板，也可以传递多个参数。对于 URL http://server/books/1/chapters/5，返回图书中 id 为 1 的第 5 章：

```
[Route("books/{bookId}/chapters/{chapterId}")]
public BookChapter GetBookChapter(int bookId, int chapterId)
{
  return books.Where(b => b.Id == bookId).Single().BookChapters.
    Where(c => c.Number == chapterId).SingleOrDefault();
}
```

也可以用类型限制参数，如下所示：

```
[Route("books/{bookId:int}/chapters/{chapterId:int}")]
```

控制器的所有动作方法常常使用相同的路由前缀。这种前缀可以使用 RoutePrefix 特性赋予控制器类型：

```
[RoutePrefix("booksamples")]
public class BookChaptersAttrController : ApiController
{
//...
```

用这种方式处理动作方法，就需要定义前缀后面的部分，例如用下面代码段中的两个参数来定义：

```
[Route("{bookId:int}/{chapterId:int}")]
public BookChapter GetBookChapter(int bookId, int chapterId)
{
  return books.Where(b => b.Id == bookId).Single().BookChapters.
    Where(c => c.Number == chapterId).SingleOrDefault();
}
```

使用下一节介绍的 OData 和 ASP.NET Web API Service 提供资源时，使用基于特性的路由是非常可行的。

44.5 使用 OData

ASP.NET Web API 为 OData(Open Data Protocol)提供了直接支持。OData 通过 HTTP 协议提供了对数据源的 CRUD 访问。发送 GET 请求会检索一组实体数据，POST 请求会创建一个新实体，PUT 请求会更新已有的实体，DELETE 请求会删除实体。前面介绍了应用于控制器中动作方法的 HTTP 方法。OData 基于 JSON 和 AtomPub(一种 XML 格式)进行数据序列化。ASP.NET Web API 也有这个功能。OData 提供的其他功能有：每个资源都可以用简单的 URL 查询来访问。为了说明其工作方式，以及 ASP.NET Web API 如何实现这个功能，下面举例说明，从一个数据库开始。

44.5.1 创建数据模型

用于 OData 的示例应用程序 Formula1ServiceSample 访问 Formula 1 数据库，该数据库可以与示例代码一起下载。

首先，使用 ADO.NET Entity Data Model 设计器创建一个模型(在 Models 目录中)，选择表 Racers、RaceResults、Races 和 Circuits，如图 44-2 所示。

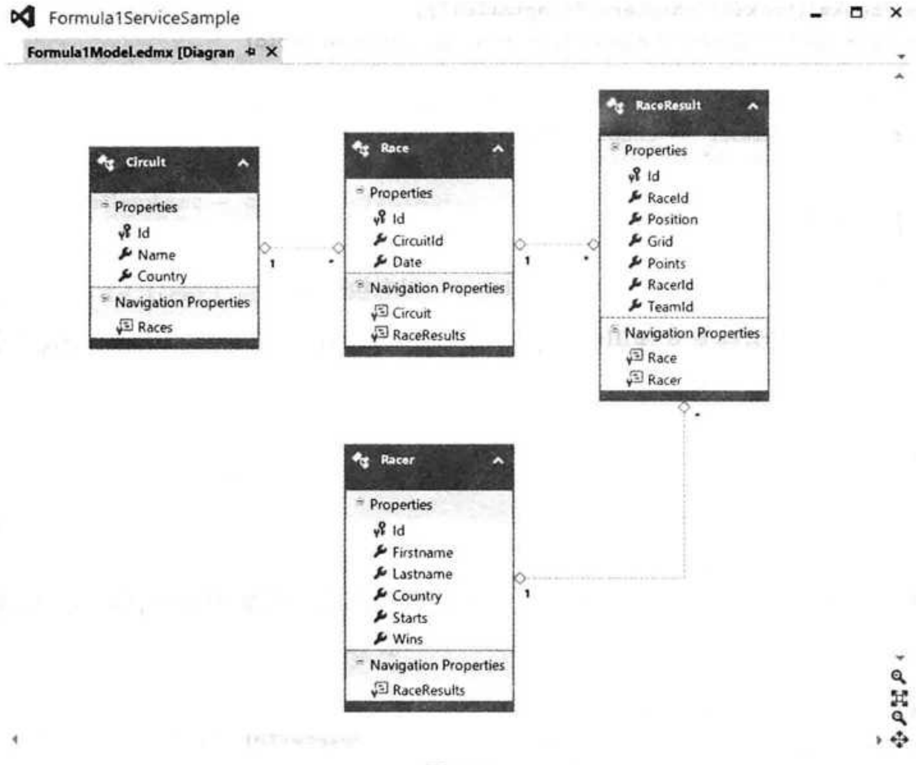


图 44-2

该设计器会创建类 Racer、RaceResult、Race 和 Circuit(它们分别映射对应的数据库表)，以及数据库上下文类 Formula1Entities。Formula1Entities 管理与数据库的连接，并查询、更新数据。

注意：

ADO.NET Entity Framework 参见第 33 章。

44.5.2 创建服务

有了 ADO.NET Entity Framework 模型，就可以使用搭建功能创建一个 OData 服务。要使用搭建功能，可以用 Add Controller 菜单添加一个新控制器，再选择选项“Web API 2 OData Controller with actions, using Entity Framework”，之后会打开如图 44-3 所示的对话框。在这里可以选择模型类和数据上下文类。新控制器的名称是 RacerController，模型类是 Racer，数据上下文类是 Formula1Entities。

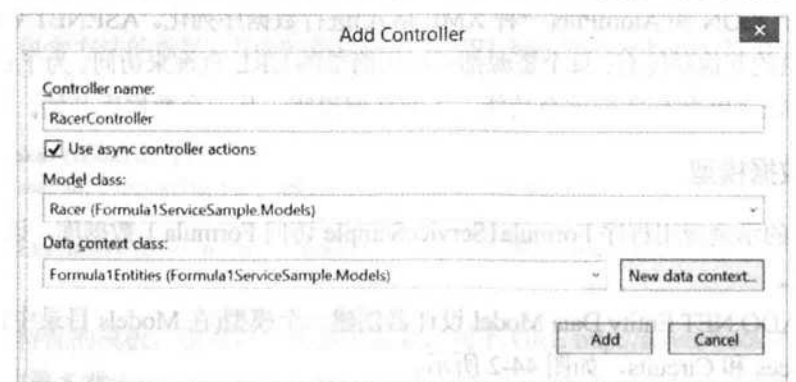


图 44-3

新生成的控制器不同于以前的控制器，它有基类 ODataController(而不是 ApiController)和特性 Queryable，其动作方法如下所示(代码文件 Formula1ServiceSample/Controllers/RacerController.cs)：

```
public class RacerController : ODataController
{
  private Formula1Entities db = new Formula1Entities();

  // GET odata/Formula1
  [Queryable]
  public IQueryable<Racer> GetRacer()
  {
    return db.Racers;
  }

  // GET odata/Formula1(5)
  [Queryable]
  public SingleResult<Racer> GetRacer([FromODataUri] int key)
  {
    return SingleResult.Create(db.Racers.Where(
      racer => racer.Id == key));
  }

  protected override void Dispose(bool disposing)
  {
    if (disposing)
    {
      db.Dispose();
    }
    base.Dispose(disposing);
  }
}
```

基类 ODataController 派生自 ApiController，提供了额外的受保护的虚拟泛型方法 Created 和 Updated。这两个方法在创建和更新实体时从控制器代码中调用，给 POST 和 PUT HTTP 方法创建动作的结果。

在服务中提供 OData，需要调用方法 MapODataRoute 来定义另一个路由(代码文件 Formula1ServiceSample/App_Start/WebApiConfig.cs)。示例代码使用 odata 作为路由的前缀，不会与其他 Web API 服务调用中的 OData 请求(它默认使用 api)冲突。方法 MapODataRoute 的第三个参数需要通过接口 IEdmModel 提供的模型元数据。ODataConventionModelBuilder 根据约定把 CLR 类型映射为 EDM 模型——因此它允许使用 Code First 和 ADO.NET Entity Framework。GetEdmModel 方法返回需要的 IEdmModel：

```
using Formula1ServiceSample.Models;
using System.Web.Http;
using System.Web.Http.OData.Builder;

namespace Formula1ServiceSample
{
  public static class WebApiConfig
  {
    public static void Register(HttpConfiguration config)
```

```
    {
      var builder = new ODataConventionModelBuilder();
      builder.EntitySet<Racer>("Racer");
      builder.EntitySet<RaceResult>("RaceResult");
      builder.EntitySet<Race>("Race");
      config.Routes.MapODataRoute("odata", "odata", builder.GetEdmModel());

      // Web API routes
      config.MapHttpAttributeRoutes();

      config.Routes.MapHttpRoute(
         name: "DefaultApi",
         routeTemplate: "api/{controller}/{id}",
         defaults: new { id = RouteParameter.Optional }
      );
    }
  }
}
```

44.5.3 OData 查询

使用下面的 URL 很容易获得数据库中的所有参赛选手(端口号可能与读者的系统不同)：

```
http://localhost:36089/odata/Racer
```

要只获取一个选手，可以把该选手的标识符和 URL 一起传递给方法。这个请求会调用 GetRacer 动作方法，并传递返回 SingleResult<Racer>的键：

```
http://localhost:36089/odata/Racer(11)
```

每个选手都有多个结果。在一个 URL 查询中，还可以获取一个选手的所有比赛结果：

```
http://localhost:36089/odata/Racer(11)/RaceResults
```

所有查询选项的要求都是应用于动作方法的 Queryable 特性。

OData 提供的查询选项多于 ASP.NET Web API 支持的选项。OData 规范允许给服务器传递参数，以分页、筛选和排序。下面介绍这些选项。

为了只给客户端返回数量有限的实体，客户端可以使用$top参数限制数量。也允许使用$skip进行分页，例如，可以跳过 20 个结果，再提取 10 个结果：

```
http://localhost:36089/odata/Racer?$top=10&$skip=20
```

使用$top 和$skip 选项，客户端可确定要检索的实体数。如果希望限制客户端可以请求的内容，例如，一个调用就不应请求上百万条记录，就可以配置 Queryable 特性来限制这个方面。把 PageSize 设置为 10，一次最多返回 10 个实体：

```
[Queryable(PageSize=10)]
```

Queryable 特性还有一些命名参数来限制查询，例如最大的 top 和 skip 值，最大的扩展深度以及排序的限制。

为了根据 Racer 类的属性筛选请求，可以将$filter 选项应用于 Racer 的属性。为了筛选出澳大利亚选手，可以使用 eq 操作符(等于)和$filter 选项：

```
http://localhost:36089/odata/Racer?$filter=Country eq 'Austria'
```

$filter 选项还可以与 lt(小于)和 gt(大于)操作符一起使用。下面的请求仅返回获胜次数超过 20 的选手：

```
http://localhost:36089/odata/Racer?$filter=Wins gt 20
```

为了请求有序的结果，$orderby 选项定义了排序顺序。添加 desc 关键字按降序排序：

```
http://localhost:36089/odata/Racer?$orderby=Wins%20desc
```

使用 HttpClient 类很容易给服务发出所有这些请求。但是，还有其他选项，例如使用 WCF Data Services 服务创建的代理，如下一节所述。

44.5.4　WCF Data Services 客户程序

WCF Data Services 是 OData 的另一个技术。WCF Data Services 包含两个部分：服务器技术和客户端技术。

服务器技术可以与 ASP.NET Web API 比较。很容易(可能更容易)创建基于 ADO.NET Entity Framework 的 OData 服务，但用自定义代码调整它就不太容易了，使用自定义代码而不是 Entity Framework 时，更新和删除也不方便。对于读取操作，需要 IQueryable 接口(这很容易，对于列表，需要实现 IEnumerable 接口)。要创建、更新和删除实体，IDataServiceUpdateProvider 接口需要做更多的工作——比使用 ASP.NET Web API 的 Post、Put 和 Delete 动作方法多得多。

WCF Data Services 的客户端技术没有与 ASP.NET Web API 对应的技术。但在服务器端有 Web API 时，很容易使用 WCF Data Services 的客户端部分。

要使用 Formula 1 服务提供的 OData 服务，创建一个控制台应用程序。要创建 WCF Data Services 的客户端，给项目添加一个服务引用，如图 44-4 所示。

图 44-4

在 Add Service Reference 中，根据 OData 服务的元数据创建实体类型。这里显示的示例是 Racer 类。生成的类(代码文件 Formula1ServiceSample/ClientApp/Service References/Reference.cs)实现了 INotifyPropertyChanged，使客户端上下文获得更改通知。设置访问器不仅触发 INotifyPropertyChanged 接口定义的通知，还调用部分方法，这些部分方法可以在该类的另一个部分实现。

```
public partial class Racer : INotifyPropertyChanged
{
  [GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
  public static Racer CreateRacer(int ID)
  {
    Racer racer = new Racer();
    racer.Id = ID;
    return racer;
  }
  [GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
  public int Id
  {
    get
    {
      return this._Id;
    }
    set
    {
      this.OnIdChanging(value);
      this._Id = value;
      this.OnIdChanged();
      this.OnPropertyChanged("Id");
    }
  }
  [GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
  private int _Id;
  partial void OnIdChanging(int value);
  partial void OnIdChanged();
  [GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
  public string Firstname
  {
    get
    {
      return this._Firstname;
    }
    set
    {
      this.OnFirstnameChanging(value);
      this._Firstname = value;
      this.OnFirstnameChanged();
      this.OnPropertyChanged("Firstname");
    }
  }
  [GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
  private string _Firstname;
  partial void OnFirstnameChanging(string value);
  partial void OnFirstnameChanged();
```

```
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public string Lastname
{
  get
  {
    return this._Lastname;
  }
  set
  {
    this.OnLastnameChanging(value);
    this._Lastname = value;
    this.OnLastnameChanged();
    this.OnPropertyChanged("Lastname");
  }
}
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
private string _Lastname;
partial void OnLastnameChanging(string value);
partial void OnLastnameChanged();
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public string Country
{
  get
  {
    return this._Country;
  }
  set
  {
    this.OnCountryChanging(value);
    this._Country = value;
    this.OnCountryChanged();
    this.OnPropertyChanged("Country");
  }
}
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
private string _Country;
partial void OnCountryChanging(string value);
partial void OnCountryChanged();
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public Nullable<int> Starts
{
  get
  {
    return this._Starts;
  }
  set
  {
    this.OnStartsChanging(value);
    this._Starts = value;
    this.OnStartsChanged();
    this.OnPropertyChanged("Starts");
  }
}
```

```
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
private global::System.Nullable<int> _Starts;
partial void OnStartsChanging(global::System.Nullable<int> value);
partial void OnStartsChanged();
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public Nullable<int> Wins
{
  get
  {
    return this._Wins;
  }
  set
  {
    this.OnWinsChanging(value);
    this._Wins = value;
    this.OnWinsChanged();
    this.OnPropertyChanged("Wins");
  }
}
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
private global::System.Nullable<int> _Wins;
partial void OnWinsChanging(global::System.Nullable<int> value);
partial void OnWinsChanged();
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public DataServiceCollection<RaceResult> RaceResults
{
  get
  {
    return this._RaceResults;
  }
  set
  {
    this._RaceResults = value;
    this.OnPropertyChanged("RaceResults");
  }
}
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
private DataServiceCollection<RaceResult> _RaceResults =
new DataServiceCollection<RaceResult>(null, TrackingMode.None);
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
public event PropertyChangedEventHandler PropertyChanged;
[GeneratedCodeAttribute("System.Data.Services.Design", "1.0.0")]
protected virtual void OnPropertyChanged(string property)
{
  if ((this.PropertyChanged != null))
  {
    this.PropertyChanged(this, PropertyChangedEventArgs(property));
  }
}
```

为了对服务执行查询，WCF Dtat Services 允许定义映射到 URL 请求的 LINQ 查询。首先，创建一个 Container 对象(代码文件 Formula1ServiceSample/ClientApp/Program.cs)。Container 派生自基类

DataServiceContext，管理连接，并把请求发送给服务。这个上下文相当于 ADO.NET Entity Framework 使用的数据上下文，用于管理到数据库的连接，并跟踪已加载的对象。对 Container 的 Racer 属性执行 LINQ 查询(这个属性属于类 DataServiceQuery<Racer>)，把 LINQ 请求转换为 URL 字符串。使用 foreach 语句迭代结果，调用服务并使用结果：

```
private static void ReadSample()
{
  Uri serviceRoot = new Uri("http://localhost:36089/odata");
  var container = new Container(serviceRoot);
  var q = from r in container.Racer
          where r.Country == "Austria"
          orderby r.Wins descending
          select r;
  foreach (var r in q)
  {
    Console.WriteLine("{0} {1}", r.Firstname, r.Lastname);
  }
}
```

这个 LINQ 查询筛选出澳大利亚选手，按获胜次数排序，将这个 LINQ 查询转换为如下 URL 请求：

```
http://localhost:36089/odata/Racer()?$filter=Country eq
'Austria'&$orderby=Wins desc
```

其中，OData 查询选项$filter 和$orderby 的用法与前面相同。使用 WCF Dtat Services，可以只考虑 LINQ 请求，OData 查询会自动完成。

要发送 HTTP POST 请求来创建一条新记录，可以调用 Container 的 AddToRacer，给上下文添加一个新对象，再调用 SaveChanges 给服务发出请求。SaveChanges 的结果是 DataServiceResponse 类型，它允许检查错误：

```
private static void CreateSample()
{
  Uri serviceRoot = new Uri("http://localhost:36089/odata");
  var container = new Container(serviceRoot);
  container.AddToRacer(
    new Racer
    {
      Firstname = "Valtteri",
      Lastname = "Botas",
      Country = "Finland",
      Wins = 0,
      Starts = 19
    });
  DataServiceResponse resp = container.SaveChanges();
}
```

更新记录在 HTTP PUT 请求中进行，它会修改已有的对象，调用 UpdateObject 方法，再调用 SaveChanges：

```
private static void UpdateSample()
{
```

```
  Uri serviceRoot = new Uri("http://localhost:36089/odata");
  Container container = new Container(serviceRoot);
  var r1 = (from r in container.Racer
            where r.Firstname == "Sebastian" && r.Lastname == "Vettel"
            select r).FirstOrDefault();
  r1.Starts = 120;
  r1.Wins = 39;
  container.UpdateObject(r1);
  DataServiceResponse resp = container.SaveChanges();
}
```

44.6 保护 Web API

为了限制对 Web API 的访问，支持与 ASP.NET Web Forms 和 ASP.NET MVC 相同的技术。创建项目时，可以修改身份验证选项，如图 44-5 所示。

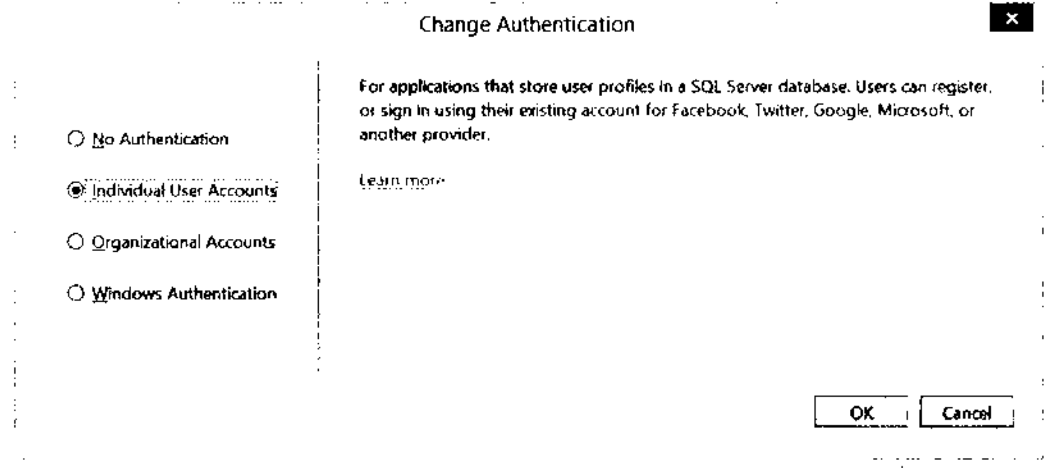


图 44-5

给身份验证选择 Individual User Account 选项，会创建一个 Web API AccountController。这个控制器定义了一些方法来管理用户，例如用户注册和修改密码。ValuesController 应用了 Authorize 特性，所以只有授权用户才被允许访问控制器的动作(代码文件 SecureWebAPI/Controllers/ValuesController.cs)：

```
[Authorize]
public class ValuesController : ApiController
{
  public IEnumerable<string> Get()
  {
    return new string[] { "value1", "value2" };
  }
  //...
```

下面使用前面的 HttpClient 类向这个 ValuesController 发出请求(代码文件 SecureWebAPI/ClientApp/Program.cs)：

```
private async static void NotAuthenticated()
{
  string valuesUri = "/api/Values";
```

```
  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11663");
  HttpResponseMessage resp = await client.GetAsync(valuesUri);

  Console.WriteLine(resp.StatusCode);
  string result = await resp.Content.ReadAsStringAsync();
}
```

在这里，返回了 HTTP 状态码 401 和 StatusCode 消息 Unauthorized。其内容包含如下字符串：Authorization has been denied for this request。还有更多信息：需要身份验证模式的信息。这些信息在 HttpResponseMessage:Headers.Www.Authenticate 的标题中。在这个标题中找到的模式是 Bearer。这是和 OAuth 一起使用的模式。bearer 令牌需要传递给 Web API 服务，以成功获得授权。

下面使它能工作。

44.6.1 创建账户

对于用户的身份验证，需要一个账户。使用新的安全模型，会直接支持带有 OAuth 验证支持的账户——例如 Facebook、Twitter、Google 和 Microsoft 账户。这样，在 Web API 服务和令牌验证服务之间建立信任的其他服务器，就会用于验证用户。本章后面将说明其他令牌服务如何使用。但是，Account 控制器也支持直接在数据库中创建用户。

这个控制器定义了一个可用于匿名用户的 Register 方法(应用了 AllowAnonyous 特性)，允许匿名用户注册并创建本地用户。这个方法的路由通过特性路由来定义(代码文件 SecureWebAPI/Controllers/AccountController.cs)。UserManager.CreateAsync 把用户信息写入数据库。这个类在名称空间 Microsoft.AspNet.Identity 中定义：

```
[AllowAnonymous]
[Route("Register")]
public async Task<IHttpActionResult> Register(RegisterBindingModel model)
{
  if (!ModelState.IsValid)
  {
    return BadRequest(ModelState);
  }

  IdentityUser user = new IdentityUser
  {
    UserName = model.UserName
  };

  IdentityResult result = await UserManager.CreateAsync(user,
    model.Password);
  IHttpActionResult errorResult = GetErrorResult(result);

  if (errorResult != null)
  {
    return errorResult;
  }
  return Ok();
}

private IHttpActionResult GetErrorResult(IdentityResult result)
```

```
{
  if (result == null)
  {
    return InternalServerError();
  }

  if (!result.Succeeded)
  {
    if (result.Errors != null)
    {
      foreach (string error in result.Errors)
      {
        ModelState.AddModelError("", error);
      }
    }

    if (ModelState.IsValid)
    {
      // No ModelState errors are available to send, so just return
      // an empty BadRequest.
      return BadRequest();
    }

    return BadRequest(ModelState);
  }
  return null;
}
```

下面调用这个服务，注册一个用户(代码文件 SecureWebAPI/ClientApp/Program.cs)。注册的链接(通过特性路由来定义)是 api/Account/Register。动作方法需要一个 RegisterBindingModel 参数。这个模型定义了属性 UserName、Password 和 ConfirmPassword。要把这些值传递给服务，应创建一个匿名类，用 PostAsJsonAsync 扩展方法格式化为一个 JSON 字符串。这个辅助方法在程序集 System.Net.Http.Formatting 的 System.Net.Http 名称空间中定义：

```
private async static void RegisterUser()
{
  string registerUri = "/api/Account/Register";
  HttpClient client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11663");

  var user = new
  {
    UserName = "christian",
    Password = "Password123",
    ConfirmPassword = "Password123"
  };

  HttpResponseMessage resp =
    await client.PostAsJsonAsync(registerUri, user);

  resp.EnsureSuccessStatusCode();
  Console.WriteLine("registered successfully");
}
```

这就是用服务创建用户所需做的工作。现在可以创建验证令牌了。

44.6.2 创建验证令牌

响应验证令牌请求的代码在 Startup 类的静态构造函数中定义(代码文件 SecureWebAPI/App_Start/Startup.Auth.cs)。这个构造函数把 OAuthAuthorizationServerOptions 的 TokenEndPointPath 设置为/Token——这是请求令牌的 URL。用 Provider 属性定义的提供程序类是 ApplicationOAuthProvider。这个类从 Visual Studio 模板中创建，可以在 SecureWebAPI/Providers 文件夹中找到该提供程序的完整实现代码:

```
static Startup()
{
  PublicClientId = "self";

  UserManagerFactory = () => new UserManager<IdentityUser>(
    new UserStore<IdentityUser>());

  OAuthOptions = new OAuthAuthorizationServerOptions
  {
    TokenEndpointPath = new PathString("/Token"),
    Provider = new ApplicationOAuthProvider(PublicClientId,
      UserManagerFactory),
    AuthorizeEndpointPath = new PathString("/api/Account/ExternalLogin"),
    AccessTokenExpireTimeSpan = TimeSpan.FromDays(14),
    AllowInsecureHttp = true
  };
}
```

注意:

属性 AllowInsecureHttp 默认设置为 true，允许发送 HTTP 请求，这适合于测试。但是在产品环境中，应使用 HTTPS 运行服务，把这个属性设置为 false。

下面从客户端向服务请求令牌。HttpClient 类现在用于对 Token URI 的 HTTP POST 请求。对于 POST 数据，需要给验证服务发送 grant_type、username 和 password。接收到的数据是一个 JSON 对象，它带有 toke-type 和 access_token 的值。要把这些信息传递给对象，可以用这些名称的属性创建一个类。示例代码说明，这里也可以使用动态类型。用泛型参数 dynamic 调用方法 ReadAsAsync，会用返回令牌的属性填充对象。使用 toke-type 和 access_token 很容易访问这些值。编译器不能验证这里使用的属性，不会弹出编译错误。这里使用属性名称，因为这些值在运行期间从 JSON 对象中填充进来。把类型和访问令牌写入控制台后，GetToken 方法返回动态的令牌对象。得到的类型是 bearer，从前面的解释可以看出，这是通过 ASP.NET Web API 引入的安全功能:

```
private async static Task<dynamic> GetToken()
{
  string tokenUri = "/Token";
  var client = new HttpClient();
  client.BaseAddress = new Uri("http://localhost:11663");

  HttpContent content = new FormUrlEncodedContent(
          new List<KeyValuePair<string, string>> {
```

```
                new KeyValuePair<string, string>("grant_type", "password"),
                new KeyValuePair<string,string>("username","christian"),
                new KeyValuePair<string,string>("password","Password123"),
            });
    content.Headers.ContentType =
      new MediaTypeHeaderValue("application/x-www-form-urlencoded");
    content.Headers.ContentType.CharSet = "UTF-8";

    HttpResponseMessage resp = await client.PostAsync(tokenUri, content);

    resp.EnsureSuccessStatusCode();
    dynamic token = await resp.Content.ReadAsAsync<dynamic>();
    Console.WriteLine("{0}", token.token_type);
    Console.WriteLine("{0}", token.access_token);
    Console.WriteLine();
    return token;
  }
```

注意：
动态类型参见第 12 章。

44.6.3 发送验证过的调用

现在可以使用令牌信息对服务发送验证过的调用。在创建令牌之前，调用 Unauthorized 结果中的 ValuesController。有了令牌后，调用现在应是成功的。访问令牌可以通过使用 HttpClient 的 DefaultRequestHeaders 属性添加到 Authorization 标题中。需要传递令牌类型和访问令牌，接着与以前一样，向 URI /api/Values 发出 GET 请求，这次是成功的，把 JSON 结果写到控制台上：

```
  private async static void Authenticated()
  {
    dynamic token = await GetToken();

    string valuesUri = "/api/Values";
    var client = new HttpClient();
    client.BaseAddress = new Uri(baseAddress);

    client.DefaultRequestHeaders.Add("Authorization",
      string.Format("{0} {1}", token.token_type, token.access_token));

    HttpResponseMessage resp = await client.GetAsync(valuesUri);

    Console.WriteLine(resp.StatusCode);

    string content = await resp.Content.ReadAsStringAsync();
    Console.WriteLine(content);
  }
```

44.6.4 获取用户信息

AccountController 定义了一个模板生成的动作方法 GetUserInfo，它以 UserInfoViewModel 对象的形式使用信息。这个类定义了属性 UserName、HasRegistered 和 LoginProvider，它们在方法的实现代码中填充。User 是基类 ApiController 的一个属性，在给用户授权时填充。使用 FromIdentity 方

法，会检查表示用户身份的 ClaimsIdentity，以检索用户信息，最终返回这些信息：

```
[HostAuthentication(DefaultAuthenticationTypes.ExternalBearer)]
[Route("UserInfo")]
public UserInfoViewModel GetUserInfo()
{
  ExternalLoginData externalLogin = ExternalLoginData.FromIdentity(
    User.Identity as ClaimsIdentity);

  return new UserInfoViewModel
  {
    UserName = User.Identity.GetUserName(),
    HasRegistered = externalLogin == null,
    LoginProvider =
      externalLogin != null ? externalLogin.LoginProvider : null
  };
}
```

下面在客户端进行一个调用。这次不同于以前：没有使用动态类型(这也是可以的)，但生成了一个新类，它定义了服务生成的 JSON 结果所要包含的成员。

```
class UserInfo
{
  public string UserName { get; set; }
  public bool HasRegistered { get; set; }
  public string LoginProvider { get; set; }
}
```

与前面已验证的调用一样，添加了授权的请求标题，然后用 GetAsync 调用 UserInfo 请求。使用泛型方法 ReasAsAsync，并传递 UserInfo 类型来填充 UserInfo 类型的对象：

```
private async static void UserInfo()
{
  string userInfoUri = "/api/Account/UserInfo";
  var token = await GetToken();
  HttpClient client = new HttpClient();
  client.BaseAddress = new Uri(baseAddress);
  client.DefaultRequestHeaders.Add("Authorization",
    string.Format("{0} {1}", token.token_type, token.access_token));

  HttpResponseMessage resp = await client.GetAsync(userInfoUri);
  resp.EnsureSuccessStatusCode();
  UserInfo userInfo = await resp.Content.ReadAsAsync<UserInfo>();
  Console.WriteLine("user: {0}, registered: {1}, provider: {2}",
    userInfo.UserName, userInfo.HasRegistered, userInfo.LoginProvider);
}
```

44.7　自驻留

WCF 的一个主要优点是，它不需要驻留在 IIS 中或存储了 ASP.NET 运行库的另一台服务器上。ASP.NET Web API 在这方面也非常灵活。很容易自驻留一个 ASP.NET Web API 服务器。

可以创建任意应用程序类型，例如可以创建一个 Console 应用程序，与 SelfHostApp 示例一样。只需要添加 NuGet 包 Microsoft ASP.NET Web API 2 Self Host，且 ID 是 Microsoft.AspNet.WebApi.SelfHost。

在可执行程序中，通过派生基类 ApiController，创建 BooksController，如本章前面所示。示例代码使用基于特性的路由。另外，从控制器返回的 Book 类型只是一个带有 Publisher 和 Title 属性的简单类：

```
[RoutePrefix("Books")]
public class BooksController : ApiController
{
  [Route("TheOne")]
  public IEnumerable<Book> GetBooks()
  {
    return new List<Book>()
    {
      new Book { Publisher="Wrox Press", Title="Professional C# 5"}
    };
  }
}
```

创建宿主所在的 Main 方法会更有趣。自驻留是非常简单的——只需几行自定义代码。路由配置用 HttpSelfHostConfiguration 类(在 System.Web.Http.SelfHost 名称空间中)完成。调用 MapHttpAttributeRoutes 支持基于特性的路由，这与在控制器中使用它相同。当然，使用方法名的传统路由也是可行的，使用 Routes 属性可以添加路由。宿主还需要一个未使用的端口号——换言之，未被 IIS 使用。配置传递给服务器——HttpSelfHostServer。用这个宿主服务器类型调用 OpenAsync，会启动服务器的监听器线程，以监听请求：

```
using System;
using System.Web.Http;
using System.Web.Http.SelfHost;

namespace SelfHostApp
{
  class Program
  {
    static void Main()
    {
      var config = new HttpSelfHostConfiguration("http://localhost:8081");
      config.MapHttpAttributeRoutes();

      using (var server = new HttpSelfHostServer(config))
      {
        server.OpenAsync().Wait();

        Console.WriteLine("Press Enter to quit.");
        Console.ReadLine();
      }
    }
  }
}
```

用户必须有权限创建所请求端口号的监听器。可以在提升模式下启动 Visual Studio 来获得这些权限，也可以使用 netsh 实用工具给用户定义权限：

```
netsh http add urlacl url=http://+:8081 user=username listen=yes
```

启动可执行程序，可以使用浏览器向 http://localhost:801/Books/TheOne 发送请求，获得服务的结果。当然，也可以用 HttpClient 类创建客户程序，而不是使用浏览器，其方式如前所述。

完成了服务后，最好解除端口号的权限。这可以使用下面的 netsh 命令实现：

```
netsh http delete urlacl url=http://+:8081
```

44.8 小结

本章描述了 ASP.NET Web API 的功能，它提供的功能允许使用 HttpClient 类创建服务，并在任何客户端(无论是 JavaScript 还是.NET 客户端)调用。返回 JSON 或 XML。

本章还介绍了 OData，它使用资源标识符，很容易引用树中的数据。ASP.NET Web API 2 添加了对 OData 的许多支持。

本章介绍了这种技术的服务器端部分和客户端部分，在客户端的一个数据服务上下文中跟踪变更信息。因为 WCF Data Services 的客户端部分实现一个 LINQ 提供程序，所以可以创建简单的 LINQ 请求，LINQ 请求可以转换为 HTTP GET/POST/PUT/DELETE 请求。

第 45 章将介绍 Windows Workflow Foundation (WF)，它允许图形化地指定不同的活动来构建工作流。这种工作流可以描述 WCF 服务，因此不仅可用于驻留工作流，还可以驻留 WCF 服务。

第45章 Windows Workflow Foundation

本章要点

- 可以创建的不同类型的工作流
- 描述一些内置活动
- 创建自定义活动
- 工作流概述

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- Hello World 示例
- 并行执行
- Pick Demo 示例
- 自定义活动
- 实参与变量
- 工作流应用程序
- 工作流即服务
- 动态更新
- 驻留设计器

45.1 工作流概述

本章将概述 Windows Workflow Foundation 4.5(本章称之为 WF 和 Workflow)，它提供了一个模型，在该模型中，可以使用一组构建块(称为活动)定义和执行进程。WF 还提供了一个设计器，在默认情况下，该设计器位于 Visual Studio 中，允许将工具箱中的活动拖放到设计界面上，从而创建一个工作流模板。

接着，这个模板就可以以许多不同的方式执行，本章将介绍这些方式。在工作流执行时，它需要访问外界，一般可以使用几个方法来访问外部世界。另外，工作流也需要保存和还原其状态，例如，需要长时间等待时。

工作流由许多活动构成，这些活动在运行期间执行。活动可以发送电子邮件、更新数据库中的一行，或在后端系统上执行一个事务。有许多内置活动，它们用于一般性的工作，也可以创建自己的自定义活动，根据需要将它们插入工作流中。

在 Visual Studio 2013 中，实际上有两个版本的 Workflow，3.x 版本随.NET Framework 3 一起发布(名称空间 System.Workflow 及其子名称空间，它们也由 SharePoint 2010 使用)，版本 4.x(名称空间 System.Activities 及其子名称空间)随.NET Framework 4 一起发布。本章介绍 Workflow 的最新版本，本章从一个规范的例子 Hello World 开始，每个人在面对一种新技术时都要使用这个例子。

45.2　Hello World 示例

Visual Studio 2013 包含在.NET Framework 3.x 和 4.x 版本中创建工作流项目的内置支持。打开 New Project 对话框，会看到一个工作流项目类型列表，如图 45-1 所示。

确保从版本组合框中选中.NET Framework 4 或 4.5，再从可用的模板中选择 Workflow Console Application，这会构建一个简单的控制台应用程序，它包含工作流模板和执行这个模板的主程序。

接着，把 WriteLine 活动从工具箱拖放到设计界面上，就会得到如图 45-2 所示的工作流。WriteLine 活动在工具箱的 Primitive 类别中。

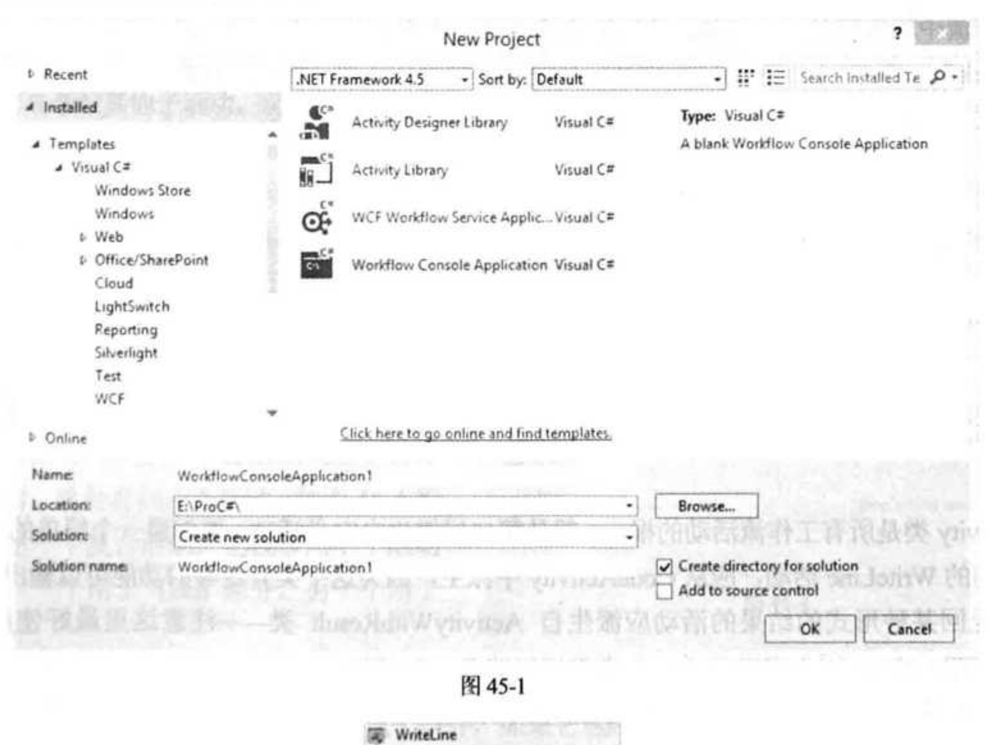


图 45-1

WriteLine
Text　Enter a C# expression

图 45-2

WriteLine 活动包含一个 Text 属性，在设置它时，可以在设计界面上直接输入文本，也可以显示属性网格。本章后面将介绍如何定义自定义活动，以使用这个相同行为。

Text 属性不是简单的字符串，实际上它定义为实参类型，该参数类型可以把一个表达式作为其源。在运行时计算表达式，得到一个结果。这个文本结果会用作 WriteLine 活动的输入。要输入简单的文本表达式，必须使用双引号——如果在 Visual Studio 中按照上面的步骤操作，就在 Text 属性中输入“Hello World”。如果省略了引号，就会得到一个编译错误，因为没有引号，所以它不是一个合法的表达式。4.5 版本中的表达式是用于 C#项目的 C#表达式，而在 4.0 版本中，表达式编辑器语法是 VB 语法，这会产生一些混淆。

如果构建并运行程序，就会在控制台上看到输出的文本。程序执行时，会在 Main()方法中创建工作流的一个实例，它使用 WorkflowInvoker 类的一个静态方法来执行该实例。这个示例的代码在 Chapter45 解决方案的 01_HelloWorld 项目中。

WorkflowInvoker 类允许同步调用工作流。还有另外两个执行工作流的方法可以异步地执行工作流，参见本章后面的内容。在 Workflow 3.x 中也可以进行同步执行，但有时启动比较困难，而且系统开销比较大。

WorkflowInvoker 类的同步功能非常适用于运行短时间的工作流，以响应某些 UI 动作——可以使用工作流启用或禁用 UI 的某些元素。

45.3 活动

工作流中的内容是一个活动，包括工作流本身。工作流其实是一组活动，在版本 4.x 中没有实际的 Workflow 类(在 3.x 中有)。活动是一个最终派生自 Activity 抽象类的类。

类层次结构比 Workflow 3.x 的定义更深，主要类如图 45-3 所示。

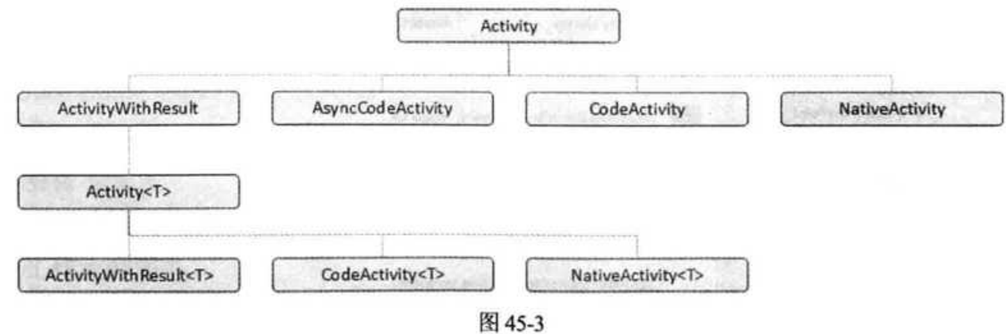


图 45-3

Activity 类是所有工作流活动的根，一般从第二层派生自定义活动。要创建一个简单的活动，如上面提到的 WriteLine 活动，应从 CodeActivity 中派生，因为这个类有足够的功能可以输出数据行。执行并返回某种形式的结果的活动应派生自 ActivityWithResult 类——注意这里最好使用泛型类 Activity<TResult>，因为它提供了一个强类型化的 Result 属性。

确定从哪个基类中派生是构建自定义活动时的主要问题，本章将通过一些示例来说明如何选择正确的基类。

为了让活动执行某个操作，一般应重写 Execute()方法，它根据所选择的基类有许多不同的签名，这些签名如表 45-1 所示。

表 45-1

基　类	Execute() 方 法
AsyncCodeActivity	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) void EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity	void Execute (CodeActivityContext)
NativeActivity	void Execute (NativeActivityContext)
AsyncCodeActivity<TResult>	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) TResult EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity<TResult>	TResult Execute (CodeActivityContext)
NativeActivity<TResult>	void Execute (NativeActivityContext)

注意，传送给 Execute()方法的参数不同，因为它使用了特定于类型的执行上下文参数。在 Workflow 3.x 中，只使用了一个 ActivityExecutionContext 类，而在 Workflow 4.x 中，可以为不同类别的活动使用不同的上下文。

主要区别是，与 NativeActivityContext 相比，CodeActivityContext 和派生的 AsyncCodeActivityContext 的功能有限。这说明，派生自 CodeActivity 和 AsyncCodeActivity 的活动可以执行的操作远远少于它们的容器。例如，前面提到的 WriteLine 活动只需写入控制台，因此，它不需要访问其运行库环境。更复杂的活动需要调度其他子活动，或与其他系统通信，此时就需要从 NativeActivity 中派生，以访问完整的运行库。在创建自己的自定义活动时，会再次探讨这个主题。

WF 提供了许多标准活动，下面几节将介绍其中一些活动的示例和使用这些活动的场合。Workflow 4.x 使用 3 个主要的程序集：System.Activities.dll、System.Activities.Core.Presentation.dll 和 System.Activities.Presentation.dll。

45.3.1　If 活动

顾名思义，这个活动的操作类似于 C# 中的 If-Else 语句。把一个 If 活动拖放到设计界面上时，就会看到一个活动，如图 45-4 所示。If 是一个复合活动，它包含两个子活动占位符，一个用于 Then 部分，另一个用于 Else 部分。

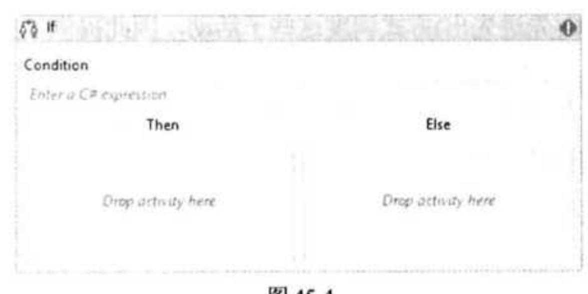


图 45-4

图 45-4 显示的 If 活动也包含一个图标，表示该活动有一个验证错误。在本例中，它表示需要定义 Condition 属性。在执行活动时判断这个条件，如果它返回 true，就执行 Then 分支；否则执行 Else 分支。

因为 Condition 属性是一个表达式，它等于一个布尔值，所以可以在这里包含任意有效的表达式。表达式可以引用工作流中定义的任意变量，也可以访问.NET Framework 中的许多静态类。所以

可以根据 Environment.Is64BitOperatingSystem 值定义表达式，假定这对工作流的某部分非常重要。一般情况下，可以定义传递到工作流中的实参，然后就可以在 If 活动中由表达式计算该实参。本章后面会讨论实参和变量。

45.3.2 InvokeMethod 活动

这是最有用的活动之一，它允许执行已有的代码，有效地把这些代码封装在工作流的执行语义中。我们一般有许多预先已有的代码，这个活动允许直接从工作流中调用这些代码。

使用 InvokeMethod 调用代码有两种方式，使用哪种方式取决于是调用静态方法还是实例方法。如果要调用静态方法，就需要定义 TargetType 和 MethodName 参数。但如果调用实例方法，就使用 TargetObject 和 MethodName 属性，在这个实例中，TargetObject 可以内联创建，它也可以是在工作流的某个地方定义的变量。02_ParallelExecution 示例中的代码显示了使用 InvokeMethod 活动的两种模式。

如果需要给调用的方法传递参数，就可以使用 Parameters 集合定义它们。集合中参数的顺序必须匹配方法中参数的顺序。另外，还有一个 Result 属性，它设置为函数调用的返回值。可以在工作流中把它绑定到一个变量上，以恰当地使用该值。

45.3.3 Parallel 活动

Parallel 活动名不符实，因为初看起来，可能认为这个活动似乎在多处理器计算机上以并行方式调度其子活动，实际上并非如此，但一些特殊情况除外。

把一个 Parallel 活动拖放到设计界面上后，就可以拖放其他子活动，如图 45-5 所示。

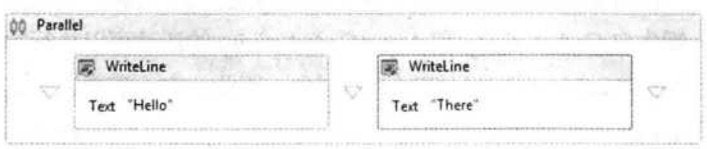


图 45-5

这些子活动可以是单一的活动，如图 45-5 所示，它们也可以来自一个复合活动，如 Sequence 或另一个 Parallel 活动。

在运行期间，Parallel 活动会调度每个直接的子活动，以执行它们。底层的运行库执行引擎会以 FIFO(先进先出)方式调度这些子活动，因此提供了并行执行的假象，但它们仅运行在一个线程上。

为了包含真正的并行执行方式，拖放到 Parallel 活动中的活动必须派生自 AsyncCodeActivity 类。02_ParallelExecution 中的示例代码包含一个例子，它演示了如何在 Parallel 活动的两个分支中异步地处理代码。图 45-6 显示了在一个 Parallel 活动中使用两个 InvokeMethod 活动的情况。

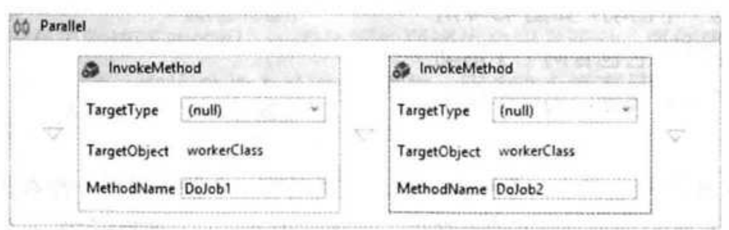


图 45-6

这里使用的 InvokeMethod 活动调用了两个简单的方法 DoJob1 和 DoJob2，它们分别休眠 2 分钟和 3 分钟。为了异步地运行这些方法，需要进行最后一个修改。InvokeMethod 活动的布尔属性

RunAsynchronously 默认为 False。在 UI 中把这个属性设置为 True，就会异步地调用目标方法，因此允许 Parallel 活动同时执行多个活动。在单处理器的计算机中，会执行两个线程，造成了同时执行的假象。而在多处理器的计算机上，这些线程可能在不同的内核上调度，所以提供了真正的并行执行方式。如果创建自己的活动，就应把它们创建为异步活动，这样最终用户就可以获得并行执行的好处。

45.3.4 Delay 活动

业务进程常常需要等待一段时间才能完成——考虑使用工作流进行费用申请的过程。工作流给经理发送一封电子邮件，要求他批准某个费用申请。之后工作流进入等待状态，等待直接经理批准(或者不批准)，这里最好定义一个超时时间。如果在 1 天时间内没有返回响应，费用申请就路由给命令链中的下一个经理。

Delay 活动可以实现这个情形的一部分(另一个部分是下一节定义的 Pick 活动)，其任务是等待预定义的时间，之后继续执行工作流。

Delay 活动包含一个 Duration 属性，它可以设置为一个离散的 TimeSpan 值，但因为该属性定义为一个表达式，所以其值可以在工作流中链接到一个变量上，或者根据需要从其他值中计算出来。

执行工作流时，工作流会进入 Idle 状态，在这个状态下，工作流会运行 Delay 活动。空闲的工作流将等待持久化——此时把工作流实例数据存储在一个永久介质中(如 SQL Server 数据库)，工作流本身可以从内存中卸载。这会节省系统资源，因为在任意给定时刻，内存中都只需要正在运行的工作流。任何延迟的工作流会持久化到磁盘上。

45.3.5 Pick 活动

一个常见的编程结构是等待一组可能的事件中的某个事件，如 System.Threading 名称空间中 WaitHandle 类的 WaitAny()方法。Pick 活动是工作流中等待事件发生的一种方式，因为它可以定义任意多个分支，每个分支在运行之前都可以等待一个触发器动作的发生。引发触发器后，就会执行工作流中的其他活动。

作为一个具体的示例，考虑上一节的费用申请工作流过程。这里的 Pick 活动有 3 个分支：第一个分支处理所接受的申请，第二个分支处理被拒绝的申请，第三个分支处理超时情况。

这个例子的代码在下载代码的 03_PickDemo 文件夹下。它包含一个示例工作流，该工作流由一个 Pick 活动和 3 个分支组成。在运行该工作流时，会提示用户接受或拒绝申请。如果过了 10 秒钟或更长时间，它就关闭这个提示，并运行延迟分支。

在这个例子中，DisplayPrompt 活动用作工作流中的第一个活动，它会调用在接口上定义的方法，该接口会提示经理批准或不批准——因为这个功能定义为接口，所以该提示可以是电子邮件、IM 消息或用其他方式通知经理，需要处理一个费用申请。之后，工作流执行 Pick 活动，等待这个外部接口的输入(批准或不批准)，同时也在等待某个延迟。

在执行 Pick 活动时，它实际上会将一个等待操作放在每条分支的第一个活动中。在触发一个事件时，会取消其他所有等待事件，并处理在其中触发了事件的其他分支。所以，在批准了费用报告的情况下，WaitForAccept 活动就完成了，接着下一个动作是输出一条确认消息。但如果经理没有批准该费用申请，WaitForReject 活动就完成了，然后输出一条拒绝消息。

最后，如果 WaitForAccept 和 WaitForReject 活动都没有完成，WaitForTimeout 活动就会在延迟

时间过后完成，并且费用报告可以路由给下一个经理——可能在 Active Directory 中查找这个人。在本示例中，执行 DisplayPrompt 活动时，会给用户显示一个对话框。所以如果执行延迟活动，则还需要关闭这个对话框，而这就是图 45-7 中 ClosePrompt 活动的作用。

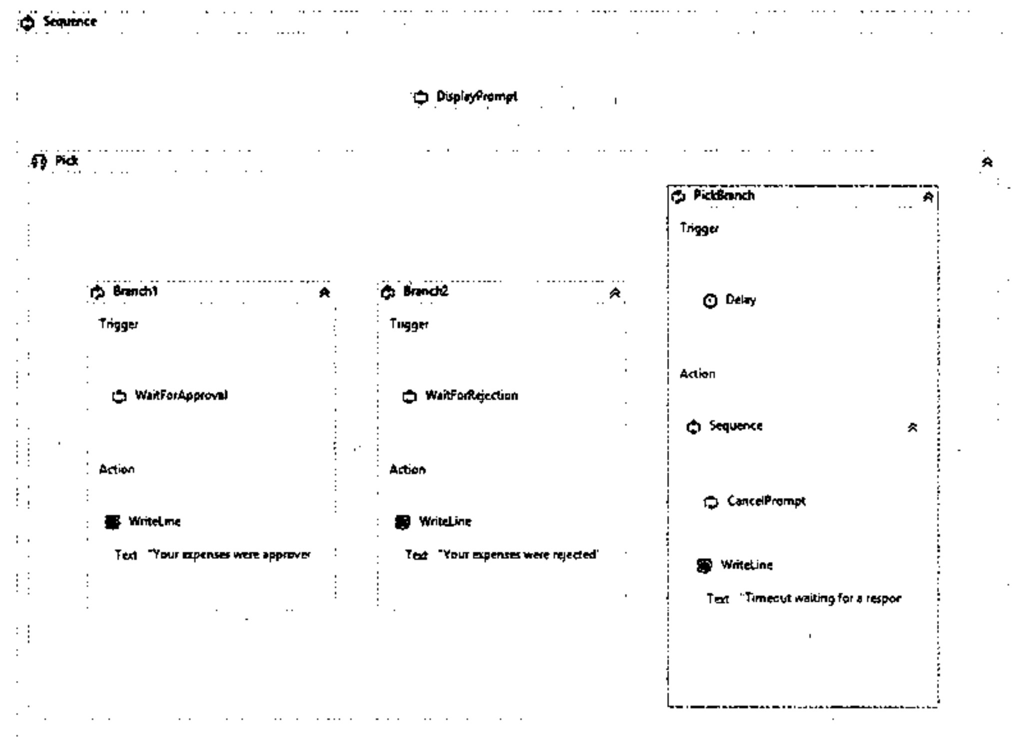


图 45-7

该例使用了一些前面没有介绍的概念，如如何编写自定义活动或等待外部事件，这些主题将在本章后面探讨。

45.4 自定义活动

前面使用的都是在 System.Activities 名称空间中定义的活动。本节将学习如何创建自定义活动，扩展它们，在设计时和运行时为用户提供更好体验。

首先，创建 DebugWrite 活动，在调试版本中将一行文本输出到控制台上。这是一个简单的示例，后面将扩展它，使用这个示例介绍自定义活动的所有可用选项。在创建自定义活动时，可以仅在工作流项目中构建一个类，但最好在一个独立的程序集中构建自定义活动，这样这个活动就是可重用的。所以，应创建一个简单的类库项目，在其中构建自定义活动。本例的代码在 04_CustomActivities 项目中。

简单的活动，如 DebugWrite 活动，直接派生自 CodeActivity 基类。下面的代码构建一个活动类，并定义一个 Message 属性，在执行活动时会显示该属性(代码文件 04_CustomActivities/Activities/DebugWrite.cs)：

```
using System;
```

```
using System.Activities;
using System.Diagnostics;
namespace Activities
{
  public class DebugWrite : CodeActivity
  {
    [Description("The message output to the debug stream")]
    public InArgument<string> Message { get; set; }
    protected override void Execute(CodeActivityContext context)
    {
      Debug.WriteLine(Message.Get(context));
    }
  }
}
```

在调度并执行 CodeActivity 时，会调用其 Execute()方法——活动实际上应在这个方法中执行一些操作。

在这个例子中，定义了 Message 属性，它看起来像是普通的.NET 属性，但它在 Execute()方法中的用法不同。在 Workflow 4 的许多改变中，一个改变是存储状态数据的位置。在 Workflow 3.x 中，常常使用标准的.NET 属性并在活动内部存储活动数据。这种方法的问题是，因为这个存储器对工作流运行库引擎实际上是不透明的，所以为了持久化工作流，必须在所有已构建的活动上实现二进制持久性，才能忠实地还原它们的数据。

在 Workflow 4 中，所有数据都存储在各个活动的外部。所以这里的模型是，应从上下文中获得实参的值，给上下文提供要设置的实参的新值。这样，工作流引擎就可以跟踪执行工作流时状态的改变，且仅存储在持久性点之间有变化的值，而不是存储整个工作流数据。

在 Message 属性上定义的[Description]特性将在 Visual Studio 的属性网格中使用，以提供该特性的额外信息，如图 45-8 所示。

显然，这个活动肯定是可以使用的，但是，有几个地方应该调整，从而使这个活动更友好。与本章前面的 Pick 活动一样，它有一些强制的属性，若没有定义，就会在设计界面上生成错误。为了使活动有相同的行为，需要扩展代码。

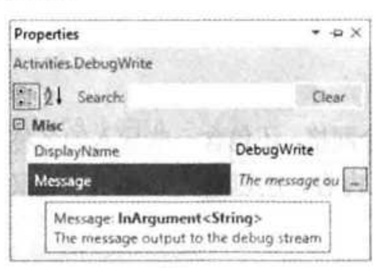


图 45-8

45.4.1 活动的验证

把活动放在设计界面上时，Designer 就会在两个地方查找验证信息。最简单的验证形式是给参数属性添加一个[RequiredArgument]特性。如果没有定义该参数，就在活动名的右边显示一个感叹号标志符号，如图 45-9 所示。

如果把鼠标悬停在感叹号上，就会显示一个工具提示“没有给活动的必选参数‘Message’提供值”。因为这是一个编译错误，

图 45-9

所以需要为这个参数定义一个值，之后就可以执行应用程序了。

如果有相关的多个属性，就可以重写 CacheMetadata()方法，以添加一些额外的验证代码。在执行活动之前调用这个方法，在其中可以检查所定义的必选参数，还可以给传递过来的参数选择性地添加额外的元数据。也可以在传递给 CacheMetadata()方法的 CodeActivityMetadata 对象上调用 AddValidationError()方法的一个重写版本，从而添加额外的验证错误(或警告)。

完成了活动的验证后，接下来就要修改活动的呈现行为，当前的 Designer 提供了这个用户体验，而且这个功能会使活动变得更有趣。

45.4.2 设计器

当活动呈现在屏幕上时，一般会把一个设计器关联到活动上。设计器的工作是为活动提供屏幕上的表示形式，在 Workflow 中，这种表示形式放在 XAML 中。如果以前没有使用过 XAML 创建用户界面，在继续之前应阅读第 35 章。

活动在设计期间的体验一般在一个独立于活动的程序集中创建，因为这个设计体验在运行期间是不需要的。Visual Studio 包含一个 Activity Designer Library 项目类型，这是一个理想的起点，因为在使用这个模板创建项目时，会得到一个默认的活动设计器，接着就可以根据需要修改它。

在设计器的 XAML 中，可以提供任何内容，包括动画。在用于用户界面时，少通常意味着多，建议查看先前已有的活动，了解应提供什么内容合适。

首先，创建一个简单的设计器，把它关联到 DebugWrite 活动上。下面的代码(代码文件 04_CustomActivities/Activities.Design/DebugWriteDesigner.xaml)显示了在项目中添加活动设计器(或构建新的活动设计器库项目)时创建的模板。

```
<sap:ActivityDesigner x:Class="Activities.<Presentation.DebugWriteDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr-namespace:System.Activities.Presentation;
              assembly=System.Activities.Presentation"
    xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
              assembly=System.Activities.Presentation">
  <Grid>
  </Grid>
</sap:ActivityDesigner>
```

所创建的 XAML 仅构建了一个网格，还包含一些导入的名称空间，本例的活动需要这些名称空间。显然，因为在模板中没有什么内容，所以首先添加用于定义消息的标签和文本框：

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <TextBlock Text="Message" Margin="0,0,5,0"/>
  <TextBox Text="{Binding Path=ModelItem.Message, Mode=TwoWay}"
    Grid.Column="1"/>
</Grid>
```

这些 XAML 在活动的 Message 属性和文本框之间构建了一个绑定。在设计器的 XAML 中，总

是可以使用 ModelItem 引用，来引用正在设计的活动。

为了把上述定义的设计器和 DebugWrite 活动关联起来，还需要修改活动，添加 Designer 属性(也可以实现 IRegisterMetadata 接口，但本章没有进一步介绍这种方式)：

```
[Designer("Activities.Presentation.DebugWriteDesigner, Activities.Presentation")]
public class DebugWrite : CodeActivity
{
    ...
}
```

这里使用[Designer]特性定义设计器和活动之间的链接。最好使用这个特性的字符串版本，因为这样可以确保在活动程序集内部不引用设计程序集。

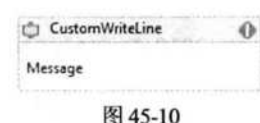


图 45-10

现在，在 Visual Studio 中使用 DebugWrite 活动的一个实例时，会得到如图 45-10 所示的结果。

但其问题是 Message 属性——它没有显示在属性网格中定义的值，如果尝试在文本框中输入它的值，就会接收到一个异常。原因是这表示我们试图把一个简单的文本值绑定到 InArgument<string>类型上，为了使这个绑定成功，需要使用 WF 中的另外一对内置类 ExpressionTextBox 和 ArgumentToExpressionConverter。现在，设计器的完整 XAML 如下所示。添加或修改的代码行用粗体表示。

```
<sap:ActivityDesigner x:Class="Activities.Presentation.DebugWriteDesigner"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sap="clr-namespace:System.Activities.Presentation;
      assembly=System.Activities.Presentation"
  xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
      assembly=System.Activities.Presentation"
  xmlns:sadc="clr-namespace:System.Activities.Presentation.Converters;
      assembly=System.Activities.Presentation">
  <sap:ActivityDesigner.Resources>
    <sadc:ArgumentToExpressionConverter x:Key="argConverter"/>
  </sap:ActivityDesigner.Resources>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="Message" Margin="0,0,5,0" />
    <sapv:ExpressionTextBox Grid.Column="1"
      Expression="{Binding Path=ModelItem.Message, Mode=TwoWay,
                   Converter={StaticResource argConverter},
                   ConverterParameter=In}"
                   OwnerActivity="{Binding ModelItem}"/>
  </Grid>
</sap:ActivityDesigner>
```

文件包含一个新的名称空间 System.Activities.Presentation.View，其中包含的转换器可用于在屏幕上的表达式和活动的 Message 属性之间进行转换。这里是 ArgumentToExpressionConverter，把它添加到 XAML 文件的资源中。

接着用 ExpressionTextBox 替换标准的 TextBox 控件。除了简单的文本之外，这个控件还允许用户输入表达式，这样 DebugWrite 活动就可以包含一个表达式，该表达式合并了正在运行的工作流中的许多值，而不仅仅是简单的文本字符串。进行了这些修改后，DebugWrite 活动就比较类似于内置活动。

如果把解决方案从.NET 4 升级到.NET 4.5，就会惊讶地发现在活动的表达式文本框中有 Enter a VB Expression 选项，该选项用于已有的工作流，还有一个 Enter a C# Expression 选项，它位于所创建的所有新工作流中。WF 4.5 在 XAML 中定义了一个特性，来确定是否显示 VB 表达式提示。查看新工作流的 XAML 时，可以看到如下指令：

```
sap:2010:ExpressionActivityEditor.ExpressionActivityEditor="C#"
```

如果省略了它，就显示 VB 表达式。不消说，找出 WF 4.5 的这个小特性花了我很多时间。

45.4.3 自定义复合活动

活动的一个常见需求是创建复合活动，即包含其他子活动的活动。例如，前面的 Pick 活动和 Parallel 活动。复合活动的执行完全由程序员负责，例如可以执行一个随机活动，它仅调度自身的其中一个子活动，或者根据当前日期是星期几，调度绕过某些子活动的一个活动。最简单的执行模式是执行所有子活动，但开发人员可以确定如何执行子活动，以及活动何时完成。

这个例子创建一个“重试”活动。我们常常会尝试一个操作，如果它失败，就重试若干次。这个活动的伪代码如下所示：

```
int iterationCount = 0;
bool looping = true;
while ( looping )
{
  try
  {
    // Execute the activity here
    looping = false;
  }
  catch (Exception ex)
  {
    iterationCount += 1;
    if ( iterationCount >= maxRetries )
      rethrow;
  }
}
```

我们会把上面的代码复制为一个活动，再把要执行的活动插入注释所在的位置。这些工作会在自定义活动的 Execute()方法中进行。还有另一种方式——可以使用其他活动编写整个活动。此时需要创建一个自定义活动，其中包含一个“洞”，最终用户可以把要重试的活动放在这里，再设置最多重试次数对应的一个属性。下面的代码演示了这个过程(代码文件 04_CustomActivities/Activities/Retry.cs)：

```
public class Retry : Activity
{
  public Activity Body { get; set; }
```

```
[RequiredArgument]
public InArgument<int> NumberOfRetries { get; set; }
public Retry()
{
  Variable<int> iterationCount =
    new Variable<int> ( "iterationCount", 0 );
  Variable<bool> looping = new Variable<bool> ( "looping", true );
  this.Implementation = () =>
  {
    return new While
    {
      Variables = { iterationCount, looping },
      Condition = new VariableValue<bool> { Variable = looping },
      Body = new TryCatch
      {
        Try = new Sequence
        {
          Activities =
          {
            this.Body,
            new Assign
            {
              To = new OutArgument<bool> ( looping ),
              Value = new InArgument<bool> { Expression = false }
            }
          }
        },
        Catches =
        {
          new Catch<Exception>
          {
            Action = new ActivityAction<Exception>
            {
              Handler = new Sequence
              {
                Activities =
                {
                  new Assign
                  {
                    To = new OutArgument<int>(iterationCount),
                    Value = new InArgument<int>
                      (ctx => iterationCount.Get(ctx) + 1)
                  },
                  new If
                  {
                    Condition = new InArgument<bool>
                      (env=>iterationCount.Get(env) >=
                       NumberOfRetries.Get(env)),
                    Then = new Rethrow()
                  }
                }
              }
            }
          }
        }
```

```
            }
          };
        };
      }
    }
```

首先定义 Activity 类型的 Body 属性——这是在重试循环中执行的活动。接着定义 RetryCount 属性，它用于指定尝试操作的次数。

这个自定义活动直接派生自 Activity 类，它把实现代码提供为一个函数。执行包含这个活动的工作流时，实际上它会执行该函数，该函数提供了一个类似于前面伪代码定义的运行时执行路径。在构造函数中，创建活动使用的局部变量，再构建一组匹配伪代码的活动。这个示例的代码也在 04_CustomActivities 解决方案中。

有了上面的代码，还可以推断出，也可以创建没有 XAML 的工作流——没有设计体验(即，不能拖放活动，以生成代码)，但是，如果我们只需要代码，就没有理由不使用它来替代 XAML。

有了自定义复合活动后，还需要定义一个设计器。这里需要一个活动，它包含一个可拖放另一个活动的占位符。如果查看其他标准活动，就会发现有几个体现类似行为的标准活动，如 If 和 Pick 活动。理想情况下，因为自定义活动的工作方式应与内置活动类似，所以现在看看它们的实现方式。

如果使用 Reflector 查看工作流库，就会发现其中缺乏设计器的 XAML。这是因为它作为一组资源编译到程序集中。使用 Reflector 可以查看这些资源，其当前版本 7.5 可以反编译 BAML 资源，以便读取它们。

在 Reflector 中，加载 System.Activities.Presentation 程序集，再导航到树型视图的 Resources 节点，打开 System.Activities.Presentation.g.resources，此时会显示当前加载的程序集中所有 BAML 资源对应的一个列表，接着就可以查看对应的示例，从而找到一些示例 XAML。

我使用这个方法了解用于内置活动的 XAML，这有助于构建如图 45-11 所示的 Retry 活动示例。

这个活动的关键是 WorkflowItemPresenter 类，它在 XAML 中用于定义子活动的占位符，其定义如下：

```
<sap:WorkflowItemPresenter IsDefaultContainer="True"
     AllowedItemType="{x:Type sa:Activity}"
     HintText="Drop an activity here" MinWidth="100" MinHeight="60"
     Item="{Binding Path=ModelItem.Body, Mode=TwoWay}"
     Grid.Column="1" Grid.Row="1" Margin="2">
```

把这个控件绑定到 Retry 活动的 Body 属性上，HintText 定义了没有给控件添加子活动时显示的帮助文本。XAML 还包含一些样式，这些样式用于显示设计器的展开或折叠版本——这确保自定义活动的工作方式与内置活动相同。这个示例的所有代码和 XAML 都在 04_CustomActivities 解决方案中。

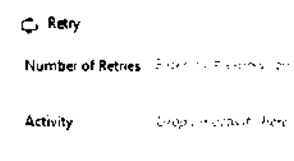


图 45-11

45.5 工作流

到目前为止，本章主要探讨了活动，但没有讨论工作流。工作流只是一个活动列表，实际上工作流本身是活动的另一个类型。使用这个模型可以简化运行库引擎，因为运行库引擎只需要知道如

何执行一种对象——派生自 Activity 类的任何对象。

前面介绍了 WorkflowExecutor 类，它可以同步执行工作流。但如前所述，这只是执行工作流的一种方式，执行工作流有 3 个不同的选项，每个选项都有不同的功能。在学习执行工作流的其他方式之前，先探讨一下实参和变量。

45.5.1 实参和变量

工作流可以看作程序，任何编程语言的其中一方面是可以创建变量，把实参传入传出程序。自然，Workflow 也支持这些结构，本节将介绍如何定义实参和变量。

首先，假定工作流处理一个保险单，所以要传递给工作流的一个实参是保险单 ID。为了定义工作流的实参，需要进入 Designer，单击左下角的 Arguments 按钮，这会列出为工作流定义的实参列表，如图 45-12 所示，也可以在这里添加自己的实参。

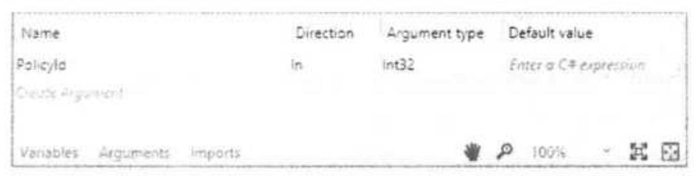


图 45-12

要定义实参，需要指定实参的 Name、Direction(可以是 In、Out 或 InOut)和数据类型。还可以选择指定默认值，如果没有提供实参，就使用默认值。

实参的方向用于确定该实参是用作工作流的输入还是输出，如果使用 InOut 方向，该实参就用作工作流的输入和输出。

本章第一节描述了如何使用 WorkflowInvoker 类执行工作流。Invoke()方法有几个重写版本，可用于给工作流传递实参。它们作为一个名称/值对字典传递，其中名称必须精确匹配实参的名称，这个匹配区分大小写。下面的代码用于给工作流传递一个 PolicyId 值(代码文件 05_ArgsAndVars/Program.cs)。

```
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("PolicyId", 123);
WorkflowInvoker.Invoke(new PolicyFlow(), parms);
```

接着调用工作流，从字典中把 PolicyId 传递给指定的参数。如果在字典中提供的名称没有对应的实参，就抛出一个 ArgumentException 异常。相反，如果没有给 In 实参提供值，就不抛出异常。这好像是错误的——我们希望对于任何未定义的 In 实参抛出一个 ArgumentException 异常，而如果传递了过多的实参，则不抛出异常。

工作流完成时，可能希望检索输出实参。为此，WorkflowInvoker.Invoke()方法有一个特定的重写版本，它返回一个字典。这个字典只包含 Out 或 InOut 实参。

在工作流中，可能希望定义变量。在 Workflow 3.x 的 XAML 工作流中，这可不容易实现，但在 Workflow 4 中，已经解决了这个问题，可以轻松地在 XAML 中定义形参。

与任何编程语言一样，工作流变量也有作用域。在工作流的根活动中定义变量，就可以定义全局变量，这些变量可用于工作流中的所有活动，它们的生命周期与工作流的生命周期联系在一起。

还可以在各个活动中定义变量，此时这些变量就只能用于定义该变量的活动，以及该活动的子活动。一旦该活动完成了，其变量就超出了作用域，且不再能访问了。

45.5.2 WorkflowApplication

尽管 WorkflowInvoker 类对于同步执行工作流很有用，但我们可能需要长时间运行的工作流，它们可以持久化到数据库中，在将来的某个时刻需要提取出来。此时应使用 WorkflowApplication 类。

WorkflowApplication 类类似于 Workflow 3 中的 WorkflowRuntime 类，在 Workflow 3 允许运行工作流，也允许响应在该工作流实例上发生的事件。可以使用 WorkflowApplication 类编写的最简单的程序如下所示：

```
WorkflowApplication app = new WorkflowApplication(new Workflow1());
ManualResetEvent finished = new ManualResetEvent(false);
app.Completed = (completedArgs) => { finished.Set(); };
app.Run();
finished.WaitOne();
```

这个程序构造一个工作流应用程序实例，并关联到该实例的 Completed 委托上，以设置一个手工重置的事件。调用 Run()方法启动工作流的执行，最后代码等待触发事件。

这揭示了 WorkflowExecutor 和 WorkflowApplication 之间的一个主要区别——后者是异步的。调用 Run()方法时，系统会使用线程池中的一个线程执行工作流，而不使用主调线程。因此，需要某种形式的同步，才能确保包含工作流的应用程序在工作流完成之后退出。

长时间运行的工作流一般有许多休眠阶段——大多数工作流的执行行为可以描述为间歇执行的阶段。在工作流的开头一般要做某项工作，接着等待某个输入或延迟，接收到这个输入后，就进行处理，再进入下一个等待状态。

所以，工作流休眠时，最好从内存中卸载它，再在事件触发工作流继续执行时，再重新加载工作流。为此，需要给 WorkflowApplication 类添加一个 InstanceStore 对象，再对前面的代码进行其他一些小的修改。在架构中，抽象类 InstanceStore 有一个实现方式 SqlWorkflowInstanceStore。为了使用这个类，首先需要一个数据库，其脚本默认位于 C:\Windows\Microsoft.NET\Framework64\v4.0.30319\SQL\en 目录下。注意版本号会改变。

在这个目录下有许多 SQL 文件，我们需要的两个是 SqlWorkflowInstanceStoreSchema.sql 和 SqlWorkflowInstanceStoreLogic.sql。可以对已有的数据库运行这两个文件，也可以根据需要创建全新的数据库。还可以使用完整的 SQL Server 安装或 SQL Express 安装。

一旦有了数据库，就需要对宿主代码进行一些修改。首先，需要构造 SqlWorkflowInstanceStore 的一个实例，再把它添加到工作流应用程序中(代码文件 06_WorkflowApplication/Program.cs)：

```
SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore
  (ConfigurationManager.ConnectionStrings["db"].ConnectionString);
AutoResetEvent finished = new AutoResetEvent(false);
WorkflowApplication app = new WorkflowApplication(new Workflow1());
app.Completed = (e) => { finished.Set(); };
app.PersistableIdle = (e) => { return PersistableIdleAction.Unload; };
app.InstanceStore = store;
app.Run();
finished.WaitOne();
```

粗体的代码行是添加到前面示例中的新代码。另外注意，在工作流应用程序中还给 PersistableIdle 委托添加了一个事件处理程序。执行工作流时，它会运行尽可能多的活动，直到没有工作可做为止。

此时，工作流就会转换到 Idle 状态，空闲的工作流是持久化的备用工作流。PersistableIdle 委托用于确定应对空闲的工作流进行什么操作。默认为什么都不做，然而，也可以指定 PersistableIdleAction.Persist，它会接受工作流的一个副本，并把它存储在数据库中，但工作流仍位于内存中，还可以指定 PersistableIdleAction.Unload，它会持久化工作流，之后卸载工作流。

也可以使用 Persist 活动请求持久化工作流，但如果调用 NativeActivityContext 的 RequestPersist()方法，使自定义活动派生自 NativeActivity，自定义活动编写器还可以请求持久化工作流。

现在有一个问题：可以从内存中卸载工作流，并把它存储在持久性存储器中，但如何从存储器中检索工作流，再次执行它？

1. 书签

书签的传统用法是标记图书中的一页，以便从同一个地方开始继续阅读。在工作流的上下文中，书签指定了一个位置，以便从这个位置开始继续运行工作流，等待外部输入时，一般会使用书签。

例如，我们可能要等待应用程序处理保险报价。最终用户在线生成一个报价单，显然，存在与这个报价单相关联的一个工作流。因为报价单可能仅在 30 天内有效，所以在 30 天后应使该报价失效。同样，也可能请求无索赔的证明，如果该证明没有在指定时间内提供，就取消保险单。于是，这个工作流有许多执行阶段，在其他时间，工作流会处于休眠状态，此时就可以从内存中卸载工作流。但在卸载之前，需要在工作流中定义一点，可以从该点处恢复处理，此时就应使用书签。

要定义书签，需要一个派生自 NativeActivity 的自定义活动(代码文件 06_WorkflowApplication/CustomActivities/Task.cs).)。接着在 Execute()方法中创建一个书签，在恢复该书签时，就继续执行代码。下面的示例活动定义一个简单的 Task 活动，该活动创建一个书签，在书签处恢复执行后，活动就完成了。

```
public class Task : NativeActivity<Boolean>
{
  [RequiredArgument]
  public InArgument<string> TaskName { get; set; }
  protected override bool CanInduceIdle
  {
    get { return true; }
  }
  protected override void Execute(NativeActivityContext context)
  {
    context.CreateBookmark(TaskName.Get(context),
      new BookmarkCallback(OnTaskComplete));
  }
  private void OnTaskComplete(NativeActivityContext context,
      Bookmark bookmark, object state)
  {
    bool taskOK = Convert.ToBoolean(state);
    this.Result.Set(context, taskOK);
  }
}
```

对 CreateBookmark 的调用传递了书签名和一个回调函数。这个回调函数在恢复书签时执行。给回调函数传递了一个任意对象，在本例中是一个布尔值，因为每个任务都应报告其成功或失败，所以使用这个布尔值可以确定工作流的后续步骤。可以给工作流传递任意对象，它可以是带许多字段

的复杂类型。

这就是编写好的活动。现在需要修改宿主代码，以便在书签处恢复执行。但这有另一个问题：宿主代码如何知道工作流创建了一个书签？如果主机应从书签处恢复执行，主机就需要知道存在一个书签。

上面创建的 Task 活动需要多做一些工作，告诉外界已经创建了一个任务。在产品系统中，这一般导致在队列表中存储一项，这个队列会对调用中心人员显示为一个作业列表。

与宿主的通信参见下一节。

2. 扩展

扩展仅是添加到工作流应用程序的运行环境中的一个类或接口。在 Workflow 3.x 中，这些扩展称为服务，但因为该服务与 WCF 服务冲突，所以在 Workflow 4 中，把它们重命名为扩展。

一般给扩展定义一个接口，再提供该接口在运行期间的实现代码。自定义活动只需要调用该接口，并允许根据需要修改该实现代码。扩展的一个例子是发送电子邮件。可以创建一个 SendEmail 活动，它在其 Execute()方法中调用扩展，再定义一个基于 SMTP 的电子邮件扩展或基于 Exchange 的 Outlook 扩展，以便在运行期间实际发送电子邮件。自定义活动不需要改为使用任意电子邮件提供程序，只需要修改应用程序配置文件，插入一个新的电子邮件提供程序即可。

对于 task 示例，需要一个扩展，在 Task 活动在其书签处等待时获得通知。这可以把书签名和其他相关信息写入一个数据库中，这样任务队列就可以显示给用户。使用下面的接口定义这个扩展(代码文件 06_WorkflowApplication/SharedInterfaces/ITaskExtension.cs)：

```
public interface ITaskExtension
{
   void ExecuteTask(string taskName);
}
```

接着，就可以修改 Execute()方法，以更新 Task 活动，从而通知任务扩展，它正在执行(代码文件 06_WorkflowApplication/CustomActivities/Task.cs)：

```
protected override void Execute(NativeActivityContext context)
{
    context.CreateBookmark(TaskName.Get(context),
       new BookmarkCallback(OnTaskComplete));
    context.GetExtension<ITaskExtension>().
       ExecuteTask(TaskName.Get(context));
}
```

在传递给 Execute()方法的 context 对象中查询 ITaskExtension 接口，接着代码调用 ExecuteTask()方法。因为 WorkflowApplication 维护着一个扩展集合，所以可以创建一个实现这个扩展接口的类，该类可用于维护任务列表。然后构建并执行一个新的工作流，每个任务都通知该扩展执行它的时间。另一个进程会查看任务列表，并显示给最终用户。

为了使示例代码简单一些，我们只创建了一个工作流实例。这个实例包含一个 Task 活动和后面的一个 If 活动，根据用户接受或拒绝任务来输出相应的消息。

3. 组合在一起

现在可以运行、持久化和卸载工作流，通过书签把事件发送给工作流。最后一部分是重新加载工作流。使用 WorkflowApplication 时，可以调用 Load，并传递工作流的唯一 ID。每个工作流都有唯一的 ID，调用 ID 属性可以从 WorkflowApplication 对象中检索该 ID。所以在伪代码中，包含工作流的应用程序如下所示(代码文件 06_WorkflowApplication/Program.cs)：

```
WorkflowApplication app = BuildApplication();
Guid id = app.Id;
app.Run();
// Wait for a while until a task is created, then reload the workflow
app = BuildApplication();
app.Load(id);
app.ResumeBookmark()
```

所提供的示例代码比前面的代码复杂一些，因为它还包含 ITaskExtension 接口的实现代码，但代码遵循前面的模式。注意对 BuildApplication()方法的两个调用。它们在代码中用于构建一个 WorkflowApplication 实例，并设置所有需要的属性，例如，InstanceStore 以及用于 Completed 和 PersistableIdle 的委托。在第一个调用之后，执行 Run()方法，这会开始执行工作流的一个新实例。

因为第二次加载应用程序是在一个持久性点后，所以在这个持久性点之前卸载了工作流，于是，应用程序实例也删除了。于是我们构建一个新的 WorkflowApplication 实例，但不是调用 Run()方法，而调用 Load()，它使用持久性提供程序从数据库中加载已有的实例。调用 ResumeBookmark()函数来恢复执行这个实例。

如果运行该示例，就会在屏幕上看到一个提示。尽管这个提示显示在屏幕上，但会持久化并卸载工作流。运行 SQL Server Management Studio，执行如图 45-13 所示的命令，就可以验证这一点。

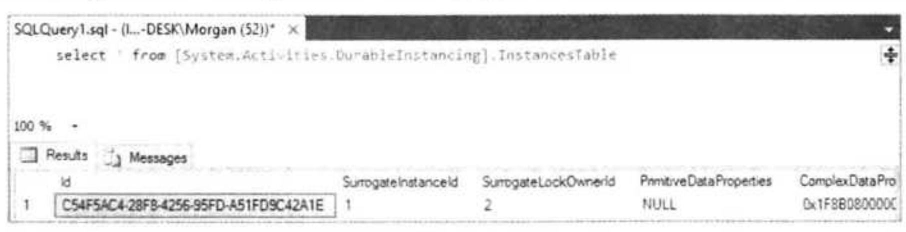


图 45-13

工作流实例存储在 System.Activities.DurableInstancing 模式的 InstancesTable 中。如图 45-13 所示的项是运行在某台计算机上的工作流的持久化实例。

继续执行这个工作流，它最终会执行完毕，此时工作流会从实例表中删除，因为关于实例存储提供了一个 InstanceCompletionAction 选项，它默认设置为 DeleteAll。这将确保一旦工作流完成，就删除在数据库中为给定的工作流实例存储的任何数据。这是一个合理的默认值，因为一旦工作流完成，通常就不应保留该工作流的任何数据。把工作流实例的完成动作设置为 DeleteNothing，就可以在定义实例存储时改变这个选项。

如果现在继续运行测试应用程序，重试图 45-13 中的 SQL 命令，就会发现工作流实例已删除。

45.5.3 存放 WCF 工作流

本章前面提到，存放工作流有 3 种方式，最后一种是使用 WorkflowServiceHost 类，它通过 WCF

来提供工作流。Workflow 的一个主要使用场合是用作 WCF 服务的后端。考虑一下典型 WCF 服务执行的操作，它通常以某种顺序调用一组相关的方法。这里的主要问题是可以以任意顺序调用这些方法，而且通常需要定义该顺序，例如，订单信息就不会在下订单之前上传。

使用 Workflow 很容易提供那些也需要考虑方法调用顺序的 WCF 服务。这里使用的主要类是 Receive 和 Send 活动。在这个示例的代码中(在 07_WorkflowsAsServices 解决方案中)，所使用的场景是房地产代理(在美国是房地产经纪人)，他希望给某网站上传资产信息。

使用 WCF 驻留工作流有两种方式：在代码中构建 WorkflowServiceHost 的一个实例(类似于常规的 WCF ServiceHost 类)，来显式保存工作流；还可以把服务构建为一个.xamlx 文件，此时工作流很可能使用 IIS 或 WAS 来提供。本示例使用这个选项，但如果需要，也可以手工完成所有工作。

这个示例还展示了 WF 4.5 的两个新特性，第一个是使用状态机工作流，第二个是这个工作流的实现代码使用了 WCF 服务协定，该协定用协定优先的方式定义。状态机工作流类型在.NET 4 中没有，但在该框架的最新版本中又回来了，它很适合这里编写的示例。图 45-14 显示了本示例要创建的状态机。

图 45-14

在状态机中，有一个起始状态、任意多个中间状态以及一个可选的结束状态。除了状态之外，还有切换，它等待一个事件的发生(通常是接收一些输入的活动)，再把状态机切换到另一个状态。

状态机的工作方式是等待一个切换活动的完成，接着把状态机切换到一个新状态。在进入一个状态时，可以调用一个活动，从一个状态中退出，也可以执行一个活动。然后状态机等待一个切换操作的完成，再转入新状态。

在图 45-14 中，第一个状态标记为“初始状态”，这只是一个切换操作，它等待一个消息通过 WCF 进入。接收到这个消息后，状态机就进入“等待”状态，该状态有两个切换操作，它们映射了两个可以在 WCF 接口上使用的操作，用户可以发送房间的细节，或者指出数据已上传完毕。这个例子使用的 WCF 服务如下所示(代码文件 07_WorkflowsAsServices/SharedInterfaces/Iproperty-Information.cs)：

```
[ServiceContract(Namespace="http://www.morganskinner.com")]
public interface IPropertyInformation
{
    [OperationContract()]
```

```
    Guid UploadPropertyInformation(string ownerName, string address, float price);

    [OperationContract(IsOneWay=true)]
    void UploadRoomInformation(Guid propertyId, string roomName, float width,
      float length);

    [OperationContract(IsOneWay = true)]
    void DetailsComplete(Guid propertyId);
}
```

UploadPropertyInformation 调用是状态机的开始。它有效地创建了一个新的工作流实例，把状态机切换到“等待”状态，再等待调用 UploadPropertyInformation 或 DetailsComplete。

要在 WF 中使用已有的 WCF 服务协定，需要给工作流项目显示上下文菜单，从菜单中选择 Import Service Contract 项。这会为服务协定中的每个操作构建活动。如果希望查看这些活动，可以单击 Show All Files 工具栏按钮，展开 Visual Studio 中当前项目的 ServiceContracts 项，这会显示服务协定名(在本示例中是 IPropertyInformation)，再显示自动生成的活动，如图 45-15 所示。

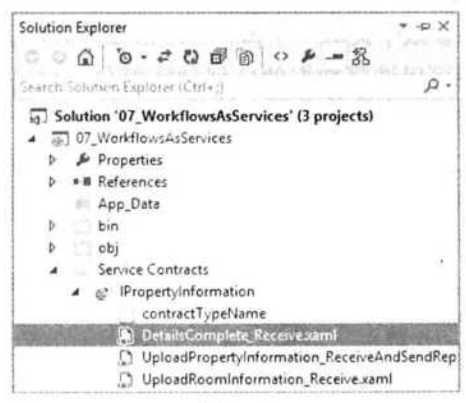


图 45-15

该图显示，从服务协定中创建了 3 个活动，它们将在后面在工作流中用于实现服务协定。

1. 校正

对于 WCF 工作流，要理解的另一个要点是如何在客户端上进行调用，该调用可以在服务器上找到正确的工作流实例。如果希望查看 WCF 服务接口，就需要使用 UploadPropertyInformation 操作，对服务器发出一个调用，来启动上传操作。

这个操作会返回一个唯一的 ID，这个 ID 会传送回后面对 UploadPropertyInformation 和 DetailsComplete 的调用。为此，工作流有校正的概念，简言之，校正就是允许使用随机的一个信息(或者是一些信息)，把调用路由到正确的工作流实例上。在返回这个 ID 的第一次调用后，只要在下一个对工作流的调用中使用同一个 ID，就可以找到正确的工作流实例，来路由消息。

为了使校正机制发挥作用，需要两个组件：CorrelationHandle 和校正初始化器。句柄只是一个用合适的作用域定义的变量(即在需要该变量的所有活动之外)，校正初始化器定义了在入站或出站消息中应使用什么项，来唯一地标识工作流实例，以相互区分工作流实例。

本例把 CorrelationHandle 定义为工作流中根活动上的一个变量。接着在 UploadInfo 转换中，有一个 Receive/Send 对。Receive 把其 CanCreateInstance 属性设置为 True，这告诉工作流，客户端调用服务器时，这个操作将是工作流执行链中的第一个操作。Receive 活动可以从传递给服务调用的参

数中读取任意数据(如果合适，就把它们存储起来，以便在工作流的后面使用)。

这里使用的 Send 活动把校正设置为在工作流中创建的唯一 ID。在本例中它使用 XPath 表达式，如图 45-16 所示，把发送回用户的 Guid 提取为 UploadPropertyInformation 方法的返回值。

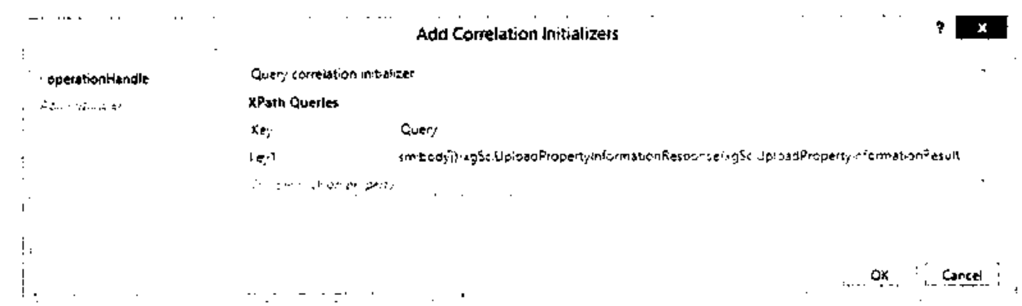


图 45-16

把工作流实例与这个 ID 有效地关联起来后，就可以再次使用校正机制，允许以后调用服务协定中的另一个方法，这里再次使用校正机制，就可以找到正确的工作流实例，本例是从传送给操作(这些操作在服务器端处理数据)的参数中提取一个值。这是很强大的，在 Workflow 3.x 中只能使用内置的工作流实例 ID，但在 4.x 中可以选择数据的任何独特部分，允许客户端调用找到正确的工作流实例。

同样，可以在工作流的生存期，校正另一个数据，这样，就可能从第一次调用中返回一个 Guid，但从后续的方法调用中返回唯一的整数键。

2. 调用工作流服务

把工作流存储为一个.xamlx 文件后，就需要某种方式来调用该工作流。这与调用标准 WCF 服务一样，因为这需要添加对该服务的引用。在本例中，服务不通过以.svc 文件结尾的 URL 来寻址，而是以.xamlx 工作流定义来结尾的 URI 进行寻址。

接着，客户端就可以进行 WCF 调用，与服务交互操作。这里，唯一的区别是工作流不接受在该 WCF 接口上的任何调用。在任意特定的时间有效的操作取决于工作流的状态。初次调用工作流时，使用了 UploadPropertyInformation 调用，这是因为该调用的接收者把其 CanCreateInstance 属性设置为 True，以允许创建新的工作流实例。

接着在接口上调用另一个方法，因为工作流实例现在正在等待它。

本例的下载代码包含一个代理类，它类似于用“添加服务引用”对话框创建的代理类。本书在代码中创建它们，是因为这个过程比使用“添加服务引用”对话框更直观，也能更好地支持大型开发团队，在大型开发团队中，服务接口常常会改变。这里使用这种方式，只需要在一个地方进行修改，无须在多个项目中重新生成多个服务引用。

接着代码创建这个代理类的一个实例，如下所示：

```
PropertyInformationClient client = new PropertyInformationClient("state");
```

这使用了在 app.config 中定义的一个端点：

```
<endpoint address="http://localhost:1353/StateService.xamlx"
        binding="basicHttpBinding"
```

```
        contract="SharedInterfaces.IPropertyInformation"
        name="state"/>
```

可以看到，服务的地址以.xamlx 结尾，这表示服务是一个工作流，而不是标准的 WCF 服务。

45.5.4 工作流的版本

因为工作流可以保存起来，并可能运行很长时间，所以需要一种方式修改工作流。例如，有一个工作流处理保险的续保。每次投保的新规发布时，都要运行这个工作流的实例，投保规定发布后 10 个月，该工作流实例都要重新启动，发布更新的报价单，根据客户的回应，发送一两封提醒函。

现在假定实现一个新的客户联系机制，允许发送文本消息，通知客户续保情况。那么该如何处理数据库中已有的所有工作流？

在 WF 4.5 之前，只有两个选项，它们都不是很合适。第一个选项是忽略已有工作流的修改，仅在新的工作流实例中实现那些修改，另一个选项是取消所有的已有实例，再重新启动它们。这两个选项都不理想，在 WF 4.5 中，添加了两个新功能来帮助实现这个特性。

1. 动态更新

动态更新功能允许修改已保存的工作流实例，对该工作流应用某个新功能。更新工作流要执行几个步骤，它们可以分为两个阶段。

第一个阶段是准备更新已保存的工作流。这里要调用 PrepareForUpdate 方法，来使用新的 DynamicUpdateServices 类，如下面的代码所示。接着根据需要修改工作流，例如添加新活动。该示例有一个新的 WriteLine 活动，它添加到当前工作流的尾部。完成了所有的修改后，就可以调用 CreateUpdateMap 方法，返回 DynamicUpdateMap 类的一个实例，在加载已有的实例，把它们更新到新工作流定义中时，要使用这个 DynamicUpdateMap 实例(代码文件 08_DynamicUpdate/Program.cs)：

```
Activity workflowDefinition = GetInitialWorkflow();
DynamicUpdateServices.PrepareForUpdate(workflowDefinition);

// Now update the workflow - add in a new activity
Sequence seq = workflowDefinition as Sequence;
seq.Activities.Add(new WriteLine { Text = "Second version of workflow" });

// And then after all the changes, create the map
return DynamicUpdateServices.CreateUpdateMap(workflowDefinition);
```

可以重复使用更新映射，来更新工作流实例，因此这里有两个阶段，第二个阶段是加载已有的工作流，升级它们。下面的代码段说明了如何加载已有的工作流实例，使用前面创建的升级映射，来更新该实例：

```
SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore(
  ConfigurationManager.ConnectionStrings["db"].ConnectionString);
WorkflowApplicationInstance instance = WorkflowApplication.GetInstance(id, store);
WorkflowApplication app = new WorkflowApplication(GetUpdatedWorkflow());
app.Load(instance, map);
app.Unload();
```

这里的 GetUpdatedWorkflow 方法加载新的工作流定义，再使用 map 对象，指出如何在已有的活动和新活动之间移动保存好的数据，把已有的实例数据加载到这个新工作流定义中。工作流保存的数据只包含数据，不包含工作流定义，所以升级仅仅是在工作流的节点之间移动已保存的信息。

工作流升级后，就必须在新的工作流定义上执行。为了简化在数据库中确定工作流版本的任务，可使用新的 WorkflowIdentity 类，在创建应用程序时，它可以关联到 WorkflowApplication 上。这个信息构成了工作流所保存信息的一部分，可以使用 WorkflowApplicationInstance 类的 DefinitionIdentity 属性来访问。使用它可以迭代存储器中的所有工作流，找出已升级的工作流和需要修改的工作流。

2. 并发工作流

使用 WorkflowApplication 类保存工作流后，就可以包含一个 WorkflowIdentity 对象，它允许把版本信息关联到工作流定义上，这样把该工作流保存到数据库中时，版本信息也会保存。接着，从数据库中读回所保存的工作流实例时，就可以找到给定版本的工作流，以便在数据库中工作流的已保存状态上映射运行期间的工作流定义。

要加载和恢复任何工作流实例，保存该工作流的 WorkflowApplication 必须用特定的工作流定义来初始化，该工作流定义用于创建工作流区域。在恢复工作流前读取版本信息，就可以使用它映射特定的工作流定义。

另外，这个版本支持还扩展到使用 WorkflowServerHost 类保存的工作流上。用 WorkflowServer-Host 保存带版本信息的工作流后，就需要在工作流服务的 DefinitionIdentity 属性中指定版本信息。

这些新增功能使工作流比以前更好用，尤其是可能需要数周或数月的商务过程。若时间更长，则需要修改的可能性更大。尽管进行这些修改并不容易，但至少现在我们有这样的选项。

45.5.5 驻留设计器

我们常常希望把最好的东西留到最后。不要打破常规，这就是本章所做的工作。Visual Studio 中使用的工作流设计器也可以驻留在自己的应用程序中，允许最终用户创建自己的工作流，而无须查看 Visual Studio 的副本。这是 Workflow 4 目前的最佳特性。传统的应用程序扩展机制总是需要某种开发人员——或者编写扩展 DLL 再把它插入系统的某个地方；或者编写宏或脚本。Windows Workflow 允许最终用户仅把活动拖放到设计界面上，来定制应用程序。

在 Workflow 3.x 中重新驻留设计器比较麻烦，但在 Workflow 4 中它就非常容易。因为设计器本身是一个 WPF 控件，所以我们使用 WPF 项目作为主应用程序。这个示例的代码在 09_DesignerRehosting 项目中。

首先需要包含工作流的程序集，接着定义主窗口的 XAML。在构建 WPF 用户界面时，总是使用 MVVM(Model-View-ViewModel，模型-视图-视图模型)模式，因为它简化了编码，并允许在需要时用不同的 XAML 覆盖相同的视图模型。主窗口的 XAML 如下所示（代码文件 09_DesignerRehosting/MainWindow.xaml）：

```
7<Window x:Class="HostApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        Title="MainWindow">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Menu IsMainMenu="True">
      <MenuItem Header="_File">
        <MenuItem Header="_New" Command="{Binding New}"/>
        <MenuItem Header="_Open" Command="{Binding Open}"/>
        <MenuItem Header="_Save" Command="{Binding Save}"/>
        <Separator/>
        <MenuItem Header="_Exit" Command="{Binding Exit}"/>
      </MenuItem>
      <MenuItem Header="Workflow">
       <MenuItem Header="_Run" Command="{Binding Run}"/>
      </MenuItem>
    </Menu>
    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="4*"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <ContentControl Content="{Binding Toolbox}" />
      <ContentControl Content="{Binding DesignerView}"
       Grid.Column="1"/>
      <ContentControl Content="{Binding PropertyInspectorView}"
       Grid.Column="2"/>
    </Grid>
  </Grid>
</Window>
```

主窗口的布局相当简单，再使用一个网格来定义用于工具箱、设计器和属性表的占位符。注意所有对象都是绑定的，包括命令在内。

前面创建的 ViewModel 包含用于每个主 UI 元素(工具箱、设计器和属性网格)的属性，除了这些属性之外，还有用于每条命令的属性，如 New、Save 和 Exit(代码文件 09_DesignerRehosting/ViewModel.cs)。

```
public class ViewModel : BaseViewModel
{
    public ViewModel()
    {
        // Ensure all designers are registered for inbuilt activities
        new DesignerMetadata().Register();
    }
    public void InitializeViewModel(Activity root)
    {
        _designer = new WorkflowDesigner();
        _designer.Load(root);
        this.OnPropertyChanged("DesignerView");
        this.OnPropertyChanged("PropertyInspectorView");
    }
```

```
        public UIElement DesignerView
        {
            get { return _designer.View; }
        }
        public UIElement ProperttInspectorView
        {
            get { return _designer.PropertyInspectorView; }
        }
        private WorkflowDesigner _designer;
    }
```

首先，ViewModel 类派生自 BaseViewModel，每次构建视图模型时，都会使用这个基类，因为它提供了 INotifyPropertyChanged 的实现方式。它来自于 Josh Twist 编写的一组代码段，可以从 www.thejoyofcode.com 上下载。

构造函数确保注册所有内置活动的元数据。没有这个构造函数的调用，就不会在用户界面上显示任何类型的特定设计器。在 InitializeViewModel()方法中，构建工作流设计器的一个实例，并给它加载一个活动。WorkflowDesigner 类比较奇怪，因为一旦给它加载了一个工作流，就不能加载另一个工作流了，所以只要创建新工作流，就需要重新创建这个类。

InitializeViewModel()方法的最后一个操作是调用属性更改通知函数，告诉用户界面，DesignerView 和 PropertyInspectorView 都更新了。由于 UI 绑定到这些属性上，因此它们是必选的，会从新的工作流设计器实例中加载新值。

用户界面中下一个要创建的部分是工具箱。在 Workflow 3.x 中，必须自己构建这个控件，而在 Workflow 4 中有一个 ToolboxControl，它使用起来非常容易。(代码文件 09_DesignerRehosting/ViewModel.cs)。

```
    public UIElement Toolbox
    {
      get
      {
        if (null == _toolbox)
        {
          _toolbox = new ToolboxControl();
          ToolboxCategory cat = new ToolboxCategory
            ("Standard Activities");
          cat.Add(new ToolboxItemWrapper(typeof(Sequence),
            "Sequence"));
          cat.Add(new ToolboxItemWrapper(typeof(Assign), "Assign"));
          _toolbox.Categories.Add(cat);
          ToolboxCategory custom = new ToolboxCategory("Custom Activities");
          custom.Add(new ToolboxItemWrapper(typeof(Message), "MessageBox"));
          _toolbox.Categories.Add(custom);
        }
        return _toolbox;
      }
    }
```

这里构建了工具箱控件，接着在第一个类别中添加两个文本框项，在第二个类别中添加一个工具箱项。ToolboxItemWrapper 类用于简化给工具箱添加给定活动所需的代码。

有了这些代码，就有了一个正在运行的应用程序。现在只需把 ViewModel 关联到 XAML 上。这在主窗口的构造函数中完成。

```
public MainWindow()
{
  InitializeComponent();
  ViewModel vm = new ViewModel();
  vm.InitializeViewModel(new Sequence());
  this.DataContext = vm;
}
```

这里构建了视图模型，并添加到默认的 Sequence 活动中。这样，在应用程序运行时，就会在屏幕上显示一些内容。

现在，唯一遗漏的部分是一些命令。我们使用 DelegateCommand 类给 WPF 编写基于 ICommand 的命令，这样视图模型中的代码比较容易理解。命令的实现非常简单，如下面的 New 命令所示：

```
public ICommand New
{
  get
  {
    return new DelegateCommand(unused =>
    {
      InitializeViewModel(new Sequence());
    });
  }
}
```

因为把这条命令绑定到 New 菜单项上，所以单击这条命令执行该委托时，在这种情况下就仅通过一个新的 Sequence 活动调用 InitializeViewModel()方法。因为这个方法也会触发设计器和属性网格的属性更改通知，所以设计器和属性网格也会更新。

Open 命令有点复杂：

```
public ICommand Open
{
  get
  {
    return new DelegateCommand(unused =>
    {
      OpenFileDialog ofn = new OpenFileDialog();
      ofn.Title = "Open Workflow";
      ofn.Filter = "Workflows (*.xaml)|*.xaml";
      ofn.CheckFileExists = true;
      ofn.CheckPathExists = true;
      if (true == ofn.ShowDialog())
        InitializeViewModel(ofn.FileName);
    });
  }
}
```

这里使用了 InitializeViewModel()方法的另一个重写版本，在这个例子中，该重写版本的参数是文件名，而不是活动。这里没有列出这些代码，但可以从代码下载中获得代码。这条命令会显示一个 OpenFileDialog，选择这条命令时，它会把工作流加载到设计器中。对应的 Save 命令会调用 WorkflowDesigner.Save()方法，把工作流的 XAML 存储到磁盘上。图 45-17 显示了此时运行应用程序时屏幕的外观。

图 45-17

视图模型中的最后一段代码是 Run 命令。如果不能执行这些代码，就不会有设计精良的工作流，所以在视图模型中也包含这个功能。它很简单——设计器包含一个 Text 属性，它是工作流中活动的 XAML 表示。我们只需要把它转换为一个 Activity，再使用 WorkflowInvoker 类执行它即可。

```
public ICommand Run
{
  get
  {
    return new DelegateCommand(unused =>
    {
      Activity root = _designer.Context.Services.
        GetService<ModelService>().Root.
        GetCurrentValue() as Activity;
      WorkflowInvoker.Invoke(root);
    },
    unused => { return !HasErrors; }
    );
  }
}

public bool HasErrors
{
  get { return (0 != _errorCount); }
}

public void ShowValidationErrors(IList<ValidationErrorInfo> errors)
{
  _errorCount = errors.Count;
  OnPropertyChanged("HasErrors");
}
private int _errorCount;
```

必须调整上述代码，以便放在页面上，因为从设计器中检索根活动的委托命令的第一行代码太长。接着只需要使用 WorkflowInvoker.Invoke()方法执行工作流。

WPF 中的命令结构包含一种禁用不能访问的命令的方式，它就是关于 DelegateCommand 的第

二个 lambda 函数。这个函数返回 HasErrors 的值，HasErrors 是一个已添加到视图模型中的布尔属性。这个属性表示在工作流中是否找到验证错误，因为视图模型实现 IValidationErrorService，只要工作流的有效状态改变了，IValidationErrorService 就会得到通知。

可以扩展这个示例，在用户界面上根据需要提供这个验证错误列表，还可以在工具箱中添加更多活动，因为我们不希望工具箱中只有 3 个活动。

45.6 小结

Windows 工作流为应用程序的构建方式带来了根本性的改变。现在可以将应用程序的复杂部分都看作活动，允许用户仅将活动拖放到工作流中，就可以修改系统的处理方式。

几乎没有应用程序不能应用工作流，从最简单的命令行工具，到包含上百个模块的最复杂的系统。以前需要开发人员给系统编写一个扩展模块，而现在可以提供一个简单的、可扩展的、几乎任何人都可以使用的定制机制。应用程序供应商以前需要提供自定义活动，与系统交互操作，并提供在应用程序中调用工作流的代码，而现在可以让客户定义在事件发生时需要应用程序执行什么操作。

Workflow 4 极大地改进了 Workflow 3.x，如果打算第一次使用工作流，最好从这个新版本开始，完全跳过 Workflow 3.x。

第46章

对 等 网 络

本章要点

- P2P 概述
- Microsoft Windows 8 Peer-to-Peer Networking 平台
- 注册和解析对等名称
- 通过对等网络收发消息
- 用.NET Framework 构建 P2P 应用程序

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码的示例文件是：

- P2PSample

46.1 P2P 网络概述

对等网络(常常称为 P2P)是近年来出现的最有用且容易误解的技术。人们提到 P2P 时，通常会想到：共享音乐视频、软件和其他文件，这常常是非法的。这是因为文件共享应用程序(如 BitTorrent)以令人吃惊的速度异军突起，而这些应用程序使用 P2P 技术工作。

尽管 P2P 在文件共享应用程序中使用，但这并不是说其他应用程序就不使用这个技术。事实上，如本章所述，P2P 可以用于大量应用程序，目前在我们生活的这个互联世界中变得越来越重要。本章第一部分将概述 P2P 技术。

Microsoft 并没有忘记 P2P 的出现，而是开发了它自己的工具和技术来使用 P2P。Microsoft Windows Peer-to-Peer Networking 平台可以用作 P2P 应用程序的通信架构。这个平台包含重要的组件 Peer Name Resolution Protocol(PNRP)。另外，.NET Framework 还包含名称空间 System.Net.PeerToPeer 和几个新类型及特性，它们可用于轻松地构建 P2P 应用程序。

对等网络是网络通信的另一种方式。为了理解 P2P 与网络通信的“标准”方法之间的区别，需

要先回过头来看看客户端-服务器通信。目前，客户端-服务器通信在网络应用程序中非常普遍。

46.1.1 客户端-服务器体系结构

传统上，我们使用客户端-服务器体系结构，通过网络(包括 Internet)与应用程序交互操作。Web 站点就是这方面的一个例子。在 Web 站点上，通过 Internet 给 Web 服务器发送一个请求，Web 服务器会返回我们需要的信息。如果要下载文件，就直接从 Web 服务器上下载。

同样，包含局域网或广域网连接的桌面应用程序一般连接到一台服务器上，如数据库服务器或包含其他服务的服务器。

客户端-服务器体系结构的这个简单形式如图 46-1 所示。

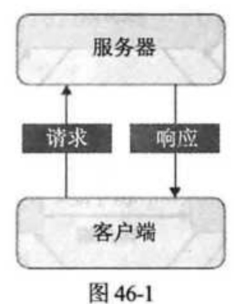


图 46-1

客户端-服务器体系结构本身并没有什么错误，而且在许多情况下，这就是我们需要的，但是它有一个可伸缩性问题。图 46-2 显示了客户端-服务器体系结构如何通过其他客户端来扩展。

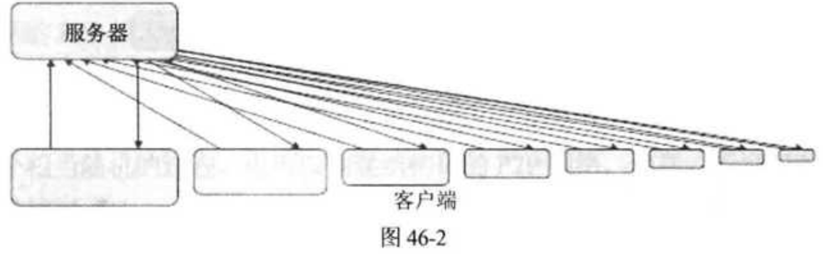


图 46-2

把每个添加了加大的负载的客户端放在服务器上，服务器就必须与每个客户端通信。再看看 Web 站点的例子，不断增加的通信负载就是 Web 站点崩溃的方式。有太多的流量，服务器简直无法响应。

当然，要缓解这种状况，可以实现一些扩展选项。可以增加服务器的功能和资源，也可以添加更多的服务器来扩展服务器。这种扩展当然受到可用技术和硬件成本的限制。扩展可能比较灵活，但需要一个额外的基础体系层，来确保客户端与各个服务器的通信，并独立于与它们正在通信的服务器来维护会话状态。这有许多解决方案，如 Web 场或服务器场产品。

46.1.2 P2P 体系结构

对等方法完全不同于扩展方法。在 P2P 中，不是集中精力并试图使服务器及其客户端之间的通信更流畅，而是考虑客户端之间的通信方式。

例如，客户端与之通信的 Web 站点是 www.wrox.com。在一个假想的情形中，Wrox 声称，本书的新版本要在 Web 站点 wrox.com 上发布，且可以免费下载，但在某一天后删除它。在本书可以从该 Web 站点上得到之前，假定有许多人都想访问这个 Web 站点，刷新其浏览器，等待下载文件的

出现。一旦文件出现在该 Web 站点上，每个人都试图同时下载它，这很可能使 wrox.com 的 Web 服务器在强大的压力下崩溃。

使用 P2P 技术可以避免这个 Web 服务器崩溃。P2P 技术不把文件直接从服务器发送给所有客户端，而是把文件仅发送给几个客户端。另外几个客户端可以从已经包含该文件的客户端上下载它，更多的客户端从这些二级客户端上下载该文件，以此类推。实际上，这个过程把文件分解成几个块，再把这些块分解到客户端上，一些客户端直接从服务器上下载文件，而一些客户端从其他客户端上下载文件，所以这个过程会更快完成。这就是文件共享技术(如 BitTorrent)的工作方式，如图 46-3 所示。

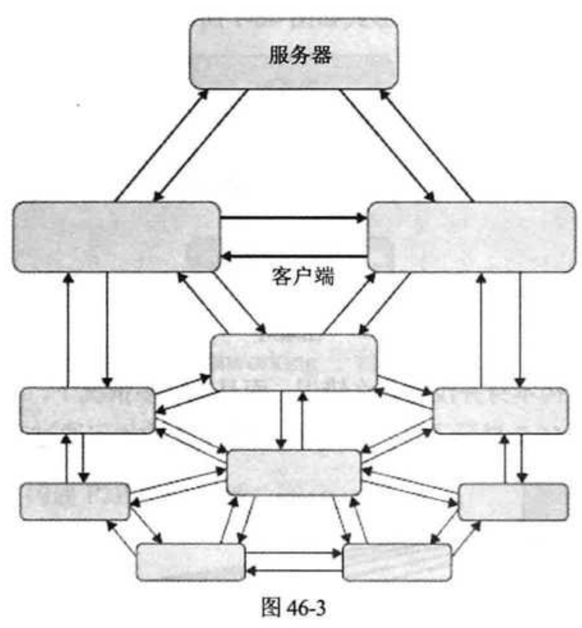


图 46-3

46.1.3 P2P 体系结构的挑战

在文件共享体系结构中，仍有一些问题需要解决。首先，客户端如何检测其他客户端的存在？如何定位其他客户端包含的文件块？另外，如何确保客户端之间的通信是按整个大陆来分割、优化的？

每个参与 P2P 网络应用程序的客户端都必须能执行如下操作，才能克服这些问题：

- 它必须能发现其他客户端
- 它必须能连接其他客户端
- 它必须能与其他客户端通信

发现问题有两个明显的解决方案。可以在服务器上保存客户端的列表，这样客户端就可以获得这个列表，并联系其他客户端(称为对等机)；也可以使用一个基础结构(如 PNRP，详见下一节)，它可以使客户端直接找到其他客户端。大多数文件共享系统都使用“服务器上的列表”解决方案，方法是把服务器称为跟踪器。另外，在文件共享系统中，任何客户端都可以用作服务器，如图 46-3 所示，只要声明客户端有一个文件，并用跟踪器注册它即可。实际上，纯粹的 P2P 网络根本不需要服务器，只需要对等机。

连接问题比较棘手，需要考虑 P2P 应用程序使用的网络的整体结构。如果有一组客户端，它们都可以彼此通信，这些客户端之间的连接拓扑结构就会非常复杂。为了提高性能，常常需要有多组客户端，每组客户端都包含该组中客户端之间的连接，但不包含其他组中的客户端连接。如果在本

地建立了这些组，性能就会得到极大的提升，因为客户端可以彼此通信，而不大需要联网的计算机之间的跳跃通信。

通信问题不太重要，因为也建立了通信协议(如TCP/IP)，并可以在这种情况下重用这些协议。但是，高端技术(例如，可以使用WCF服务，因此可以使用WCF提供的所有功能)和低端协议(例如，把数据同时发送给多个端点的多播协议)都有性能提升的空间。

发现、连接和通信是所有P2P实现方案的核心。本章介绍的实现方案使用System.Net.PeerToPeer类型和PNM进行发现，使用PNRP进行连接。如下面几节所述，这些技术涵盖了上述3个操作。

46.1.4 P2P术语

前面几节介绍了对等机的概念，这是在P2P网络中引用客户端的方式。“客户端”在P2P网络中没有意义，因为客户端不需要服务器。

彼此连接的对等机的组合称为网格(mesh)、云(cloud)或图(graph)，这些术语可以互换。如果满足如下条件之一，给定的组就是连接好的：

- 每对对等机之间都有一条连接路径，这样每个对等机都可以根据需要连接到任何其他对等机上。
- 在任意一对对等机之间遍历都有一个相对较小的连接数。
- 删除一个对等机不会妨碍其他对等机的彼此连接。

这并不意味着，每个对等机都必须能直接连接到其他所有对等机上。如果用数学方法分析网络，那么会发现对等机只需要连接数量相当少的其他对等机，即可满足这些条件。

另一个要注意的P2P概念是溢出(flooding)。溢出是单块数据通过网络传播到所有对等机上的方式，或者在网络中查询其他节点以定位特定数据块的方式。在未结构化的P2P网络中，先连接与自己最近的对等机，这个对等机再连接与它最近的对等机，以此类推，直到连接了网络中的所有对等机为止，这是一个相当随机的过程。也可以创建结构化的P2P网络，这样，查询和对等机之间的数据流就有定义好的路径了。

46.1.5 P2P解决方案

为P2P建立了基础结构后，不仅可以开发客户端-服务器应用程序的改进版本，还可以开发全新的应用程序。P2P特别适合于下述类型的应用程序：

- 内容发布应用程序，包括前面讨论的文件共享应用程序。
- 合作应用程序，例如，桌面共享和共享白板应用程序。
- 多用户通信应用程序，允许用户直接通信和交换数据，而不是通过服务器通信。
- 分布式处理应用程序，作为超级计算应用程序的另一种方式，处理海量数据。
- Web 2.0应用程序，在下一代动态Web应用程序中合并上述的一部分或全部功能。

46.2 PNRP

Microsoft Windows Peer-to-Peer Networking平台是Microsoft的P2P技术实现方式。它是Windows XP SP2及其以后版本的一部分，使用Peer Name Resolution Protocol(PNRP)可以发布和解析对等机的地址。本节学习这种技术。

当然，可以使用任意协议实现P2P应用程序，但如果在Microsoft Windows 8环境下工作(如果读者在阅读本书，就可能在Microsoft Windows环境下工作)，考虑PNRP至少是有意义的。PNRP本身并没有提供创建P2P应用程序所需的功能，而只是其中一种可用于解析对等机地址的底层技术。PNRP允许客户端注册一个端点(称为对等机名称)，该端点自动在云中的对等机之间循环。这个对等机名称封装在PNRP ID中。发现这个PNRP ID的对等机可以使用PNRP把它解析为实际的对等机名称，然后它就可以直接与相关联的客户端通信。

例如，定义一个表示WCF服务端点的对等机名称。可以使用PNRP把云中的这个对等机名称注册为PNRP ID。运行合适的客户端应用程序的对等机使用一个发现机制，该机制可以标识为服务提供的对等机名称，接着这个对等机可能发现这个PNRP ID。之后对等机就使用PNRP定位WCF服务的端点，并使用该服务。

PNRP没有假定对等机名称实际上表示什么。而是在发现对等机名称时，由对等机决定如何使用它们。在解析PNRP ID时，对等机从PNRP中获得的信息包括ID发布者的IPv6(实际上还有IPv4)地址和一个端口号，有时还有少量其他数据。除非对等机知道对等机名称的含义，否则不可能利用该信息执行有意义的操作。

1. PNRP ID

PNRP ID是一个256位的标识符。低位上的128位用于唯一标识对等机，高位上的128位标识一个对等机名称。高位上的128位是来自发布对等机的一个散列公钥和一个至多149个字符的字符串的散列组合，该字符串用于标识对等机名称。散列公钥(称为授权)和这个字符串(称为分类器)统称为P2P ID。也可以使用0代替散列公钥，此时对等机名称是不安全的(与使用公钥的安全对等机名称相反)。

PNRP ID的结构如图46-4所示。

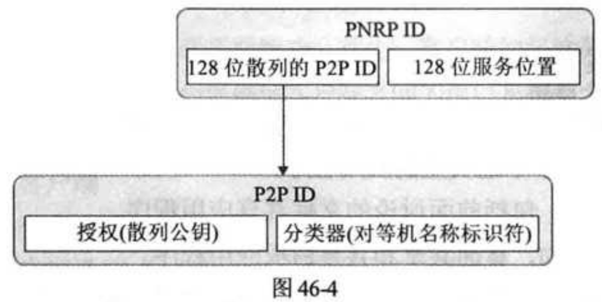


图46-4

对等机上的PNRP服务负责维护一个PNRP ID列表，包括它发布的PNRP ID和通过云中其他地方的PNRP服务实例获得的PNRP ID缓存列表。对等机尝试解析PNRP ID时，PNRP服务会使用端点的一个缓存副本来解析发布PNRP的对等机，如果对等机的邻居能解析它，就请求它们解析。最终建立与发布对等机之间的连接，PNRP服务能解析PNRP ID。

在进行上述所有操作时，不需要外界干预。我们只需确保对等机在用本地的PNRP服务解析了PNRP ID后，知道如何处理对等机名称。

对等机可以使用PNRP定位匹配某个特定P2P ID的PNRP ID。使用这个功能可以为不安全的对等机名称实现一个基本的发现机制。这是因为如果几个对等机提供了一个使用相同分类器且不安全

的对等机名称，P2P ID 就会相同。当然，因为任意对等机都可以使用不安全的对等机名称，用户必须确保所连接的端点就是希望的端点类型，所以这只是在本地网络中发现对等机的一种可行解决方案。

2. PNRP 云

在前面的讨论中，学习了 PNRP 如何注册和解析云中的对等机名称。云通过种子服务器维护，该服务器可以是运行 PNRP 服务的任意服务器，但 PNRP 服务至少要维护一个对等机记录。PNRP 服务可以使用两种类型的云：

- **本地链接**——这类云包含连接到本地网络上的计算机。如果 PC 有多个网络适配器，它就可以连接到多个本地链接的云上。
- **全局**——这类云包含默认连接到 Internet 上的计算机，尽管也可以定义一个私有的全局云。其区别是 Microsoft 为全局的 Internet 云维护种子服务器，而如果定义了私有的全局云，就必须使用自己的种子服务器。如果使用自己的种子服务器，就必须配置策略设置，从而确保所有对等机都连接到种子服务器上。

在 PNRP 的以前版本中，有第 3 种云：本地站点。现在不再使用它，所以本章不探讨它。

使用下面的命令可以确定自己连接到什么类型的云上：

```
netsh p2p pnrp cloud show list
```

通常结果如下所示。

```
Scope  Id     Addr  State            Name
-----  -----  ----- ---------------- -----
    1     0      1  Virtual          Global_
    3    13      1  Virtual          LinkLocal_ff00::%13/8
    3    19      1  Virtual          LinkLocal_ff00::%19/8
```

输出显示，有三个云可用，一个是全局云，另外两个是本地链接云。这可以从名称和 Scope 值看出，本地链接云的 Scope 值是 3，而全局云的 Scope 值是 1。为了连接到全局云，必须有一个 IPv6 地址。

云的状态如下：

- Active——如果云的状态是 active，就可以使用它发布和解析对等机名称。
- Alone——如果从云中查询的对等机没有连接到其他对等机上，其状态就是 alone。
- No Net——如果对等机没有连接到网络上，云的状态就从 active 改为 no net。
- Synchronizing——对等机连接到云上时，云的状态就是 synchronizing。这个状态可以很快变成其他状态，因为连接不需要很长时间，所以可能从来看不到云在这个状态下。
- Virtual——PNRP 服务仅根据需要，通过对等机名称注册和解析功能连接到云上。如果云的连接停用的时间超过 15 分钟，它就会进入这个 virtual 状态。

如果读者遇到网络连接问题，就应检查防火墙，以防它禁止通过 UDP 端口 3540 或 1900 的本地网络流量。UDP 端口 3540 由 PNRP 使用，UDP 端口 1900 由 SSDP(Simple Service Discovery Protocol，简单服务发现协议)使用，SSDP 由 PNRP 服务(以及 UPnP 设备)使用。

3. 自从 Windows 7 以来的 PNRP

自从 Windows 7 以来，PNRP 使用一个组件 DRT (Distributed Routing Table，分布式路由表)，这个组件负责确定 PNRP 使用的密钥的结构，其默认的实现方式是前面描述的 PNRP ID。使用 DRT API 可以定义另一种密钥模式，但密钥必须是 256 位的整数值(与 PNRP ID 一样)。这表示，虽然可以使用任意模式，但必须自己负责密钥的生成和安全性。使用这个组件可以在 PNRP 的范围之外创建新的云拓扑结构。这是一种高级技术，超出了本章的范围。

Windows 7 还引入了一种新方式 Easy Connect，用于连接到 Remote Assistance 应用程序的其他用户上。这个连接选项使用 PNRP 定位要连接的用户。一旦创建了一个会话，用户就可以通过 Easy Connect 或其他方式(如电子邮件邀请)共享其桌面，并通过 Remote Assistance 接口互相帮助。

46.3 构建 P2P 应用程序

既然学习了 P2P 网络的概念和.NET 开发人员可用于实现 P2P 应用程序的技术，就该讨论如何构建它们了。从前面的讨论中知道，因为使用 PNRP 可以发布、分配和解析对等机名称，所以这里首先要了解如何使用.NET 完成这个任务。接着陈述如何把 PNM 作为 P2P 应用程序的架构。这很有优势，因为如果使用 PNM，就不必实现自己的发现机制。

为了研究这些主题，需要学习 System.Net.PeerToPeer 名称空间中的类；要使用这些类，必须引用 System.Net.dll 程序集。

System.Net.PeerToPeer 名称空间中的类封装了 PNRP 的 API，允许与 PNRP 服务交互操作。使用这些类可以完成两个主要任务：

- 注册对等机名称
- 解析对等机名称

在下面几节中，除非特别指出，否则提到的所有类型都来自 System.Net.PeerToPeer 名称空间。

1. 注册对等机名称

要注册对等机名称，必须执行如下步骤：

(1) 用特定的分类器创建一个安全或不安全的对等机名称。

(2) 为对等机名称配置一个注册项，指定如下可选信息：

- TCP 端口号
- 用于注册对等机名称的云(如果未指定，PNRP 就在所有可用的云中注册对等机名称)
- 至多 39 个字符的注释

- 至多4096个字节的其他数据
- 是否自动为对等机名称生成端点(默认行为是，端点应从对等机的IP地址和端口号(假设指定了端口号)中生成)
- 端点集合

(3) 使用对等机名称注册项，通过本地PNRP服务注册对等机名称

在第3步之后，对等机名称就可以用于所选云中的所有对等机。对等机注册在明确地停止后不再能使用，或者在注册对等机名称的过程终止后不再能使用。

要创建对等机名称，可以使用PeerName类。以authority.classifier的形式从P2P ID的字符串表示中创建这个类的一个实例，或者从一个分类器字符串和PeerNameType中创建。可以使用PeerNameType.Secured或PeerNameType.Unsecured。例如：

```
varpn = new PeerName("Peer classifier", PeerNameType.Secured);
```

因为不安全的对等机名称使用的authority值是0，所以下面的代码行与上述代码等价：

```
varpn = new PeerName("Peer classifier", PeerNameType.Unsecured);

varpn = new PeerName("0.Peer classifier");
```

有了PeerName实例后，就可以使用它和一个端口号初始化一个PeerNameRegistration对象：

```
varpnr = new PeerNameRegistration(pn, 8080);
```

另外，还可以在使用其默认参数创建的PeerNameRegistration对象上设置PeerName和Port(可选)属性。也可以把Cloud实例指定为PeerNameRegistration构造函数的第3个参数，或者通过Cloud属性来指定。从云的名称中可以得到一个Cloud实例，或者使用Cloud如下的一个静态成员：

- Cloud.Global——这个静态属性获取全局云的一个引用。根据对等机策略配置，这可以是私有的全局云。
- Cloud.AllLinkLocal——这个静态字段获取一个云，该云包含可用于对等机的所有本地链接。
- Cloud.Available——这个静态字段获取一个云，该云包含可用于对等机的所有云，其中包含本地链接云和(如果可用)全局云。

创建了Cloud实例后，可以根据需要设置Comment和Data属性。但应注意这些属性的限制。如果试图把Comment设置为超过39个Unicode字符的字符串，就会接收到一个PeerToPeerException异常，如果试图把Data设置为超过4096个字节的byte[]，就会得到一个ArgumentOutOfRangeException异常。还可以使用EndPointCollection属性添加端点。这个属性是System.Net.IPEndPoint对象的一个System.Net.IPEndPointCollection集合。如果使用EndPointCollection属性，还需要把UseAutoEndPointSelection属性设置为false，以禁止自动生成端点。

准备注册对等机名称时，可以调用PeerNameRegistration.Start()方法。要从PNRP服务中删除对等机名称注册项，可以使用PeerNameRegistration.Stop()方法。

下面的代码通过注释注册一个安全对等机名称：

```
varpn = new PeerName("Peer classifier", PeerNameType.Unsecured);
varpnr = new PeerNameRegistration(pn, 8080);
pnr.Comment = "Get pizza here";
pnr.Start();
```

2. 解析对等机名称

要解析对等机名称，必须执行如下步骤：

(1) 从已知的P2P ID或通过发现技术得到的P2P ID中生成一个对等机名称。

(2) 使用解析器解析对等机名称，得到一个对等机名称记录集合。可以把解析器限制为只用于某个特定的云，和/或要返回的最多结果数。

(3) 对于获得的每个对等机名称记录，需要获取对等机名称、端点、注释和额外的数据信息。

这个过程从一个类似于对等机名称注册项的PeerName对象开始。这里的区别是使用通过一个或多个远程对等机注册的对等机名称。从本地链接云中获取活动对等机列表的最简单的方式是使用同一个分类器为每个对等机注册一个不安全的对等机名称，再在解析阶段使用同一个对等机名称。但是对于全局云，不推荐使用这个方法，因为不安全的对等机名称很容易被盗取。

要解析对等机名称，可以使用PeerNameResolver类。有了这个类的一个实例后，就可以使用Resolve()方法同步解析对等机名称，或者使用ResolveAsync()方法异步解析对等机名称。

用一个PeerName参数可以调用Resolve()方法，也可以给该方法传递要解析的可选Cloud实例，或者传递要返回的最大对等机数(int)，或者传递这两个参数。这个方法返回一个PeerNameRecord-Collection实例，这是PeerNameRecord对象的集合。例如，下面的代码解析所有本地链接云中一个不安全的对等机名称，最多返回5个结果：

```
varpn = new PeerName("0.Peer classifier");
varpnres = new PeerNameResolver();
PeerNameRecordCollectionpnrc = pnres.Resolve(pn, Cloud.AllLinkLocal, 5);
```

ResolveAsync()方法使用一个标准的异步方法调用模式。给该方法传递一个唯一的userState对象，为正在查找的对等机侦听ResolveProgressChanged事件，在该方法终止时侦听ResolveCompleted事件。可以用ResolveAsyncCancel()方法取消未决的异步请求。

ResolveProgressChanged事件的处理程序使用ResolveProgressChangedEventArgs事件参数，该参数派生自标准的System.ComponentModel.ProgressChangedEventArgs类。可以使用在事件处理程序中接收到的事件参数对象的PeerNameRecord属性获得对查找到的对等机名称记录的引用。

同样，ResolveCompleted 事件也需要一个事件处理程序，它使用ResolveCompletedEventArgs类型的参数，该参数派生自AsyncCompletedEventArgs。这个类型包含一个PeerNameRecordCollection参数，可用于获得查找到的对等机名称记录的完整列表。

下面的代码显示了这两个事件的事件处理程序的实现代码：

```
privatepnres_ResolveProgressChanged(object sender,
    ResolveProgressChangedEventArgs e)
{
  // Use e.ProgressPercentage (inherited from base event args)
  // Process PeerNameRecord from e.PeerNameRecord
}

privatepnres_ResolveCompleted(object sender,
    ResolveCompletedEventArgs e)
```

```
{
  // Test for e.IsCancelled and e.Error (inherited from base event args)
  // Process PeerNameRecordCollection from e.PeerNameRecordCollection
}
```

有一个或多个 PeerNameRecord 对象后，就可以处理它们了。这个 PeerNameRecord 类提供 Comment 和 Data 属性，用于检查在对等机名称注册项(如果存在)中设置的注释和数据，PeerName 属性可获取对等机名称记录的 PeerName 对象，最重要的是 EndPointCollection 属性。与 PeerNameRecordRegistration 一样，这个属性也是 System.Net.IPEndPoint 对象的一个 System.Net.IPEndPointCollection 集合。这些对象可用于以任意方式连接对等机提供的端点。

3. System.Net.PeerToPeer 中的代码访问安全性

System.Net.PeerToPeer 名称空间还包含如下两个用于 CAS(参见第 20 章)的类，更多信息可参见第 22 章：

- PnrpPermission，它继承自 CodeAccessPermission
- PnrpPermissionAttribute，它继承自 CodeAccessSecurityAttribute

这两个类可以用通常的 CAS 方式给 PNRP 访问提供权限功能。

4. 示例应用程序

本章可下载的代码包含一个示例 P2P 应用程序(P2PSample)，它使用本节介绍的概念和名称空间。它是一个 WPF 应用程序，它为一个对等机端点使用一个 WCF 服务。

该应用程序用一个应用程序配置文件来配置，在该文件中，可以指定对等机的名称和侦听的端口，如下所示（代码文件 App.config）：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="username" value="Christian" />
    <add key="port" value="8731" />
  </appSettings>
</configuration>
```

构建应用程序后，或者把它复制到本地网络的其他计算机上，并运行所有实例来测试它；或者在一台计算机上运行多个实例来测试它。如果选择后一个选项，就必须修改每个配置文件(复制本地计算机上 Debug 目录的内容，依次编辑每个配置文件)，以修改每个实例使用的端口。用这两种方式测试该应用程序时，如果还修改了每个实例的用户名，结果就会更清楚。

运行对等机应用程序后，就可以使用 Refresh 按钮异步地获得对等机列表。定位了一个对等机后，就可以单击对等机的 Message 按钮，发送一条默认消息。

图 46-5 显示了这个应用程序，它在一台计算机上运行 3 个实例。在图 46-5 中，一个对等机给另一个对等机发送了消息，这会生成一个对话框。

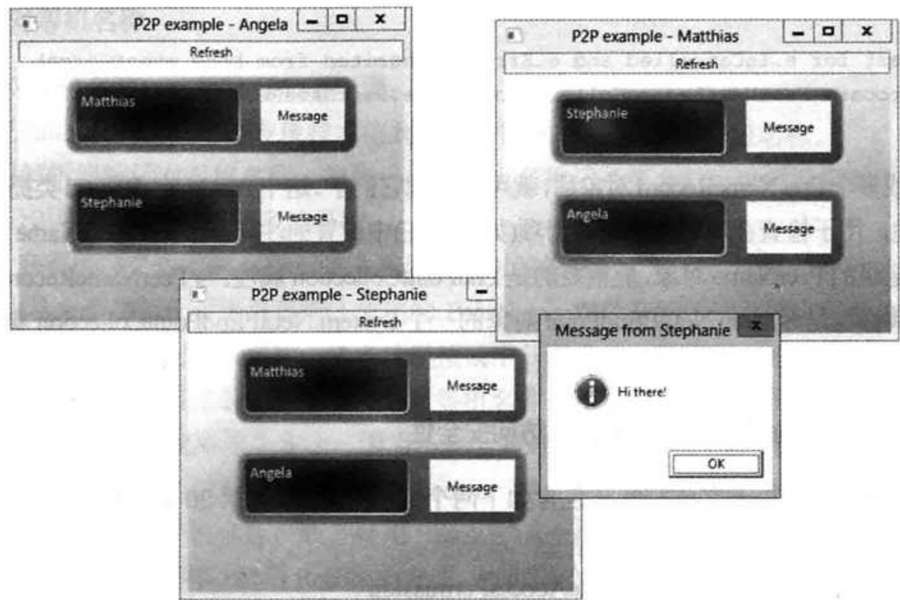


图 46-5

现在看看代码。在 MainWindow 类的字段成员中（代码文件 MainWindow.xaml.cs），定义了一个可观察的集合，它包含所有的对等机。在该类的构造函数中，只添加了一个 PeerEntry，它为用户提供信息，以单击 Refresh 按钮，获取所有的对等机。

```
public partial class MainWindow : Window
{
  private P2PService localService;
  privateServiceHost host;
  privatePeerNamepeerName;
  privatePeerNameRegistrationpeerNameRegistration;
  privateObservableCollection<PeerEntry>peerList =
    newObservableCollection<PeerEntry>();
  private object peersLock = new object();

  publicMainWindow()
  {
    InitializeComponent();
    this.DataContext = peerList;
    peerList.Add(
      newPeerEntry
      {
        DisplayString = "Refresh to look for peers.",
        ButtonsEnabled = false
      });
    BindingOperations.EnableCollectionSynchronization(peerList, peersLock);
  }
```

在这个应用程序中，大多数工作都在 MainWindow 窗口的 Window_Loaded()事件处理程序中完成。这个方法首先加载配置信息，并用用户名设置窗口的标题：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // Get configuration from app.config
```

```
string port = ConfigurationManager.AppSettings["port"];
string username = ConfigurationManager.AppSettings["username"];
stringmachineName = Environment.MachineName;
stringserviceUrl = null;

// Set window title
this.Title = string.Format("P2P example - {0}", username);
```

接着使用对等机主机地址和配置的端口确定要把 WCF 服务存放在哪个端点上。因为该服务使用 NetTcpBinding 绑定，所以端点的 URL 使用 net.tcp 协议：

```
// Get service url using IPv4 address and port from config file
serviceUrl = Dns.GetHostAddresses(Dns.GetHostName())
  .Where(address =>address.AddressFamily == AddressFamily.InterNetwork)
  .Select(address =>
  string.Format("net.tcp://{0}:{1}/P2Pservice", address, port))
  .FirstOrDefault();
```

验证端点的 URL 后，就注册并启动 WCF 服务：

```
// Check for null address
if (serviceUrl == null)
{
  // Display error and shutdown
  MessageBox.Show(this, "Unable to determine WCF endpoint.",
    "Networking Error", MessageBoxButton.OK,MessageBoxImage.Stop);
  Application.Current.Shutdown();
}

// Register and start WCF service.
localService = new P2PService(this, username);
host = new ServiceHost(localService, new Uri(serviceUrl));
var binding = new NetTcpBinding();
binding.Security.Mode = SecurityMode.None;
host.AddServiceEndpoint(typeof(IP2PService), binding, serviceUrl);
try
{
  host.Open();
}
catch (AddressAlreadyInUseException)
{
  // Display error and shutdown
  MessageBox.Show(this, "Cannot start listening, port in use.",
    "WCF Error", MessageBoxButton.OK, MessageBoxImage.Stop);
  Application.Current.Shutdown();
}
```

该服务类的单一实例用于启动主机应用程序和服务之间的简便通信(用于发送和接收消息)。另外要注意，在绑定配置中，为了简单起见，禁用了安全性。

之后，使用 System.Net.PeerToPeer 名称空间中的类注册对等机名称：

```
// Create peer name
peerName = new PeerName("P2P Sample", PeerNameType.Unsecured);
```

```
        // Prepare peer name registration in link local clouds
        peerNameRegistration = new PeerNameRegistration(peerName, int.Parse(port));
        peerNameRegistration.Cloud = Cloud.AllLinkLocal;

        // Start registration
        peerNameRegistration.Start();
    }
```

单击 Refresh 按钮时，RefreshButton_Click()事件处理程序使用 PeerNameResolve.ResolveAsync()异步地获得对等机：

```
privateasync void RefreshButton_Click(object sender, RoutedEventArgs e)
{
  // Create resolver and add event handlers
  var resolver = new PeerNameResolver();
  resolver.ResolveProgressChanged +=
    newEventHandler<ResolveProgressChangedEventArgs>(
      resolver_ResolveProgressChanged);
  resolver.ResolveCompleted +=
    newEventHandler<ResolveCompletedEventArgs>(
      resolver_ResolveCompleted);

  // Prepare for new peers
  peerList.Clear();
  RefreshButton.IsEnabled = false;

  // Resolve unsecured peers asynchronously
  resolver.ResolveAsync(new PeerName("0.P2P Sample"), 1);
```

接收到对等信息时，就会触发接收了对等信息的事件 ResolveProgressChanged 和 ResolveCompleted。如果对等机还没有激活，就定义一个超时时间，来取消解析过程，从而触发 ResolveCompleted。超时时间用 Task.Delay 方法来处理。在超时时间过后，就用相同的用户状态值调用 ResolveAsyncCancel，这个用户状态值传递给了 ResolveAsync 方法。有了用户状态值后，同一个解析任务就映射为取消操作。

```
  awaitTask.Delay(5000);
  resolver.ResolveAsyncCancel(1);
}
```

剩余的代码负责显示对等机，并与对等机通信，读者可以自己研究它们。

通过 P2P 云提供 WCF 端点是定位企业中的服务的一种好方法，也是在对等机之间通信的好方法，如本例所示。

46.4 小结

本章介绍了如何在应用程序中使用 P2P 类实现对等网(P2P)功能。

我们探讨了 P2P 可以实现的解决方案类型，这些解决方案的构造方式，如何使用 PNRP 和 System.Net.PeerToPeer 名称空间中的类型，还了解了作为 P2P 端点提供 WCF 服务的有用技术。

下一章介绍消息队列，它涉及 System.Messaging 名称空间中的类和 WCF。

第47章

消息队列

本章要点

- 消息队列体系结构
- 使用消息队列管理工具
- 编程创建消息队列
- 收发消息
- 课程订单示例应用程序
- 使用消息队列和WCF

本章源代码下载地址(wrox.com)：

打开网页 http://www.wrox.com/go/procsharp，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- 使用队列
- 创建消息队列
- 寻找队列
- 打开队列
- 发送消息
- 接收消息
- 课程订单示例
- 发送者
- 接收者
- WCF 课程订单示例
- 发送者
- 接收者

47.1 概述

System.Messaging 名称空间包含的类可以用 Windows 操作系统的消息队列功能读写消息。消息传递功能可以在断开连接的环境下使用，在该环境下，客户端和服务器不需要同时运行。

本章介绍消息队列的体系结构和用法，探讨 System.Messaging 名称空间中用于创建队列和收发消息的类，学习如何使用确认队列和响应队列从服务器中获得应答，如何通过 WCF 消息队列绑定使用消息队列。

在开始学习消息队列编程之前，本节先讨论消息传递的基本概念，将它与同步和异步编程比较。在同步编程中，调用一个方法时，调用者必须等待该方法执行完毕。在异步编程中，主调线程可以启动同时运行的方法。异步编程可以通过委托、支持异步方法的类库(例如，Web 服务代理，System.Net 和 System.IO 类)，或使用自定义线程和任务(详见第 21 章)来实现。在同步和异步编程中，客户端和服务器必须同时运行。

尽管消息队列是异步进行的，但因为客户端(发送者)不等待服务器(接收者)读取发送给它的数据，所以消息队列与异步编程有很大的区别。消息队列可以在断开连接的环境下进行。在发送数据时，接收者可以离线。在以后的某个时刻，接收者上线时，就会接收到数据，而无须来自发送应用程序的干预。

可以把打开连接的编程和断开连接的编程与给他人打电话和发送电子邮件做比较。在给他人打电话时，电话两端的人都必须同时在线；这种通信是同步的。而利用电子邮件，发送者不能确定处理电子邮件的时间。使用这个技术的人工作在断开连接的模式。当然，电子邮件可能从来不被处理，而它可能被忽略。这就是断开连接的通信的本质。为了避免这个问题，可以要求发送一个应答，以确认已读取电子邮件。如果在指定的时间内没有收到应答，就要处理这个“异常”。这也可以使用消息队列来实现。

在某些方面，消息队列就是应用程序之间通信的电子邮件，而不是人与人之间通信的电子邮件。但是，它提供了邮件服务没有的许多功能，例如，有保证的交付、事务、确认、使用内存的快捷模式等。如下一节所述，消息队列有许多可用于应用程序之间通信的特性。

通过消息队列功能，可以在打开连接或断开连接的环境下发送、接收和路由消息。图 47-1 显示了使用消息的一种非常简单的方式。发送者给消息队列发送消息，接收者从队列中接收消息。

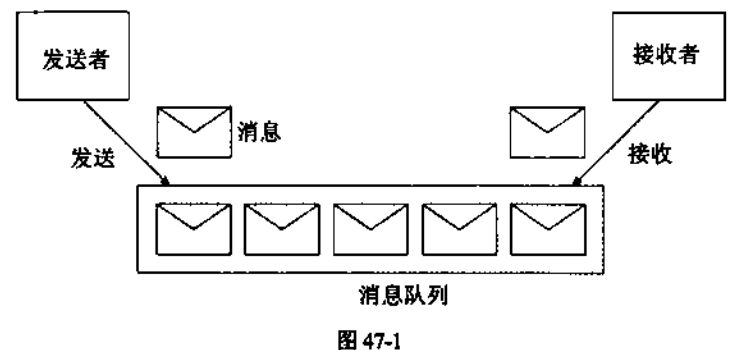


图 47-1

47.1.1 使用消息队列的场合

使用消息队列的一个场合是，客户端应用程序常常从网络上断开连接(例如，销售员在站点上访

问顾客)。销售员可以直接在顾客的站点上输入订购数据。应用程序把每个订单的消息发送给位于客户端系统上的消息队列(如图 47-2 所示)。只要销售员回到办公室，订单就会自动从客户端系统的消息队列传输到目标系统的消息队列上，在目标系统上处理消息。

除了使用笔记本电脑之外，销售员还可以使用支持消息队列的小型设备。

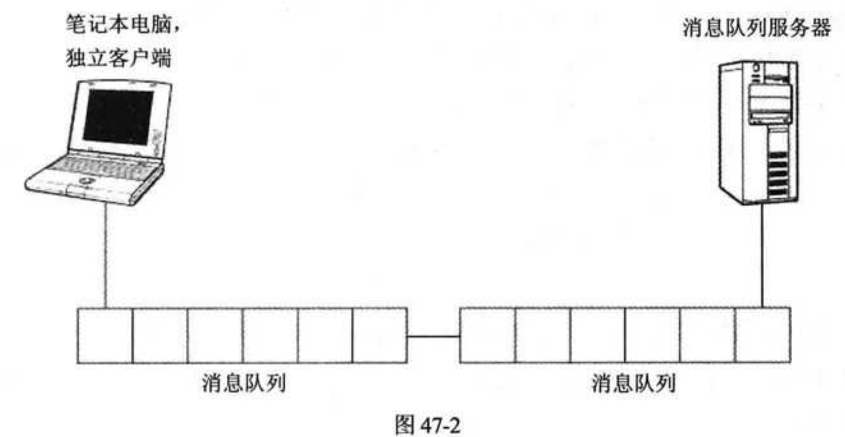


图 47-2

消息队列还可以在打开连接的环境下使用。假定在一个电子商务站点上(如图 47-3 所示)，其中服务器在某些时刻(如傍晚和周末)的订单事务负载是满的，但在晚上，其负载很低。一种解决方案是购买一台更快的服务器或在系统中添加更多服务器，以处理高峰时的订单。还有一种成本较低的解决方案：把事务从高负载的时段移动到低负载的时段，即削峰平谷。在这种方案中，把订单发送到消息队列中，接收端按对数据库系统有利的速度读取订单。现在，系统的负载被平摊到各个时间段内，这样处理事务的服务器就可以比数据库系统升级的系统开销少。

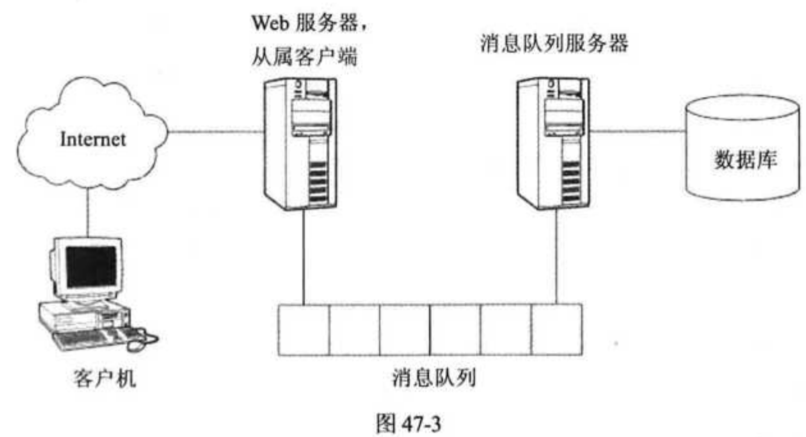


图 47-3

47.1.2 消息队列功能

消息队列是 Windows 8 操作系统的一部分。这个服务的主要功能如下：

- 消息可以在断开连接的环境下发送。不需要同时运行正在发送和正在接收的应用程序。
- 使用快捷模式，消息可以非常快地发送。在快捷模式下，消息存储在内存中。
- 对于可恢复的机制，消息可以使用有保证的交付方式发送。可恢复的消息存储在文件中。在服务器重新启动时发送它们。

- 用访问控制列表来保护消息队列，可以确定哪些用户可以发送或接收队列中的消息。消息还可以加密，避免网络嗅探器读取其中的数据。消息在发送时可以指定优先级，这样可以更快地处理高优先级的项。
- Message Queuing 3.0 支持多播消息的发送。
- Message Queuing 4.0 支持病毒消息。病毒消息不能解析。可以定义一个病毒队列，把不能解析的消息移动到病毒队列中。例如，如果从正常的队列中读取消息后，对应作业要把消息插入数据库中，但消息不能插入数据库，因此该作业失败，该消息就会发送到病毒队列中。有人负责处理病毒队列，这个人应以能解析病毒消息的方式来处理该消息。
- Message Queuing 5.0 支持更安全的身份验证算法，可以处理大量队列(Message Queuing 4.0 在处理几千个队列时有性能问题)。

因为消息队列是操作系统的一部分，所以不能在 Windows Vista 和 Windows Server 2008 系统上安装 Message Queuing 5.0。Message Queuing 5.0 是 Windows 7 及以后版本的一部分。

本章剩余的内容探讨这些功能的用法。

47.2 Message Queuing 产品

Message Queuing 5.0 是 Windows Server 2008 R2 和 Windows 7 及以后版本的一部分。Windows 2000 和 Message Queuing 2.0 一起发布，它不支持 HTTP 协议和多播消息。Message Queuing 3.0 是 Windows Server 2003 和 Windows XP 的一部分。Message Queuing 4.0 是 Windows Server 2008 和 Windows Vista 的一部分。

使用 Windows 8.1 的 Configuring Programs and Features 中的 Turn Windows Features on or off 链接时，有一个单独用于 Message Queuing 选项的部分。在这个部分中，可以选择如下组件：

- Microsoft Message Queue(MSMQ) Server Core——Core 子组件是消息队列基本功能所必需的。
- Active Directory Domain Services Integration——有了它，消息队列名就会写入 Active Directory。利用这个选项可以使用 Active Directory Integration 查找队列，用 Windows 用户和组保护队列。
- MSMQ HTTP Support——它允许使用 HTTP 协议发送和接收消息。
- Triggers——利用 Triggers，可以在接收到新消息时实例化应用程序。
- Multicast Support——有了它，就可以把一条消息发送给一组服务器。
- MSMQ DCOM Proxy——有了 DCOM 代理，系统就可以使用 DCOM API 连接到远程服务器上。

在安装 Message Queuing 时，必须启动 Message Queuing 服务(如图 47-4 所示)。这条服务读写消息，与其他 Message Queuing 服务器通信，把消息路由到网络上。

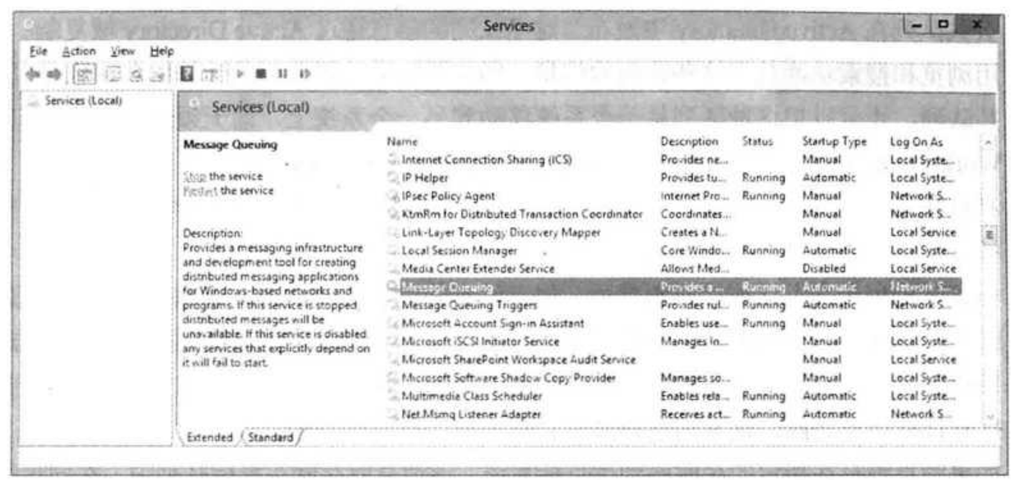


图 47-4

47.3 消息队列体系结构

通过消息队列，可以从消息队列中读写消息。消息和消息队列有几个属性需要进一步阐述。

47.3.1 消息

把消息发送到消息队列中。消息包含正文(包含要发送的数据)和一个标签(消息的标题)。在消息的正文中可以放置任意信息。在.NET 中，有几个格式化程序转换要放在正文中的数据。除了标签和正文之外，消息还包含发送者、超时配置、事务 ID 或优先级等信息。

消息队列有几种类型的消息：

- 一般消息——由应用程序发送。
- 确认消息——报告一般消息的状态。把确认消息发送到管理队列中，来报告一般消息的发送是否成功。
- 响应消息——当原始发送者需要某种特殊应答时，由接收应用程序发送响应消息。
- 报告消息——由消息队列系统生成。测试消息和路由跟踪消息属于此类。

消息可以有优先级，优先级定义从队列中读取消息的顺序。因为消息在队列中按照其优先级排序，所以从队列中读取的下一条消息就是优先级最高的那条消息。

消息有两种传递模式：快捷模式和可恢复模式。快捷消息的传送速度非常快，因为消息只使用消息存储器来存储。可恢复消息在路由的每一阶段都要存储在文件中，直到消息传递到目的地为止。这样，即使计算机重新启动或网络失败，消息的传递也能得到保证。

事务消息是可恢复消息的一种特殊版本。在事务消息传递过程中，可以确保消息只到达目的地一次，且按照它们发送的顺序到达目的地。优先级不能在事务消息中使用。

47.3.2 消息队列

消息队列是一个消息存储库。存储在磁盘上的消息位于<windir>\system32\msmq\storage 目录。

公共队列或私有队列通常用于发送消息，但还有其他队列类型：

- 公共队列在 Active Directory 中发布。这些队列的信息通过 Active Directory 域复制。可以使用浏览和搜索功能获得这些队列的信息。即使不知道放置队列的计算机名，也可以访问公共队列。还可以把这种队列从一个系统移动到另一个系统上，而无须通知客户。但不能在 Workgroup 环境下创建公共队列，因为需要 Active Directory。
- 私有队列不在 Active Directory 中发布。只有在知道队列的完整路径名时才能访问这些队列。私有队列可以在 Workgroup 环境下使用。
- 日志队列用于在发送或接收消息后，保存消息的副本。启动公共或私有队列的日志功能，就会自动创建一个日志队列。在日志队列中，可以有两种不同的队列类型：源日志队列和目标日志队列。通过消息的属性打开源日志功能，用源系统存储日志消息。用队列的属性打开目标日志功能，这些消息存储在目标系统的日志队列中。
- 如果消息没有在指定的超时前到达目标系统，该消息就存储在死信队列中。在同步编程中，错误会被立即检测出来，但使用消息队列处理错误的方式必须不同。死信队列可以用于检查未到达目的地的消息。
- 管理队列包含发送消息的确认消息。发送者可以指定一个管理队列，发送者从中接收消息是否成功发送的通知。
- 如果需要把多条简单的确认消息用作接收端的应答，就可以使用响应队列。接收应用程序可以把响应消息发送回原始发送者。
- 报告队列用于测试消息。把公共或私有队列的类型改为预定义的 ID{55EE8F33-CCE9-11CF-B108-0020AFD61CE9}，就可以创建报告队列。报告队列可以用作跟踪其路由中的消息的测试工具。
- 系统队列是私有的，由消息队列系统使用。这些队列用于管理消息，存储通知消息，保证事务消息的正确顺序。

47.4 Message Queuing 管理工具

在介绍如何以编程方式处理 Message Queuing 之前，本节先讨论 Windows 操作系统中的管理工具，以创建和管理队列和消息。

这里介绍的工具不仅能用于 Message Queuing。只有安装了 Message Queuing，才能使用这些工具的 Message Queuing 功能。

47.4.1 创建消息队列

消息队列可以使用 Computer Management MMC 插件创建。在树型视图面板上，Message Queuing 位于 Services and Applications 项的下面。选择 Private Queues 或 Public Queues，就可以从 Action 菜单中创建新队列，如图 47-5 所示。只有在 Active Directory 模式下配置 Message Queuing，公共队列才可用。

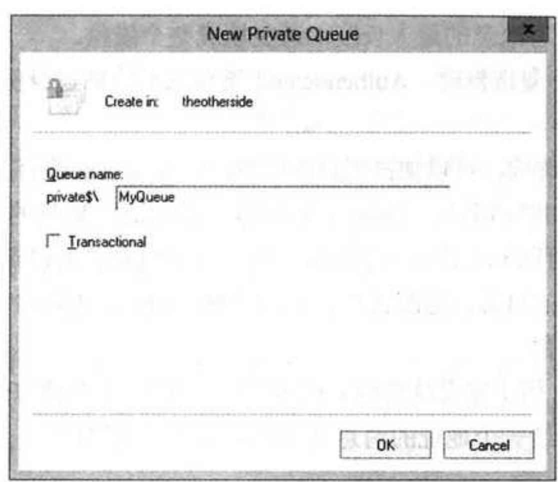


图 47-5

47.4.2　消息队列属性

在创建队列后，可以使用 Computer Management 插件，在树型面板上选择队列，选择 Action | Properties 菜单，来修改队列的属性，如图 47-6 所示。

图 47-6

可以配置几个选项：

- Label 是队列的名称，它可用于搜索队列。
- Type ID 默认设置为{00000000-0000-0000-0000-000000000000}，把多个队列映射到一个类别或类型上。报告队列使用特定的类型 ID，如前所述。类型 ID 是通用唯一 ID(UUID)或 GUID。

可以使用 uuidgen.exe 或 guidgen.exe 实用程序创建自定义类型标识符。uuidgen.exe 是一个命令行实用程序，用于创建唯一的 ID，guidgen.exe 是创建 UUID 的图形版本。

- 可以限制队列中所有消息的最大长度，避免填满整个磁盘。
- 勾选 Authenticated 复选框时，Authenticated 选项只允许通过身份验证的用户读写队列中的消息。
- 使用 Privacy level 选项，可以加密消息的内容。可以设置的值有 None、Optional 和 Body。None 表示不接收加密的消息，Body 只接收加密的消息，默认的 Optional 值接收两者。
- 使用 Journal 设置可以配置目标日志功能。使用这个选项，可以把接收的消息副本存储到日志中。可以为队列的日志消息配置能使用的磁盘空间的最大容量。当到达这个最大容量时，就停止目标日志功能。
- 配置选项 Multicast 用于定义队列的多播 IP 地址。同一个多播 IP 地址可以用于网络上的不同节点，这样发送给一个地址的消息就可以用多个队列接收。

47.5　消息队列的编程实现

既然理解了消息队列的体系结构之后，就可以探讨其编程了。下面几节将学习如何创建和控制队列，如何发送和接收消息。

还要构建一个小型课程订单应用程序，它由发送部分和接收部分组成。

47.5.1　创建消息队列

前面了解了如何使用 Computer Management 实用程序创建消息队列。消息队列还可以用 MessageQueue 类的 Create()方法以编程方式创建。

在 Create()方法中，必须传递新队列的路径。路径包括队列所在主机的名称和队列的名称。在下面的例子中，要在本地主机上创建 MyNewPublicQueue 队列。为了创建私有队列，路径名必须包含 Private$，如\Private$\MyNewPrivateQueue。

调用 Create()方法之后，就可以修改队列的属性。例如，使用 Label 属性，把队列的标签设置为 Demo Queue。示例程序把队列的路径和格式名写到控制台上。格式名用 UUID 自动创建，UUID 可用于访问队列，且无需服务器名(代码文件 WorkingWithQueues/CreateMessageQueue/Program.cs)：

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      using (var queue = MessageQueue.Create(@".\MyNewPublicQueue"))
      {
        queue.Label = "Demo Queue";
        Console.WriteLine("Queue created:");
        Console.WriteLine("Path: {0}", queue.Path);
        Console.WriteLine("FormatName: {0}", queue.FormatName);
      }
    }
  }
}
```

创建队列时需要管理权限。通常不希望应用程序的用户拥有管理权限。这就是队列通常用安装程序创建的原因。本章后面将介绍如何使用 MessageQueueInstaller 类创建消息队列。

47.5.2 查找队列

路径名和格式名可以用于标识队列。要查找队列，必须区分公共队列和私有队列。公共队列在 Active Directory 中发布。对于这些队列，无须知道它们所在的系统。只有在已知队列所在的系统私有队列名称，才能找到私有队列。

在 Active Directory 域中搜索队列的标签、类别或格式名，就可以找到公共队列。还可以获得计算机上的所有队列。MessageQueue 类的静态方法 GetPublicQueuesByLabel()、GetPublicQueuesByCategory()和 GetPublicQueuesByMachine()可以搜索队列。GetPublicQueues()方法返回包含域中所有公共队列的数组(代码文件 WorkingWithQueues/FindQueues/Program.cs)。

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      foreach (var queue in MessageQueue.GetPublicQueues())
      {
        Console.WriteLine(queue.Path);
      }
    }
  }
}
```

GetPublicQueues()方法是重载的。它的一个重载版本允许传递 MessageQueueCriteria 类的一个实例。利用这个类可以搜索在某个时刻之前或之后创建或修改的队列，还可以查找队列的类别、标签或计算机名。

可以使用静态方法 GetPrivateQueuesByMachine()搜索私有队列。这个方法返回指定系统中的所有私有队列。

47.5.3 打开已知队列

如果队列名已知，就不需要搜索它。使用路径或格式名就可以打开队列。路径或格式名都在 MessageQueue 类的构造函数中设置。

1. 路径名

路径指定了打开队列需要的计算机名和队列名。下面的代码示例打开本地主机上的 MyPublicQueue 队列。为了确定队列是否存在，可以使用静态方法 MessageQueue.Exists()(代码文件

WorkingWithQueues/OpenQueue/Program.cs):

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      if (MessageQueue.Exists(@".\MyPublicQueue"))
      {
        var queue = new MessageQueue(@".\MyPublicQueue");
        //...

      }
      else
      {
        Console.WriteLine("Queue .\MyPublicQueue not existing");
      }
    }
  }
}
```

根据队列的类型，在打开队列时需要不同的标识符。表 47-1 列出了指定类型的队列名的语法。

表 47-1

队 列 类 型	语 法
公共队列	MachineName\QueueName
私有队列	MachineName\Private$\QueueName
日志队列	MachineName\QueueName\Journal$
计算机日志队列	MachineName\Journal$
计算机死信队列	MachineName\DeadLetter$
计算机事务死信队列	MachineName\XactDeadLetter$

在使用路径名打开公共队列时，需要传递计算机名。如果计算机名未知，则可以使用格式名代替。私有队列的路径名只能在本地系统上使用，必须使用格式名远程访问私有队列。

2. 格式名

除了路径名之外，还可以使用格式名打开队列。格式名用于在 Active Directory 中搜索队列，获得队列所在的主机。在断开连接的环境下，在发送消息时队列不能到达，此时就需要使用格式名：

```
var queue = new MessageQueue(
    @"FormatName:PUBLIC=09816AFF-3608-4c5d-B892-69754BA151FF");
```

格式名还有一些其他用途。它可以用于打开私有队列，并指定要使用的协议：

- 要访问私有队列，必须给构造函数传递字符串 FormatName:PRIVATE=MachineGUID\QueueNumber。在创建队列时，会生成私有队列的队列号。队列号在<windows>\System32\msmq\storage\lqs 目录下。
- 使用 FormatName:DIRECT=Protocol:MachineAddress\QueueName，可以指定用于发送消息的协议。Message Queuing 3.0 及以后版本支持 HTTP 协议。
- FormatName:DIRECT=OS:MachineName\QueueName 是使用格式名指定队列的另一种方式。此时不需要指定协议，但仍可以使用计算机名和格式名。

47.5.4 发送消息

可以使用 MessageQueue 类的 Send()方法给队列发送消息。作为参数传递给 Send()方法的对象序列化到相关联的队列上。Send()方法是重载的，这样才能传递标签和 MessageQueueTransaction 对象。Message Queuing 的事务行为在后面论述。

下面的代码示例先检查队列是否存在，如果不存在，就创建一个队列。接着打开队列，使用 Send()方法给队列发送 Sample Message 消息。

路径名给服务器名指定"."，表示它是本地系统。私有队列的路径名只能在本地使用(代码文件 WorkingWithQueues/SendMessage/Program.cs)。

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      try
      {
        if (!MessageQueue.Exists(@".\Private$\MyPrivateQueue"))
        {
          MessageQueue.Create(@".\Private$\MyPrivateQueue");
        }
        var queue = new MessageQueue(@".\Private$\MyPrivateQueue");

        queue.Send("Sample Message", "Label");
      }
      catch (MessageQueueException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

图 47-7 显示了 Computer Management 管理工具，其中可以看到到达队列的消息。

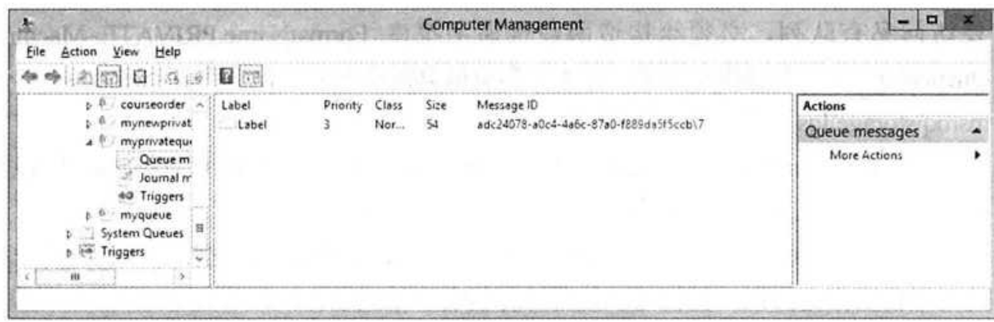


图 47-7

打开消息，选择对话框的 Body 选项卡(如图 47-8 所示)，就可以看到用 XML 格式化后的消息。判断消息的格式化方式是与消息队列相关联的格式化程序的功能。

图 47-8

1. 消息格式化程序

消息传输给队列的格式取决于格式化程序。MessageQueue 类有一个 Formatter 属性，通过它可以指定格式化程序。默认的格式化程序 XmlMessageFormatter 会用 XML 语法格式化消息，如前面的例子所示。

消息格式化程序实现 IMessageFormatter 接口。System.Messaging 名称空间中有 3 个消息格式化程序：

- XmlMessageFormatter 是默认的格式化程序，它使用 XML 序列化对象，XML 格式的内容详见第 34 章。
- 使用 BinaryMessageFormatter，可以用二进制格式对消息进行序列化。这些消息比使用 XML 格式化的消息短。
- ActiveXMessageFormatter 是一个二进制格式化程序，这样可以用 COM 对象读写消息。使用这个格式化程序，可以用.NET 类把消息写入队列中，使用 COM 对象从队列中读取消息，反之亦然。

图 47-8 中用 XML 显示的示例消息是用图 47-9 中的 BinaryMessageFormatter 格式化的。

Label Properties

General | Queues | Sender | Body

The body size is 38 bytes.

```
00 01 00 00 00 FF FF FF .....ÿÿÿ
FF 01 00 00 00 00 00 00 ÿ.......
00 06 01 00 00 00 0E 53 .......S
61 6D 70 6C 65 20 4D 65 ample Me
73 73 61 67 65 0B       ssage.
```

OK | Cancel | Apply

图 47-9

2. 发送复杂的消息

除了传递字符串之外，还可以给 MessageQueue 类的 Send()方法传递对象。虽然该类的类型必须满足一些特定的要求，但它们取决于格式化程序。

对于二进制格式化程序，该类必须用[Serializable]属性序列化。使用.NET 运行库的序列化功能，序列化所有字段(包括私有字段)。实现 ISerializable 接口就可以定义自定义序列化。.NET 运行库的序列化功能详见第 24 章。

XML 序列化在使用 XML 格式化程序时进行。在 XML 序列化过程中，会序列化所有公共字段和属性。使用 System.Xml.Serialization 名称空间中的属性可以影响 XML 序列化。XML 序列化详见第 34 章。

47.5.5 接收消息

要读取消息，也可以使用 MessageQueue 类。通过 Receive()方法可以读取一条消息，再将该消息从队列中删除。如果使用不同的优先级发送消息，就读取优先级最高的消息。读取优先级相同的消息时，第一条发送的消息不一定是第一条读取的消息，因为消息在网络中的传递顺序无法保证。要保证发送顺序和读取顺序相同，可以使用事务消息队列。

在下面的例子中，要从私有队列 MyPrivateQueue 中读取一条消息。之前把一个简单的字符串传递给该消息。在使用 XmlMessageFormatter 格式化程序读取消息时，必须把要读取的对象的类型传递给该格式化程序的构造函数。在本例中，将 System.String 类型传递给 XmlMessageFormatter 的构造函数的参数数组。这个构造函数可以接收一个 String 数组，该数组包含要作为字符串传递的类型；也可以接收一个 Type 数组。

用 Receive()方法读取消息，再把消息正文写入控制台中(代码文件 WorkingWithQueues/SendMessage/Program.cs)：

```
using System;
```

```
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
      queue.Formatter = new XmlMessageFormatter(
          new string[] {"System.String"});

      Message message = queue.Receive();
      Console.WriteLine(message.Body);
    }
  }
}
```

Receive()方法将同步执行，如果队列中没有消息，它就会等待队列中有消息时再执行。

1. 枚举消息

除了使用Receive()方法逐条消息地读取之外，还可以使用枚举器遍历所有消息。因为MessageQueue类实现IEnumerable接口，所以可以在foreach语句中使用。使用迭代器时，虽然消息不会从队列中删除，但可以查看消息，从而获得它们的内容：

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
      new string[] {"System.String"});

foreach (Message message in queue)
{
   Console.WriteLine(message.Body);
}
```

除了使用IEnumerable接口外，还可以使用MessageEnumerator类。虽然MessageEnumerator类实现IEnumerator接口，但它有更多功能。实现IEnumerable接口，就表示不从队列中删除消息。MessageEnumerator类的RemoveCurrent()方法可以从枚举器的当前光标位置删除消息。

在下面的例子中，使用MessageQueue类的GetMessageEnumerator()方法访问MessageEnumerator类。通过MessageEnumerator类的MoveNext()方法，可以逐条查看消息。MoveNext()方法重载为允许把一个时间段作为参数。这是使用这个枚举器的一个主要优点。现在，线程可以在指定的时间段内等待消息到达队列，之后就不等待了。IEnumerator接口定义的Current属性返回消息的一个引用：

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
      new string[] {"System.String"});

using (MessageEnumerator messages = queue.GetMessageEnumerator())
{
  while (messages.MoveNext(TimeSpan.FromMinutes(30)))
  {
    Message message = messages.Current;
```

```
        Console.WriteLine(message.Body);
    }
}
```

2. 异步读取

MessageQueue 类的 Receive()方法会等到队列中的消息可以读取为止。为了避免阻碍线程的执行，可以在 Receive()方法的一个重载版本中指定一个超时期限。要在超时后读取队列中的消息，必须再次调用 Receive()方法。除了轮询消息外，还可以调用 BeginReceive()异步方法。在使用 BeginReceive()开始异步读取消息之前，应设置 ReceiveCompleted 事件。ReceiveCompleted 事件需要 ReceiveCompletedEventHandler 委托，在消息到达队列并可以读取时，该委托会引用要调用的方法。在下面的例子中，把 MessageArrived()方法传递给 ReceivedCompletedEventHandler 委托(代码文件 WorkingWithQueues/ReceiveMessageAsync/Program.cs)：

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
      new string[] {"System.String"});

queue.ReceiveCompleted += MessageArrived;
queue.BeginReceive();
// thread does not wait
```

MessageArrived()处理程序方法需要两个参数。第一个参数是 MessageQueue 事件源。第二个参数是 ReceiveCompletedEventArgs 类型，它包含消息和异步结果。在下面的例子中，调用队列中的 EndReceive()方法，以获得异步方法的结果，即消息：

```
public static void MessageArrived(object source, ReceiveCompletedEventArgs e)
{
  MessageQueue queue = (MessageQueue)source;
  Message message = queue.EndReceive(e.AsyncResult);
  Console.WriteLine(message.Body);
}
```

如果不应从队列中删除消息，BeginPeek()和 EndPeek()方法就可以与异步 I/O 一起使用。

47.6 课程订单应用程序

为了演示消息队列的用法，本节将创建一个示例解决方案，用于订购课程。示例解决方案由 3 个程序集组成：

- 组件库(CourseOrder)，它包含在队列中发送和接收的消息的实体类
- WPF 应用程序(CourseOrderSender)，它给消息队列发送消息
- WPF 应用程序(CourseOrderReceiver)，它从消息队列中接收消息

47.6.1 课程订单类库

发送和接收应用程序都需要订单信息。所以，把实体类放在一个单独的程序集中。CourseOrder

程序集包含3个实体类：CourseOrder、Course和Customer，还有一个基类BindableBase。在示例应用程序中，并不像实际应用程序中的实现方式那样实现所有属性，而只实现用来揭示概念的属性。

在BindableBase.cs文件中，定义了BindableBase类。这个类实现了INotifyPropertyChanged接口，提供了SetProperty方法，该方法在派生类的属性设置器中调用(代码文件CourseOrderApplication/CourseOrder/BindableBase.cs)：

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Wrox.ProCSharp.Messaging
{
  public abstract class BindableBase : INotifyPropertyChanged
  {
    protected void SetProperty<T>(ref T prop, T value,
        [CallerMemberName] string callerName = "")
    {
      if (!EqualityComparer<T>.Default.Equals(prop, value))
      {
        prop = value;
        OnPropertyChanged(callerName);
      }
    }

    protected virtual void OnPropertyChanged(string propertyName)
    {
      PropertyChangedEventHandler propertyChanged = PropertyChanged;
      if (propertyChanged != null)
      {
       propertyChanged(this, new PropertyChangedEventArgs(propertyName));
      }
    }

    public event PropertyChangedEventHandler PropertyChanged;
  }
}
```

在Course.cs文件中定义Course类。这个类只有一个属性，表示课程的名称(代码文件CourseOrderApplication/CourseOrder/Course)：

```
public class Course : BindableBase
{
  private string title;
  public string Title
  {
    get { return title; }
    set
    {
      SetProperty(ref title, value);
    }
  }
}
```

Customer.cs 文件包含 Customer 类，该类的属性用于表示公司名称和联系人姓名(代码文件 CourseOrderApplication/CourseOrder/Customer.cs)：

```
public class Customer : BindableBase
{
  private string company;
  public string Company
  {
    get { return company; }
    set
    {
      SetProperty(ref company, value);
    }
  }

  private string contact;
  public string Contact
  {
    get { return contact; }
    set
    {
      SetProperty(contact, value);
    }
  }
}
```

在 CourseOrder.cs 文件中的 CourseOrder 类映射订单中的一个顾客和一门课程，并确定订单是否有高优先级。这个类还定义了队列名，把队列名设置为公共队列的格式名。即使当前不能访问队列，也可以使用格式名发送消息。我们可以使用 Computer Management 插件获得格式名，来读取消息队列的 ID。如果不能访问 Active Directory 来创建公共队列，就可以轻松地修改代码来使用私有队列(代码文件 CourseOrderApplication/CourseOrder/CourseOrder.cs)。

```
public class CourseOrder : BindableBase
{
  public const string CourseOrderQueueName =
     "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13";

  private Customer customer;
  public Customer Customer
  {
    get { return customer; }
    set
    {
      SetProperty(ref customer, value);
    }
  }

  private Course course;
  public Course Course
  {
    get { return course; }
    set
    {
```

```
        SetProperty(ref course, value);
      }
    }
  }
```

47.6.2　课程订单消息发送程序

解决方案的第二部分是一个 Windows 应用程序 CourseOrderSender。在这个应用程序中，把课程订单发送给消息队列。必须引用 System.Messaging 和 CourseOrder 程序集。

这个应用程序的用户界面如图 47-10 所示。一个组合框允许选择一个可用的课程，一些文本框控件允许用户输入一些文本，之后提交订单。

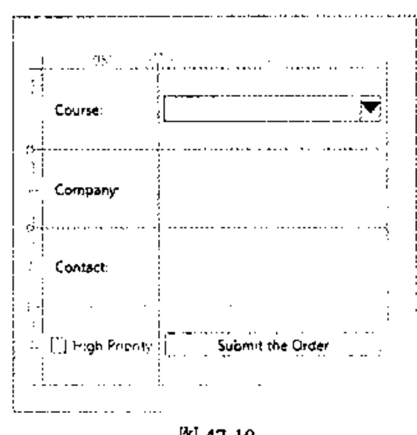


图 47-10

XAML 代码使用 WPF 数据绑定，如下所示。ComboBox 绑定到 Courses 属性上，该属性返回一组可用的课程(代码文件 CourseOrderApplication/CourseOrderSender/ CourseOrderWindow.xaml)。

```
<CheckBox Grid.Row="3" Grid.Column="0"
    IsChecked="{Binding MessageConfiguration.HighPriority,
      Mode=OneWayToSource}">
  High Priority</CheckBox>
<ComboBox ItemsSource="{Binding Courses}"Grid.Row="0" Grid.Column="1"
  SelectedItem="{Binding CourseOrder.Course.Title, Mode=OneWayToSource}"/>
<TextBox Text="{Binding CourseOrder.Customer.Company}"Grid.Row="1"
  Grid.Column="1" />
<TextBox Text="{Binding CourseOrder.Customer.Contact}"Grid.Row="2"
  Grid.Column="1" />
<Button Click="buttonSubmit_Click" Grid.Row="3" Grid.Column="1">
  Submit the Order</Button>
```

在代码隐藏文件中绑定的属性如下所示。Courses 属性只返回一个字符串集合，其中包含了可用的课程。CourseOrder 类型的 CourseOrder 属性接收来自用户的输入数据，前面所示的 CourseOrder 类创建了课程订单类库(代码文件 CourseOrderApplication/CourseOrderSender/CourseOrderWindow.xaml.xs):

```
public partial class CourseOrderWindow : Window
{
  private readonly ObservableCollection<string> courseList =
      new ObservableCollection<string>();
  private readonly CourseOrder courseOrder = new CourseOrder();
  private readonly MessageConfiguration messageConfiguration =
```

```
        new MessageConfiguration();

    public CourseOrderWindow()
    {
      InitializeComponent();
      FillCourses();
      this.DataContext = this;
    }

    public IEnumerable<string> Courses
    {
      get
      {
        return courseList;
      }
    }

    private void FillCourses()
    {
      courseList.Add("Parallel .NET Programming");
      courseList.Add("Data Access with the ADO.NET Entity Framework");
      courseList.Add("Distributed Solutions with WCF");
      courseList.Add("Windows 8 Metro Apps with XAML and C#");
    }

    public CourseOrder CourseOrder
    {
      get
      {
        return courseOrder;
      }
    }
    public MessageConfiguration MessageConfiguration
    {
      get
      {
        return messageConfiguration;
      }
    }
```

单击 Submit the Order 按钮，就调用 buttonSubmit_Click()处理程序方法。在这个方法中，创建一个 MessageQueue 实例，以便用格式名打开一个消息队列。在 Send 方法中，传递了 courseOrder 对象，将消息写入队列(代码文件 CourseOrder Application/CourseOrderSender/CourseOrderWindow.xaml.cs):

```
  private void buttonSubmit_Click(object sender, RoutedEventArgs e)
  {
    try
    {
      using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
      {
        queue.Send(courseOrder, String.Format("Course Order {{{0}}}",
```

```
            courseOrder.Customer.Company));
      }

      MessageBox.Show("Course Order submitted", "Course Order",
          MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
      MessageBox.Show(ex.Message, "Course Order Error",
          MessageBoxButton.OK, MessageBoxImage.Error);
    }
  }
```

47.6.3 发送优先级和可恢复的消息

通过设置 Message 类的 Priority 属性，就可以给消息指定优先级。如果消息是特别配置的，就必须创建一个 Message 对象，其中消息的正文在构造函数中传递。

在这个例子中，根据用户用 MessageConfiguration.HighPriority 属性(该属性绑定到一个复选框设置上)进行的选择，将优先级设置为 MessagePriority.High 或 MessagePriority.Normal。MessagePriority 是一个枚举，可以把值设置为从 Lowest (0)到 Highest (7)，默认值 Normal 的优先级是 3。

为了使消息可以恢复，应把 Recoverable 属性设置为 true：

```
  private void buttonSubmit_Click(object sender, RoutedEventArgs e)
  {
    try
    {
      using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
      using (var message = new Message(courseOrder)
        {
          Recoverable = true,
          Priority = MessageConfiguration.HighPriority == true ?
              MessagePriority.High : MessagePriority.Normal
        })
      {
        queue.Send(message, String.Format("Course Order {{{0}}}",
            courseOrder.Customer.Company));
      }

      MessageBox.Show("Course Order submitted", "Course Order",
          MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
      MessageBox.Show(ex.Message, "Course Order Error",
          MessageBoxButton.OK, MessageBoxImage.Error);
    }
  }
```

运行应用程序，就可以把课程订单添加到消息队列中，如图 47-11 所示。

图 47-11

47.6.4 课程订单消息接收应用程序

课程订单接收应用程序从队列中读取消息，其设计视图如图 47-12 所示。这个应用程序在 listOrders 列表框中显示每个订单的标签。选中一个订单后，订单的内容就会显示在应用程序右边的控件中。

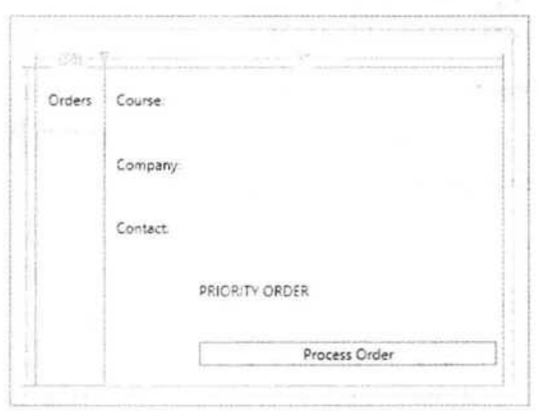


图 47-12

与以前的 WPF 应用程序一样，接收应用程序也使用了数据绑定。这里把 ListBox 绑定到 OrdersList 属性上，该属性返回所有的订单。一个 Grid 包含了用于列表框中所选订单的控件，该网格绑定到 SelectedCourseInfo 属性上。SelectedCourseInfo 属性的类型是 CourseOrderInfo。创建这个类是为了定义这个网格的子控件需要的信息。CourseOrderInfo 实现了属性 Course、Company、Contact 等(代码文件 CourseOrderApplication/CourseOrderReceiver/CourseOrderReceiverWindow.xaml)：

```
<Grid Grid.Column="0">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="4*" />
  </Grid.RowDefinitions>
  <Label Grid.Row="0" Content="Orders"/>
  <ListBox x:Name="listOrders" Grid.Row="1" ItemsSource="{Binding OrdersList}"
      SelectionChanged="listOrders_SelectionChanged"/>
</Grid>
<GridSplitter Grid.Column="1" HorizontalAlignment="Left" Width="3" />
<Grid Grid.Column="1" IsEnabled="True"
```

```
    DataContext="{Binding SelectedCourseInfo}">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Grid.Row="0" Grid.Column="0" Content="Course:"/>
  <Label Grid.Row="1" Grid.Column="0" Content="Company:"/>
  <Label Grid.Row="2" Grid.Column="0" Content="Contact:"/>
  <TextBlock Text="{Binding Course}"Grid.Row="0" Grid.Column="1" />
  <TextBlock Text="{Binding Company}"Grid.Row="1" Grid.Column="1" />
  <TextBlock Text="{Binding Contact}"Grid.Row="2" Grid.Column="1" />
  <TextBlock Grid.Row="3" Grid.Column="1" Text="PRIORITY ORDER"
      Visibility="{Binding HighPriority}"/>
  <Button Grid.Row="4" Grid.Column="1" Content="Process Order"
      IsEnabled="{Binding EnableProcessing}"
      Click="buttonProcessOrder_Click"/>
```

CourseOrderInfo 类的代码如下所示。这个类派生自前面的 BindableBase 基类，以便实现 INotifyPropertyChanged 接口，定义 XAML 中数据绑定所需的所有属性(代码文件 CourseOrderApplication/CourseOrderReceiver/CourseOrderInfo.cs)：

```
using System.Windows;

namespace Wrox.ProCSharp.Messaging
{
  public class CourseOrderInfo : BindableBase
  {
    public CourseOrderInfo()
    {
      Clear();
    }

    private MessageInfo messageInfo;
    public MessageInfo MessageInfo
    {
      get { return messageInfo; }
      set
      {
        SetProperty(ref messageInfo, value);
      }
    }

    private string course;
    public string Course
    {
      get { return course; }
      set
```

```
    {
      SetProperty(ref course, value);
    }
  }

  private string company;
  public string Company
  {
    get { return company; }
    set
    {
      SetProperty(ref company, value);
    }
  }

  private string contact;
  public string Contact
  {
    get { return contact; }
    set
    {
      SetProperty(ref contact, value);
    }
  }

  private bool enableProcessing;
  public bool EnableProcessing
  {
    get
    {
      return enableProcessing;
    }
    set
    {
      SetProperty(ref enableProcessing, value);
    }
  }

  private Visibility highPriority;
  public Visibility HighPriority
  {
    get
    {
      return highPriority;
    }
    set
    {
      SetProperty(ref highPriority, value);
    }
  }

  public void Clear()
  {
    Course = string.Empty;
    Company = string.Empty;
```

```
            Contact = string.Empty;
            EnableProcessing = false;
            HighPriority = Visibility.Hidden;
        }
    }
}
```

在窗体类 CourseOrderReceiverWindow 的构造函数中，创建一个 MessageQueue 对象，它引用由发送应用程序使用的队列。要读取消息，应使用 Formatter 属性把 XmlMessageFormatter、读取的类型和队列关联起来。

要在列表中显示可用的消息，应创建一个新任务，该任务在后台中读取消息。该任务的主方法是 PeekMessages()(代码文件 CourseOrderApplication/CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs)。

任务的更多内容详见第 21 章。

```
using System;
using System.Collections.ObjectModel;
using System.Messaging;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Threading;

namespace Wrox.ProCSharp.Messaging
{
  public partial class CourseOrderReceiverWindow : Window
  {
    private MessageQueue ordersQueue;
    private ObservableCollection<MessageInfo> ordersList =
      new ObservableCollection<MessageInfo>();
  private object syncOrdersList = new object();

  public ObservableCollection<MessageInfo> OrdersList
  {
    get
    {
      return ordersList;
    }
  }

  protected override void OnClosed(EventArgs e)
  {
    base.OnClosed(e);
    if (ordersQueue != null)
      ordersQueue.Dispose();
  }

  public CourseOrderReceiverWindow()
```

```
{
  InitializeComponent();
  this.DataContext = this;
  BindingOperations.EnableCollectionSynchronization(ordersList,
      syncOrdersList);

  ordersQueue = new MessageQueue(CourseOrder.CourseOrderQueueName);

  ordersQueue.Formatter = new XmlMessageFormatter(
    new Type[]
      {
        typeof(CourseOrder),
        typeof(Customer),
        typeof(Course)
      });

  // start the task that fills the ListBox with orders
  Task.Factory.StartNew(PeekMessages);
}
```

任务的主方法 PeekMessages()使用消息队列的枚举器显示所有消息。在 while 循环中，MessageEnumerator 检查队列中是否还有新消息。如果队列中没有消息，任务就等待下一条消息进入队列中，等待 3 个小时后它才会退出。

```
private void PeekMessages()
{
  try
  {
    using (MessageEnumerator messagesEnumerator =
        ordersQueue.GetMessageEnumerator2())
    {
      while (messagesEnumerator.MoveNext(TimeSpan.FromHours(3)))
      {
        var messageInfo = new MessageInfo
        {
          Id = messagesEnumerator.Current.Id,
          Label = messagesEnumerator.Current.Label
        };

        ordersList.Add(messageInfo);
      }
    }
    MessageBox.Show("No orders in the last 3 hours. Exiting thread",
          "Course Order Receiver", MessageBoxButton.OK,
          MessageBoxImage.Information);
  }
  catch (MessageQueueException ex)
  {
    MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
        MessageBoxImage.Error);
  }
}
```

ListBox 控件包含 LabelIdMapping 类的元素。虽然这个类用于在列表框中显示消息的标签，但这里它隐藏消息的 ID。消息的 ID 可以用于以后读取消息(代码文件 CourseOrderApplication/CourseOrderReceiver/MessageInfo.cs)：

```
private class MessageInfo
{
   public string Label { get; set; }
   public string Id { get; set; }

   public override string ToString()
   {
      return Label;
   }
}
```

ListBox 控件的 SelectedIndexChanged 事件与 OnOrders_SelectionChanged()方法相关联。这个方法从当前选中的项中获得 LabelIdMapping 对象，并使用 ID 通过 PeekById()方法再次查看消息。接着在 TextBox 控件中显示消息的内容。因为默认情况下不读取消息的优先级，所以 MessageReadPropertyFilter 属性必须设置为接收 Priority(代码文件 CourseOrderApplication/CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs)：

```
private void listOrders_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
  var messageInfo = (sender as ListBox).SelectedItem as MessageInfo;
  if (messageInfo == null)
    return;

  ordersQueue.MessageReadPropertyFilter.Priority = true;
  Message message = ordersQueue.PeekById(messageInfo.Id);

  var order = message.Body as CourseOrder;
  if (order != null)
  {
    selectedCourseInfo.MessageInfo = messageInfo;
    selectedCourseInfo.Course = order.Course.Title;
    selectedCourseInfo.Company = order.Customer.Company;
    selectedCourseInfo.Contact = order.Customer.Contact;
    selectedCourseInfo.EnableProcessing = true;

    if (message.Priority > MessagePriority.Normal)
    {
      selectedCourseInfo.HighPriority = Visibility.Visible;
    }
    else
    {
      selectedCourseInfo.HighPriority = Visibility.Hidden;
    }
  }
  else
  {
    MessageBox.Show("The selected item is not a course order",
```

```
            "Course Order Receiver", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
```

单击 Process Order 按钮时，会调用 OnProcessOrder()处理程序方法。这里会再次引用列表框中当前选择的消息，并调用 ReceiveById()方法从队列中删除该消息：

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
  Message message = ordersQueue.ReceiveById(
     SelectedCourseInfo.MessageInfo.Id);

  ordersList.Remove(SelectedCourseInfo.MessageInfo);

  listOrders.SelectedIndex = -1;
  selectedCourseInfo.Clear();

  MessageBox.Show("Course order processed", "Course Order Receiver",
     MessageBoxButton.OK, MessageBoxImage.Information);
}
```

图 47-13 显示了正在运行的接收程序，该程序列出了队列中的 4 个订单，且当前选中了一个订单。

图 47-13

47.7 接收结果

在示例应用程序的当前版本中，发送应用程序并不知道消息是否已处理。为了得到接收程序的结果，可以使用确认队列或响应队列。

47.7.1 确认队列

使用确认队列，发送应用程序可以获得消息的状态信息。在确认消息中，可以定义是否要接收应答，来判断是一切正常，还是某些地方出错。例如，可以在消息到达目标队列或读取消息时，发送确认消息；或在指定的超时过后，消息未到达目标队列或未读取消息时，发送确认消息。

在下面的例子中，将 Message 类的 AdministrationQueue 设置为 CourseOrderAck 队列，这个队列的创建方式类似于一般队列，但它以另外一种方式使用：原始发送程序接收到确认消息时使用它。

把 AcknowledgementType 属性设置为 AcknowledgementTypes.FullReceive，会在读取消息时获得一条确认消息：

```
var message = new Message(order);

message.AdministrationQueue = new MessageQueue(@".\CourseOrderAck");
message.AcknowledgeType = AcknowledgeTypes.FullReceive;

queue.Send(message, String.Format("Course Order {{0}}",
      order.Customer.Company);

string id = message.Id;
```

关联 ID 用于确定什么确认消息属于发送的哪条消息。发送的每条消息都有一个 ID，响应该消息所发送的确认消息把源消息的 ID 作为其关联 ID。可以用 MessageQueue.ReceiveByCorrelationId()方法读取确认队列中的消息，以接收相关的确认消息。

除了使用确认消息之外，还可以为没有到达其目的地的消息使用死信队列。把 Message 类的属性 UseDeadLetterQueue 设置为 true，如果消息在超时过后没有到达目标队列，该消息就会复制到死信队列中。

超时用 Message 类的 TimeToReachQueue 和 TimeToBeReceived 属性设置。

47.7.2 响应队列

如果需要从接收程序中获得比确认消息更多的信息，就可以使用响应队列。响应队列类似于一般队列，但原始发送程序把该队列用作接收程序，原始接收程序把响应队列用作发送程序。

发送程序必须用 Message 类的 ResponseQueue 属性指定响应队列。下面的示例代码揭示了接收程序如何使用响应队列返回一条响应消息。在响应消息 responseMessage 中，把 CorrelationId 属性设置为原始消息的 ID。这样，客户端应用程序就知道该应答属于哪条消息。这类似于确认队列。响应消息用 MessageQueue 对象的 Send()方法发送，MessageQueue 对象从 ResponseQueue 属性中返回：

```
public void ReceiveMessage(Message message)
{
  var responseMessage = new Message("response")
  {
    CorrelationId = message.Id
  }

  message.ResponseQueue.Send(responseMessage);
}
```

47.8 事务队列

对于可恢复的消息，不能保证消息的到达顺序，也不能保证消息只到达一次。网络失败可能会使消息到达多次，如果发送程序和接收程序安装了供消息队列使用的多个网络协议，也会发生这种情况。

在需要确保如下条件的情况下，可以使用事务队列：

- 消息的到达顺序与其发送顺序相同。
- 消息只到达一次。

对于事务队列，一个事务不能横跨消息的发送和接收过程。消息队列的本质是发送和接收的时间间隔可能非常长。而事务应该很短。在消息队列中，第一个事务用于把消息发送到队列中，第二个事务用于把消息转发到网络上，第三个事务用于接收消息。

下一个例子揭示了如何创建事务消息队列，如何使用事务发送消息。

给 MessageQueue.Create()方法的第二个参数传递 true，就会创建事务消息队列。

如果要在一个事务中把多条消息写入队列中，就必须实例化 MessageQueueTransaction 对象，并调用 Begin()方法。在发送完属于该事务的所有消息后，必须调用 MessageQueueTransaction 对象的 Commit()方法。要取消一个事务(且不把消息写入队列中)，就必须在 catch 块中调用 Abort()方法，如下所示：

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
  class Program
  {
    static void Main()
    {
      if (!MessageQueue.Exists(@".\MyTransactionalQueue"))
      {
        MessageQueue.Create(@".\MyTransactionalQueue", true);
      }

      var queue = new MessageQueue(@".\MyTransactionalQueue");
      var transaction = new MessageQueueTransaction();

      try
      {
        transaction.Begin();
        queue.Send("a", transaction);
        queue.Send("b", transaction);
        queue.Send("c", transaction);
        transaction.Commit();
      }
      catch
      {
        transaction.Abort();
      }
    }
  }
}
```

47.9 消息队列和 WCF

第 43 章介绍了 WCF 的体系结构和核心功能。使用 WCF 可以配置一个使用 Windows Message Queuing 体系结构的消息队列绑定。WCF 通过它为消息队列提供了一个抽象层。图 47-14 用一幅简单的图片解释了该体系结构。客户端应用程序调用 WCF 代理的一个方法，来给队列发送一条消息。该消息由代理创建。客户端开发人员不需要知道消息发送给了队列，只调用代理的方法即可。代理抽象了处理 System.Messaging 名称空间中的类的过程，并给队列发送一个消息。服务端的 MSMQ 侦听器信道从队列中读取消息，把它们转换为方法调用，再用服务调用这些方法调用。

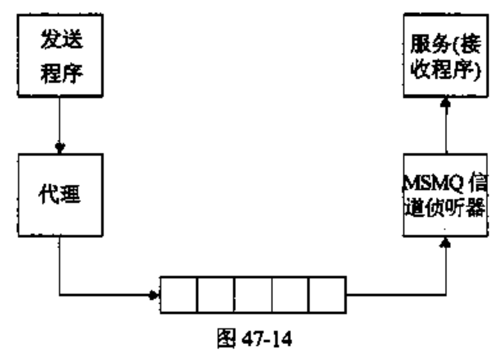


图 47-14

接着，从 WCF 的角度来看，把课程预订应用程序转换为使用消息队列。在这个解决方案中，修改前面的 3 个项目，再添加一个程序集，其中包含 WCF 服务的协定。

- 组件库(CourseOrder)包含在网络上发送消息的实体类。这些实体类改为满足数据协定的要求，来与 WCF 进行序列化。
- 添加一个新库(CourseOrderService)，它定义了服务提供的协定。
- 修改 WPF 发送应用程序(CourseOrderSender)，不发送消息，而调用 WCF 代理的方法。
- 修改 WPF 接收应用程序(CourseOrderReceiver)，使用实现了协定的 WCF 服务。

47.9.1 带数据协定的实体类

在 CourseOrder 库中，修改 Course、Customer 和 CourseOrder 类，通过属性[DataContract]及[DataMember]来应用数据协定。要使用这些属性，必须引用 System.Runtime.Serialization 程序集，并导入 System.Runtime.Serialization 名称空间(代码文件 CourseOrderApplicationWCF/CourseOrder/Course.cs)。

```
using System.Runtime.Serialization;

namespace Wrox.ProCSharp.Messaging
{
  [DataContract]
  public class Course
  {
    [DataMember]
    public string Title { get; set; }
  }
}
```

Customer 类还需要数据协定属性(代码文件 CourseOrderApplicationWCF/CourseOrder/Customer.cs)：

```
[DataContract]
public class Customer
{
  [DataMember]
  public string Company { get; set; }

  [DataMember]
  public string Contact { get; set; }
}
```

对于 CourseOrder 类，不仅要添加数据协定属性，还要添加 ToString()方法的一个重写版本，以包含这些对象的默认字符串表示(代码文件 CourseOrderApplicationWCF/CourseOrder/CourseOrder.cs)：

```
[DataContract]
public class CourseOrder
{
  [DataMember]
  public Customer Customer { get; set; }

  [DataMember]
  public Course Course { get; set; }

  public override string ToString()
  {
    return String.Format("Course Order {{{0}}}", Customer.Company);
  }
}
```

47.9.2 WCF 服务协定

为了给服务提供 WCF 服务协定，需要添加一个 WCF 服务库 CourseOrderServiceContract。这个协定由 ICourseOrderService 接口定义。该协定需要[ServiceContract]特性。如果希望仅把这个接口用于消息队列，就可以应用[DeliveryRequirements]特性，并指定 QueuedDeliveryRequirements 属性。QueuedDeliveryRequirementsMode 枚举的值有 Required、Allowed 和 NotAllowed。AddCourseOrder()方法由服务提供。消息队列使用的方法只能有输入参数。因为发送程序和接收程序都可以彼此独立地运行，所以发送程序不能期望立即得到结果。使用[OperationContract]属性设置 IsOneWay 特性。这个操作的调用者不等待服务的应答(代码文件 CourseOrderApplicationWCF/CourseOrderServiceContract/ICourseOrderService.cs)：

```
using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
  [ServiceContract]
  [DeliveryRequirements(
     QueuedDeliveryRequirements=QueuedDeliveryRequirementsMode.Required)]
  public interface ICourseOrderService
  {
    [OperationContract(IsOneWay = true)]
```

```
        void AddCourseOrder(CourseOrder courseOrder);
    }
}
```

可以使用确认队列和响应队列给客户提供应答。

47.9.3 WCF 消息接收应用程序

WPF 应用程序 CourseOrderReceiver 现在修改为实现 WCF 服务，并接收消息。它需要引用 System.ServiceModel 程序集和 WCF 协定程序集 CourseOrderServiceContract。

CourseOrderService 类实现 ICourseOrderService 接口。在实现代码中，触发 CourseOrderAdded 事件。WPF 应用程序注册这个事件，来接收 CourseOrder 对象。

因为把 WPF 控件绑定到单个线程上，所以 UseSynchronizationContext 属性用[ServiceBehavior]特性设置。这是 WCF 运行库的一个功能，可以把方法调用传递给 WPF 应用程序的同步上下文定义的线程(代码文件 CourseOrderApplicationWCF/CourseOrderReceiver/CourseOrderService.cs)。

```
using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
  [ServiceBehavior(UseSynchronizationContext=true)]

  public class CourseOrderService: ICourseOrderService
  {
    public static event EventHandler<CourseOrderEventArgs>
        CourseOrderAdded;

    public void AddCourseOrder(CourseOrder courseOrder)
    {
      var courseOrderAdded = CourseOrderAdded;
      if (courseOrderAdded != null)
      {
        courseOrderAdded(this, new CourseOrderEventArgs(courseOrder));
      }
    }
  }

  public class CourseOrderEventArgs : EventArgs
  {
    public CourseOrderEventArgs(CourseOrder courseOrder)
    {
      this.CourseOrder = courseOrder;
    }
    public CourseOrder CourseOrder { get; private set; }
  }
}
```

第 21 章介绍了同步上下文。

在 CourseReceiverWindow 类的构造函数中，实例化一个 ServiceHost 对象，并打开它，以启动侦听器。侦听器的绑定在应用程序配置文件中完成。

在构造函数中，订阅 CourseOrderReceiver 类的 CourseOrderAdded 事件。因为这里只需要把接收到的 CourseOrder 对象添加到集合中，所以使用了一个简单的 lambda 表达式。

第 8 章介绍了 lambda 表达式。

这里使用的集合类是 System.Collections.ObjectModel 名称空间中的 ObservableCollection<T>。由于这个集合类实现 INotifyCollectionChanged 接口，因此绑定到集合上的 WPF 控件知道列表的动态变化(代码文件 CourseOrderApplicationWCF/CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs)。

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.Windows;

namespace Wrox.ProCSharp.Messaging
{
  public partial class CourseOrderReceiverWindow : Window
  {
    private ObservableCollection<CourseOrder> courseOrders =
        new ObservableCollection<CourseOrder>();

    public CourseOrderReceiverWindow()
    {
      InitializeComponent();
      this.DataContext = courseOrders;
      CourseOrderService.CourseOrderAdded += (sender, e) =>
        {
          courseOrders.Add(e.CourseOrder);
          buttonProcessOrder.IsEnabled = true;
        };

      var host = new ServiceHost(typeof(CourseOrderService));
      try
      {
        host.Open();
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
```

XAML 代码中 WPF 元素的数据绑定现在改为使用新集合。把 ListBox 绑定到数据上下文中，把单项控件绑定到数据上下文的当前项的属性上(代码文件 CourseOrderApplicationWCF/CourseOrderReceiver/CourseOrderReceiverWindow.xaml)。

```
<ListBox x:Name="listOrders" Grid.Row="1" ItemsSource="{Binding}"
   IsSynchronizedWithCurrentItem="True"/>
<!-- ... -->

<TextBlock Text="{Binding Course.Title}" Grid.Row="0" Grid.Column="1" />
<TextBlock Text="{Binding Customer.Company}" Grid.Row="1" Grid.Column="1" />
<TextBlock Text="{Binding Customer.Contact}" Grid.Row="2" Grid.Column="1" />
```

应用程序配置文件定义 netMsmqBinding。为了可靠地传递消息，需要事务队列。要接收和发送非事务队列中的消息，必须把 exactlyOnce 属性设置为 false(配置文件 CourseOrderApplicationWCF/CourseOrderReceiver/app.config)。

```
<?xml version="1.0"?>
<configuration>
  <startup>
   <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
  </startup>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="NonTransactionalQueueBinding" exactlyOnce="false">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <services>
      <service name="Wrox.ProCSharp.Messaging.CourseOrderService">
        <endpoint address="net.msmq://localhost/private/courseorder"
          binding="netMsmqBinding"
          bindingConfiguration="NonTransactionalQueueBinding"
          name="OrderQueueEP"
          contract="Wrox.ProCSharp.Messaging.ICourseOrderService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

buttonProcessOrder 按钮的 Click 事件处理程序从集合类中删除选中的课程订单(代码文件 CourseOrderApplicationWCF/CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs)：

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
  var courseOrder = listOrders.SelectedItem as CourseOrder;
  courseOrders.Remove(courseOrder);
  listOrders.SelectedIndex = -1;
  buttonProcessOrder.IsEnabled = false;

  MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);

}
```

47.9.4 WCF 消息发送应用程序

把发送应用程序修改为使用 WCF 代理类。对于服务协定，需要引用 CourseOrderServiceContract 程序集，而要使用 WCF 类，需要引用 System.ServiceModel 程序集。

在 buttonSubmit 控件的 Click 事件处理程序中，ChannelFactory 类返回一个代理。通过调用 AddCourseOrder()方法，该代理给队列发送一条消息(代码文件 CourseOrderApplicationWCF/CourseOrderSender/CourseOrderWindow.xaml.cs：

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
  try
  {
    var factory = new ChannelFactory<ICourseOrderService>("queueEndpoint");
    ICourseOrderService proxy = factory.CreateChannel();
    proxy.AddCourseOrder(CourseOrder);
    factory.Close();

    MessageBox.Show("Course Order submitted", "Course Order",
          MessageBoxButton.OK, MessageBoxImage.Information);
  }
  catch (MessageQueueException ex)
  {
    MessageBox.Show(ex.Message, "Course Order Error",
          MessageBoxButton.OK, MessageBoxImage.Error);
  }
}
```

应用程序配置文件定义 WCF 连接的客户端部分。这里也使用 netMsmqBinding(配置文件 CourseOrderApplicationWCF/CourseOrderSender/app.config)：

```
<?xml version="1.0"?>
<configuration>
  <startup>
   <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
  </startup>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="nonTransactionalQueueBinding" exactlyOnce="false">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <client>
      <endpoint address="net.msmq://localhost/private/courseorder"
        binding="netMsmqBinding"
        bindingConfiguration="nonTransactionalQueueBinding"
        contract="Wrox.ProCSharp.Messaging.ICourseOrderService"
        name="queueEndpoint"
        kind="" endpointConfiguration="" />
    </client>
  </system.serviceModel>
</configuration>
```

现在启动应用程序，它的工作方式与以前类似。但不再需要使用 System.Messaging 名称空间中的类发送和接收消息。而是使用 TCP 或 HTTP 信道和 WCF 以类似方式编写应用程序。

但是，要创建消息队列和删除消息，仍需要 MessageQueue 类。WCF 只是发送和接收消息的一个抽象。

如果需要使用 System.Messaging 应用程序与 WCF 应用程序通信，就可以使用 Binding 替代 netMsmqBinding。这个绑定使用用于 COM 和 System.Messaging 的消息格式。

47.10 消息队列的安装

可以用 MessageQueue.Create()方法创建消息队列。但运行应用程序的用户通常没有创建消息队列所需的管理特权。

一般，可以通过安装程序使用 MessageQueueInstaller 类创建消息队列。如果安装程序类是应用程序的一部分，命令行实用程序 installutil.exe(或 Windows 安装软件包)就会调用安装程序的 Install()方法。

Visual Studio 专门支持在 Windows 窗体应用程序中使用 MessageQueueInstaller 类。如果把 MessageQueue 组件从工具箱拖放到窗体上，该组件的智能标记就允许添加一个安装程序及其菜单项 Add Installer。可以使用属性编辑器配置 MessageQueueInstaller 对象，以定义事务队列、日志队列、格式化程序的类型、基本优先级等。

47.11 小结

本章介绍了消息队列的用法。消息队列是一种重要的技术，它不仅提供异步通信，还提供断开连接的通信。发送程序和接收程序可以在不同的时间内运行，此时智能客户端就可以使用消息队列，消息队列还可以把负载分布到不同的时间段上，以充分利用服务器的潜能。

消息队列最重要的类是 Message 和 MessageQueue。MessageQueue 类可以发送、接收和查看消息，Message 类定义发送的内容。

WCF 提供消息队列的一个抽象。可以使用 WCF 提供的概念，调用代理的方法来发送消息，实现一个服务来接收消息。

封面
书名
版权
前言
目录
第Ⅰ部分　C＃语言
第1章　NET体系结构
1.1　C＃与．NET的关系
1.2　公共语言运行库
1.2.1　平台无关性
1.2.2　提高性能
1.2.3　语言的互操作性
1.3　中间语言
1.3.1　面向对象和接口的支持
1.3.2　不同的值类型和引用类型
1.3.3　强数据类型化
1.3.4　通过异常处理错误
1.3.5　特性的使用
1.4　程序集
1.4.1　私有程序集
1.4.2　共享程序集
1.4.3　反射
1.4.4　并行编程
1.4.5　异步编程
1.5　.NET Framework类
1.6　名称空间
1.7　用C＃创建．NET应用程序
1.7.1　创建ASP.NET应用程序
1.7.2　使用WPF
1.7.3　windowsStore应用程序
1.7.4　Windows服务
1.7.5　WCF
1.7.6　Windows WF
1.8　C＃在．NET企业体系结构中的作用
1.9　小结
第2章　核心C＃
2.1　C＃基础
2.2　第一个C＃程序
2.2.1　代码
2.2.2　编译并运行程序
2.2.3　详细介绍
2.3　变量
2.3.1　变量的初始化
2.3.2　类型推断
2.3.3　变量的作用域

2.3.4　常量
2.4　预定义数据类型
2.4.1　值类型和引用类型
2.4.2　CTS类型
2.4.3　预定义的值类型
2.4.4　预定义的引用类型
2.5　流控制
2.5.1　条件语句
2.5.2　循环
2.5.3　跳转语句
2.6　枚举
2.7　名称空间
2.7.1　using语句
2.7.2　名称空间的别名
2.8　Main（）方法
2.8.1　多个Main（）方法
2.8.2　给Main（）方法传递参数
2.9　有关编译C＃文件的更多内容
2.10　控制台I/O
2.11　使用注释
2.11.1　源文件中的内部注释
2.11.2　XML文档
2.12　C＃预处理器指令
2.12.1　#define和＃undef
2.12.2　#if、#elif、#else和＃endif
2.12.3　＃warning和＃error
2.12.4　＃region和＃endregion
2.12.5　#line
2.12.6　＃pragma
2.13　C＃编程规则
2.13.1　关于标识符的规则
2.13.2　用法约定
2.14　小结
第3章　对象和类型
3.1　创建及使用类
3.2　类和结构
3.3　类
3.3.1　数据成员
3.3.2　函数成员
3.3.3　只读字段
3.4　匿名类型
3.5　结构
3.5.1　结构是值类型
3.5.2　结构和继承
3.5.3　结构的构造函数

3.6　弱引用
3.7　部分类
3.8　静态类
3.9　Object类
3.9.1　System.Objcct（）方法
3.9.2　ToString（）方法
3.10　扩展方法
3.11　小结
第4章　继承
4.1　继承
4.2　继承的类型
4.2.1　实现继承和接口继承
4.2.2　多重继承
4.2.3　结构和类
4.3　实现继承
4.3.1　虚方法
4.3.2　隐藏方法
4.3.3　调用函数的基类版本
4.3.4　抽象类和抽象函数
4.3.5　密封类和密封方法
4.3.6　派生类的构造函数
4.4　修饰符
4.4.1　可见性修饰符
4.4.2　其他修饰符
4.5　接口
4.5.1　定义和实现接口
4.5.2　派生的接口
4.6　小结
第5章　泛型
5.1　泛型概述
5.1.1　性能
5.1.2　类型安全
5.1.3　二进制代码的重用
5.1.4　代码的扩展
5.1.5　命名约定
5.2　创建泛型类
5.3　泛型类的功能
5.3.1　默认值
5.3.2　约束
5.3.3　继承
5.3.4　静态成员
5.4　泛型接口
5.4.1　协变和抗变
5.4.2　泛型接口的协变
5.4.3　泛型接口的抗变

5.5　泛型结构
5.6　泛型方法
5.6.1　泛型方法示例
5.6.2　带约束的泛型方法
5.6.3　带委托的泛型方法
5.6.4　泛型方法规范
5.7　小结
第6章　数组
6.1　同一类型和不同类型的多个对象
6.2　简单数组
6.2.1　数组的声明
6.2.2　数组的初始化
6.2.3　访问数组元素
6.2.4　使用引用类型
6.3　多维数组
6.4　锯齿数组
6.5　Array类
6.5.1　创建数组
6.5.2　复制数组
6.5.3　排序
6.6　数组作为参数
6.6.1　数组协变
6.6.2　ArraySegment<T>
6.7　枚举
6.7.1　IEnumerator接口
6.7.2　foreach语句
6.7.3　yield语句
6.8　元组
6.9　结构比较
6.10　小结
第7章　运算符和类型强制转换
7.1　运算符和类型转换
7.2　运算符
7.2.1　运算符的简化操作
7.2.2　运算符的优先级
7.3　类型的安全性
7.3.1　类型转换
7.3.2　装箱和拆箱
7.4　比较对象的相等性
7.4.1　比较引用类型的相等性
7.4.2　比较值类型的相等性
7.5　运算符重载
7.5.1　运算符的工作方式
7.5.2　运算符重载的示例：Vector结构
7.6　用户定义的类型强制转换

7.6.1　实现用户定义的类型强制转换
7.6.2　多重类型强制转换
7.7　小结
第8章　委托、lambda表达式和事件
8.1　引用方法
8.2　委托
8.2.1　声明委托
8.2.2　使用委托
8.2.3　简单的委托示例
8.2.4　Action〈T〉和Func〈T〉委托
8.2.5　BubbleSorter示例
8.2.6　多播委托
8.2.7　匿名方法
8.3　lambda表达式
8.3.1　参数
8.3.2　多行代码
8.3.3　闭包
8.3.4　使用foreach语句的闭包
8.4　事件
8.4.1　事件发布程序
8.4.2　事件侦听器
8.4.3　弱事件
8.5　小结
第9章　字符串和正则表达式
9.1　System.String类
9.1.1　创建字符串
9.1.2　StringBuilder成员
9.1.3　格式字符串
9.2　正则表达式
9.2.1　正则表达式概述
9.2.2　RegularExpressionsPlayaround示例
9.2.3　显示结果
9.2.4　匹配、组合和捕获
9.3　小结
第10章　集合
10.1　概述
10.2　集合接口和类型
10.3　列表
10.3.1　创建列表
10.3.2　只读集合
10.4　队列
10.5　栈
10.6　链表
10.7　有序列表
10.8　字典

10.8.1　键的类型
10.8.2　字典示例
10.8.3　Lookup类
10.8.4　有序字典
10.9　集
10.10　可观察的集合
10.11　位数组
10.11.1　BitArray类
10.11.2　BitVector32结构
10.12　不变的集合
10.13　并发集合
10.13.1　创建管道
10.13.2　使用BlockingCollection
10.13.3　使用ConcurrentDictionary
10.13.4　完成管道
10.14　性能
10.15　小结
第11章　LINQ
11.1　LINQ概述
11.1.1　列表和实体
11.1.2　LINQ查询
11.1.3　扩展方法
11.1.4　推迟查询的执行
11.2　标准的查询操作符
11.2.1　筛选
11.2.2　用索引筛选
11.2.3　类型筛选
11.2.4　复合的from子句
11.2.5　排序
11.2.6　分组
11.2.7　对嵌套的对象分组
11.2.8　内连接
11.2.9　左外连接
11.2.10　组连接
11.2.11　集合操作
11.2.12　合并
11.2.13　分区
11.2.14　聚合操作符
11.2.15　转换操作符
11.2.16　生成操作符
11.3　并行LINQ
11.3.1　并行查询
11.3.2　分区器
11.3.3　取消
11.4　表达式树

11.5　LINQ提供程序
11.6　小结
第12章　动态语言扩展
12.1　DLR
12.2　dynamic类型
12.3　包含DLR ScriptRuntime
12.4　DynamicObject和ExpandoObject
12.4.1　DynamicObject
12.4.2　ExpandoObject
12.5　小结
第13章　异步编程
13.1　异步编程的重要性
13.2　异步模式
13.2.1　同步调用
13.2.2　异步模式
13.2.3　基于事件的异步模式
13.2.4　基于任务的异步模式
13.3　异步编程的基础
13.3.1　创建任务
13.3.2　调用异步方法
13.3.3　延续任务
13.3.4　同步上下文
13.3.5　使用多个异步方法
13.3.6　转换异步模式
13.4　错误处理
13.4.1　异步方法的异常处理
13.4.2　多个异步方法的异常处理
13.4.3　使用AggregateException信息
13.5　取消
13.5.1　开始取消任务
13.5.2　使用框架特性取消任务
13.5.3　取消自定义任务
13.6　小结
第14章　内存管理和指针
14.1　内存管理
14.2　后台内存管理
14.2.1　值数据类型
14.2.2　引用数据类型
14.2.3　垃圾回收
14.3　释放非托管的资源
14.3.1　析构函数
14.3.2　IDisposable接口
14.3.3　实现Disposable接口和析构函数
14.4　不安全的代码
14.4.1　用指针直接访问内存

14.4.2　指针示例：PointerPlayground
14.4.3　使用指针优化性能
14.5　小结
第15章　反射
15.1　在运行期间处理和检查代码
15.2　自定义特性
15.2.1　编写自定义特性
15.2.2　自定义特性示例：WhatsNewAttributes
15.3　反射
15.3.1　System.Type类
15.3.2　TypeView示例
15.3.3　Assembly类
15.3.4　完成WhatsNewAttributes示例
15.4　小结
第16章　错误和异常
16.1　简介
16.2　异常类
16.3　捕获异常
16.3.1　实现多个catch块
16.3.2　在其他代码中捕获异常
16.3.3　System.Exception属性
16.3.4　没有处理异常时所发生的情况
16.3.5　嵌套的try块
16.4　用户定义的异常类
16.4.1　捕获用户定义的异常
16.4.2　抛出用户定义的异常
16.4.3　定义用户定义的异常类
16.5　调用者信息
16.6　小结
第Ⅱ部分　Visual Studio
第17章　Visual Studio 2013
17.1　使用Visual Studio 2013
17.1.1　项目文件的改进
17.1.2　Visual Studio的版本
17.1.3　Visual Studio设置
17.2　创建项目
17.2.1　面向多个版本的. NET Framework
17.2.2　选择项目类型
17.3　浏览并编写项目
17.3.1　Solution Explorer
17.3.2　使用代码编辑器
17.3.3　学习和理解其他窗口
17.3.4　排列窗口
17.4　构建项目
17.4.1　构建、编译和生成

17.4.2　调试版本和发布版本
17.4.3　选择配置
17.4.4　编辑配置
17.5　调试代码
17.5.1　设置断点
17.5.2　使用数据提示和调试器可视化工具
17.5.3　监视和修改变量
17.5.4　异常
17.5.5　多线程
17.5.6　IntelliTrace
17.6　重构工具
17.7　体系结构工具
17.7.1　依赖项关系图
17.7.2　层关系图
17.8　分析应用程序
17.8.1　代码地图
17.8.2　序列图
17.8.3　探查器
17.8.4　Concurrency Visualizer
17.8.5　Code Analysis
17.8.6　Code Metrics
17.9　单元测试
17.9.1　创建单元测试
17.9.2　运行单元测试
17.9.3　预期异常
17.9.4　测试全部代码路径
17.9.5　外部依赖
17.9.6　Fakes Framework
17.10　Windows Store应用程序、WCF、WF等
17.10.1　使用Visual Studio生成WCF应用程序
17.10.2　使用Visual Studio生成WF应用程序
17.10.3　使用Visual Studio 2013生成Windows Store应用程序
17.11　小结
第18章　部署
18.1　部署是应用程序生命周期的一部分
18.2　部署的规划
18.2.1　部署选项
18.2.2　部署要求
18.2.3　部署.NET运行库
18.3　传统的部署选项
18.3.1　xcopy部署
18.3.2　xcopy和Web应用程序
18.3.3　Windows Installer
18.4　ClickOnce
18.4.1　ClickOnce操作

18.4.2　发布ClickOnce应用程序
18.4.3　ClickOnce设置
18.4.4　ClickOnce文件的应用程序缓存
18.4.5　应用程序的安装
18.4.6　ClickOnce部署API
18.5　Web部署
18.5.1　Web应用程序
18.5.2　配置文件
18.5.3　创建Web Deploy包
18.6　Windows Store应用程序
18.6.1　创建应用程序包
18.6.2　Windows App Certification Kit
18.6.3　旁加载
18.6.4　Windows部署API
18.7　小结
第Ⅲ部分　基础
第19章　程序集
19.1　程序集的含义
19.1.1　程序集的功能
19.1.2　程序集的结构
19.1.3　程序集清单
19.1.4　名称空间、程序集和组件
19.1.5　私有程序集和共享程序集
19.1.6　附属程序集
19.1.7　查看程序集
19.2　构建程序集
19.2.1　创建模块和程序集
19.2.2　程序集的特性
19.2.3　创建和动态加载程序集
19.3　应用程序域
19.4　共享程序集
19.4.1　强名
19.4.2　使用强名获得完整性
19.4.3　全局程序集缓存
19.4.4　创建共享程序集
19.4.5　创建强名
19.4.6　安装共享程序集
19.4.7　使用共享程序集
19.4.8　程序集的延迟签名
19.4.9　引用
19.4.10　本机映像生成器
19.5　配置.NET应用程序
19.5.1　配置类别
19.5.2　绑定程序集
19.6　版本问题

19.6.1　版本号
19.6.2　通过编程方式获取版本
19.6.3　绑定到程序集版本
19.6.4　发行者策略文件
19.6.5　运行库的版本
19.7　在不同的技术之间共享程序集
19.7.1　共享源代码
19.7.2　可移植类库
19.8　小结
第20章　诊断
20.1　诊断概述
20.2　代码协定
20.2.1　前提条件
20.2.2　后置条件
20.2.3　不变量
20.2.4　纯粹性
20.2.5　接口的协定
20.2.6　简写
20.2.7　协定和遗留代码
20.3　跟踪
20.3.1　跟踪源
20.3.2　跟踪开关
20.3.3　跟踪侦听器
20.3.4　筛选器
20.3.5　相关性
20.3.6　使用ETW进行跟踪
20.3.7　使用EventSource
20.3.8　使用EventSource进行高级跟踪
20.4　事件日志
20.4.1　事件日志体系结构
20.4.2　事件日志类
20.4.3　创建事件源
20.4.4　写入事件日志
20.4.5　资源文件
20.5　性能监视
20.5.1　性能监视类
20.5.2　性能计数器生成器
20.5.3　添加PerformanceCounter组件
20.5.4　perfmon.exe
20.6　小结
第21章　任务、线程和同步
21.1　概述
21.2　Parallel类
21.2.1　用Parallel.For（）方法循环
21.2.2　使用Parallel.ForEach（）方法循环

21.2.3　通过Parallel.Invoke（）方法调用多个方法
21.3　任务
21.3.1　启动任务
21.3.2　Future——任务的结果
21.3.3　连续的任务
21.3.4　任务层次结构
21.4　取消架构
21.4.1　Parallel.For（）方法的取消
21.4.2　任务的取消
21.5　线程池
21.6　Thread类
21.6.1　给线程传递数据
21.6.2　后台线程
21.6.3　线程的优先级
21.6.4　控制线程
21.7　线程问题
21.7.1　争用条件
21.7.2　死锁
21.8　同步
21.8.1　lock语句和线程安全
21.8.2　Interlocked类
21.8.3　Monitor类
21.8.4　SpinLock结构
21.8.5　WaitHandle基类
21.8.6　Mutex类
21.8.7　Semaphore类
21.8.8　Events类
21.8.9　Barrier类
21.8.10　ReaderWriterLockSlim类
21.9　Timer类
21.10　数据流
21.10.1　使用动作块
21.10.2　源和目标数据块
21.10.3　连接块
21.11　小结
第22章　安全性
22.1　概述
22.2　身份验证和授权
22.2.1　标识和Principal
22.2.2　角色
22.2.3　声明基于角色的安全性
22.2.4　声称
22.2.5　客户端应用程序服务
22.3　加密
22.3.1　签名

22.3.2　交换密钥和安全传输
22.4　资源的访问控制
22.5　代码访问安全性
22.5.1　第2级安全透明性
22.5.2　权限
22.6　使用证书发布代码
22.7　小结
第23章　互操作
23.1　.NET和COM技术
23.1.1　元数据
23.1.2　释放内存
23.1.3　接口
23.1.4　方法绑定
23.1.5　数据类型
23.1.6　注册
23.1.7　线程
23.1.8　错误处理
23.1.9　事件
23.1.10　封送
23.2　在.NET客户端中使用COM组件
23.2.1　创建COM组件
23.2.2　创建运行库可调用包装
23.2.3　使用RCW
23.2.4　通过动态语言扩展使用COM服务
23.2.5　线程问题
23.2.6　添加连接点
23.3　在COM客户端中使用.NET组件
23.3.1　COM可调用包装
23.3.2　创建.NET组件
23.3.3　创建类型库
23.3.4　COM互操作特性
23.3.5　COM注册
23.3.6　创建COM客户端应用程序
23.3.7　添加连接点
23.3.8　使用sink对象创建客户端
23.4　平台调用
23.5　小结
第24章　文件和注册表操作
24.1　文件和注册表
24.2　管理文件系统
24.2.1　表示文件和文件夹的.NET类
24.2.2　Path类
24.2.3　FileProperties示例
24.3　移动、复制和删除文件
24.3.1　FilePropertiesAndMovement示例

24.3.2　FilePropertiesAndMovement示例的代码
24.4　读写文件
24.4.1　读取文件
24.4.2　写入文件
24.4.3　流
24.4.4　缓存的流
24.4.5　使用FileStream类读写二进制文件
24.4.6　读写文本文件
24.5　映射内存的文件
24.6　读取驱动器信息
24.7　文件的安全性
24.7.1　从文件中读取ACL
24.7.2　从目录中读取ACL
24.7.3　添加和删除文件中的ACL项
24.8　读写注册表
24.8.1　注册表
24.8.2　.NET注册表类
24.9　读写独立存储器
24.10　小结
第25章　事务处理
25.1　简介
25.2　概述
25.2.1　事务处理阶段
25.2.2　ACID属性
25.3　数据库和实体类
25.4　传统的事务
25.4.1　ADO.NET事务
25.4.2　System.EnterpriseServices
25.5　System.Transactions
25.5.1　可提交的事务
25.5.2　事务处理的升级
25.5.3　依赖事务
25.5.4　环境事务
25.6　隔离级别
25.7　自定义资源管理器
25.8　文件系统事务
25.9　小结
第26章　网络
26.1　网络
26.2　HttpClient类
26.2.1　异步调用Web服务
26.2.2　标题
26.2.3　HttpContent
26.2.4　HttpMessageHandler
26.3　把输出结果显示为HTML页面

26.3.1　从应用程序中进行简单的Web浏览
26.3.2　启动Internet Explorer实例
26.3.3　给应用程序提供更多IE类型的功能
26.3.4　使用WebBrowser控件打印
26.3.5　显示所请求页面的代码
26.3.6　WebRequest类和WebResponse类的层次结构
26.4　实用工具类
26.4.1　URI
26.4.2　IP地址和DNS名称
26.5　较低层的协议
26.5.1　使用SmtpClient
26.5.2　使用TCP类
26.5.3　TcpSend和TcpReceive示例
26.5.4　TCP和UDP
26.5.5　UDP类
26.5.6　Socket类
26.5.7　WebSocket
26.6　小结
第27章　Windows服务
27.1　Windows服务
27.2　Windows服务的体系结构
27.2.1　服务程序
27.2.2　服务控制程序
27.2.3　服务配置程序
27.2.4　Windows服务的类
27.3　创建Windows服务程序
27.3.1　创建服务的核心功能
27.3.2　QuoteClient示例
27.3.3　Windows服务程序
27 3.4　线程化和服务
27.3.5　服务的安装
27.3.6　安装程序
27.4　Windows服务的监控和控制
27.4.1　MMC管理单元
27.4.2　net.exe实用程序
27.4.3　se.exe实用程序
27.4.4　Visual Studio Server Explorer
27.4.5　编写自定义ServiceController类
27.5　故障排除和事件日志
27.6　小结
第28章　本地化
28.1　全球市场
28.2　System.Globalization名称空间
28.2.1　Unicode问题
28.2.2　区域性和区域

28.2.3　使用区域性
28.2.4　排序
28.3　资源
28.3.1　创建资源文件
28.3.2　资源文件生成器
28.3.3　ResourceWriter
28.3.4　使用资源文件
28.3.5　System.Resources名称空间
28.4　使用Visual Studio的Windows Forms本地化
28.4.1　通过编程方式修改区域性
28.4.2　使用自定义资源消息
28.4.3　资源的自动回退
28.4.4　外包翻译
28.5　ASP.NET Web Forms的本地化
28.6　用WPF本地化
28.6.1　用于WPF的．NET资源
28.6.2　XAML资源字典
28.7　自定义资源读取器
28.7.1　创建DatabaseResourceReader类
28.7.2　创建DatabaseResourceSet类
28.7.3　创建DatabaseResourceManager类
28.7.4　DatabaseResourceReader的客户端应用程序
28.8　创建自定义区域性
28.9　用Windows Store应用程序进行本地化
28.9.1　使用资源
28.9.2　使用多语言应用程序工具集进行本地化
28.10　小结
第29章　核心XAML
29.1　XAML的作用
29.2　XAML概述
29.2.1　元素如何映射到．NET对象上
29.2.2　使用自定义．NET类
29.2.3　把属性用作特性
29.2.4　把属性用作元素
29.2.5　基本的．NET类型
29.2.6　使用集合和XAML
29.2.7　用XAML代码调用构造函数
29.3　依赖属性
29.3.1　创建依赖属性
29.3.2　强制值回调
29.3.3　值变更回调和事件
29.3.4　事件的冒泡和隧道
29.4　附加属性
29.5　标记扩展
29.6　创建自定义标记扩展

29.7　XAML定义的标记扩展
29.8　读写XAML
29.9　小结
第30章　Managed Extensibility Framework
30.1　概述
30.2　MEF的体系结构
30.2.1　使用属性的MEF
30.2.2　基于约定的部件注册
30.3　定义协定
30.4　导出部件
30.4.1　创建部件
30.4.2　导出属性和方法
30.4.3　导出元数据
30.4.4　使用元数据进行惰性加载
30.5　导入部件
30.5.1　导入连接
30.5.2　部件的惰性加载
30.5.3　用惰性实例化的部件读取元数据
30.6　容器和出口提供程序
30.7　类别
30.8　小结
第31章　Windows运行库
31.1　概述
31.1.1　.NET与Windows运行库的比较
31.1.2　名称空间
31.1.3　元数据
31.1.4　语言投射
31.1.5　Windows运行库中的类型
31.2　Windows运行库组件
31.2.1　集合
31.2.2　流
31.2.3　委托与事件
31.2.4　异步操作
31.3　Windows Store应用程序
31.4　应用程序的生命周期
31.4.1　应用程序的执行状态
31.4.2　Suspension Manager
31.4.3　导航状态
31.4.4　测试暂停
31.4.5　页面状态
31.5　应用程序的设置
31.6　小结
第IV部分　数据
第32章　核心ADO.NET
32.1　ADO.NET概述

32.1.1　名称空间
32.1.2　共享类
32.1.3　数据库专用类
32.2　使用数据库连接
32.2.1　管理连接字符串
32.2.2　高效地使用连接
32.2.3　事务
32.3　命令
32.3.1　执行命令
32.3.2　调用存储过程
32.4　快速数据访问：数据读取器
32.5　异步数据访问：使用Task和await
32.6　管理数据和关系：DataSet类
32.6.1　数据表
32.6.2　数据列
32.6.3　数据关系
32.6.4　数据约束
32.7　XML架构：用XSD生成代码
32.8　填充DataSet类
32.8.1　用数据适配器填充DataSet
32.8.2　从XML中填充DataSet类
32.9　持久化DataSet类的修改
32.9.1　通过数据适配器进行更新
32.9.2　写入XML输出结果
32.10　使用ADO.NET
32.10.1　分层开发
32.10.2　生成SQL Server的键
32.10.3　命名约定
32.11　小结
第33章　ADO.NET Entity Framework
33.1　用Entity Framework编程
33.2　Entity Framework映射
33.2.1　逻辑层
33.2.2　概念层
33.2.3　映射层
33.2.4　连接字符串
33.3　实体
33.4　对象上下文
33.5　关系
33.5.1　一个层次结构一个表
33.5.2　一种类型一个表
33.5.3　懒惰加载、延迟加载和预先加载
33.6　查询数据
33.6.1　Entity SQL
33.6.2　使用DbSqlQuery

33.6.3　LINQ to Entities
33.7　把数据写入数据库
33.7.1　对象跟踪
33.7.2　改变信息
33.7.3　附加和分离实体
33.7.4　使用最后一个更改操作写入实体的更改
33.7.5　使用第一个更改操作写入实体的更改
33.7.6　写入实体的更改并处理冲突
33.8　使用Code First编程模型
33.8.1　定义实体类型
33.8.2　创建数据上下文
33.8.3　创建数据库，存储实体
33.8.4　数据库
33.8.5　查询数据
33.8.6　定制数据库的生成
33.8.7　数据库的自动填充
33.8.8　连接的弹性
33.8.9　架构的迁移
33.9　小结
第34章　处理XML
34.1　XML
34.2　.NET支持的XML标准
34.3　System.Xml名称空间
34.4　使用System.Xml类
34.5　读写流格式的XML
34.5.1　使用XmlReader类
34.5.2　使用XmlReader类进行验证
34.5.3　使用XmlWriter类
34.6　在．NET中使用DOM
34.7　使用XPathNavigator类
34.7.1　System.Xml.XPath名称空间
34.7.2　System.Xml.Xsl名称空间
34.7.3　调试XSLT
34.8　XML和ADO.NET
34.8.1　将ADO.NET数据转换为XML文档
34.8.2　把XML文档转换为ADO.NET数据
34.9　在XML中序列化对象
34.10　LINQ to XML和．NET
34.11　使用不同的XML对象
34.11.1　XDocument对象
34.11.2　XElement对象
34.11.3　XNamespace对象
34.11.4　XComment对象
34.11.5　XAttribute对象
34.12　使用LINQ查询XML文档

34.12.1　查询静态的XML文档
34.12.2　查询动态的XML文档
34.13　XML文档的更多查询技术
34.13.1　读取XML文档
34.13.2　写入XML文档
34.14　小结
第Ⅴ部分　显示
第35章　核心WPF
35.1　理解WPF
35.1.1　名称空间
35.1.2　类层次结构
35.2　形状
35.3　几何图形
35.4　变换
35.5　画笔
35.5.1　SolidColorBrush
35.5.2　LinearGradientBrush
35.5.3　RadialGradientBrush
35.5.4　DrawingBrush
35.5.5　ImageBrush
35.5.6　VisualBrush
35.6　控件
35.6.1　简单控件
35.6.2　内容控件
35.6.3　带标题的内容控件
35.6.4　项控件
35.6.5　带标题的项控件
35.6.6　修饰
35.7　布局
35.7.1　StackPanel
35.7.2　WrapPanel
35.7.3　Canvas
35.7.4　Dock.Panel
35.7.5　Grid
35.8　样式和资源
35.8.1　样式
35.8.2　资源
35.8.3　系统资源
35.8.4　从代码中访问资源
35.8.5　动态资源
35.8.6　资源字典
35.9　触发器
35.9.1　属性触发器
35.9.2　多触发器
35.9.3　数据触发器

35.10 模板
35.10.1 控件模板
35.10.2 数据模板
35.10.3 样式化列表框
35.10.4 ItemTemplate
35.10.5 列表框元素的控件模板
35.11 动画
35.11.1 时间轴
35.11.2 非线性动画
35.11.3 事件触发器
35.11.4 关键帧动画
35.12 可见状态管理器
35.12.1 可见的状态
35.12.2 变换
35.13 3-D
35.13.1 模型
35.13.2 照相机
35.13.3 光线
35.13.4 旋转
35.14 小结
第36章 用WPF编写业务应用程序
36.1 概述
36.2 菜单和功能区控件
36.2.1 菜单控件
36.2.2 功能区控件
36.3 Commanding
36.3.1 定义命令
36.3.2 定义命令源
36.3.3 命令绑定
36.4 数据绑定
36.4.1 BooksDemo应用程序内容
36.4.2 用XAML绑定
36.4.3 简单对象的绑定
36.4.4 更改通知
36.4.5 对象数据提供程序
36.4.6 列表绑定
36.4.7 主从绑定
36.4.8 多绑定
36.4.9 优先绑定
36.4.10 值的转换
36.4.11 动态添加列表项
36.4.12 动态添加选项卡中的项
36.4.13 数据模板选择器
36.4.14 绑定到XML上
36.4.15 绑定的验证和错误处理

36.5 Tree View
36.6 DataGrid
36.6.1 自定义列
36.6.2 行的细节
36.6.3 用DataGrid进行分组
36.6.4 实时成型
36.7 小结
第37章 用WPF创建文档
37.1 简介
37.2 文本元素
37.2.1 字体
37.2.2 TextEffect
37.2.3 内联
37.2.4 块
37.2.5 列表
37.2.6 表
37.2.7 块的锚定
37.3 流文档
37.4 固定文档
37.5 XPS文档
37.6 打印
37.6.1 用PrintDialog打印
37.6.2 打印可见元素
37.7 小结
第38章 Windows Store应用程序：用户界面
38.1 概述
38.2 Microsoft的现代设计
38.2.1 内容，不是边框
38.2.2 快速流畅
38.2.3 可读性
38.3 示例应用程序的核心功能
38.3.1 文件和目录
38.3.2 应用程序页面
38.4 应用程序工具栏
38.5 启动与导航
38.6 布局的变化
应用程序数据
38.7 存储
38.7.1 定义数据协定
38.7.2 写入移动数据
38.7.3 读取数据
38.7.4 写入图像
38.7.5 读取图像
38.8 选择器
38.9 活动的磁贴

38.10 小结
第39章 Windows Store应用程序：协定和设备
39.1 概述
39.2 搜索
39.3 共享协定
39.3.1 共享源
39.3.2 共享目标
39.4 相机
39.5 定位
39.6 感应器
39.6.1 光线
39.6.2 罗盘
39.6.3 加速计
39.6.4 倾斜计
39.6.5 陀螺仪
39.6.6 方向
39.6.7 Rolling Marble示例
39.7 小结
第40章 核心ASP.NET
40.1 用于Web应用程序的.NET Framework
40.1.1 ASP.NET Web Forms
40.1.2 ASP.NET Web Pages
40.1.3 ASP.NET MVC
40.2 Web技术
40.2.1 HTML
40.2.2 CSS
40.2.3 JavaScript和jQuery
40.3 托管和配置
40.4 处理程序和模块
40.4.1 创建自定义处理程序
40.4.2 ASP.NET处理程序
40.4.3 创建自定义模块
40.4.4 通用模块
40.5 全局的应用程序类
40.6 请求和响应
40.6.1 使用HttpRequest对象
40.6.2 使用HttpResponse对象
40.7 状态管理
40.7.1 视图状态
40.7.2 cookie
40.7.3 会话
40.7.4 应用程序状态
40.7.5 缓存
40.7.6 配置文件
40.8 ASP.NET身份系统

40.8.1 基础知识
40.8.2 存储和检索用户信息
40.8.3 安全启动
40.8.4 使用注册和身份验证
40.9 小结
第41章 ASP.NET Web Forms
41.1 概述
41.2 ASPX页面模型
41.2.1 添加控件
41.2.2 使用事件
41.2.3 使用回送
41.2.4 使用自动回送
41.2.5 回送到其他页面
41.2.6 定义强类型化的跨页面回送
41.2.7 使用页面事件
41.2.8 ASPX代码
41.2.9 服务器端控件
41.3 母版页
41.3.1 创建母版页
41.3.2 使用母版页
41.3.3 在内容页中定义母版页内容
41.4 导航
41.4.1 站点地图
41.4.2 Menu控件
41.4.3 菜单路径
41.5 验证用户输入
41.5.1 使用验证控件
41.5.2 使用验证摘要
41.5.3 验证组
41.6 访问数据
41.6.1 使用Object Framework
41.6.2 创建库
41.6.3 使用Object Data Source
41.6.4 编辑
41.6.5 定制列
41.6.6 在网格中使用模板
41.7 安全性
41.7.1 建立ASP.NET身份
41.7.2 用户注册
41.7.3 用户的身份验证
41.7.4 用户授权
41.8 Aiax
41.8.1 ASP.NET AJAX的概念
41.8.2 ASP.NET AJAX网站示例
41.8.3 支持ASP.NET AJAX的网站配置

41.8.4　添加ASP.NET AJAX功能
41.9　小结
第42章　ASP.NET MVC
42.1　ASP.NET MVC概述
42.2　定义路由
42.2.1　添加路由
42.2.2　路由约束
42.3　创建控制器
42.3.1　动作方法
42.3.2　参数
42.3.3　返回数据
42.4　创建视图
42.4.1　向视图传递数据
42.4.2　Razor语法
42.4.3　强类型视图
42.4.4　布局
42.4.5　部分视图
42.5　从客户端提交数据
42.5.1　模型绑定器
42.5.2　注释和验证
42.6　HTML Helper
42.6.1　简单的Helper
42.6.2　使用模型数据
42.6.3　定义HTML特性
42.6.4　创建列表
42.6.5　强类型化的Helper
42.6.6　编辑器扩展
42.6.7　创建自定义Helper
42.6.8　模板
42.7　创建数据驱动的应用程序
42.7.1　定义模型
42.7.2　创建控制器和视图
42.8　动作过滤器
42.9　身份验证和授权
42.9.1　登录模型
42.9.2　登录控制器
42.9.3　登录视图
42.10　小结
第VI部分　通信
第43章　WCF
43.1　WCF概述
43.1.1　SOAP
43.1.2　WSDL
43.1.3　REST
43.1.4　JSON

43.2　创建简单的服务和客户端
43.2.1　定义服务和数据协定
43.2.2　数据访问
43.2.3　服务的实现
43.2.4　WCF服务宿主和WCF测试客户端
43.2.5　自定义服务宿主
43.2.6　WCF客户端
43.2.7　诊断
43.2.8　与客户端共享协定程序集
43.3　协定
43.3.1　数据协定
43.3.2　版本问题
43.3.3　服务协定
43.3.4　消息协定
43.3.5　错误协定
43.4　服务的行为
43.5　绑定
43.5.1　标准的绑定
43.5.2　标准绑定的特性
43.5.3　Web套接字
43.6　宿主
43.6.1　自定义宿主
43.6.2　WAS宿主
43.6.3　预配置的宿主类
43.7　客户端
43.7.1　使用元数据
43.7.2　共享类型
43.8　双工通信
43.8.1　双工通信的协定
43.8.2　双工通信的服务
43.8.3　双工通信的客户端应用程序
43.9　路由
43.9.1　示例应用程序
43.9.2　路由接口
43.9.3　WCF路由服务
43.9.4　为失败使用路由器
43.9.5　改变协定的桥梁
43.9.6　过滤器的类型
43.10　小结
第44章　ASP.NET Web API
44.1　概述
44.2　创建服务
44.2.1　定义模型
44.2.2　创建控制器
44.2.3　错误处理

44.3 创建.NET客户程序
44.3.1 发送GET请求
44.3.2 发送POST请求
44.3.3 发送PUT请求
44.3.4 发送DELETE请求
44.4 Web API路由和操作
44.4.1 给操作添加HTTP方法
44.4.2 基于特性的路由
44.5 使用OData
44.5.1 创建数据模型
44.5.2 创建服务
44.5.3 OData查询
44.5.4 WCF Data Services客户程序
44.6 保护Web API
44.6.1 创建账户
44.6.2 创建验证令牌
44.6.3 发送验证过的调用
44.6.4 获取用户信息
44.7 自驻留
44.8 小结
第45章 Windows Workflow Foundation
45.1 工作流概述
45.2 Hello World示例
45.3 活动
45.3.1 If活动
45.3.2 InvokeMethod活动
45.3.3 Parallel活动
45.3.4 Delay活动
45.3.5 Pick活动
45.4 自定义活动
45.4.1 活动的验证
45.4.2 设计器
45.4.3 自定义复合活动
45.5 工作流
45.5.1 实参和变量
45.5.2 WorkflowApplication
45.5.3 存放WCF工作流
45.5.4 工作流的版本
45.5.5 驻留设计器
45.6 小结
第46章 对等网络
46.1 P2P网络概述
46.1.1 客户端-服务器体系结构
46.1.2 P2P体系结构
46.1.3 P2P体系结构的挑战

46.1.4　P2P术语
46.1.5　P2P解决方案
46.2　PNRP
46.3　构建P2P应用程序
46.4　小结
第47章　消息队列
47.1　概述
47.1.1　使用消息队列的场合
47.1.2　消息队列功能
47.2　Message Queuing产品
47.3　消息队列体系结构
47.3.1　消息
47.3.2　消息队列
47.4　Message Queuing管理工具
47.4.1　创建消息队列
47.4.2　消息队列属性
47.5　消息队列的编程实现
47.5.1　创建消息队列
47.5.2　查找队列
47.5.3　打开已知队列
47.5.4　发送消息
47.5.5　接收消息
47.6　课程订单应用程序
47.6.1　课程订单类库
47.6.2　课程订单消息发送程序
47.6.3　发送优先级和可恢复的消息
47.6.4　课程订单消息接收应用程序
47.7　接收结果
47.7.1　确认队列
47.7.2　响应队列
47.8　事务队列
47.9　消息队列和WCF
47.9.1　带数据协定的实体类
47.9.2　WCF服务协定
47.9.3　WCF消息接收应用程序
47.9.4　WCF消息发送应用程序
47.10　消息队列的安装
47.11　小结