单击 OK 按钮。

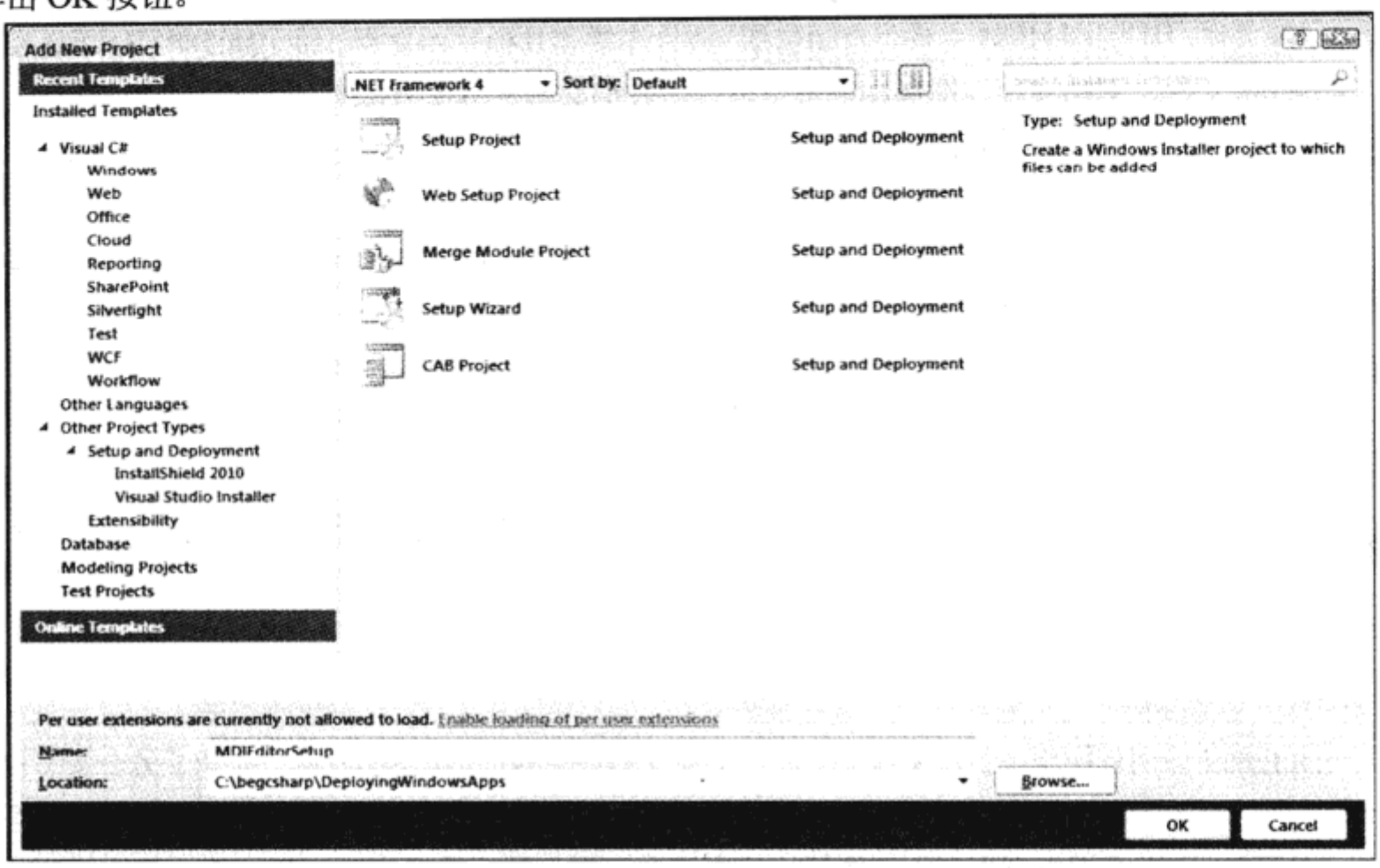


图 17-22

17.5.3 项目属性

到目前为止，我们仅仅拥有一个项目文件可以用于安装解决方案。必须定义要安装的文件，还必须配置项目属性。为此，必须了解 Packaging 和 Bootstrapper 选项的含义。

1. 打包

MSI 是启动安装的数据库，但是我们可以定义如何使用如图 17-23 所示的 3 个选项打包要安装的文件。右击 MDIEditorSetup 项目，选择 Properties，就会打开此对话框。

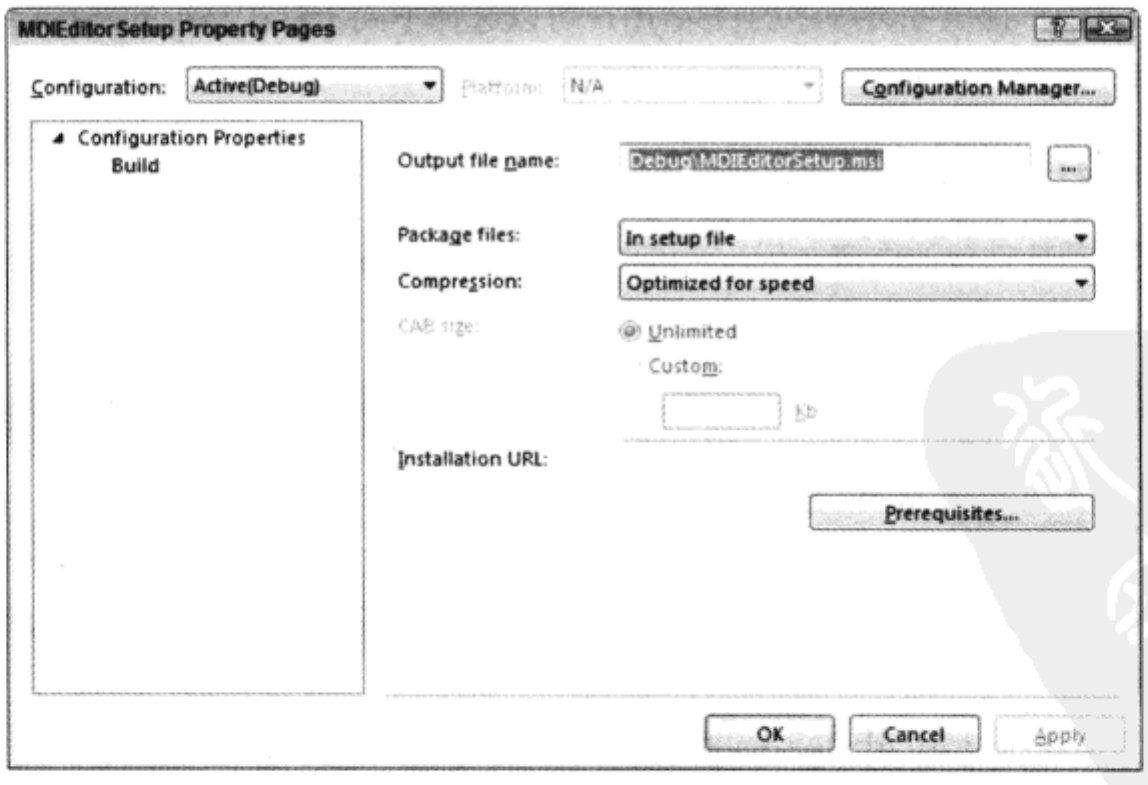


图 17-23

Package files 下拉列表中的选项如下所示：

- As loose uncompressed files 选项将所有程序和数据文件都原样存储，不进行文件压缩。
- In setup file 选项会把所有的文件合并、压缩到 MSI 文件中。可以为软件包中的单一组件重写此选项。如果将所有文件放到一个 MSI 文件中，就必须注意安装程序的大小要适合于希望使用的介质，比如 CD 或软盘。如果安装的文件太多，超过了一张软盘的容量，可以试试在 Compression 下拉列表中选择 Optimized for size 选项，改变压缩选项。如果其容量仍然不适合，则可以选择下一个打包选项。
- 对文件打包的第三种方法是 In cabinet file(s)。在此方法中，MSI 文件仅用于加载和安装 CAB 文件。使用 CAB 文件可以设置文件的大小，以便从 CD 或软盘上安装(对于从软盘的安装，可以设置 1440KB 的安装容量)。

2. 预先安装的软件包

在上面的对话框中可以配置预先安装的软件包，即在安装应用程序之前必须安装的部分。单击 Installation URL 文本框旁边的 Prerequisites 按钮，就会弹出 Prerequisites 对话框，如图 17-24 所示。可以看出，.NET Framework 4 Client Profile 默认选中，作为预先安装的软件包。如果客户系统没有安装.NET Framework，就从安装程序中安装它。还可以选择其他预先安装选项，如下所示。

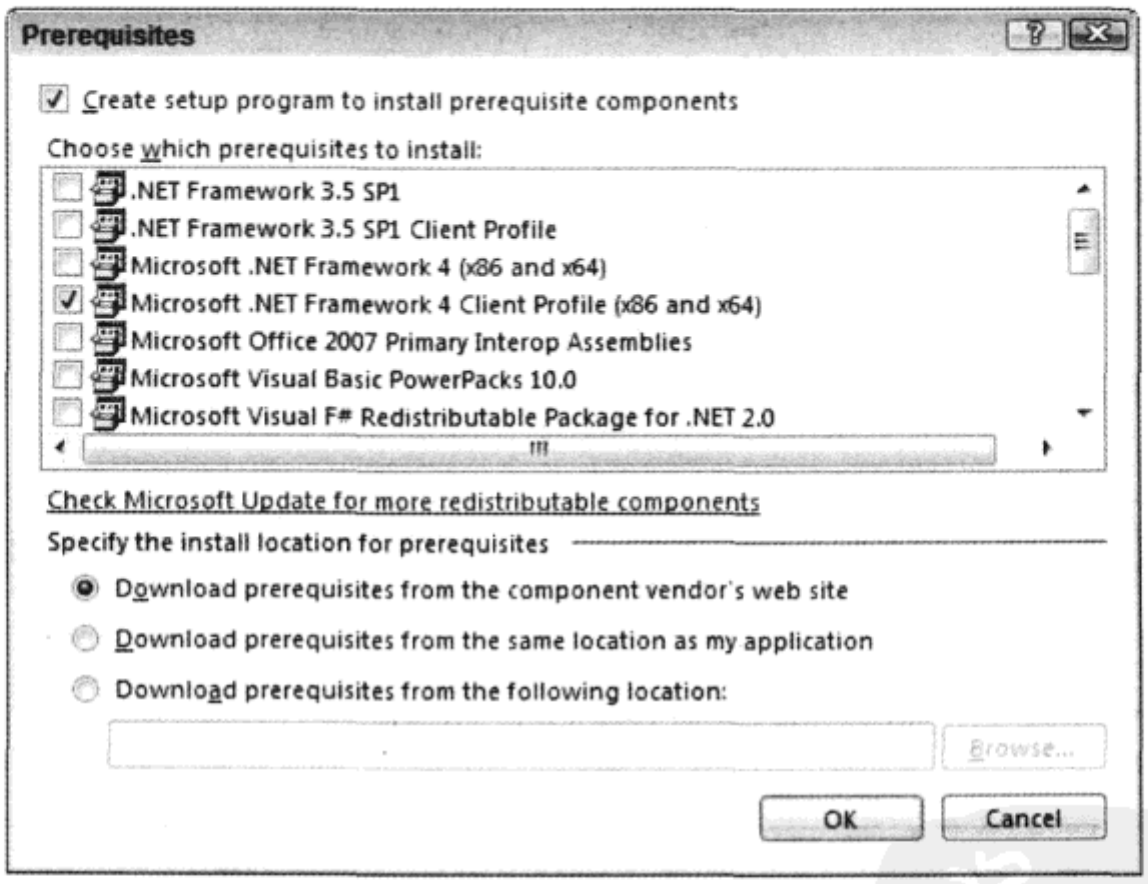


图　17-24

- Windows Installer 3.1：Windows Installer 3.1 用于通过 Visual Studio 2010 创建的安装软件包。如果目标系统是 Windows Vista 或 Windows Server 2008，该安装程序就已经安装在系统上了。在较旧的系统上，可能没有 Windows Installer 的正确版本，所以应选择这个选项，在安装程序中包含 Windows Installer 3.1。Windows 7 使用 Windows Installer 5 版本，而 Visual Studio 2010 使用 Windows Installer 4.5 版本。

- **SQL Server 2008 Express**：如果需要在客户系统上有一个数据库，就可以在安装程序中包含 SQL Server 2008 Express 版本。有关用 ADO.NET 访问 SQL Server 的详细内容见第 24 章。
- **Microsoft Office 2007 Primary Interop Assemblies**：对于使用 Office 自动功能的应用程序，可以用这个组件安装 Office 2007 的主要互操作程序集。
- **Visual Basic PowerPacks 10.0**：提供了用 Visual Basic 编程的额外特性，Visual Basic 可以用这个组件来安装。
- **Visual F# Redistributable Package**：对于用编程语言 F#编写的应用程序，需要这个软件包。

试一试：配置项目

(1) 将 Property 页面上的 Prerequisites 选项(View | Property Pages)改为包含 Windows Installer 3.1，这样应用程序就可以安装在没有 Windows Installer 3.1 的系统上。将输出文件名改为 WroxMDIEditor.msi，如图 17-25 所示。然后单击 OK 按钮。

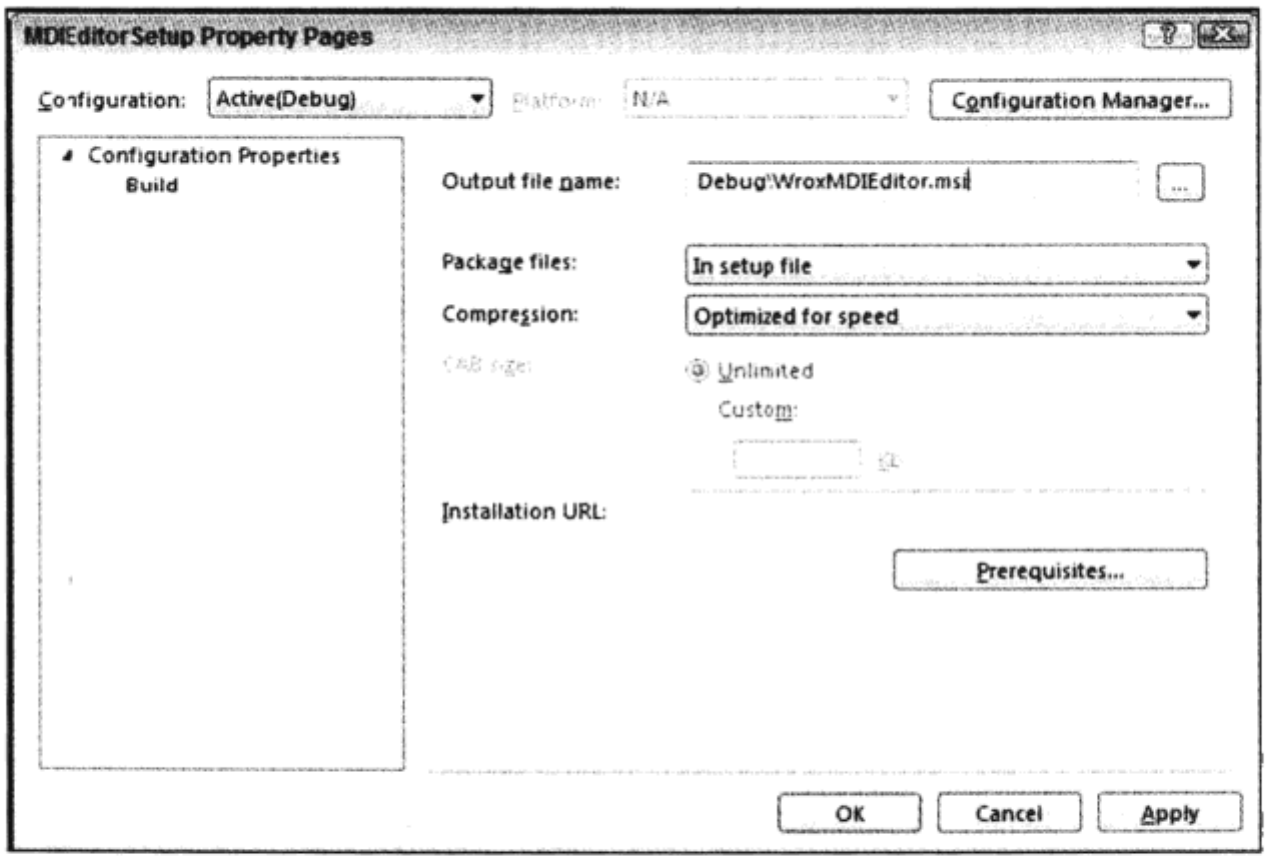


图　17-25

(2) 使用 Properties 窗口将项目属性设置为表 17-1 中的值。

表　17-1

属　性	值
Author	Wrox Press
Description	要打印和编辑文本文件的 MDI Editor
Keywords	Installer、Wrox Press、MDI Editor
InstallAllUsers	True
Manufacturer	Wrox Press
ManufacturerUrl	http://www.wrox.com

(续表)

属　性	值
Product Name	Wrox MDI Editor
SupportUrl	http://p2p.wrox.com
Title	MDI Editor 的安装演示
Version	1.0.0

17.5.4　安装编辑器

Visual Studio 2010 的 Setup Project 有 6 个编辑器。选择编辑器的方式如下：打开部署项目，选择 View | Editor 菜单项。

- File System 编辑器用于向安装软件包中添加文件。
- 使用 Registry 编辑器，可以为应用程序创建注册表项。
- File Types 编辑器允许注册应用程序的特定文件扩展名。
- 使用 User Interface 编辑器，可以添加和配置在安装程序期间显示的对话框。
- Custom Actions 编辑器允许在安装和卸载期间启动定制的程序。
- 使用 Launch Conditions 编辑器，可以指定对应用程序的要求，比如，必须已安装NET 运行库。

17.5.5　File System 编辑器

使用 File System 编辑器，可以向安装软件包中添加文件，并配置安装它们的位置。通过 View | Editor | File System 菜单可以打开此编辑器。还会自动打开一些预定义的特殊文件夹，如图 17-26 所示。

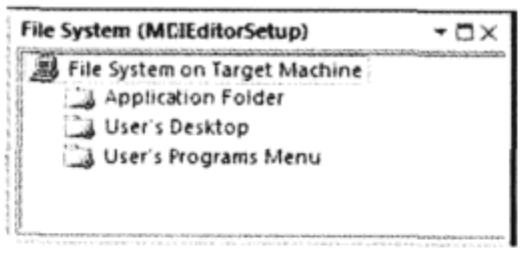


图　17-26

- Application Folder 文件夹用于存储可执行文件和库。其位置定义为[ProgramFilesFolder]\[Manufacturer]\[ProductName]。在 English 语言系统上，[ProgramFilesFolder]解析为 C:\Program Files。[Manufacturer]和[ProductName]的目录由 Manufacturer 和 ProductName 项目属性定义。
- 如果希望在桌面上放置图标，可以使用 User's Desktop 文件夹。此文件夹的默认路径是 C:\Users\username\Desktop 或 C:\Users\All Users\Desktop，这取决于安装是用于单个用户还是所有用户。
- 用户通常在 All Programs 菜单中启动程序。默认路径是 C:\Documents and Settings*username*\Start Menu\Programs。在此菜单中可以把快捷方式放到应用程序上。快捷方式的名称包括公司和应用程序的名称，这样用户就很容易识别应用程序，如 Microsoft Excel。

一些应用程序创建了子菜单，由此可以启动多个应用程序，如 Microsoft Visual Studio 2010。根据 Windows 准则，许多程序由于错误的原因而列举了大量不需要的程序：在此菜单中不应该放置卸载程序，因为此功能可以在 Control Panel 的 Programs and Features 中获取，卸载程序应该在控制面

板中进行。此菜单中也不应该放置帮助文件，因为帮助文件可以直接从应用程序中获取。因此对于许多应用程序而言，在 All Programs 菜单中直接放置应用程序的快捷方式就足矣。这些限制的目的是不要在 Start 菜单中放置太多的项，避免使它过于拥挤。

此信息的参考信息位于“Microsoft Windows 7 桌面应用程序规范”中。该文档在 http://msdn.microsoft.com/en-us/windows/dd203105.aspx 上。

右击并选择 Add Special Folder，还可以添加其他文件夹，其中一些文件夹如下：

- Global Assembly Cache (GAC)Folder 表示安装共享程序集的文件夹。GAC 用于在多个应用程序之间共享的程序集。
- User's Personal Data Folder 是用户的默认文件夹，其中存储了文档。C:\Users\[username] \My Documents 是默认路径。此路径是 Visual Studio 用于存储项目的默认目录。
- 当选择文件时，User's Send To Menu 中的快捷方式可以扩展 Send To 关联菜单。使用此关联菜单，用户通常可以将文件发送到目标位置，如软驱、邮件接收者或 My Documents 文件夹。

1. 向特殊文件夹添加项

选择文件夹，再选择菜单 Action | Add Special Folder，可以从列表中选择一些项，添加到特定的文件夹中。可以选择 Project Output、Folder、File 或 Assembly。给文件夹添加项目的输出，会自动添加生成的输出文件，以及一个.dll 或.exe，这取决于所添加的项目是组件库还是应用程序。选择 Project Output 或 Assembly 可以给文件夹自动添加所有的依赖关系(所有引用的程序集)。

2. 文件属性

在文件夹中选择文件的属性，就可以设置表 17-2 中所示的属性。根据文件类型，可能无法应用其中一些属性，也可能应用其中未列出的一些属性。

表 17-2

属 性	说 明
Condition	可以使用此属性定义一个条件，确定是否选择应安装的文件。如果仅在特定的操作系统版本上添加该文件，或者用户必须在对话框中进行一些选择，该属性就非常有用
Exclude	如果不安装此文件，则该属性设置为 True。这样文件就可以保留在项目中，而不会安装。如果可以肯定它不是相关文件，或它已存在于要部署应用程序的每个系统中，则可以排除该文件
PackageAs	利用 PackageAs 可以重写文件添加到安装软件包的默认方式；例如，如果项目配置是 in setup file，就可以使用此选项将特定文件的软件包配置改为 Loose，这样此文件就不会添加到 MSI 数据库文件中。如果希望添加用户在开始安装之前读取的 ReadMe 文件，这就非常有用。显然，即使压缩了其他文件，也不应压缩此文件
Permanent	将此属性设置为 True 意味着，在卸载之后文件仍会保留在目标计算机上。这可以用于配置文件。例如，安装 Microsoft Outlook 的新版本时，如果配置了 Microsoft Outlook，则在卸载并再次安装它时，就不需要再次配置它，因为上一次的安装配置并未删除
ReadOnly	此属性在安装时设置只读文件特性
Vital	此属性表示，此文件对于此产品的安装至关重要。如果此文件的安装失败，就终止完整的安装进程，只能回滚

下面的示例给 Windows 安装程序包添加应部署的文件。

试一试：向安装软件包添加文件

(1) 向安装程序项目的 Application 文件夹添加 MDI Editor 项目的主要输出，为此使用 Project | Add | Project Output 菜单项。在 Add Project Output Group 对话框中，选择 Primary Output，如图 17-27 所示。

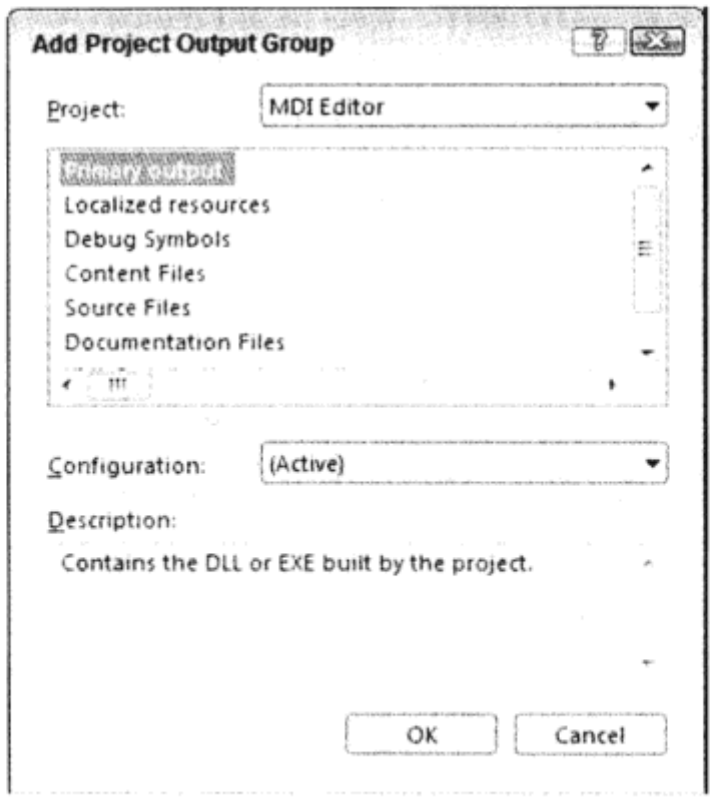


图　17-27

单击 OK 按钮，把 MDI Editor 项目的主要输出结果添加到自动打开的 File System Editor 中的 Application Folder。在此，主要输出就是 MDI Editor.exe。

(2) 要添加的附加文件有徽标、许可和 ReadMe 文件。在 File System Editor 中，选择 Application Folder，并选择菜单项 Action | Add | Folder，在 Application Folder 中创建名为 Setup 的子目录。

在 Visual Studio 中，只有选择了 setup editor 中的项，才能使用 Action 菜单。如果选择了 Solution Explorer 或 Class View 中的项，就不能使用 Action 菜单。

(3) 右击 Setup 文件夹，选择 Add | File，将文件 wroxlogo.bmp、wroxsetuplogo.bmp、readme.rtf 和 license.rtf 添加到 Setup 文件夹中。这些文件可以从本书的下载代码中得到，也可以自己创建这些文件。可以使用许可和 ReadMe 信息填充这些文本文件。无需改变这些文件的属性。这些文件会用在安装程序的对话框中。

位图 wroxsetuplogo.bmp 的大小应是 500 像素宽，70 像素高。该位图左边的 420 像素应只是一个背景图形，因为安装对话框的文本会覆盖这个区域。

(4) 将文件 readme.txt 添加到 Application Folder 中。在开始安装之前，用户应能读取这个文件。将属性 PackageAs 设置为 vsdpaLoose，这样此文件就不会压缩到 Installer 软件包中。还要将属性 ReadOnly 设置为 True，这样就不会改变文件。

项目现在包含两个 ReadMe 文件：readme.txt 和 readme.rtf。文件 readme.txt 可以由安装应用程序的用户在安装开始之前读取，文件 readme.rtf 用于在安装对话框中显示一些信息。

(5) 将文件 demo.txt 拖放到 User's Desktop 文件夹中。只有在询问过用户是否希望安装之后，才能安装此文件。因此，将此文件的 Condition 属性设置为 CHECKBOXDEMO。CHECKBOXDEMO 是用户可以设置的条件，其值必须大写。此条件只有设置为 TRUE，才能安装此文件。稍后将定义一个对话框，来设置此属性。

(6) 要从 Start | Programs 菜单中启动程序，需要 MDI Editor 程序的一个快捷方式。

在 Application Folder 中选择 Primary output from MDI Editor 项，并打开菜单项 Action | Create Shortcut to Primary output from MDIEditor。将所生成的快捷方式的 Name 属性设置为 Wrox MDI Editor，并将此快捷方式拖放到 User's Programs 菜单中。

17.5.6 File Types 编辑器

如果应用程序使用了定制的文件类型，并注册文件扩展名，希望用户双击带有该扩展名的文件时就启动应用程序，就可以使用 File Types 编辑器。此编辑器可以使用 View | Editor | File Types 启动，图 17-28 显示的是添加了一个定制文件扩展名的 File Types 编辑器。

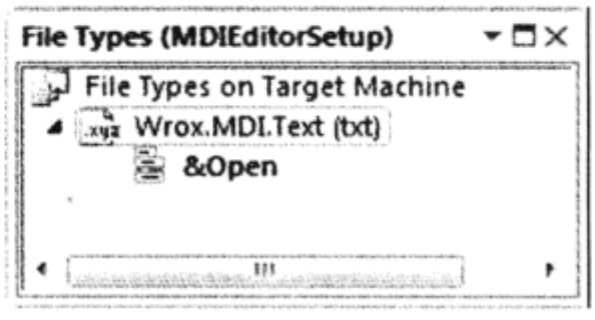


图 17-28

利用 File Type 编辑器可以配置可在应用程序中处理的文件扩展名。文件扩展名具有表 17-3 所示的属性。

表 17-3

属 性	说 明
Name	在此添加描述文件类型的有效名称。此名称显示在 File Type Editor 中，并写入到注册表中。名称应该是唯一的。.doc 文件类型的示例是 Word.Document.12。无须像在 Word 中那样使用 ProgID，也可以为.dochtml 文件扩展名使用简单的文本，如 wordhtmlfile
Command	使用 Command 属性可以指定可执行文件，当用户打开此类型的文件时，就会启动这个可执行文件
Description	在此可以添加说明
Extensions	应用程序可以进行注册的文件扩展名，在注册表中注册的文件扩展名
Icon	为文件扩展名指定要显示的图标

创建操作

在 File Type Editor 中创建了文件类型之后，就可以添加操作(action)。自动添加的默认操作是 Open。也可以添加其他动作，如 New 和 Print，或者是程序可以对文件执行的任何操作。对于操作，

必须定义 Arguments 和 Verb 属性。Arguments 属性指定传递给应用程序的参数，已经为文件扩展名对其进行了注册。例如，"%1"的意思是，文件名传递给应用程序。Verb 属性指定要发生的操作。如果应用程序支持的话，对于打印操作可以添加/print。

下面给 MDIEditor 安装程序添加操作。我们希望注册文件扩展名，这样就可以在 Windows Explorer 中使用 MDIEditor 应用程序，打开带有扩展名.txt 的文件。在注册后，可以双击这些文件打开它们，自动启动 MDIEditor 应用程序。

试一试：设置文件扩展名

(1) 通过 View | Editor | File Types 菜单项启动 File Types 编辑器。使用菜单项 Action | Add File Type 添加新文件类型，其属性如表 17-4 所示。

表　17-4

属　性	值
(Name)	Wrox.MDIEditor.Text
Command	MDI Editor 的主要输出
Description	文本文档
Extensions	Txt

也可以设置 Icon 属性，为打开的文件定义图标。还可以设置 MIME 类型。

Open 操作的属性使用默认值，这样就将文件名作为应用程序参数传递。

17.5.7　Launch Condition 编辑器

通过 Launch Condition 编辑器，可以在安装之前对目标系统指定一些要求。选择菜单项 View | Editor | Launch Conditions，启动 Launch Condition 编辑器，如图 17-29 所示。

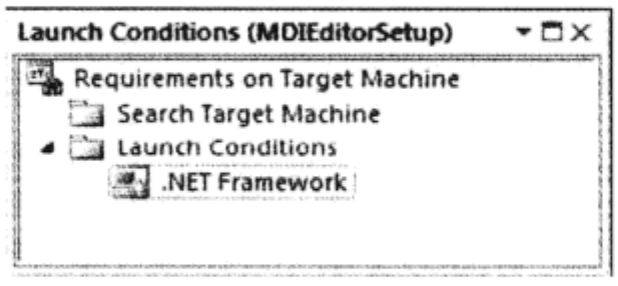


图　17-29

可以在这个编辑器的如下两部分指定要求：Search Target Machine 和 Launch Conditions。在第一部分中，可以指定要搜索什么文件或注册表项，如果搜索失败，则在第二部分中定义错误消息。下面研究一些可以使用 Action 菜单定义的启动条件。

- File Launch Condition 在开始安装前在目标系统上搜索文件。
- Registry Launch Condition 允许在安装之前查找注册表项。
- Windows Installer Launch Condition 可以搜索必须存在的 Windows Installer 组件。
- .NET Framework Launch Condition 检查目标系统是否已经安装了.NET Framework。

- Internet Information Services Launch Condition 检查已经安装的 Internet Information Services。在 Internet Information Services 已安装的情况下，如果添加这个启动条件，会搜索定义好的特定注册表项，并添加检查特定版本的条件。

在默认情况下包含了.NET Framework Launch Condition，其属性设置为预定义的值：message 属性设置为[VSDNETMSG]，这是预定义的错误消息。如果未安装.NET Framework 4，就弹出一个消息，告诉用户安装.NET Framework。InstallUrl 默认设置为 http://go.microsoft.com/ fwlink/?linkid=131000，这样用户就很容易地启动.NET Framework 的安装。

17.5.8 User Interface 编辑器

使用 User Interface 编辑器，可以定义用户在配置安装时看到的对话框。通过它可以向用户显示许可协议，询问安装路径和配置应用程序的其他信息。

下面的示例将启动 User Interface 编辑器，以配置安装应用程序时显示的对话框。

试一试：启动 User Interface 编辑器

(1) 选择 View | Editor | User Interface 菜单项，启动 User Interface 编辑器。

(2) 使用 User Interface 编辑器可以为预定义的对话框设置属性。图 17-30 显示了自动生成的对话框和两个安装模式。

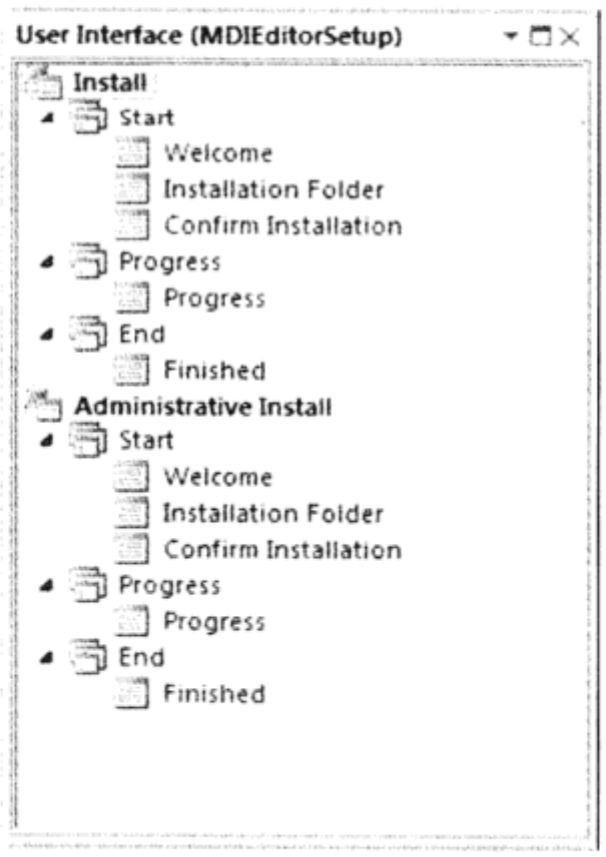


图 17-30

示例的说明

如图 17-30 所示，有两种安装模式：Install 和 Administrative Install。Install 模式一般用于在目标系统上安装应用程序。而 Administrative Install 模式可以在网络共享上安装应用程序的映像，然后用户就可以从网络安装应用程序。

在两种安装模式中，显示对话框有 3 个阶段：Start、Progress 和 End。下面介绍默认的对话框：

- Welcome 对话框向用户显示欢迎消息。可以使用自己的消息代替默认的欢迎文本。用户只能取消安装或单击 Next 按钮。
- 通过第二个对话框 Installation Folder，用户可以选择安装应用程序的文件夹。如果添加了定制的对话框(稍后介绍)，就必须在此对话框之前添加。
- Confirm Installation 对话框是安装开始之前的最后一个对话框。
- Progress 对话框显示一个进度控件，以便用户查看安装进度。
- 安装结束后会显示 Finished 对话框。

即使没有在解决方案中打开 User Interface 编辑器，默认对话框也会在安装时自动显示，但是应配置这些对话框，显示对应用程序有用的消息。

下面的示例将配置在安装应用程序时显示的默认对话框。这里忽略了 Administrative Install 路径，只配置一般安装路径。

试一试：配置默认对话框

(1) 选择 Welcome 对话框。在属性窗口中，可以看到用于对话框的 3 个属性：BannerBitmap、CopyrightWarning 和 WelcomeText。选择 BannerBitmap 属性，方式是在组合框中单击 Browse，然后选择 Application Folder\Setup 文件夹中的文件 wroxsetuplogo.bmp。此文件存储的位图会显示在对话框顶部。

属性 CopyrightWarning 的默认文本为：

```
WARNING: This computer program is protected by copyright law and international treaties.
Unauthorized duplication or distribution of this program, or any portion of it, may result
in severe civil or criminal penalties, and will be prosecuted to the maximum extent possible
under the law.(警告：此计算机程序受到版权法和国际公约保护。未经授权，复制或销售此程序，或其中的一部分，
都会受到法律许可的最大限度的民事或刑事处罚。)
```

此文本也会显示在 Welcome 对话框中。如果希望发出更严厉的警告，可以改变此文本。属性 WelcomeText 可以定义在对话框中显示的更多文本，其默认值如下：

```
The installer will guide you through the steps required to install [ProductName] on your
computer.(安装程序会引导您完成在计算机上安装[ProductName]的全部步骤)。
```

也可以改变此文本。字符串[ProductName]会自动被在项目属性中定义的属性 ProductName 代替。

(2) 选择对话框 Installation Folder。此对话框仅有两个属性 BannerBitmap 和 InstallAllUsersVisible。InstallAllUsersVisible 的默认值是 true。如果这个值设置为 false，就只有已登录的用户可以安装该应用程序。按照 Welcome 对话框的方式，把 BannerBitmap 属性改为 wroxsetuplogo.bmp 文件。每个对话框都可以用这个属性显示位图，所以也可以在其他的对话框中改变此属性。

1. 其他对话框

如果设计了一个定制对话框，您无法将其添加到 Visual Studio 安装程序的安装序列中。完成这个任务需要更专业的工具，如 Windows 的 InstallShield 或 Wise。但是有了 Visual Studio 安装程序，可以在 Add Dialog 窗口中添加和定制许多预定义的对话框。

在 User Interface 编辑器中选择 Start 序列，再选择菜单项 Action | Add Dialog，就可以打开 Add Dialog 对话框，如图 17-31 所示。所有这些对话框都是可以配置的。

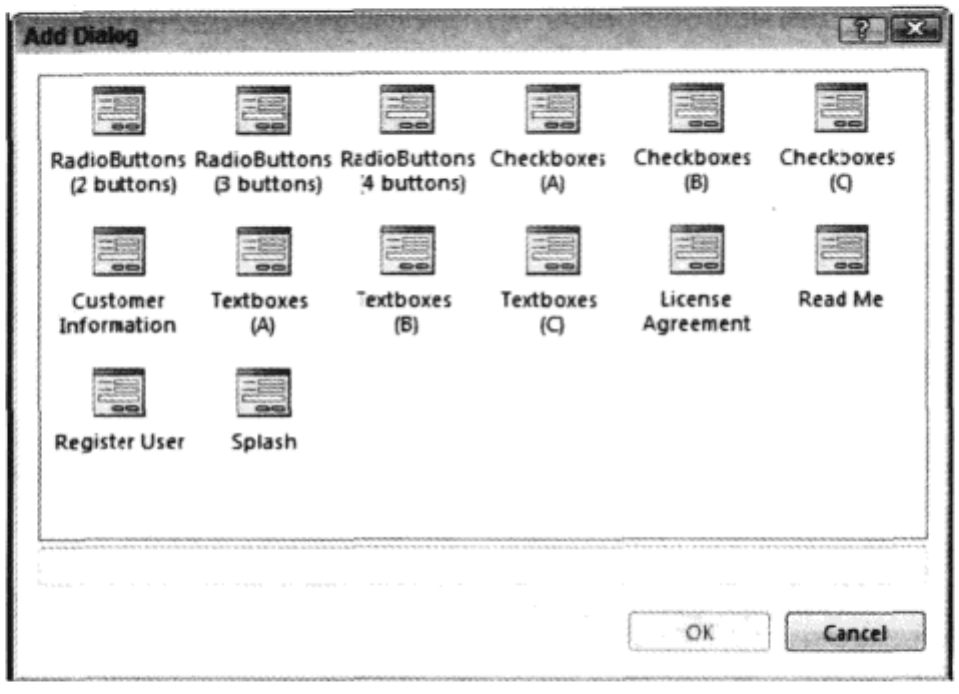


图 17-31

其中的对话框，有的可以显示 2、3 或 4 个单选按钮，复选框对话框至多可以显示 4 个复选框，文本对话框至多可以显示 4 个文本框。设置属性，就可以配置这些对话框。

下面快速浏览一下其中一些对话框：

- Customer Information 对话框询问用户的姓名、公司和产品的序列号。如果没有为产品提供序列号，可将 ShowSerialNumber 设置为 false，隐藏 Serial Number 文本框。
- 使用 License Agreement 对话框，用户可以在开始安装之前接受一个许可。许可文件通过 LicenseFile 属性定义。
- 用户可以在 Register User 对话框中单击 Register Now 按钮，启动使用 Executable 属性定义的一个程序。该定制程序可以向 FTP 服务器发送数据，或通过电子邮件传输数据。
- Splash 对话框在开始安装之前显示一个闪屏，其中使用了由 SplashBitmap 属性指定的位图。

在下面的示例中要添加一些对话框，如 Read Me、License Agreement 和 Checkboxes 对话框。

试一试：添加其他对话框

(1) 使用 Action | Add Dialog 菜单项，可以在 Start 序列中添加 Read Me、License Agreement 和 Checkboxes(A)对话框。以拖放方式，定义启动序列中的顺序：

```
Welcome - Read Me - License Agreement - Checkboxes(A) - Installation Folder - Confirm Installation.
```

(2) 按以前的方式为所有这些对话框配置属性 BannerBitmap。对于 Read Me 对话框，将属性 ReadmeFile 设置为 readme.rtf，即前面给 Application Folder\Setup 添加的文件。

(3) 对于 License Agreement 对话框，将 LicenseFile 属性设置为 license.rtf。

(4) Checkbox(A)对话框用于询问用户是否安装 User's Desktop 文件夹中的文件 demo. wroxtext。根据表 17-5 来改变此对话框的属性。

表 17-5

属　性	值
BannerText	Optional Files
BodyText	Installation of optional files
Checkbox1Label	Do you want a demo file put on to the desktop?
Checkbox1Property	CHECKBOXDEMO
Checkbox2Visible	False
Checkbox3Visible	False
Checkbox4Visible	False

Checkbox1Property 属性值设置为与文件 demo.txt 的 Condition 属性相同——前面使用 File System 编辑器向软件包中添加文件时设置了这个 Condition 属性。如果用户选中此复选框，则 CHECKBOXDEMO 的值是 true，就安装文件；如果未选中复选框，其值是 false，就不安装文件。

其他复选框的 CheckboxXVisible 属性设置为 false，因为仅需要一个复选框。

17.6　生成项目

现在开始生成安装程序项目。

试一试：生成项目

(1) 要创建 Windows 安装程序软件包，右击 SimpleEditorSetup 项目，选择 Build 选项。

(2) 如果生成成功，则可以在 Debug 或 Release 目录中找到 setup.exe、WroxSimpleEditor.msi 和 readme.txt 文件(取决于生成设置)。

示例的说明

setup.exe 启动 MSI 数据库文件 WroxSimpleEditor.msi 的安装。向安装程序项目添加的所有文件(只有一个例外)都合并、压缩到 MSI 文件中，因为项目属性设置为 Package Files in Setup File。其中一个例外是文件 readme.txt。由于改变了 PackageAs 属性，该文件可以在安装应用程序之前读取。.NET Framework 的安装软件包位于 DotNetFx 子目录中。

17.7　安装

现在可以开始安装 MDI Editor 应用程序。双击 setup.exe 文件，或选择文件 WroxMDIEditor. Msi。按下鼠标右键，打开关联菜单，选择 Install 选项。也可以在 Visual Studio 2010 中启动安装，即右击在 Solution Explorer 中打开的安装项目，选择 Install 选项。

从下面的屏幕图中可以看到，所有的对话框都具有 Wrox 标志，插入的 Read Me 和 License Agreement 对话框带有已配置的文件。

17.7.1 Welcome

第一个显示的对话框是 Welcome 对话框(如图 17-32 所示)。在此对话框中，可以看到 Wrox 徽标，它是通过设置 BannerBitmap 属性的值而插入的。所看到的文本由属性 WelcomeText 和 CopyrightWarning 定义。此对话框的标题是由属性 ProductName 设置的，该属性是通过项目属性设置的。

图 17-32

17.7.2 Read Me

单击 Next 按钮之后，就会显示 Read Me 对话框。它显示通过属性 ReadmeFile 配置的格式文本文件 readme.rtf，如图 17-33 所示。

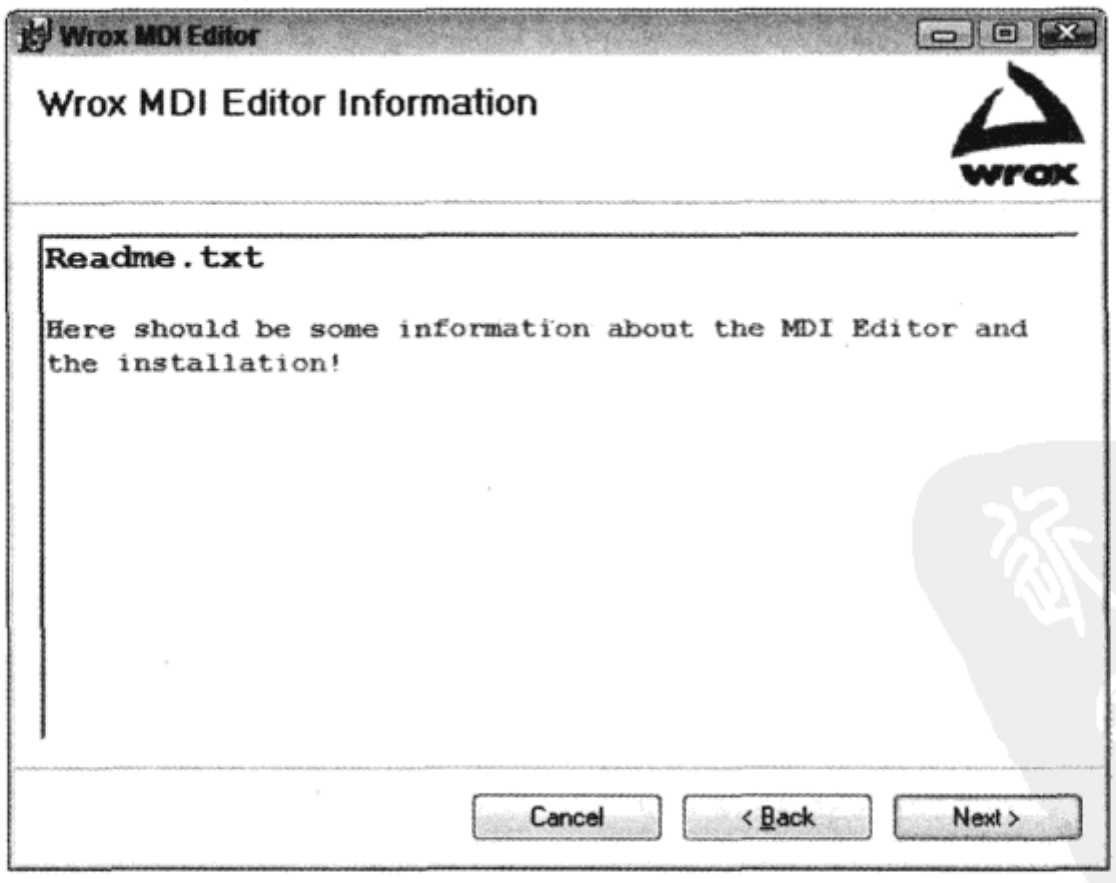


图 17-33

17.7.3　License Agreement

第 3 个对话框是 license agreement。在此仅配置了 BannerBitmap 和 LicenseFile 属性。同意此许可的单选按钮是自动添加的。如图 17-34 所示，只有选中 I Agree 按钮，才能激活 Next 按钮。这个功能是自动完成的。

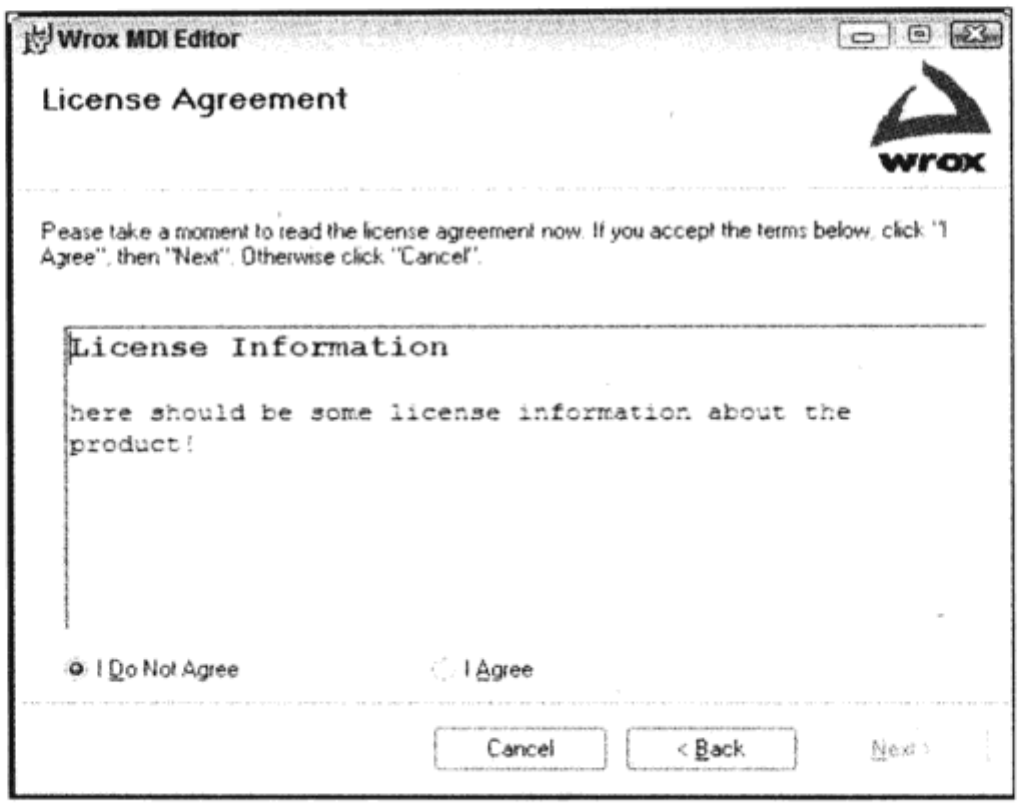


图　17-34

17.7.4　Optional Files

同意许可信息并单击 Next 按钮后，就显示 Checkboxes(A)对话框，如图 17-35 所示。在此对话框中，可以看到用属性 BannerText、BodyText 和 Checkbox1Label 定义的文本。其他复选框都不可见，因为特定 CheckboxVisible 属性设置为 false。

图　17-35

选中复选框就会在桌面上安装 demo.txt 文件。

17.7.5　选择安装文件夹

在 Select Installation Folder 对话框中，用户可以选择安装应用程序的路径，如图 17-36 所示。此对话框只允许设置属性 BannerBitmap。默认路径是[Program Files]\[manufacturer]\[Product Name]。

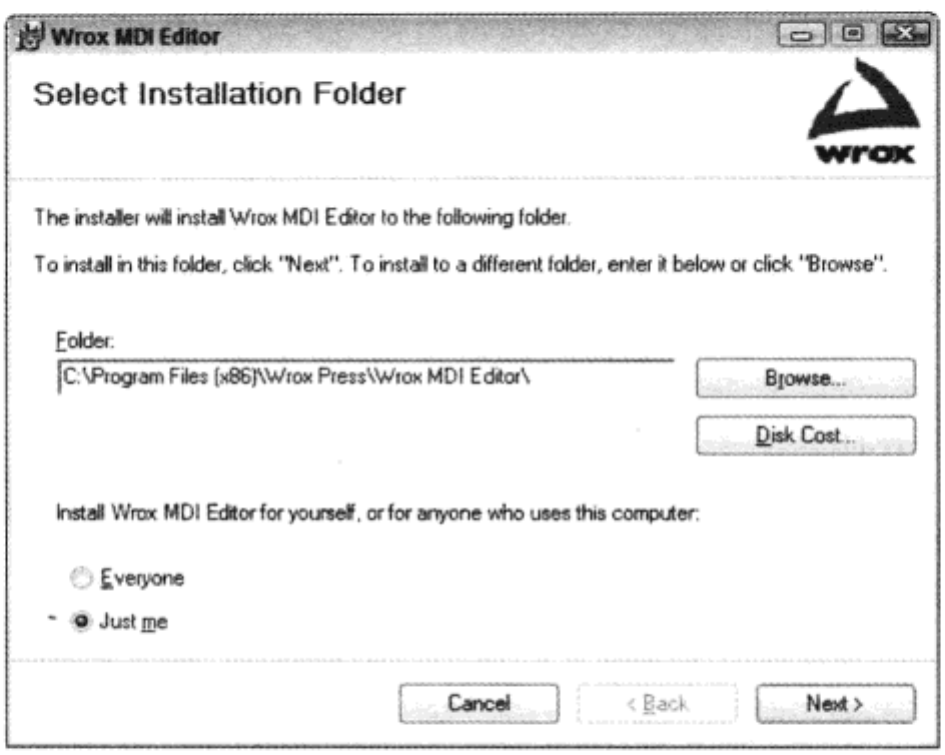


图　17-36

用户也可以指定是为所有人安装应用程序，还是只为当前登录的用户安装应用程序。根据这个选项的响应，程序文件的快捷方式会放在用户指定的目录或 All Users 目录中。

Disk Cost

单击 Disk Cost 按钮，可以打开如图 17-37 所示的对话框，其中显示了所有硬盘的磁盘空间，并计算每个磁盘所需的空间。这有助于用户选择将应用程序安装在哪个磁盘上。

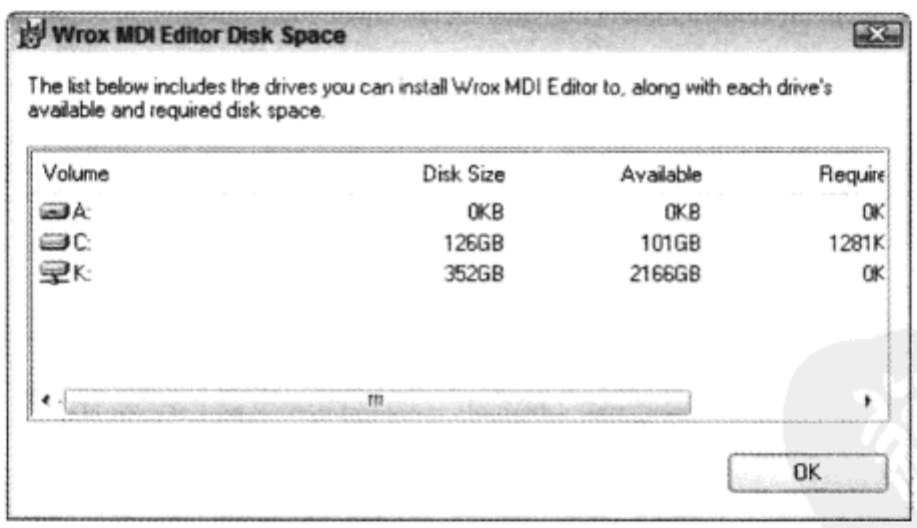


图　17-37

17.7.6　确认安装

Confirm Installation 对话框是开始安装前显示的最后一个对话框。在此不向用户询问什么问题，它仅仅是实际安装之前取消安装的最后一次机会，如图 17-38 所示。

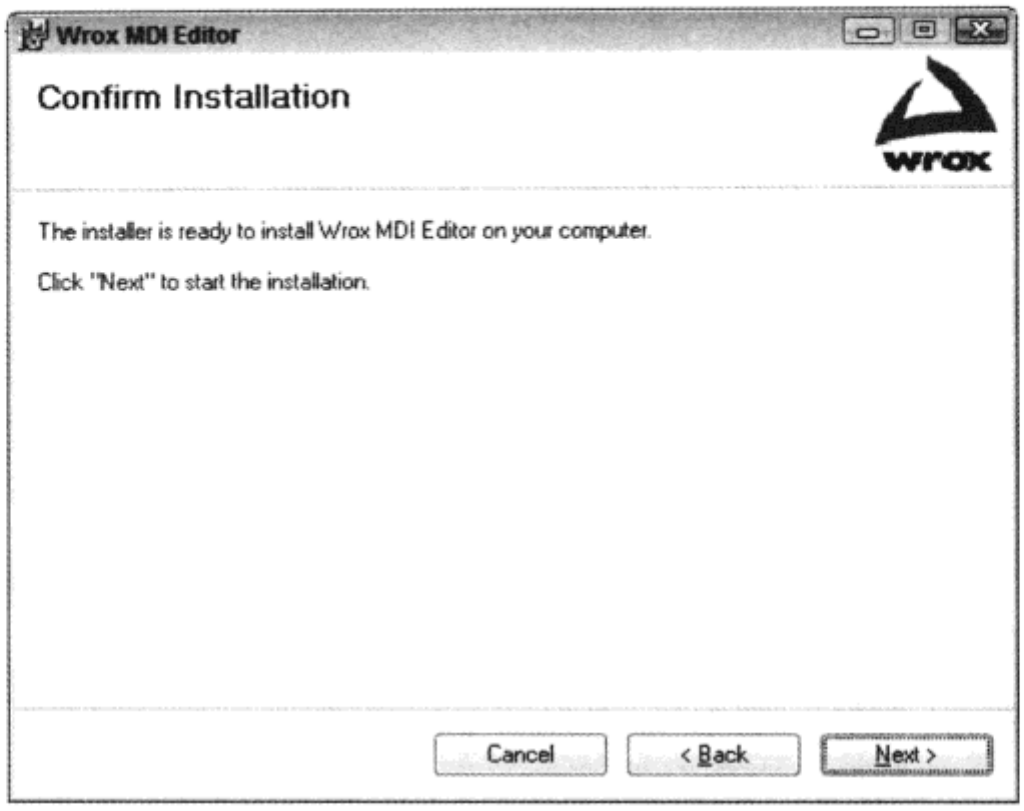


图　17-38

17.7.7　进度

Installing 对话框显示安装期间的进度控件，告诉用户安装还在进行，并大致估算安装所需要的时间。因为 MDI Editor 是一个小程序，所以此对话框很快便会结束，如图 17-39 所示。

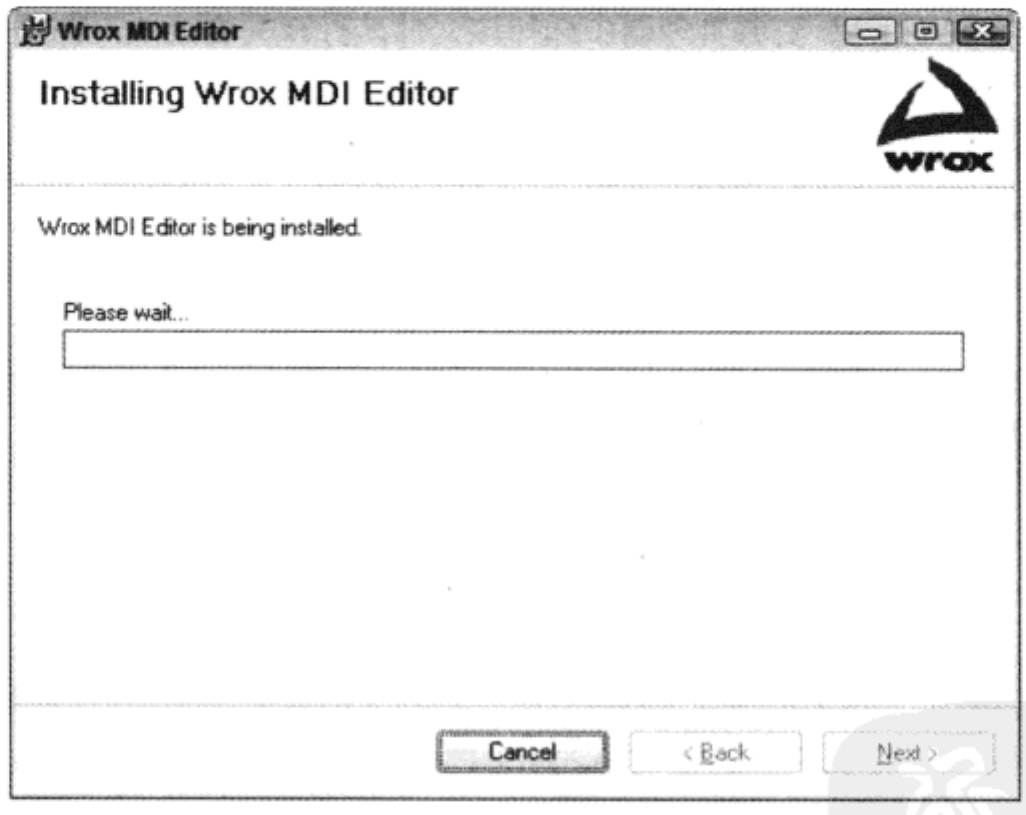


图　17-39

17.7.8　完成安装

安装成功后，会显示最后一个对话框 Installation Complete，如图 17-40 所示。

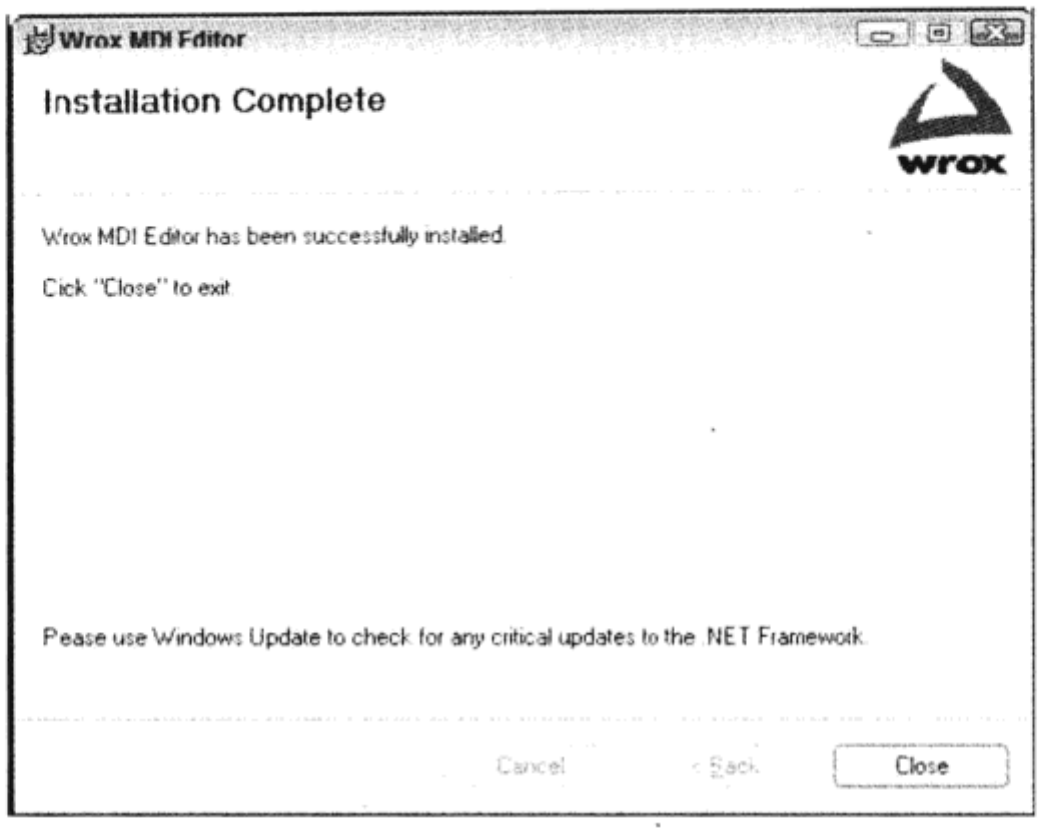


图 17-40

17.7.9 运行应用程序

选择 Start | All Programs | Wrox MDI Editor 菜单项可以启动这个编辑器。我们注册了文件扩展名，所以可以通过另一种方式启动应用程序：双击带有文件扩展名.txt 的文件。要选择用于打开文件的应用程序，应在 Windows 资源管理器中选择该文件，打开关联菜单。在关联菜单中选择 Open with，再选择需要的应用程序。要定义通过双击打开文件的应用程序，应打开关联菜单，选择 Open with，再选择 Choose default program，之后就会使用这里选择的应用程序。

如果选中了 Optional Files 对话框中的复选框，就可以在桌面上找到 demo.txt。

17.7.10 卸载

如果希望清除 Wrox MDI Editor，可以使用控制面板中的 Add / Remove Programs 命令，并在 Wrox MDI Editor 中单击 Remove 按钮。

17.8 小结

本章介绍了 ClickOnce 部署的用法和 Windows Installer 的功能，以及如何使用 Visual Studio 2010 创建 Installer 软件包。Windows Installer 使标准化的安装、卸载和修复变得更易于完成。

ClickOnce 是一种新技术，它使得安装 Windows 应用程序更容易，且不需要以系统管理员的身份进行登录。ClickOnce 提供了简单的部署和客户应用程序更新方式。

如果需要的功能比 ClickOnce 能提供的还多，就应使用 Windows 安装程序。Visual Studio 2010 安装程序在功能上受到了限制，不能提供 Windows 安装程序的全部功能，但是对于许多应用程序而言，Visual Studio 2010 安装程序提供的功能已经足够了。有几个编辑器可以配置生成的 Windows 安装程序文件。使用 File System 编辑器可以指定所有文件和快捷方式，Launch Conditions 编辑器可以定义一些必须预先安装的软件包，File Types 编辑器用于为应用程序注册文件扩展名，User Interface

编辑器允许更加方便地修改用于安装的对话框。

附录 A 给出了练习答案。

17.9　练习

(1) ClickOnce 部署有什么优点？

(2) 用 ClickOnce 清单能定义什么？

(3) 何时需要使用 Windows Installer？

(4) 在 Visual Studio 中，可以使用哪些编辑器创建 Windows 安装软件包？

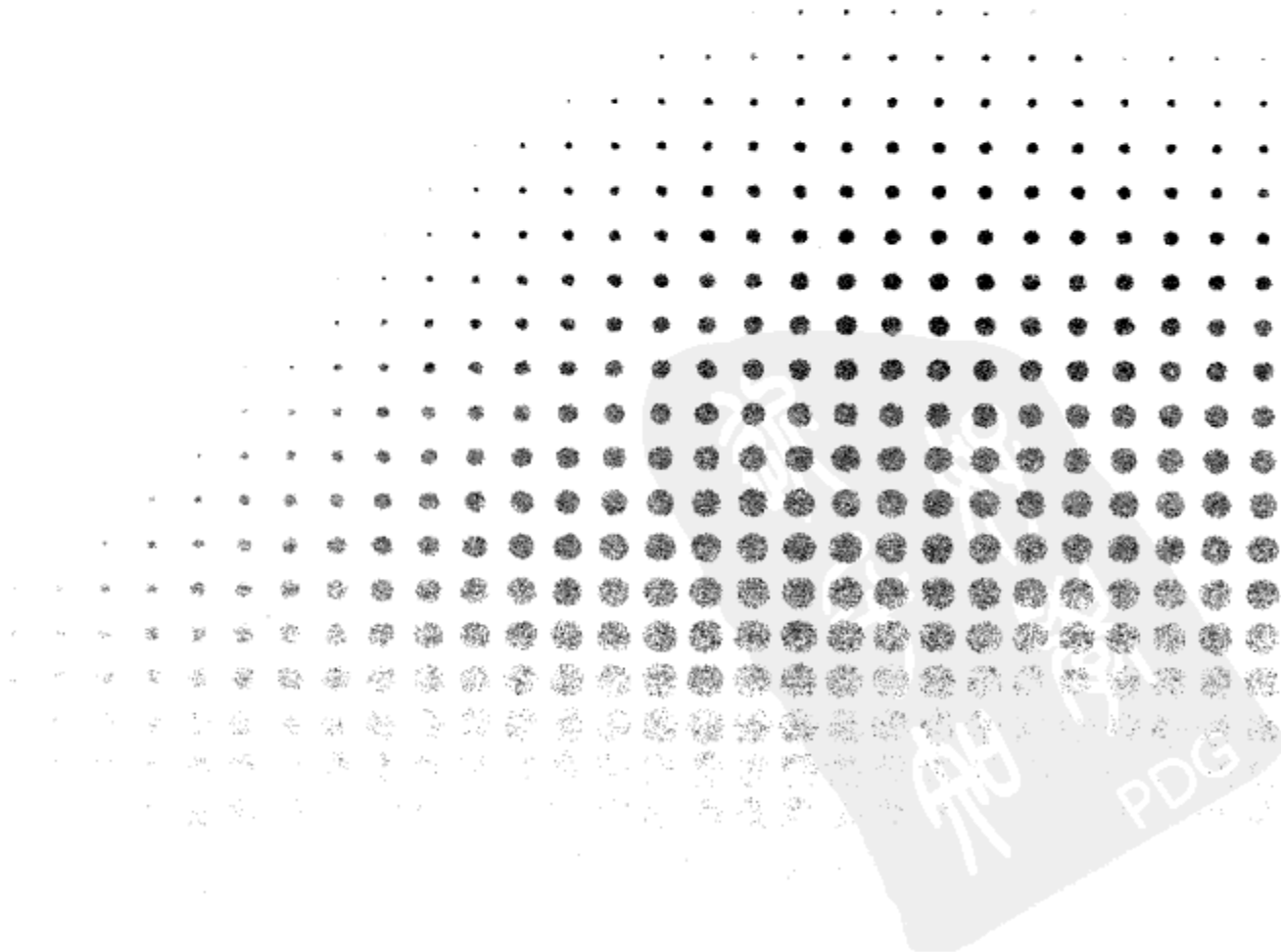
17.10　本章要点

主　题	重 要 概 念
ClickOnce	ClickOnce 可用于在没有管理权限的情况下部署应用程序。只需单击 Web 页面上的一个链接，就可以安装 Windows 窗体或 WPF 应用程序。这是 ClickOnce 的主要优点，因为它没有给 IT 管理员增加什么负担。可以在项目属性的 Publish 部分创建 ClickOnce 部署
Windows 安装程序包	Windows 安装程序允许在系统上安装共享的应用程序。使用这个技术，可以安装需要管理权限的应用程序组件。安装程序包很容易用 Visual Studio Installer 模板 Setup Project 创建
定制安装对话框	Visual Studio Setup Project 提供了一些预定义的对话框，可通过设置属性来定制安装，例如在安装过程中显示的版权文本和徽标

第III部分

Web 编 程

- 第 18 章　ASP.NET Web 编程
- 第 19 章　Web 服务
- 第 20 章　部署 Web 应用程序



PDG

第18章

ASP.NET Web 编程

本章内容：

- ASP.NET 开发概述
- 如何使用 ASP.NET 服务器控件
- 如何把 ASP.NET 回送信息发送给不同页面
- 如何创建 ASP.NET Ajax 回送
- 如何验证用户的输入
- 如何管理状态
- 如何给 Web 页面添加样式
- 如何使用母版页
- 如何实现页面的导航
- 如何验证用户的身份，以及给用户授权
- 如何读写 SQL Server 数据库

Windows 窗体是编写 Windows 应用程序的技术，而使用 ASP.NET 可以创建能在任意浏览器上显示的 Web 应用程序。使用 ASP.NET 编写 Web 应用程序的方式类似于开发 Windows 应用程序的方式，这是因为服务器端控件抽象出了 HTML 代码，并模仿了 Windows 控件的行为。当然，Windows 和 Web 应用程序仍有许多区别，因为 Web 应用程序的底层技术是 HTTP 和 HTML。

本章将概述如何用 ASP.NET 编写 Web 应用程序，如何使用 Web 控件、如何进行状态管理(这与 Windows 应用程序大相径庭)、如何进行身份验证以及如何在数据库中读写数据。

18.1 Web 应用程序概述

Web 应用程序会让 Web 服务器给客户机发送 HTML 代码。这些代码显示在 Web 浏览器，例如 Internet Explorer 中。当用户在浏览器中输入 URL 字符串时，就会把 HTTP 请求发送给 Web 服务器。HTTP 请求包含所请求的文件名和其他信息，例如标识客户应用程序的字符串、客户机支持的语言

和请求所属的其他数据。Web 服务器会返回一个 HTTP 响应，其中包含 HTML 代码。Web 浏览器将解释这些 HTML 代码，向用户显示文本框、按钮和列表。

ASP.NET 技术可用于动态创建带有服务器端代码的 Web 页面。这些 Web 页面在开发时有许多地方都类似于客户端 Windows 程序。不是直接处理 HTTP 请求和响应并手工创建发送给客户端的 HTML 代码，而是使用 TextBox、Label、ComboBox 和 Calendar 等控件创建 HTML 代码。

18.2 ASP.NET 运行库

使用 ASP.NET 在客户系统上创建 Web 应用程序，只需一个简单的 Web 浏览器。可以使用 Internet Explorer、Opera、Netscape Navigator、Firefox 或其他支持 HTML 的 Web 浏览器。客户系统不需要安装.NET。

在服务器系统上，需要 ASP.NET 运行库。如果系统上有 Internet Information Services (IIS)，就会在安装.NET Framework 时为服务器配置 ASP.NET 运行库。在开发过程中，不需要 IIS，因为 Visual Studio 发布了自己的 ASP.NET Web Development Server，可以用于测试和调试 Web 应用程序。

下面讨论浏览器上的典型 Web 请求，以说明 ASP.NET 运行库的工作原理，如图 18-1 所示。客户机向服务器请求一个文件，如 default.aspx。所有的 ASP.NET Web 页面通常带有扩展名.aspx。因为这个文件扩展名是用 IIS 注册的，或者 ASP.NET Web Development Server 能识别它，所以 ASP.NET 运行库和 ASP.NET 辅助进程(worker process)就会开始工作。对文件 default.aspx 的第一次请求会启动 ASP.NET 分析器，编译器会把该文件和一个与.aspx 文件相关的 C#文件一起编译，创建一个程序集。然后.NET 运行库的 JIT 编译器把程序集编译为本机代码。该程序集包含一个 Page 类，调用它会把 HTML 代码返回给客户端。之后删除 Page 对象。但是，会保留程序集，用于以后的请求，所以在第二个请求中，不需要再次编译程序集。

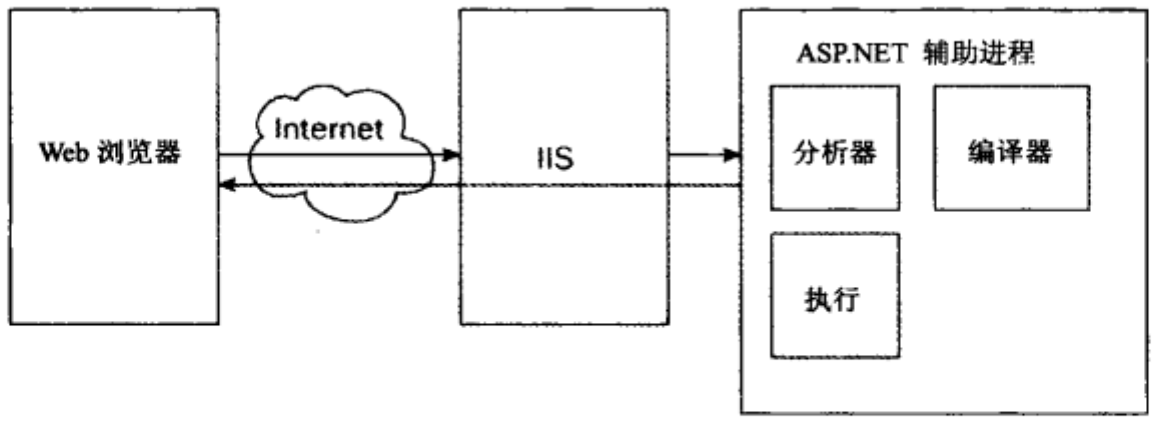


图 18-1

18.3 创建简单的 Web 页面

下面的示例将创建一个简单的 Web 页面。在本章和第 19 章介绍的示例应用程序中，将创建一个简单的 Event 网站，网友可以在这个网站上注册事件。

试一试：创建一个简单的 Web 页面

(1) 在 Visual Studio 中选择 File | New | Project，创建一个新的 Web 项目，如图 18-2 所示。在 New

Project 对话框中，选择 Visual C#类别，再选择子类别 Web，最后选择 ASP.NET Empty Web Application 模板，将项目命名为 EventRegistrationWeb。

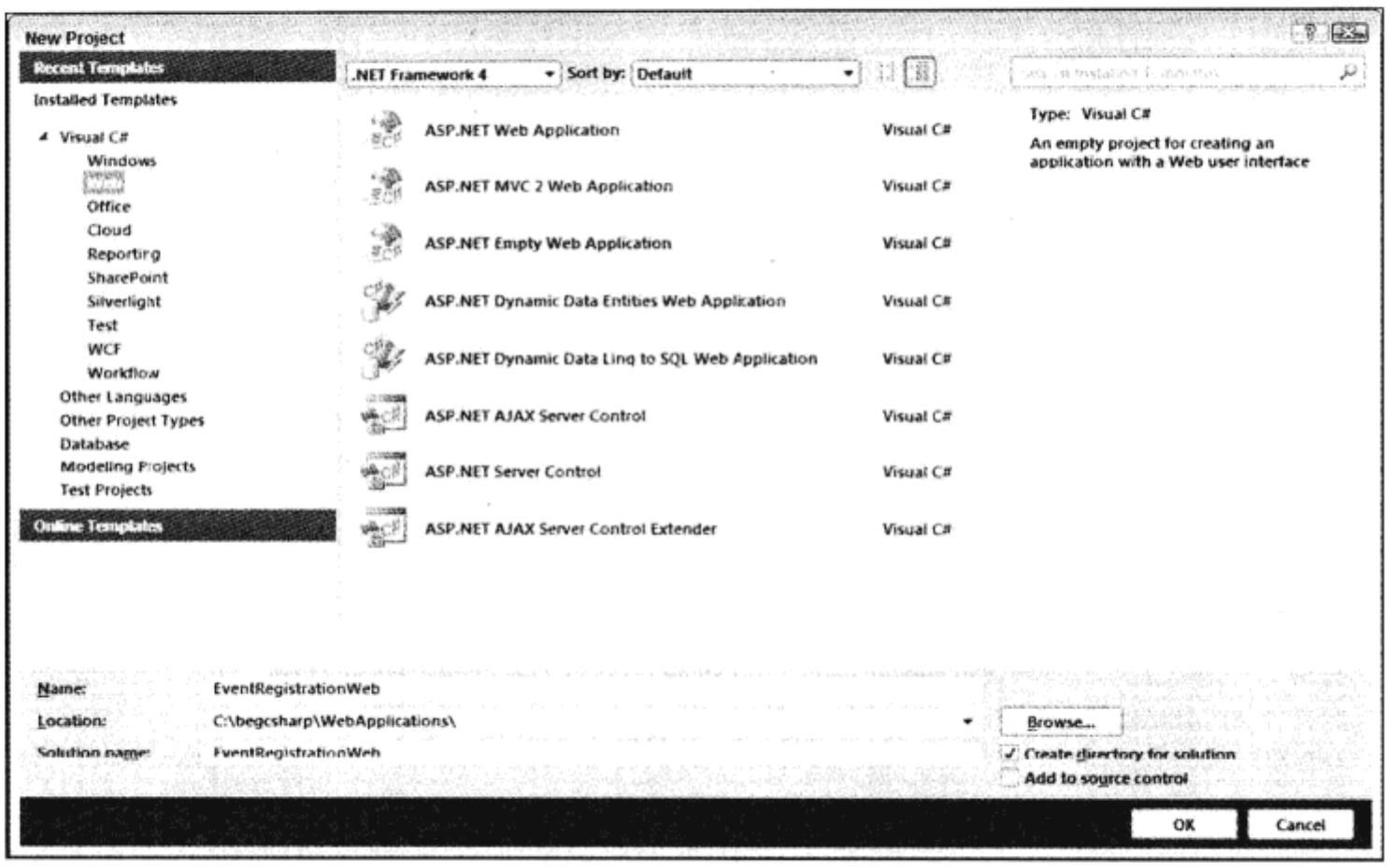


图　18-2

(2) 创建 Web 项目后，就使用菜单 Project | Add New Item，选择 Web Form 模板，创建一个新的 Web 页面，命名为 Registration.aspx，如图 18-3 所示。

图　18-3

(3) 把控件放在一个表格中是比较有效的。单击进入设计视图，选择 Table | Insert Table 菜单，添加一个表格。在 Insert Table 对话框中，设置 5 行 2 列，如图 18-4 所示。

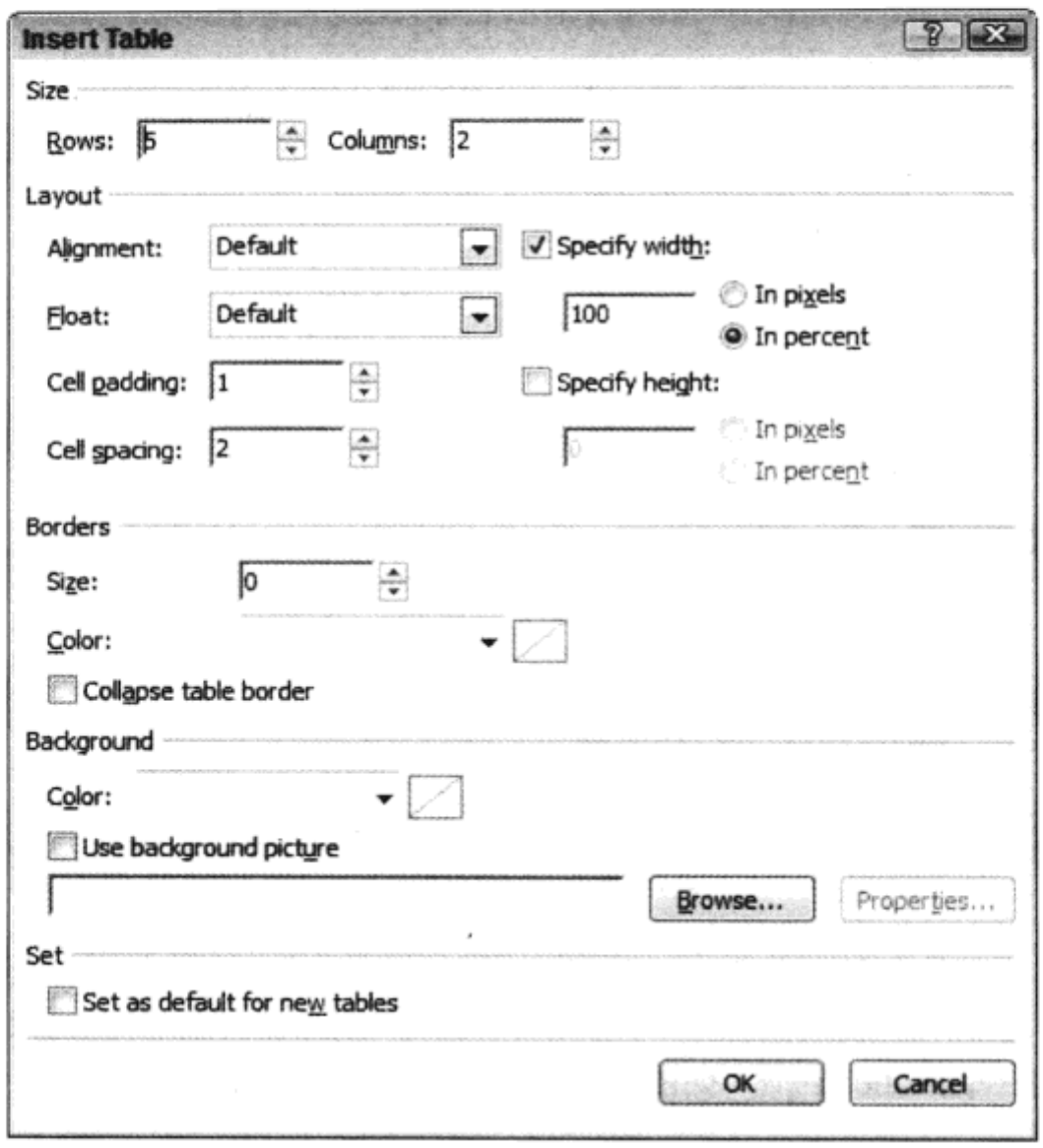


图 18-4

(4) 在表格中添加 4 个标签、3 个文本框、1 个下拉列表和 1 个按钮，如图 18-5 所示。

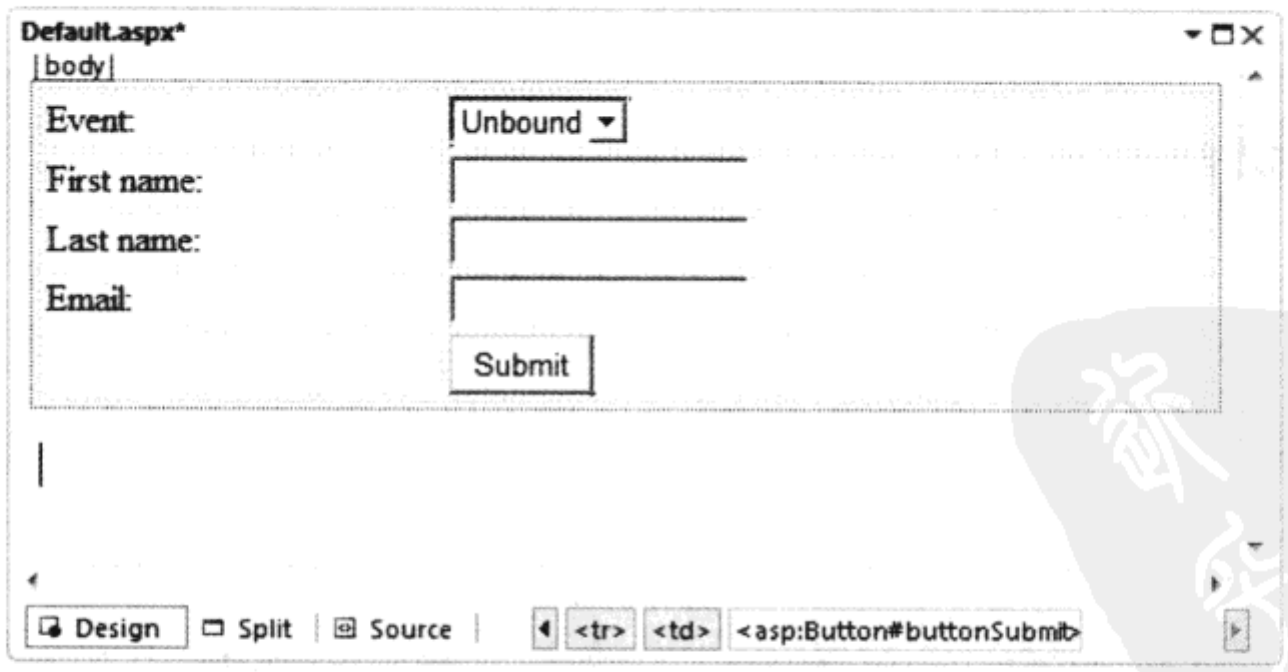


图 18-5

(5) 设置控件属性，如表 18-1 所示。

表　18-1

控 件 类 型	(ID)	文　本
Label	labelEvent	Event:
Label	labelFirstname	Firstname:
Label	labelLastname	Lastname:
Label	labelEmail	Email:
DropDownList	dropDownListEvents	
TextBox	textFirstName	
TextBox	textLastName	
TextBox	textEmail	
Button	buttonSubmit	Submit

(6) 在下拉列表中，选择 Properties 窗口中的 Items 属性，在 ListItem Collection Editor 中输入字符串 Introduction to ASP.NET、Introduction to Windows Azure 和 Take off to .NET 4.0，如图 18-6 所示。

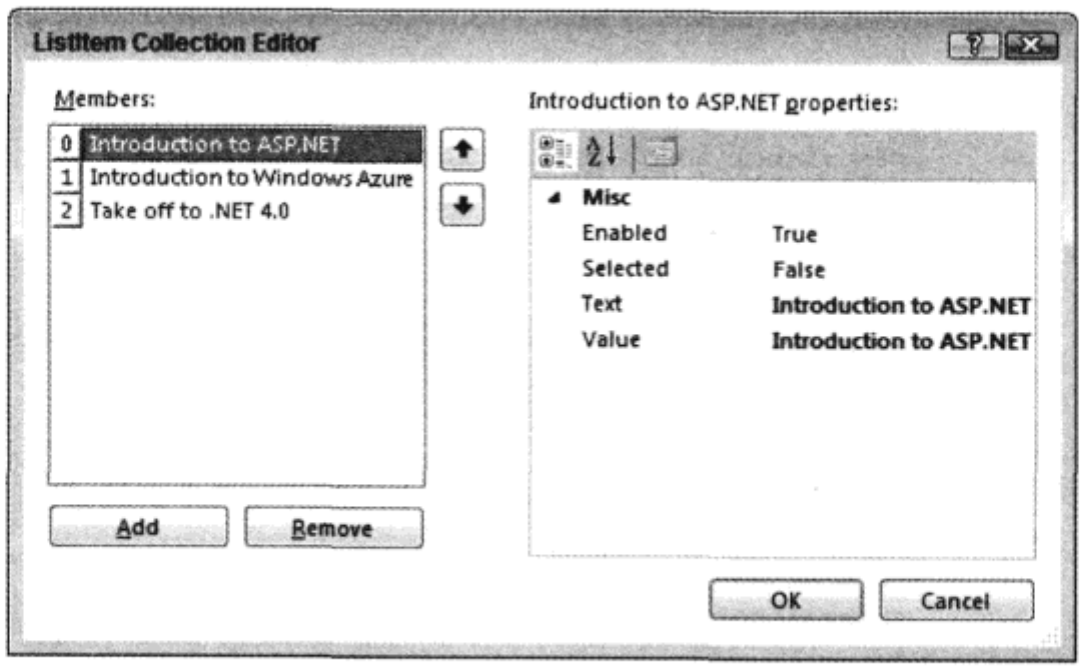


图　18-6

(7) 把编辑器切换到源代码视图，生成的代码如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
        Inherits="EventRegistrationWeb.Registration" %>

<!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>
   <style type="text/css">
      .style1
      {
```

```
            width: 100%;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <table class="style1">
            <tr>
                <td>
                    <asp:Label ID="labelEvent" runat="server" Text="Event:">
                    </asp:Label>
                </td>
                <td>
                    <asp:DropDownList ID="dropDownListEvents" runat="server">
                        <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
                        <asp:ListItem>Introduction to Windows Azure</asp:ListItem>
                        <asp:ListItem>Take off to .NET 4.0</asp:ListItem>
                    </asp:DropDownList>
                </td>
            </tr>
            <tr>
                <td>
                    <asp:Label ID="labelFirstName" runat="server"
                        Text="First name:"></asp:Label>
                </td>
                <td>
                    <asp:TextBox ID="textFirstName" runat="server"></asp:TextBox>
    </td>
</tr>
<tr>
    <td>
        <asp:Label ID="labelLastName" runat="server" Text="Last name:">
        </asp:Label>
    </td>
    <td>
        <asp:TextBox ID="textLastName" runat="server"></asp:TextBox>
    </td>
</tr>
<tr>
    <td>
            <asp:Label ID="labelEmail" runat="server" Text="Email:">
            </asp:Label>
        </td>
     <td>
        <asp:TextBox ID="textEmail" runat="server"></asp:TextBox>
    </td>
</tr>
<tr>
    <td>
```

```
                 </td>
            <td>
                <asp:Button ID="buttonSubmit" runat="server" Text="Submit" />
            </td>
        </tr>
    </table>
</div>
</form>
</body>
</html>
```

代码段 Registration.aspx

(8) 在启动应用程序之前，先进入项目属性，打开 Web 设置，如图 18-7 所示。验证 Start Action 设置为当前页面，在 Servers 组，验证配置了 Visual Studio Development Server。

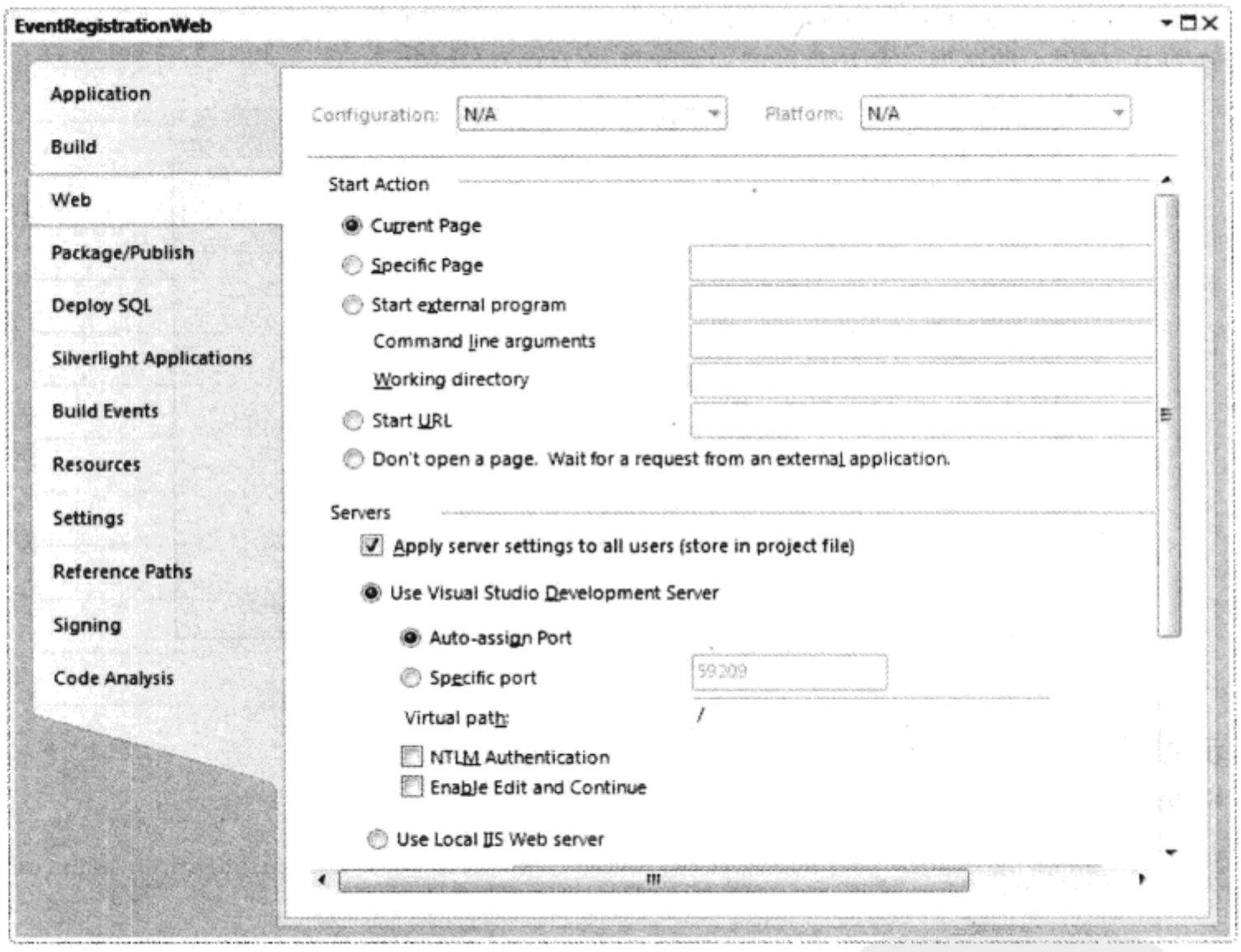


图　18-7

(9) 在编辑器中再次打开 Registration.aspx 文件。选择 Debug | Start Without Debugging 选项，启动 Web 应用程序。此时，会自动启动 ASP.NET Development Server。Windows Explorer 任务栏中有 ASP.NET Development Server 的图标。双击这个图标，就会打开如图 18-8 所示的对话框。该对话框显示了 Web 服务器的物理路径和虚拟路径，以及 Web 服务器监听的端口。该对话框还可以用于停止 Web 服务器。

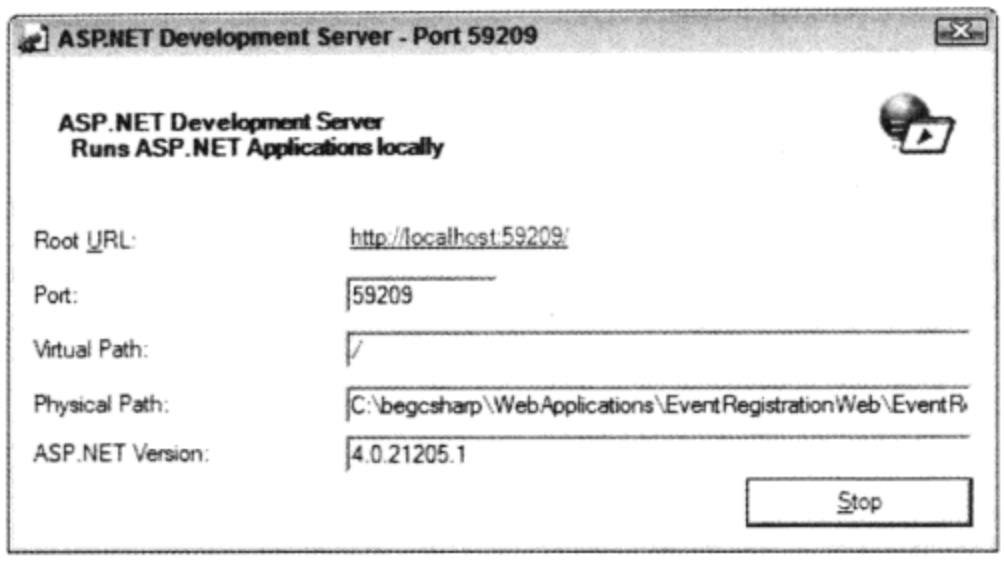


图 18-8

启动应用程序，Internet Explorer 会显示 Web 页面，如图 18-9 所示。选择 View | Source 就可以查看 HTML 代码。服务器端的控件此时已转换为纯 HTML 代码。

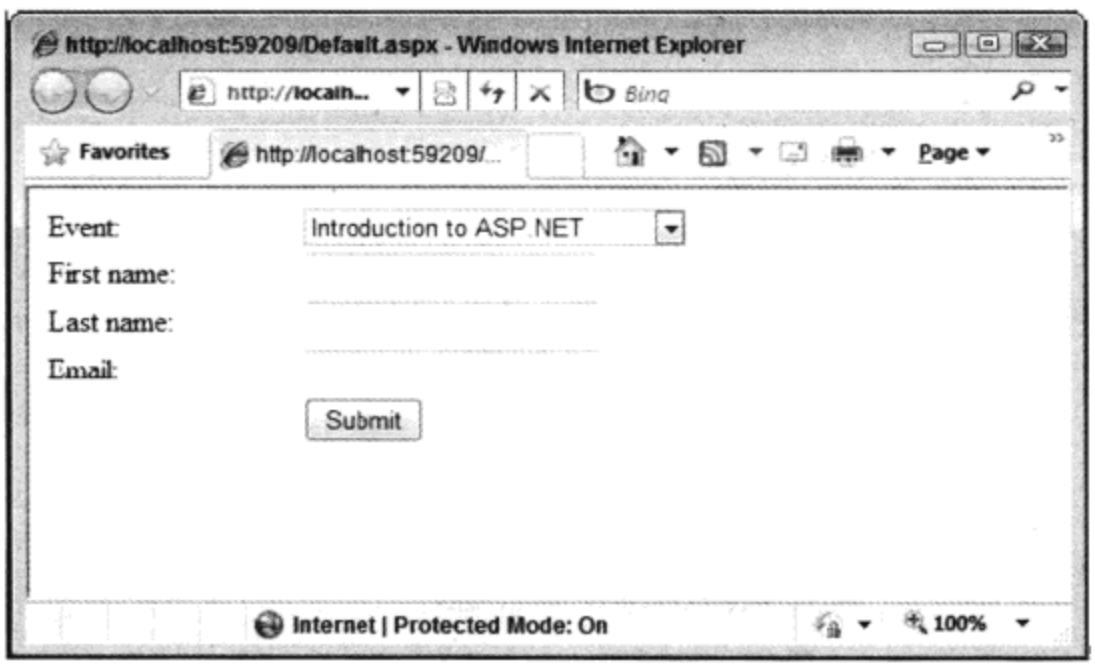


图 18-9

示例的说明

Registration.aspx 文件的第一行是 Page 指令：

可从 wrox.com 下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
        Inherits="EventRegistrationWeb.Registration" %>
```

代码段 Registration.aspx

这个指令定义了要使用的编程语言和类。属性 AutoEventWireup ="true"表示，页面的事件处理程序自动链接到特定方法名上，如稍后所示。Inherits="EventRegistrationWeb.Registration"表示在 ASPX 文件中动态生成的类派生于基类 Registration。这个基类位于用 CodeFile 属性定义的代码隐藏文件 Registration.aspx.cs 中。本章将在后面给.cs 文件添加处理代码。生成的代码隐藏文件 Registration.aspx.cs 如下所示：

可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
    public partial class Registration : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}
```

代码段 Registration.aspx.cs

上面代码中使用的 partial 关键字详见第 10 章。

下面是 ASPX 页面的代码，客户端接收简单的 HTML 代码。页面发送给客户端时，只是从<head>标记中删除了 runat="server"特性。

可从
wrox.com
下载源代码

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title></title>
    <style type="text/css">
      .style1
     {
        width: 100%;
     }
   </style>
</head>
<body>
```

代码段 Registration.aspx

还有一些带有 runat="server"特性的 HTML 元素，如<form>元素。通过 runat="server"特性，ASP.NET 服务器控件就会与 HTML 标记关联起来。这个控件可以用于写入服务器端代码。在<form>元素的后面是 System.Web.UI.HtmlControls.HtmlForm 类型的一个对象，该对象有一个用 id 特性定义的变量名 form1。form1 可以用于调用 HtmlForm 类的方法和属性。

HtmlForm 对象创建一个发送给客户端的<form>标记。

可从
wrox.com
下载源代码

```
<form id="form1" runat="server">
```

代码段 Default.aspx

当然，不会将 runat 特性发送给客户端。

从工具箱拖放到窗体设计器上的标准控件拥有以<asp:开头的元素<asp:Label>和<asp:DropDownList>。它们是服务器端的 ASP.NET Web 控件，它们与 System.Web.UI.WebControls 名称空间中的.NET 类相关。<asp:Label>由 Label 类表示，<asp:DropDownList>由 DropDownList 类表示。

可从 wrox.com 下载源代码

```
<td>
    <asp:LabelID="labelEvent"runat="server"Text="Event:"></asp:Label>
</td>
<td>
    <asp:DropDownList ID="dropDownListEvents" runat="server">
        <asp:ListItem>SQL Server 2008 and XML</asp:ListItem>
        <asp:ListItem>Office 2007 and XML</asp:ListItem>
        <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
    </asp:DropDownList>
</td>
```

代码段 Registration.aspx

<asp:Label>不向客户端发送<asp:Label>元素，因为它不是一个有效的 HTML 元素，<asp:Label>返回一个标记。同样，<asp:DropDownList>返回<select>元素，<asp:TextBox>返回<input type="text">元素。

ASP.NET 在名称空间 System.Web.UI.HtmlControls 和 System.Web.UI.WebControls 中有 UI 控件类。这两个名称空间有一些类似的控件，也称为 HTML 服务器控件和 Web 服务器控件。例如，HtmlInputText 是 HTML 服务器控件，TextBox 是 Web 服务器控件。HTML 服务器控件提供的方法和属性类似于 HTML 控件，但在 Web 服务器控件中，有更复杂的控件，如 Calendar、DataGrid 和 Wizard。如果不需要在服务器端代码中给控件编程，只需在 JavaScript 中给它编程，就可以只使用 HTML 控件，而不添加 runat="server"特性。

18.4 服务器控件

表 18-2 列出了 ASP.NET 提供的一些主要 Web 服务器控件，以及这些控件返回的 HTML 代码。

表 18-2

控 件	HTML	说 明
Label		返回一个包含文本的 span 元素
Literal	static text	返回简单的静态文本。使用 Literal 控件，可以根据客户应用程序转换内容
TextBox	<input type="text">	返回 HTML <input type="text">，在该元素中用户可以输入一些值。当文本改变时可以编写服务器端事件处理程序
Button	<input type="submit">	把窗体值发送给服务器

(续表)

控　件	HTML	说　明
LinkButton	<a href="javascript: dopostback()">	创建一个 anchor 标记，其中包含给服务器回送的 JavaScript
ImageButton	<input type="image">	生成 image 类型的 input 标记，显示引用的图像
HyperLink	<a>	创建一个简单的 anchor 标记，来引用 Web 页面
DropDownList	<select>	创建一个 select 标记，用户能看到其中的一项，并可以单击下拉选择框，从多个项中选择一项
ListBox	<select size="">	创建一个带有 size 特性的 select 标记，可以一次显示多个项
CheckBox	<input type="checkbox">	返回一个 check box 类型的 input 元素，显示一个可以选中或取消选中的按钮。如果不使用 CheckBox，还可以使用 CheckBoxList 创建一个表格，其中包含多个 check box 元素
RadioButton	<input type="radio">	返回一个 radio 类型的 input 元素。使用单选按钮时，只能选择组中的一个按钮。与 CheckBoxList 类似，RadioButtonList 也提供了一个按钮列表
Image	<img src="">	返回一个 img 标记，在客户端上显示 GIF 或 JPG 文件
Calendar	<table>	显示完整的日历，用户可以在日历中选择日期，修改月份等。对于输出，会生成带有 JavaScript 代码的 HTML 表
TreeView	<table>	返回一个 div 标记，其中根据内容包含多个 table 标记。JavaScript 用于打开和关闭客户端上的树

18.5　ASP.NET 回送

Web 服务器控件可以包含服务器上调用的事件处理程序。Button 控件可以包含 Click 事件，DropDownList 控件则提供了 SelectedIndexChanged 事件，TextBox 提供了 TextChanged 事件。

只有进行回送时，才在服务器上触发事件。文本框中的值改变时，TextChanged 事件不会立即触发，只有单击 Submit 按钮，提交了窗体，并发送给服务器，才会触发 TextChanged 事件。ASP.NET 运行库会验证控件的状态已改变，并调用相应的事件处理程序。如果 DropDownList 的选项改变了，就调用 SelectedIndexChanged 事件，如果文本框的值改变了，就调用 TextChanged 事件。

如果希望把更改事件立即传送给服务器(例如，改变了 DropDownList 的选项)，可以把 AutoPostback 属性设置为 true。这样就会使用客户端的 JavaScript 把窗体数据立即提交给服务器。当然，网络通信量也会增加。使用这个功能时要小心。

页面返回给服务器后，比较控件的新值和旧值是由 View State 完成的。View State 是一个隐藏字段，它会同页面的内容一起发送给浏览器。当把页面发送给客户端时，View State 会包含与窗体中

控件相同的值。向服务器回送时，View State 也会同控件的新值一起发送给服务器。这样它就可以验证值是否改变，并调用事件处理程序。

前面的示例应用程序只给客户端发送了一个简单的页面。下面需要处理用户输入的结果。在第一个示例中，用户输入显示在同一个页面中，然后使用另一个页面。下面的示例将显示用户输入。

试一试：显示用户输入

(1) 在 Visual Studio 中，打开前面创建的 Web 应用程序 EventRegistrationWeb。

(2) 要显示用户输入，以便进行事件注册，给 Web 页面 Registration.aspx 添加一个标签 labelResult，清空这个标签的 Text 属性。

(3) 双击 Submit 按钮，给它添加 Click 事件处理程序，在 Registration.aspx.cs 文件中给处理程序添加如下代码：

可从 wrox.com 下载源代码

```
protected void buttonSubmit_Click(object sender, EventArgs e)
{
   string selectedEvent = dropDownListEvents.SelectedValue;
   string firstname = textFirstname.Text;
   string lastname = textLastname.Text;
   string email = textEmail.Text;
   labelResult.Text = String.Format("0) {1} selected the event {2}",
         firstName, lastName, selectedEvent);
}
```

代码段 Registration.aspx.cs

(4) 再次使用 Visual Studio 启动 Web 页面。输入数据，单击 Submit 按钮后，该页面会在新标签中显示用户输入。

示例的说明

双击 Submit 按钮，在 Registration.aspx 文件中给<asp:Button>元素添加 OnClick 特性：

可从 wrox.com 下载源代码

```
<asp:Button ID="buttonSubmit" Runat="server" Text="Submit"
      OnClick="buttonSubmit_Click" />
```

代码段 Registration.aspx

对于 Web 服务器控件，OnClick 定义了服务器端的 Click 事件，在单击按钮时，就会调用该事件。

在 buttonSubmit_Click()方法的实现代码中，控件的值可以使用属性来读取。dropDownList Events 是引用 DropDownList 控件的变量。在 ASPX 文件中，ID 设置为 dropDownListEvents，所以会自动创建变量。SelectedValue 属性返回当前的选择。对于 TextBox 控件，Text 属性返回用户输入的字符串。

可从 wrox.com 下载源代码

```
string selectedEvent = dropDownListEvents.SelectedValue;
string firstName = textFirstName.Text;
string lastName = textLastName.Text;
string email = textEmail.Text;
```

代码段 Registration.aspx.cs

labelResult 标签的 Text 属性设置了结果：

```
labelResult.Text = String.Format("{0} {1} selected the event {2}",
                                 firstName, lastName, selectedEvent);
```

代码段 Registration.aspx.cs

可从 wrox.com 下载源代码

ASP.NET 不是在同一个页面上显示结果，而是在另一个页面上显示，如下面的示例所示。

试一试：在第二个页面上显示结果

(1) 创建一个新的 WebForm，命名为 ResultsPage.aspx。

(2) 给 ResultsPage 添加一个标签 labelResult。

(3) 给 ResultsPage 类的 Page_Load 方法添加代码，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
    public partial class ResultsPage : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            try
            {
                DropDownList dropDownListEvents =
                    (DropDownList)PreviousPage.FindControl("dropDownListEvents");
                string selectedEvent = dropDownListEvents.SelectedValue;
                string firstName = ((TextBox)PreviousPage.FindControl(
                                    "textFirstName")).Text;
                string lastName = ((TextBox)PreviousPage.FindControl(
                                    "textLastName")).Text;
                string email = ((TextBox)PreviousPage.FindControl(
                                    "textEmail")).Text;
                labelResult.Text = String.Format("{0} {1} selected the event {2}",
                    firstName, lastName, selectedEvent);
            }
            catch
            {
                labelResult.Text = "The originating page must contain " +
                    "textFirstName, textLastName, textEmail controls";
            }
        }
    }
}
```

代码段 ResultsPage.aspx.cs

(4) 把 Registration.aspx 页面上 Submit 按钮的 PostbackUrl 属性设置为 ResultsPage.aspx。

(5) 可以删除 Submit 按钮的 Click 事件处理程序，因为不再需要它了。

(6) 启动 Default.aspx 页面，填充一些数据，单击 Submit 按钮，就会重定向到 ResultsPage.aspx 页面，其中显示了输入的数据。

示例的说明

在 ASP.NET 中，Button 控件有一个新属性 PostbackUrl，定义了应从 Web 服务器上请求的页面。这个属性创建客户端 JavaScript 代码，用 Submit 按钮的客户端 onclick 处理程序请求所定义的页面。

```
<input type="submit" name="buttonSubmit" value="Submit"
  onclick="javascript:webForm_DoPostBackWithOptions(
    new WebForm_PostBackOptions(&quot;buttonSubmit&quot;,
      &quot;&quot;, false, &quot;&quot;, &quot;ResultPage.aspx&quot;,
      false, false))" id="buttonSubmit" />
```

浏览器把第一个页面中窗体的所有数据都发送到新页面上，但是，在新请求的页面上，需要从前面页面定义的控件中获取数据。为了访问前面页面中的控件，Page 类定义了属性 PreviousPage。PreviousPage 返回一个 Page 对象，这个页面的控件可以使用 FindControl()方法来访问。FindControl()定义为返回一个 Control 对象，所以必须把返回值的类型转换为所搜索的控件类型。

可从 wrox.com 下载源代码

```
DropDownList dropDownListEvents =
    (DropDownList)PreviousPage.FindControl("dropDownListEvents");
```

代码段 ResultsPage.aspx.cs

在开发过程中，如果不使用 FindControl()方法访问前面页面的值，就可以把对前面页面的访问强类型化，以减少错误。为此，应使用 default_aspx 类的一个属性返回一个定制结构，如下面的示例所示。

试一试：创建强类型化的 PreviousPage

(1) 选择 Project | Add New Class，为项目添加一个新类，命名为 RegistrationInfo。

(2) 实现 RegistrationInfo 类，如下所示：

可从 wrox.com 下载源代码

```
public class RegistrationInfo
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string SelectedEvent { get; set; }
}
```

代码段 RegistrationInfo.cs

(3) 在 Registration.aspx.cs 文件中给 Registration 类添加公共属性 RegistrationInfo：

可从 wrox.com 下载源代码

```
public RegistrationInfo RegistrationInfo
{
    get
    {
        return new RegistrationInfo
        {
            FirstName = textFirstName.Text,
            LastName = textLastName.Text,
            Email = textEmail.Text,
```

```
                SelectedEvent = dropDownListEvents.SelectedValue
            };
        }
    }
```

代码段Registration.aspx.cs

(4) 在 ResultPage.aspx 文件的 Page 指令下面添加 PreviousPageType 指令：

```
<%@ Page Language="C#"AutoEventWireup="true"CodeFile="ResultPage.aspx.cs"
   Inherits="ResultPage_aspx" %>
<%@ PreviousPageType VirtualPath="~/Registration.aspx" %>
```

代码段ResultPage.aspx

(5) 在 ResultsPage 类的 Page_Load()方法中，代码可以简化为：

```
protected void Page_Load(object sender, EventArgs e)
{
   try
   {
      RegistrationInformation ri = PreviousPage.RegistrationInformation;

          labelResult.Text = String.Format("{0} {1} selected the event {2}",
             ri.FirstName, ri.LastName, ri.SelectedEvent);
   }
   catch
   {
      labelResult.Text = "The originating page must contain " +
           "textFirstName, textLastName, textEmail controls";
   }
}
```

代码段ResultPage.aspx.cs

示例的说明

PreviousPageType 指令创建了一个 PreviousPage 类型的属性，它返回与该指令关联的类型。在其实现代码中，调用了基类的 PreviousPage 属性，如下面的代码所示：

```
public new EventRegistrationWeb.Default PreviousPage {
   get {
      return ((EventRegistrationWeb.Default)(base.PreviousPage));
   }
}
```

这里没有使用 PreviousPageType 指令的 VirtualPath 特性定义上一个页面的类型，而使用了 TypeName 特性。如果前面有多个页面，就可以使用这个特性。此时，需要为前面的所有页面定义一个基类，并把该基类赋予 TypeName 特性。

18.6 ASP.NET AJAX 回送

在一般的 ASP.NET 回送中，会请求整个页面。回送用户已经加载的同一个页面时，也会再次返回整个页面。为了减少网络上的传输量，可以使用 ASP.NET Ajax 回送。在 Ajax 回送中，只使用 JavaScript 返回并刷新页面的一部分，使用 UpdatePanel 可以方便地做到这一点。

为了便于比较 ASP.NET 回送和 ASP.NET Ajax 回送，下面的示例将当前时间输出到两个标签中，其中一个标签在 UpdatePanel 中，另一个不在 UpdatePanel 中。

试一试：使用 UpdatePanel 控件

(1) 使用 Visual Studio 打开前面创建的项目 EventRegistrationWeb。

(2) 给现有的网站添加一个新 AJAX Web 窗体 UpdatePanelDemo.aspx。

(3) 在工具箱的 AJAX Extensions 类别中，给页面添加一个 UpdatePanel 控件。

(4) 在 UpdatePanel 控件内部添加一个标签和一个按钮，在 UpdatePanel 控件外部再添加一个标签和一个按钮，把 UpdatePanel 控件内部和外部的按钮的 Text 属性分别设置为 AJAX Postback 和 ASP.NET Postback。

可从 wrox.com 下载源代码

```
<form id="form1" runat="server">
<div>
    <asp:ScriptManager ID="ScriptManager1" runat="server">
    </asp:ScriptManager>
</div>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:Label ID="Label1"runat="erver"Text="Label"> </asp:Label>
        <asp:Button ID="Button1"runat="server" Text="AJAX Postback"/>
           OnClick="OnButtonClick" />
    </ContentTemplate>
</asp:UpdatePanel>
<asp:Label ID="Label2" runat="server" Text="Label"> </asp:Label>
<asp:Button ID="Button2" runat="server" Text="ASP.NET Postback"/>
</form>
```

代码段 UpdatePanelDemo.aspx

(5) 给两个按钮指定 Click 事件处理程序 OnButtonClick()，其实现代码如下所示：

可从 wrox.com 下载源代码

```
protected void OnButtonClick(object sender, EventArgs e)
{
    DateTime now = DateTime.Now;
    Label1.Text = now.ToLongTimeString();
    Label2.Text = now.ToLongTimeString();
}
```

代码段 UpdatePanelDemo.aspx.cs

(6) 启动应用程序，单击两个按钮。单击 AJAX Postback 按钮仅刷新第一个标签。单击 ASP.NET Postback 会刷新整个页面，如图 18-10 所示。

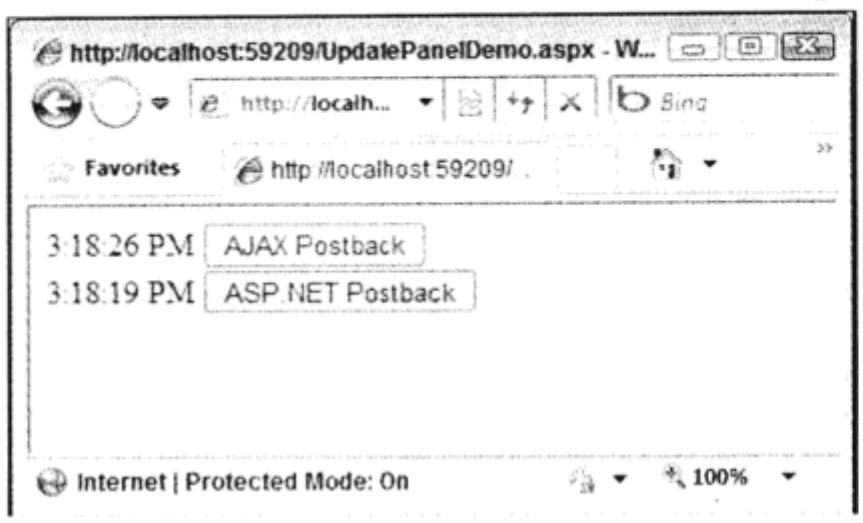


图　18-10

示例的说明

在 ASP.NET AJAX 页面上，需要一个 ScriptManager 对象。这个对象使用 AJAX Web Form 模板添加。ScriptManager 类加载了包含几个功能的 JavaScript 函数。也可以使用这个类加载自己的定制脚本。ScriptManager 的属性如表 18-3 所示。

表　18-3

属　性	说　明
EnablePageMethods	指定在 ASPX 页面上定义的公共静态方法是否可以从客户脚本中作为 Web 服务方法调用
EnablePartialRendering	为了启用 UpdatePanel 的部分呈现功能，这个属性必须设置为 true(默认)
LoadScriptsBeforeUI	指定脚本放在所返回的 HTML 页面的什么地方。把它们放在<head>元素内部，脚本就会在加载 UI 之前加载
ScriptMode	指定应使用脚本的调试版本还是发布版本
ScriptPath	指定定制脚本所在的目录的根路径
Scripts	包含应呈现在客户端的定制脚本文件集合
Services	包含可以从客户脚本中调用的 Web 服务引用集合

页面上的 ASP.NET 控件 Button 让客户机创建 HTML Submint 按钮。Button2 给服务器发送一个一般的 HTTP POST 请求。Button1 在 UpdatePanel 的内部，所以客户脚本关联上该按钮的 Click 事件，发出一个 Ajax POST 请求。Ajax POST 请求使用 XmlHttpRequest 对象给服务器发送一个请求。服务器只返回更新 UI 所需的数据。解释了数据后，JavaScript 代码就修改 UpdatePanel 内部的 HTML 控件，显示一个新的 UI。

一个页面上可以有多个 UpdatePanel。只需在一个页面上添加多个 UpdatePanel，每个 UpdatePanel 就会在 Ajax POST 请求时更新。更新可以用触发器控制，如下面的示例所示。

试一试：使用触发器更新面板

(1) 在 Visual Studio 中打开前面创建的 EventRegistrationWeb 项目。

(2) 在现有网站上添加一个新的 AJAX Web 窗体 UpdatePanelWithTrigger.aspx。

(3) 添加两个 UpdatePanel 控件。

(4) 在每个 UpdatePanel 控件中添加一个 Label 和一个 Button 控件。

(5) 把两个 Button 控件的 Click 事件处理程序赋予 OnButtonClick()方法，实现该方法的代码如下：

可从 wrox.com 下载源代码

```
protected void OnButtonClick(object sender, EventArgs e)
{
    DateTime now = DateTime.Now;
    Label1.Text = now.ToLongTimeString();
    Label2.Text = now.ToLongTimeString();
}
```

代码段 UpdatePanelWithTrigger.aspx.cs

(6) 运行应用程序。无论单击哪个按钮，两个标签都会改变，如图 18-11 所示。

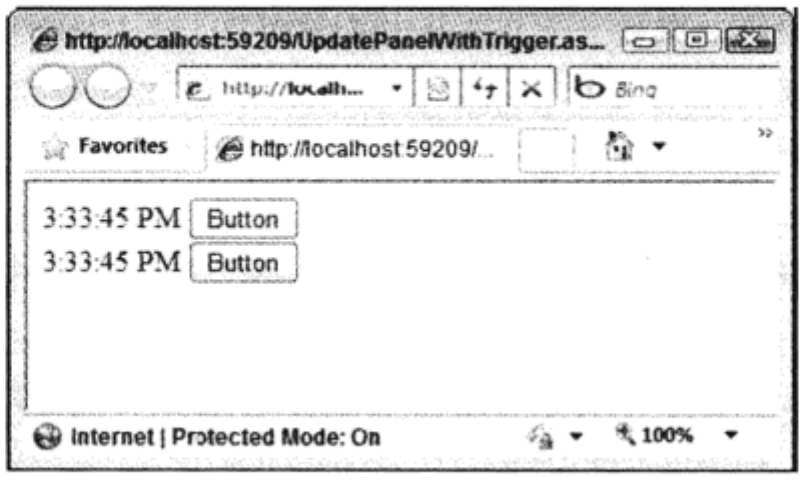


图 18-11

(7) 把两个 UpdatePanel 控件的 UpdateMode 属性都从 Always 改为 Conditional。

(8) 再次运行应用程序。现在单击一个按钮，就只有该按钮所在的 UpdatePanel 控件中的标签会改变。

(9) 选择第一个 UpdatePanel，单击 Triggers 属性旁边的省略号，打开如图 18-12 所示的对话框。添加一个 AsynchronousPostback 触发器，把 ControlID 属性设置为第二个 UpdatePanel 中的按钮 Button2，把 EventName 设置为 Click。

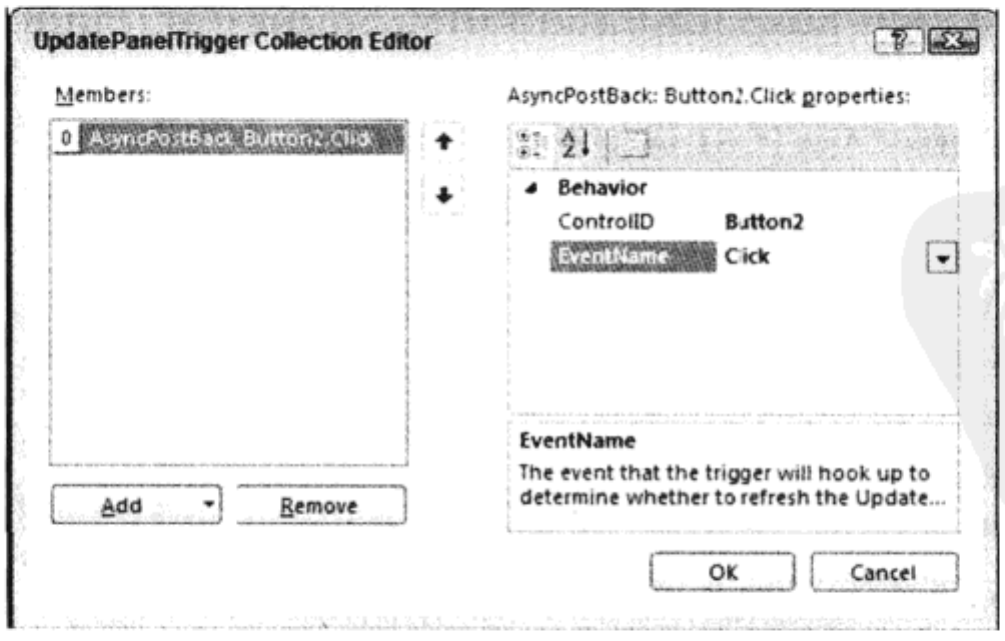


图 18-12

(10) 运行应用程序。单击 Button2，会更新两个 UpdatePanel 的内容，而单击 Button1，只更新第一个 UpdatePanel 控件的内容。

示例的说明

可以改变 UpdatePanel 的更新操作。它默认为每次发生 Ajax 回送事件时更新。可以改变这个更新操作，只有更新事件发生在 UpdatePanel 内部时才更新控件，或者定义一个触发器，由 UpdatePanel 外部的控件引发更新操作。

下面的 ASPX 代码定义了 UpdatePanel1 的 AsyncPostbackTrigger：

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Conditional">
    <ContentTemplate>
        <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        <asp:Button ID="Button1" runat="server" Text="Button"
            OnClick="OnButtonClick" />
    </ContentTemplate>
    <Triggers>
        <asp:AsyncPostBackTrigger ControlID="Button2" EventName="Click" />
    </Triggers>
</asp:UpdatePanel>
```

代码段 UpdatePanelWithTrigger.aspx

表 18-4 中描述了 UpdatePanel 控件的属性。

表 18-4

属　性	说　明
ChildrenAsTriggers	如果这个属性设置为 true，则 UpdatePanel 的子控件执行回送时，会更新 UpdatePanel 的内容
RenderMode	指定 UpdatePanel 的呈现方式。其值可以是 UpdatePanelRenderMode.Block 或 UpdatePanel RenderMode.Inline。Block 枚举值指定应呈现 标记，而 Inline 指定显示标记
UpdateMode	设置为 UpdatePanelUpdateMode 的一个枚举值。Always 会在每个 Ajax 回送时更新 UpdatePanel，Conditional 则取决于触发器
Triggers	指定 AsyncPostbackTrigger 和 PostbackTrigger 元素的集合，指定了更新 UpdatePanel 内容的时间

18.7　输入的有效性验证

在用户输入数据时，应检查数据是否有效。这个检查可以在客户端和服务器上进行。在客户端上检查数据可以使用 JavaScript 来进行。但是，如果使用 JavaScript 在客户端检查数据，就一定要在服务器上也进行检查，因为客户端是永远不能完全信任的。可以在浏览器上禁用 JavaScript，但黑客能使用其他 JavaScript 函数，接收不正确的输入。在服务器上检查数据是绝对必须的。在客户端上检查数据还能提供更好的性能，因为在客户端上验证完数据之前，不需要把数据发送到服务器上。

在 ASP.NET 中，不需要自己编写验证函数。许多已有的验证控件能进行客户端和服务器验证。

下面这个示例显示了与文本框 textFirstname 相关的验证控件 RequiredFieldValidator。所有的验证控件都有 ErrorMessage 和ControlToValidate 属性。如果输入不正确，ErrorMessage 可以定义显示的消息。默认的错误消息显示在验证控件所在的位置上。ControlToValidate 属性定义对输入进行检查的控件。

```
<asp:TextBox ID="textFirstName" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
   ErrorMessage="Enter your first name" ControlToValidate="textFirstName">
</asp:RequiredFieldValidator>
```

表 18-5 列出了所有的验证控件。

表 18-5

控 件	说 明
RequiredFieldValidator	指定所验证的控件需要输入一些内容。如果所验证的控件要设置初始值，而用户必须改变初始值，就可以使用验证控件的 InitialValue 属性设置这个初始值
RangeValidator	定义允许用户输入的最大值和最小值。该控件的特定属性有 MinimumValue 和 MaximumValue
RegularExpressionValidator	通过 ValidationExpression 属性，可以使用 Perl 5 语法设置一个正则表达式，来检查用户的输入
CompareValidator	比较多个值(如密码)。这个验证控件不仅可以比较两个值是否相等，还可以使用 Operator 属性设置多个选项。Operator 属性是 ValidationCompareOperator 类型，定义了枚举值，如 Equal、NotEqual、GreaterThan 和 DataTypeCheck。使用 DataTypeCheck 可以检查输入值，确定它是否是指定的数据类型，例如，查看输入的日期是否正确
CustomValidator	如果其他验证控件都不满足验证要求，就可以使用 CustomValidator。通过它可以定义客户端和服务器端验证功能
ValidationSummary	编写页面的小结，而不是编写与输入控件直接相关的错误消息

在前面的示例应用程序中，用户可以输入姓、名和电子邮件地址。下面将使用验证控件扩展这个应用程序。

试一试：检查需要的输入和电子邮件地址

(1) 打开前面在 Visual Studio 中创建的项目 EventRegistrationWeb。

(2) 打开文件 Registration.aspx。

(3) 在编辑器的设计视图中选择右边的列，再选择菜单项 Table | Insert | Column to the Right，给表添加一个新列。

(4) 需要输入姓、名和电子邮件地址。应检查电子邮件地址的语法是否正确。添加 3 个 RequiredFieldValidator 控件和 1 个 RegularExpressionValidator 控件，如图 18-13 所示。

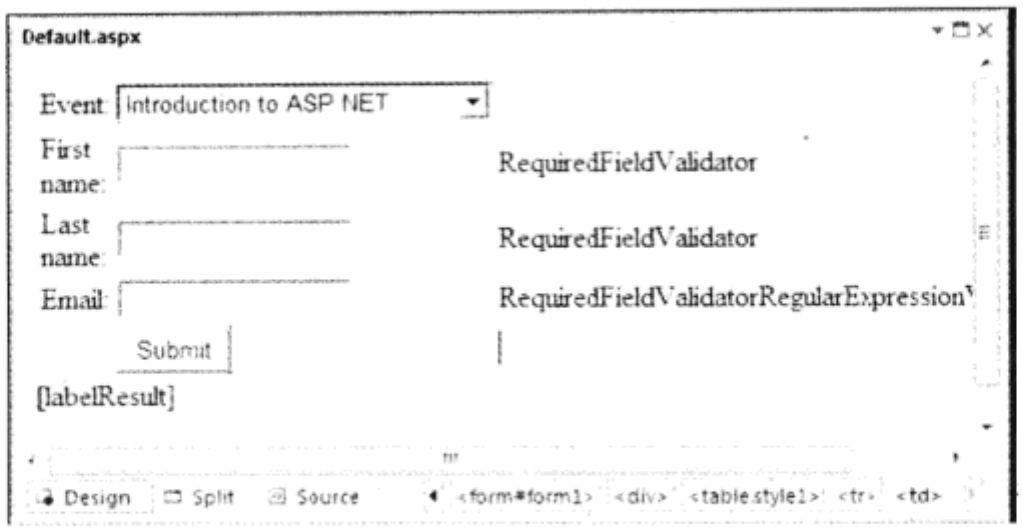


图　18-13

(5) 按照表 18-6 中的定义配置验证控件。

表　18-6

验 证 控 件	属　　性	值
RequiredFieldValidator	ErrorMessage	Firstname is required.
	ControlToValidate	textFirstName
RequiredFieldValidator	ErrorMessage	Last name is required.
	ControlToValidate	textLastName
RequiredFieldValidator	ErrorMessage	Email is required.
	ControlToValidate	textEmail
	Display	Dynamic
RegularExpressionValidator1	ErrorMessage	Enter a valid email.
	ControlToValidate	textEmail
	ValidationExpression	\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
	Display	Dynamic

(6) 不需要手工输入正则表达式，而可以在 Properties 窗口中单击 ValidationEpression 的省略号按钮，启动 Regular Expression Editor，如图 18-14 所示。这个编辑器提供了一些预定义的正则表达式，包括用来选择检查 Internet e-mail address 的正则表达式。

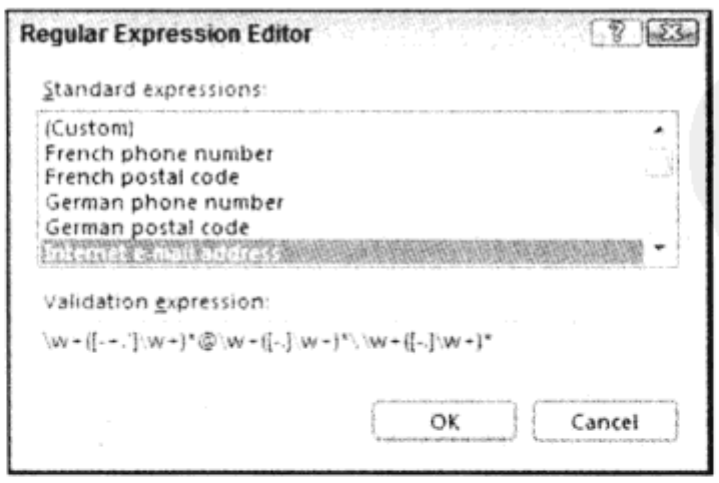


图　18-14

(7) 如果回送的页面不同于包含验证控件的页面(使用前面设置的 PostBackUrl 属性)，则在新的页面中必须使用 IsValid 属性，验证上一个页面的结果是否有效。把以下代码添加到 ResultsPage 类的 Page_Load 方法中：

可从 wrox.com 下载源代码

```
protected void Page_Load(object sender, EventArgs e)
{
   try
   {
      if (!PreviousPage.IsValid)
      {
         labelResult.Text = "Error in previous page";
         return;
      }
      //...
```

代码段 ResultsPage.aspx.cs

(8) 现在就可以启动应用程序了。如果没有输入数据或数据不正确，验证控件就会显示错误消息，如图 18-15 所示。

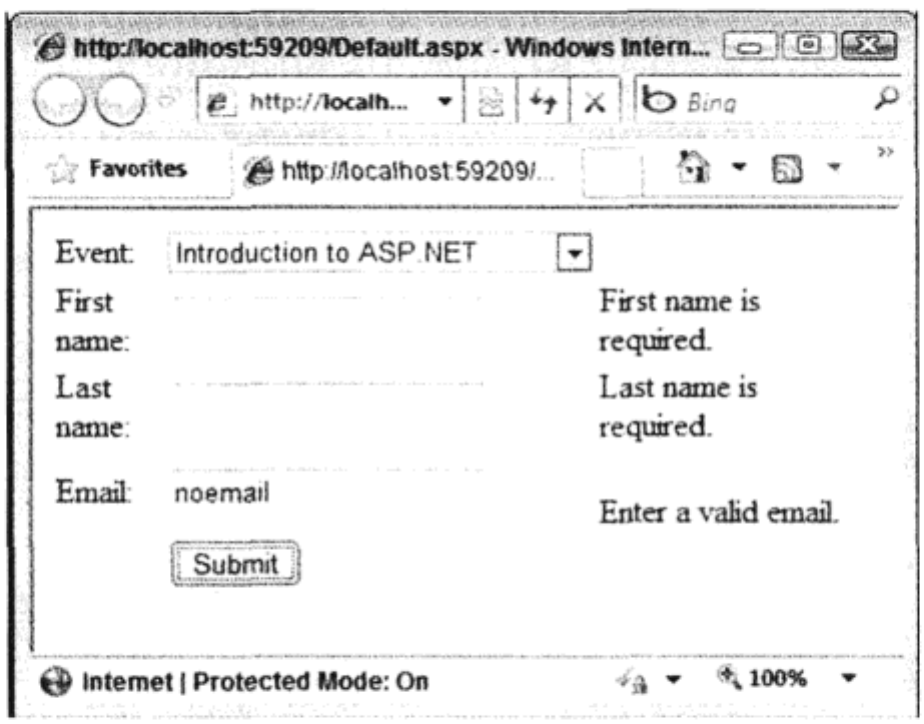


图 18-15

示例的说明

验证控件创建了客户端 JavaScript 代码，在客户端上验证输入，还创建了服务器端代码，在服务器上验证输入。把验证属性 EnableClientScript 设置为 false，就可以关闭 JavaScript。设置 Page 类的 ClientTarget 属性，也可以关闭 JavaScript，而不需要对每个验证控件执行改变属性值的操作。

根据客户的类型，ASP.NET 控件可以向客户端返回 JavaScript，也可以不返回 JavaScript。这取决于 ClientTarget 属性，该属性默认设置为 automatic，此时，根据 Web 浏览器的功能，返回或不返回脚本代码。如果 ClientTarget 设置为 downlevel，脚本代码就不返回给任何客户端，而把属性 ClientTarget 设置为 uplevel，就总是返回脚本代码。

ClientTarget 属性的设置可以在 Page 类的 Page_Load()方法中进行：

```
protected void Page_Load(object sender, EventArgs e)
```

```
{
    ClientTarget = "downlevel";
}
```

18.8　状态管理

HTTP 协议是无状态的。从客户端到服务器的连接可以在每个请求之后关闭。但是一般需要把一些客户端信息从一个页面传送给另一个页面。这可以通过几种方式来实现。

在保存状态的各种方式中，主要区别是：状态是存储在客户端还是服务器上。表 18-7 中简要列出了各种状态管理技术以及状态保持有效的时间。

表　18-7

状 态 类 型	客户端或服务器资源	有 效 时 间
ViewState	客户端	只在一个页面中
Cookie	客户端	关闭浏览器时会删除临时 cookie，永久 cookie 存储在客户系统的磁盘上
Session	服务器	会话状态与浏览器会话相关。会话在超时 (默认为 20 分钟) 后变得无效
Application	服务器	在所有的客户端上共享应用程序状态，这个状态在服务器重新启动之前都是有效的
Cache	服务器	类似于应用程序状态，缓存是共享的。但是，使缓存无效有更好的控制方式

下面详细介绍这些技术。

18.8.1　客户端的状态管理

首先介绍客户端的状态管理：ViewState 和 cookie。

1. ViewState

前面已讨论过在客户端存储状态的一种技术：ViewState。Web 服务器控件自动使用 ViewState 来使事件工作。ViewState 包含的状态与控件发送给客户端时包含的状态相同。当浏览器把窗体发送回服务器时，ViewState 包含了初始值，但所发送的控件包含新值。如果初始值和新值有区别，就调用相应的事件处理程序。

使用 ViewState 的缺点是，数据总是要从服务器传送给客户端，再从客户端传送给服务器，增加了网络流量。为了减少网络流量，可以关闭 ViewState。在 Page 指令中把 EnableViewState 属性设置为 false，就可以关闭页面中所有控件的 ViewState。

```
<%@ Page Language="C#" AutoEventWireUp="true" CodeFile="Registration.aspx.cs"
    Inherits="Registration_aspx" EnableViewState="false" %>
```

设置一个控件的 EnableViewState 属性，也可以配置该控件的 ViewState。无论页面进行了什么

配置，只要定义了控件的 EnableViewState 属性，就使用相应的控件值。只有没有配置 ViewState 的控件才使用页面配置的值。

还可以把定制的数据存储在 ViewState 中。为此可以使用索引符和 Page 类的 ViewState 属性。可以使用 index 参数定义一个名称，用于访问 ViewState 值。

```
ViewState["mydata"] = "my data";
```

可以读取前面存储的 ViewState，如下所示：

```
string mydata = (string)ViewState["mydata"];
```

在发送给客户端的 HTML 代码中，整个页面的 ViewState 存储在一个隐藏字段中：

```
<input type="hidden" name="__VIEWSTATE"
value="/wEPDwUKLTU4NzY5NTcwNw8WAh4HbXlzdGF0ZQUFbXl2YWwWAgIDD2QWAg
IFDw8WAh4EVGV4dAUFbXl2YWxkZGTCdCywUOcAW97aKpcjt1tzJ7ByUA==" />
```

使用隐藏字段的优点是，每个浏览器都可以使用这个特性，用户不能关闭它。

ViewState 只保存在页面中。如果状态应保存在多个不同的页面中，就应使用 cookie 在客户端保存状态。

2. cookie

cookie 在 HTTP 头中定义。使用 HttpResponse 类可以把 cookie 发送给客户端。Response 是 Page 类的一个属性，它返回一个 HttpResponse 类型的对象。HttpResponse 类定义了返回 HttpCookieCollection 的 Cookies 属性。使用 HttpCookieCollection 可以向客户端返回多个cookie。

下面的示例代码说明了如何把 cookie 发送给客户端。首先实例化一个 HttpCookie 对象。在这个类的构造函数中，设置 cookie 的名称，这里是 mycookie。HttpCookie 类的 Values 属性可以添加多个 cookie 值。如果只需要返回一个 cookie 值，就可以使用 Value 属性。但如果要发送多个 cookie 值，最好把值添加到一个 cookie 中，而不是使用多个 cookie。

```
string myval = "myval";
HttpCookie cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", myval);
Response.Cookies.Add(cookie);
```

cookie 可以是临时的，仅在一个浏览器会话中有效，也可以存储在客户端的磁盘上。为了使 cookie 变成永久的，必须使用 HttpCookie 对象设置 Expires 属性。使用 Expires 属性可以定义 cookie 不再有效的日期，这里把它设置为三个月后。

```
Var cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", "myval");
cookie.Expires = DateTime.Now.AddMonths(3);
Response.Cookies.Add(cookie);
```

尽管可以设置特定的日期，但无法保证 cookie 会一直存储到该日期为止。用户可以删除 cookie，如果在本地存储了太多的 cookie，浏览器应用程序也会删除 cookie。浏览器只能给一个服务器存储 20 个 cookie，给所有的服务器存储 300 个 cookie。达到这个界限时，就会删除一段时间内不再使用的 cookie。

客户从服务器上请求一个页面时，这个服务器的 cookie 就可以在客户端上使用，并作为 HTTP 请求的一部分发送给服务器。要在 ASP.NET 页面中读取 cookie，可以访问 HttpRequest 对象中的 cookie 集合。

与 HTTP 响应一样，Page 类也有一个 Request 属性返回 HttpRequest 类型的对象。Cookies 属性返回 HttpCookieCollection，它可以用于读取客户端发送的 cookie。可以用索引符通过其名称来访问 cookie，然后使用 HttpCookie 的 Values 属性从 cookie 中获取值。

```
HttpCookie cookie = Request.Cookies["mycookie"];
string myval = cookie.Values["mystate"];
```

ASP.NET 允许更加方便地使用 cookie。但必须了解 cookie 的限制。如前所述，浏览器只能接收来自一个服务器的 20 个 cookie，以及来自所有服务器的 300 个 cookie。另外， cookie 存储的数据量不能超过 4K。这些限制使客户端磁盘不会被 cookie 占满。

18.8.2　服务器端的状态管理

除了在客户端上保存状态之外，还可以在服务器上保存状态。使用客户端的状态，其缺点在于增加了数据在网络之间的传送。使用服务器端状态的缺点在于，服务器必须给其客户端分配资源。下面详细讨论服务器端的状态管理技术。

1. 会话

会话状态与浏览器会话相关。客户在服务器上第一次打开 ASP.NET 页面时，会话就开始了。当客户在 20 分钟之内没有访问服务器时，会话结束。

可以在 Global Application 类中定义自己的代码，在会话开始或结束时运行。Global Application 类使用菜单项 Project | Add New Item | Global Application Class 创建。在新的 Global Application 类中，会创建 global.asax 文件。在这个文件中，在派生于基类 HttpApplication 的 Global 类中定义了一些事件处理程序例程：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;

namespace EventRegistrationWeb
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
        }

        protected void Session_Start(object sender, EventArgs e)
        {
            // Code that runs when a new session is started
        }
        protected void Application_BeginRequest(object sender, EventArgs e)
        {
```

```
    }

    protected void Application_AuthenticateRequest(object sender, EventArgs e)
    {

    }

    protected void Application_Error(object sender, EventArgs e)
    {
        // Code that runs when an unhandled error occurs
    }

    protected void Session_End(object sender, EventArgs e)
    {
        // Code that runs when a session ends.
        // Note: The Session_End event is raised only when the session state
        // mode is set to InProc in the Web.config file. If session mode is
        // set to StateServer or SQLServer, the event is not raised.
    }

    protected void Application_End(object sender, EventArgs e)
    {
        // Code that runs on application shutdown
    }
  }
}
```

代码段 Global.asax.cs

会话状态可以存储在 HttpSessionState 对象中。可以使用 Page 类的 Session 属性来访问与当前 HTTP 上下文相关的会话状态对象。在 Session_Start()事件处理程序中，可以初始化会话变量。在下面的示例中，名为 mydata 的会话状态被初始化为 0:

```
void Session_Start(Object sender, EventArgs e) {
    // Code that runs on application startup
    Session["mydata"] = 0;
}
```

可以使用会话状态名读取会话状态，这通过 Session 属性来完成：

```
void Button1_Click(object sender, EventArgs e)
{
  int val = (int)Session["mydata"];
  Label1.Text = val.ToString();
  val += 4;
  Session["mydata"] = val;
}
```

要把客户端与其会话变量关联起来，ASP.NET 默认使用一个临时 cookie 和一个会话标识符。ASP.NET 也支持没有 cookie 的会话，其中的 URL 标识符用于把 HTTP 请求映射到同一个会话。

2. 应用程序

如果应在多个客户端之间共享数据，就可以使用应用程序状态。应用程序状态的使用方式与会话状态非常类似。对于应用程序状态，应使用 HttpApplicationState 类，通过 Page 类的 Application 属性可以访问它。

在下面的示例中，在启动 Web 应用程序时，初始化应用程序变量 userCount。Application _Start()是 global.asax 文件中的事件处理方法，在启动网站的第一个 ASP.NET 页面时调用该方法。这个变量用于统计访问网站的用户数。

```
void Application_Start(Object sender, EventArgs e) {
    // Code that runs on application startup
    Application["userCount"] = 0;
}
```

在 Session_Start()事件处理程序中，应用程序变量 userCount 的值会递增。在改变应用程序变量之前，必须用 Lock()方法锁定应用程序对象，否则就会出现线程问题，因为多个客户可以同时访问一个应用程序变量。在改变了应用程序变量的值后，还必须调用 Unlock()方法。注意锁定和解锁之间的时间非常短暂，在此时间段内，不应读取文件或数据库中的数据。否则，其他客户就必须等到数据访问完成之后才能操作。

```
void Session_Start(Object sender, EventArgs e) {
    // Code that runs when a new session is started
    Application.Lock();
    Application["userCount"] = (int)Application["userCount"] + 1;
    Application.UnLock();
}
```

读取应用程序状态中的数据与读取会话状态中的数据一样简单：

```
Label1.Text = this.Application["userCount"].ToString();
```

不要在应用程序状态中存储太多的数据，因为应用程序状态需要占用服务器资源，直到服务器停止或重启动之后，才会释放这些资源。

3. 缓存

缓存是服务器端状态，它类似于应用程序状态，因为它在所有的客户端上共享。缓存与应用程序状态的区别是，缓存要灵活得多：可以通过多种方式来定义状态的失效时间。我们不是给每个请求读取文件或数据库，而是把数据存储在缓存中。

对于缓存，需要使用 System.Web.Caching 名称空间和 Cache 类。给缓存添加对象的过程如下：

```
Cache.Add("mycache", myobj, null, DateTime.MaxValue,
    TimeSpan.FromMinutes(10), CacheItemPriority.Normal, null);
```

Page 类的 Cache 属性返回一个 Cache 对象。使用 Cache 类的 Add()方法，可以把任意对象赋予缓存。Add()方法的第一个参数定义了缓存项的名称。第二个参数是应缓存的对象。第三个参数定义了依赖关系，例如，缓存项可以依赖于一个文件：当文件改变时，缓存对象就会失效。在上面的示

例中，没有定义依赖关系，因为这个参数设置为 null。

第四和五个参数用于设置缓存项的有效时间。第四个参数定义了缓存项失效的绝对时间。第五个参数定义了使缓存项失效的相对时间(即从最后一次访问以来经历的时间)。上面的示例使用了相对时间，如果缓存项 10 分钟以来未受到访问，就会失效。

第六个参数定义了缓存的优先级。CacheItemPriority 是一个设置缓存优先级的枚举。如果 ASP.NET 辅助进程有很高的内存利用率，ASP.NET 运行库就根据优先级删除缓存项。优先级较低的项先删除。最后一个参数用于定义一个方法，在删除缓存项时调用该方法。当缓存依赖于一个文件时，就可以使用最后一个参数：当文件改变时，就删除缓存项，调用事件处理程序。通过这个事件处理程序，可以再次读取文件，重新加载缓存。

使用前面介绍的索引符和会话状态或应用程序状态，就可以读取缓存项。在使用从 Cache 属性返回的对象之前，必须检查结果是否为 null，当缓存失效时，结果就是 null。如果从 Cache 索引符返回的值不是 null，就可以对返回的对象进行类型转换，用于存储缓存项：

```
object o = Cache["mycache"];
if (o == null)
{
   // Reload the cache.
}
else
{
    // Use the cache.
   MyClass myObj = (MyClass)o;
   //...
}
```

18.9 样式

Visual Studio 支持使用层叠样式表(Cascading Style Sheets，CSS)给 Web 页面指定样式。使用 CSS 可以定义 HTML 页面的外观和格式。使用 CSS 无需定制每个 HTML 元素，而可以为特定的元素定义样式(参见下面的示例)，再按名称引用它们，以便于重用。

试一试：给元素定义样式

(1) 打开前面创建的 Web 应用程序项目 EventRegistrationWeb。

(2) 选择 Project | New Folder，添加一个新文件夹 Styles。

(3) 在 Solution Explorer 中选择这个文件夹，再选择 Project | Add New Item，之后选择 Style Sheet，创建一个新的样式表，命名为 Site.css。

(4) 这个样式表默认包含一个空的 body 元素。

(5) 在 body 元素的花括号内部单击，打开关联菜单，选择 Build Style，打开如图 18-16 所示的 Modify Style 对话框。

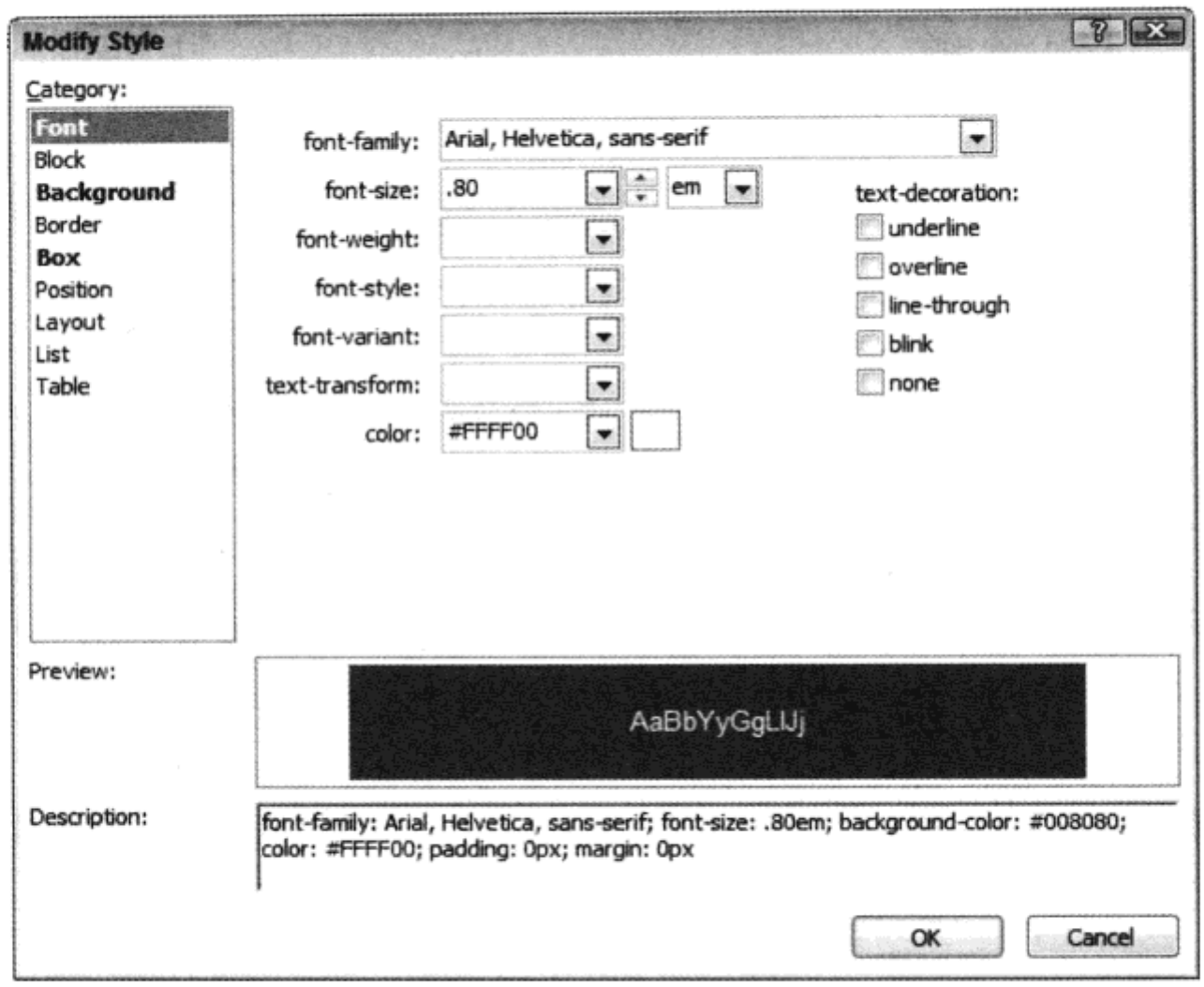


图　18-16

(6) 选择 Font 类别，把 font-family 设置改为 Arial, Helvetica, sans-serif，把 font-size 设置改为.80 em，把 color 改为#FFFF00。

(7) 在同一个对话框中选择 Background 类别，把背景色改为#008080。

(8) 选择 Box 类别，把 padding 和 margin 都改为 0。

(9) 样式表现在应如下面的代码段所示：

可从 wrox.com 下载源代码

```
body
{
    font-family: Arial, Helvetica, sans-serif;
    font-size: .80em;
    background-color: #008080;
    color: #FFFF00;
    padding: 0px;
    margin: 0px;
}
```

代码段 Site.css

(10) 在 CSS 编辑器的源代码视图中，选择关联菜单 Add Style Rule，再选择元素 a:hover，如图 18-17 所示，单击 OK 按钮。

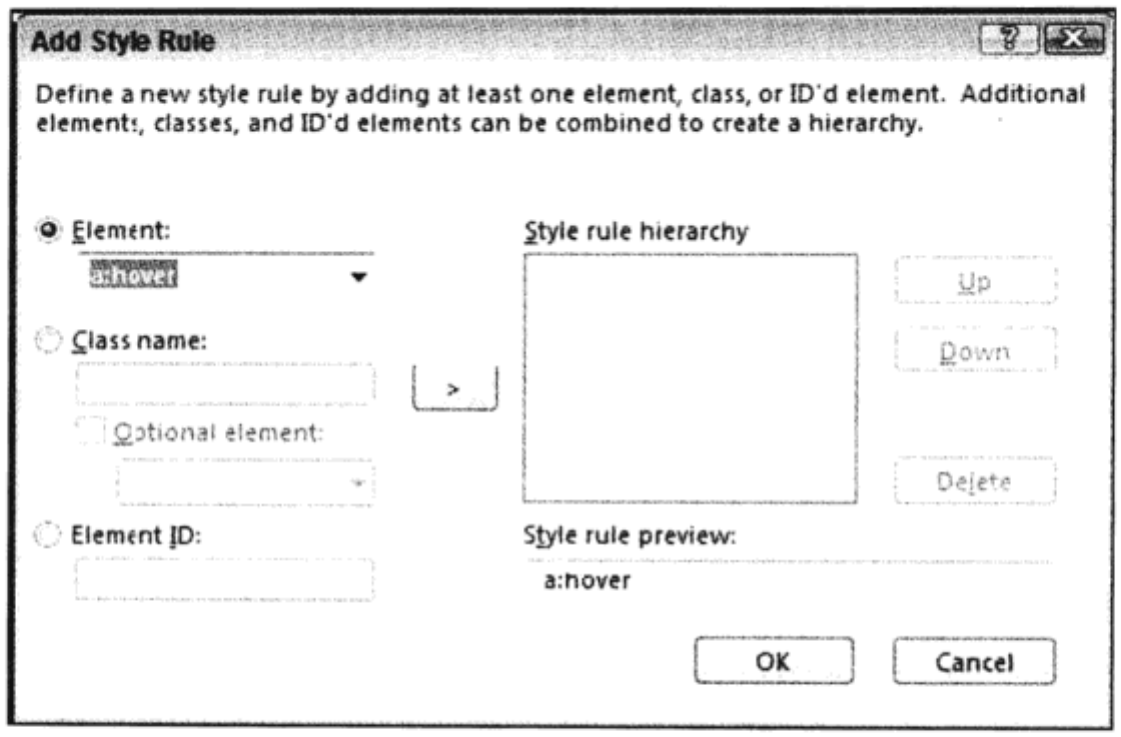


图 18-17

(11) 选择 Build Style 菜单，再次打开 Modify Style 对话框。选择 Font 类别，把颜色改为#FF0000，选择 text-decoration 中的 underline 和 overline，得到的 CSS 代码如下所示。

可从 wrox.com 下载源代码

```
a:hover
{
    color: #FF0000;
    text-decoration: underline overline;
}
```

代码段 Site.css

(12) 根据下面的代码给 a:active、a:link、a:visited 和 h1 添加样式规则：

可从 wrox.com 下载源代码

```
a:link, a:visited
{
    color: #00FFFF;
}

a:active
{
    color: #00FFFF;
}

a:hover
{
    color: #FF0000;
    text-decoration: underline overline;
}
h1
{
    text-align: center;
}
```

代码段 Site.css

(13) 创建一个新的 Web 页面 StylesDemo.aspx，把 Site.css 文件从 Solution Explorer 拖放到编辑器的设计视图上。页面的背景色会立即改变。

(14) 切换到源代码视图，验证新的 link 项引用了样式表。

```
<head runat="server">
    <title></title>
    <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>
```

代码段 StylesDemo.aspx

(15) 在页面的 body 元素内部添加一个 h1 标记和一个锚定标记，如下所示：

```
<body>
    <form id="form1" runat="server">
    <h1>
        Styles Demo</h1>
    <div>
        <a href="http://www.wrox.com">Wrox Press</a>
    </div>
    </form>
</body>
```

代码段 StylesDemo.aspx

(16) 选择 Debug | Start without Debugging，启动浏览器，来查看该页面。页面应如图 18-18 所示。验证是否把页面样式应用于标题和链接。把鼠标停放在链接上，查看颜色的变化和文本装饰。

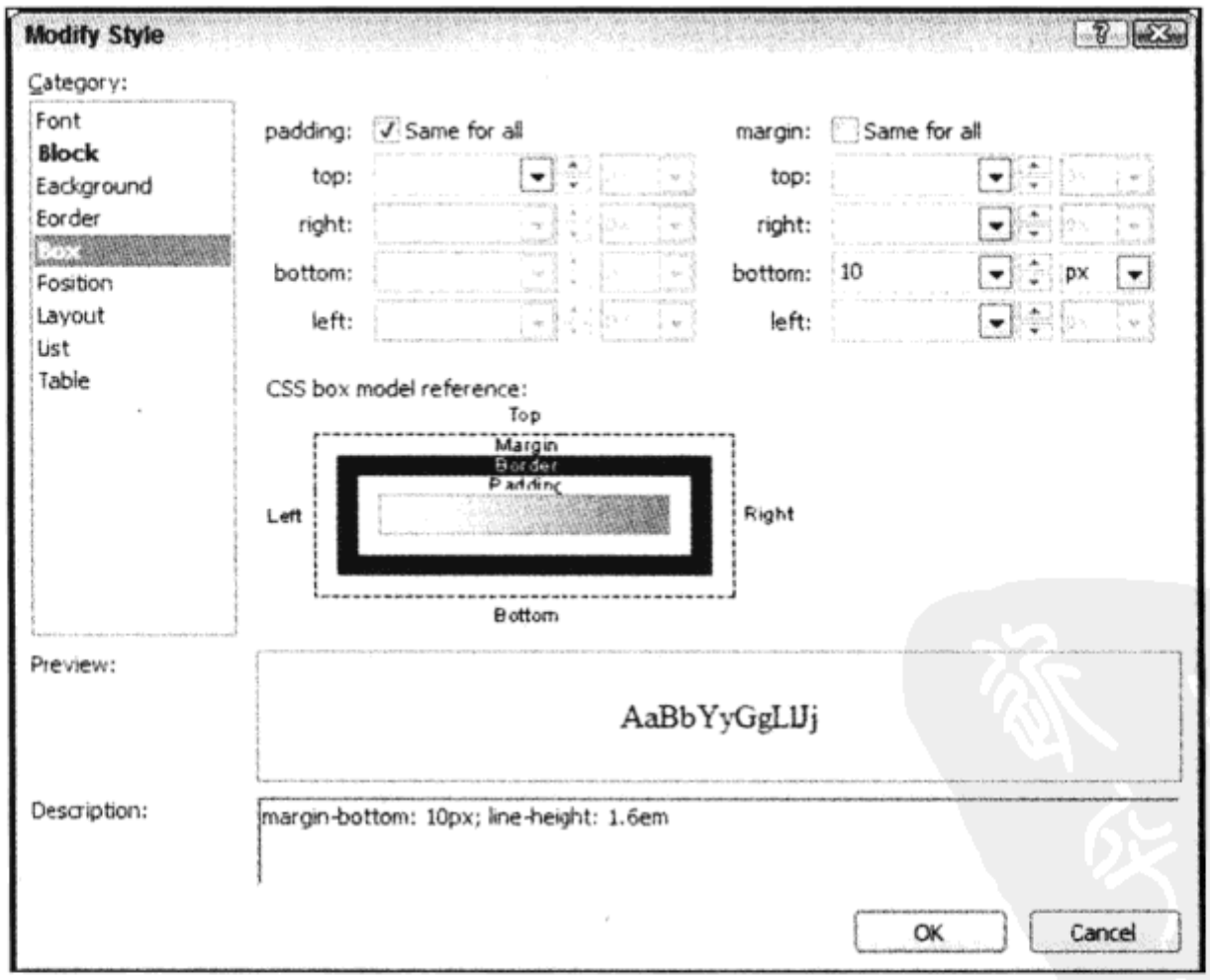


图　18-18

示例的说明

由于给链接元素引用了 CSS 文件，所以浏览器请求这个页面和 HTML 代码。接着浏览器使用 CSS 文件中已应用了样式的元素，来改变 HTML 元素的外观。

使用 CSS 不仅可以改变特定 HTML 标记的样式，还可以定义在 HTML 标记上引用的类，如下面的示例所示。

试一试：定义样式类

(1) 打开样式表 Site.css。

(2) 打开 Add Style Rule 编辑器，添加一个新类 bottom。对话框中的样式规则预览在类名之前加了一个句点(.)。单击 OK 按钮。

(3) 打开 Modify Style 编辑器。

(4) 选择 Font 类别，给 font-size 选择 x-small。

(5) 选择 Block 类别，将 vertical-align 定义为 text-bottom，将 text-align 定义为 center。

(6) 选择 Box 类别，将上、下、左、右的页边距都定义 5。

(7) 选择 Position 类别，把高度定义为 40 px。

(8) 在 Site.css 文件中验证结果。

可从 wrox.com 下载源代码

```
.bottom
{
    margin: 5px;
    height: 40px;
    text-align: center;
    vertical-align: text-bottom;
    font-size: x-small;
}
```

代码段 Site.css

(9) 打开 StylesDemo.aspx 文件，添加一个包含如下文本的 div 元素：

可从 wrox.com 下载源代码

```
<div class="bottom">
    Copyright (c) 2010 Wrox Press
</div>
```

代码段 StylesDemo.aspx

(10) 启动浏览器，查看 StylesDemo.aspx 文件。验证给 div 元素应用了样式。

示例的说明

不需要为网站的每个页面中的每个元素定义样式，而可以在一个公共位置定义样式。Modify Style 编辑器可以预览所有可以改变的样式。打开页面时，浏览器会自动应用样式，并排列元素。

警告：在不同的浏览器上，一些样式的效果会不同。确保在所有支持的浏览器上查看页面的外观。

使用样式不仅可以应用字体大小和颜色，还可以定义 Web 页面的布局。另外，不仅可以利用样式来指定布局，还可以用 HTML 表指定布局。CSS 在布局方面比 HTML 表更灵活，而且还把内容与可见的信息分开，提供了可访问性方面的优势。因此目前无表格的 Web 设计通常是首选方案。

由于本书介绍 C#编程，而不是 HTML 页面的设计，所以仅简要介绍了 CSS。如需了解 CSS 的更多信息，请参阅 Jon Ducket 编著的图书 *Beginning HTML, XHTML, CSS, and JavaScript*(Wrox 于 2009 出版)。

18.10　母版页

大多数 Web 站点都在每个页面上重用了母版页的部分内容，例如，公司徽标和菜单常常会出现在所有的页面上。每个页面不需要重复共同的用户界面元素，可以将共同的元素添加到母版页上。母版页看起来类似于一般的 ASP.NET 页面，但定义了由内容页(content page)替换的占位符。

母版页的文件扩展名是.master，它在文件的第一行上使用 Master 指令，如下所示：

```
<%@ MasterLanguage="C#"AutoEventWireup="true"CodeFile="MasterPage.master.cs"
    Inherits="MasterPage" %>
```

在 Web 站点中，只有母版页才使用<html>、<head>、<body>和<form>等 HTML 元素。Web 页面本身只包含内嵌到<form>元素中的内容。Web 页面可以把它自己的内容嵌入 ContentPlaceHolder 控件中。如果 Web 页面没有这么做，母版页可以为 ContentPlaceHolder 定义默认内容。

```
<html xmlns="http://www.w3.org/1999/xhtml>
<head runat="server">
   <title> </title>
      <asp:contentplaceholder id="head" runat="server">
      </asp:contentplaceholder>
</head>
<body>
   <form id="form1" runat="server">
     <div>
       <asp:contentplaceholder id="ContentPlaceHolder1" runat="server">
       </asp:contentplaceholder>
     </div>
   </form>
</body>
</html>
```

要使用母版页，必须把 MasterPageFile 特性应用于 Page 指令。要替换母版页的内容，可以使用 Content 控件。该控件把 ContentPlaceHolder 与 ContentPlaceHolderID 关联起来。

```
<%Page Language="C#" MasterPageFile="~/MasterPage.master"
  AutoEventWireUp="true" CodeFile="Default.aspx.cs" Inherits="default"
  Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head"
```

```
    Runat="Server"></asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
    Runat="Server"></asp:Content>
```

除了用 Page 指令定义母版页之外，还可以在 Web 配置文件 web.config 中使用<pages>元素，把默认的母版页赋予所有 Web 页面：

```
<configuration>
  <system.web>
    <pages masterPageFile="~/MasterPage.master" />
     <!--…-->
   </pages>
  </system.web>
</configuration>
```

使用在 web.config 中配置的母版页文件时，ASP.NET 页面需要该文件中的 Content 元素配置，如前所示。否则，masterPageFile 特性就没有用了。如果同时使用 Page 指令的 MasterPageFile 特性和 web.config 中的项，Page 指令的设置将覆盖 web.config 中的设置。这样，就可以既使用 web.config 定义默认的母版页文件，又允许特定的 Web 页面覆盖默认设置。

还可以使用编程方式修改母版页。这样就可以把不同的母版页用于不同的设备或不同的浏览器类型。可以修改母版页的最后一个地方是在 Page_PreInit 处理方法中。在下面的示例代码中，如果浏览器发送 MSIE 字符串和浏览器名称(由 Microsoft IE 发送)，则 Page 类的 MasterPageFile 属性就设置为 IE.master，否则给其他所有浏览器发送 Default.master：

```
public partial class changeMaster : System.Web.UI.Page
{
  void Page_Load(object sender, EventArgs e)
  {
  }

  void Page_PreInit(object sender, EventArgs e)
  {
    if (Request.UserAgent.Contains("MSIE")
    {
      this.MasterPageFile = "~/IE.master";
    }
    else
    {
      this.MasterPageFile = "~/Default.master";
    }
  }
}
```

在下面的示例中将创建自己的母版页。这里的示例母版页包含标题和主体，母版页的主要部分由各个页面替代。

试一试：创建母版页

(1) 打开 Web 应用程序项目 EventRegistrationWeb。

(2) 添加一个新的 Master Page 项，命名为 Events.master。

(3) 切换到编辑器的设计视图，应用样式表 Site.css。把 Site.css 文件从 Solution Explorer 拖放到编辑器中。

(4) 把第二个 ContentPlaceHolder 的 ID 重命名为 ContentPlaceHolderMain，把 CSS 类的内容赋予 div 元素，如下所示：

可从 wrox.com 下载源代码

```
<div class="content">
    <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
    </asp:ContentPlaceHolder>
</div>
```

代码段 Events.Master

(5) 在前面修改的 div 元素前面，添加下面的 div 和 h1 元素：

可从 wrox.com 下载源代码

```
<div class="header">
    <h1>
        Event Registration
    </h1>
</div>
<div class="navigation">
    Menu will go here
</div>
```

代码段 Events.Master

(6) 在包含 ContentPlaceHolder 的 div 元素后面添加如下 div 元素：

可从 wrox.com 下载源代码

```
<div class="bottom">
    Copyright (c) 2010 Wrox Press
</div>
```

代码段 Events.Master

(7) 完整的页面如下所示：

可从 wrox.com 下载源代码

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Events.master.cs"
            Inherits="EventRegistrationWeb.Events" %>

<!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>
    <asp:ContentPlaceHolder ID="head" runat="server">
    </asp:ContentPlaceHolder>
    <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div class="header">
        <h1>
            Event Registration
```

```
            </h1>
        </div>
        <div class="navigation">
            Menu will go here
        </div>
        <div class="content">
            <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
            </asp:ContentPlaceHolder>
        </div>
        <div class="bottom">
            Copyright (c) 2010 Wrox Press
        </div>
        </form>
    </body>
    </html>
```

代码段 Events.Master

示例的说明

如前所述，母版页包含 HTML，其中的 FORM 标记包括内容占位符，其内容会被使用母版页的页面替代。HTML 和链接的 CSS 定义了页面的布局。只有内容占位符会被内容页面替代。如果应替换页面的不同部分，就可以使用多个内容占位符。

创建母版页后，就可以在 Web 页面中使用它，如下面的示例所示。

试一试：使用母版页

(1) 在 Web 应用程序中使用母版页添加一个 Web Form 类型的新项，命名为 Default.aspx。

(2) 打开 Select a Master Page 对话框，如图 18-19 所示。选择母版页 Events.master，单击 OK 按钮。

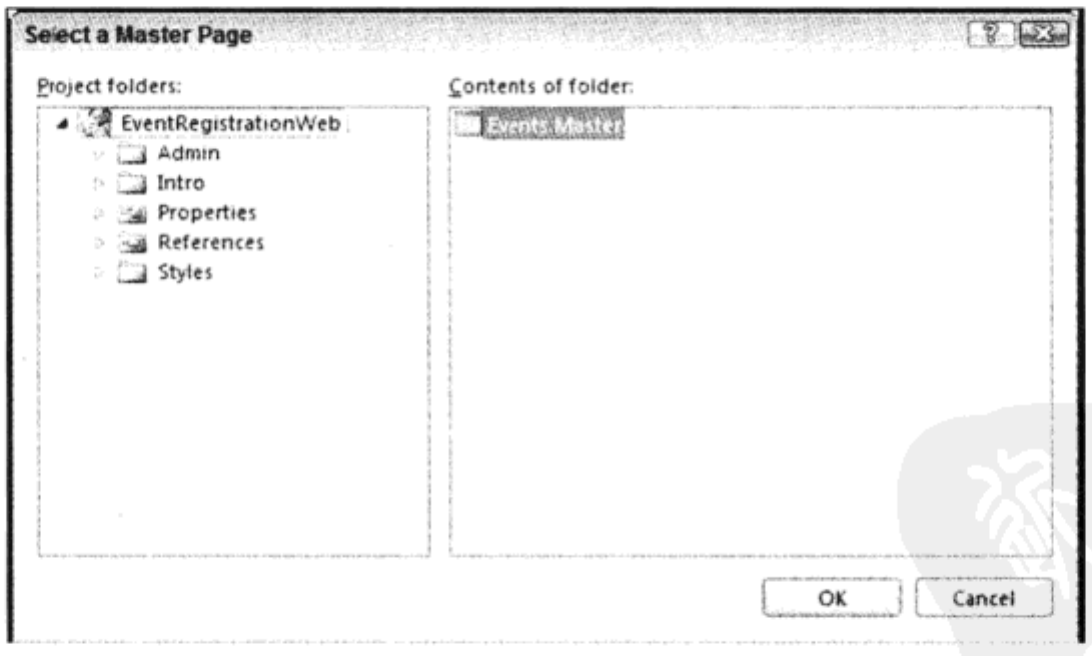


图 18-19

(3) 在 Default.aspx 文件的源代码视图中，Page 指令引用了母版页中的 ContentPlaceHolder 控件，在该指令后只显示了 2 个 Content 控件。把这两个 Content 控件的 ID 属性改成 ContentHead 和 ContentMain。

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.master"
         AutoEventWireup="true" CodeBehind="Default.aspx.cs"
         Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
         runat="server">
</asp:Content>
```

代码段 Default.aspx

(4) 切换到 Visual Studio 的设计视图。这个视图显示了不能在页面中修改的母版页内容，包括标题和版权信息，给 Content 控件添加一些文本，并居中对齐。

(5) 切换到源代码视图，会看到如下代码。居中对齐改为 CSS 样式。

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
         AutoEventWireup="true" CodeBehind="Default.aspx.cs"
         Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
    <style type="text/css">
        .style1
        {
            text-align: center;
        }
    </style>
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
    <p class="style1">
        Welcome to the</p>
    <p class="style1">
        Event Registration</p>
    <p class="style1">
        Sample application for Beginning Visual C# 2010!</p>
</asp:Content>
```

代码段 Default.aspx

(6) 用浏览器查看页面。结果应如图 18-20 所示。

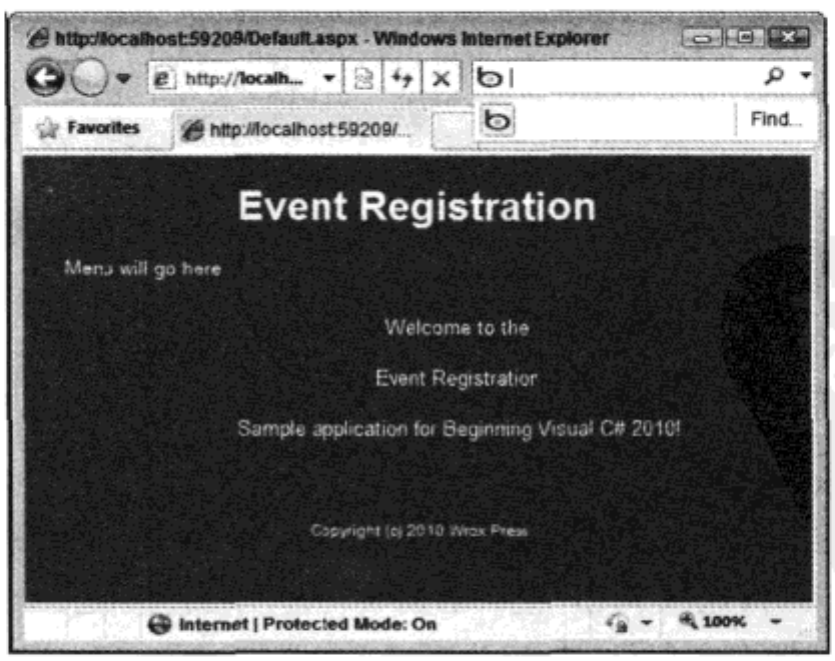


图　18-20

18.11 站点导航

在 Web 站点的多个页面上导航时，可以定义一个包含 Web 站点结构的 XML 文件，使用一些 UI 控件显示导航选项。用于导航的重要控件如表 18-8 所示。

表 18-8

控 件	说 明
SiteMapDataSource	SiteMapDataSource 控件是一个数据源控件，它引用站点地图数据提供程序。在 Visual Studio 工具箱中，该控件位于 Data 部分
Menu	Menu 控件按照站点地图数据源的定义显示页面的链接。菜单可以水平或垂直显示，而且有许多配置其样式的选项
SiteMapPath	SiteMapPath 控件使用很小的空间显示页面在 Web 站点层次结构中的当前位置。可以显示文本或图像超链接
TreeView	TreeView 控件显示 Web 站点的层次结构视图

下面的示例添加一个站点地图和一个菜单控件，用于在网站的页面之间导航。

试一试：添加导航控件

(1) 打开 Web 应用程序项目 EventRegistrationWeb。

(2) 在 Solution Explorer 中右击项目，选择 Add New Item，给 Web 站点添加一个新的 Site Map 项，名称 Web.sitemap 不变。

(3) 修改文件的内容，如下所示：

可从 wrox.com 下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="Default.aspx" title="Home">
      <siteMapNode url="EventRegister.aspx" title="Register"
                   description="Register to an Event" />
      <siteMapNode url="EventList.aspx" title="Event List"
                   description="Lists Events Worldwide" />
      <siteMapNode url="Admin/EventManagement.aspx" title="Event Management"
                   description="Management of Events" roles="Editors" />
    </siteMapNode>
</siteMap>
```

代码段 Web.sitemap

(4) 打开 Events.master 文件。

(5) 在工具箱的 Data 选项卡上，找到 SiteMapDataSource 控件，把它添加到页面上。

(6) 从工具箱的 Navigation 选项卡上，把一个 Menu 控件添加到标题 Registration Demo Web 的下面。把数据源设置为 SiteMapDataSource1。

(7) 配置 Menu 控件：把 Orientation 属性设置为 Horizontal，StaticDisplayLevels 属性设置为 2，

CssClass 属性设置为 menu。

```
<div class="navigation">
     
    <asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1"
        Orientation="Horizontal" StaticDisplayLevels="2" CssClass="menu">
    </asp:Menu>
    <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</div>
```

代码段 Events.Master

(8) 在 Site.css 文件中添加如下样式规则，给菜单指定样式：

```
.menu ul li a
{
    background-color: #008085;
    border: 1px #4e667d solid;
    color: #dde4ec;
    display: block;
    line-height: 1.35em;
    padding: 4px 20px;
    text-decoration: none;
    white-space: nowrap;
}

.menu ul li a:hover
{
    background-color: #bfcbd6;
    color: #465c71;
    text-decoration: none;
}
```

代码段 Site.css

(9) 在 Menu 控件的下面添加一个 SiteMapPath 控件。

(10) 在浏览器中打开 Default.aspx 文件。注意，菜单和路径显示了当前文件在 Web 站点中的位置。

(11) 使用 Web Form using Master Page 模板创建新页面 EventRegister.aspx 和 EventList.aspx，其中使用了母版页 Events.Master。

(12) 创建一个新文件夹 Admin，再在这个文件夹中创建一个新页面 EventManagement.aspx。再次使用 Web Form using Master Page 模板创建该页面。

(13) 根据需要添加在 Web.sitemap 文件中引用的其他页面。只需引用同一个母版页，显示定义好的菜单即可。

(14) 在 system.web 元素中，给 web.config 文件添加 siteMap 元素，如下所示：

```
<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">
  <providers>
    <clear />
    <add name="XmlSiteMapProvider"
         description="Default SiteMap Provider"
```

```
            type="System.Web.XmlSiteMapProvider"
            siteMapFile="Web.sitemap"
            securityTrimmingEnabled="true" />
    </providers>
</siteMap>
```

代码段 Web.config

示例的说明

Web 站点的结构由 Web.sitemap 文件中的 Web 页面定义。这个 XML 文件在<sitemap>根元素中包含 XML 元素<siteMapNode>。<siteMapNode>元素定义了 Web 页面。页面的文件名用 url 特性设置，title 特性指定显示在菜单中的名称。把<siteMapNode>元素编写为页面的子元素(包含子元素的链接)，就定义了页面的层次结构。

SiteMapDataSource 控件是一个数据源控件，类似于第 17 章介绍的数据源控件。这个控件可以使用不同的提供程序。默认使用 XmlSiteMapProvider 类来获取数据。XmlSiteMapProvider 类默认使用 Web.sitemap 文件。

因为 roles 特性应用于 siteMapNode EventManagement.aspx，所以只有特定角色 Editors 中的用户才能看到这个菜单项。因为 XmlSiteMapProvider 的这个授权特性默认情况下不启用，所以把 web.config 文件改为设置 XmlSiteMapProvider 的 securityTrimmingEnabled 属性。如果菜单不需要角色，就根本不需要 web.config 中的这个配置。

使用 Menu 控件可以编辑显示在 ASPX 源文件中的菜单项，也可以通过编程方式添加菜单项。添加菜单项最简单的方式是通过配置数据源来使用站点地图数据源。

18.12 身份验证和授权

为了保护网站，应进行身份验证，检查用户是否有权登录。而授权检查的是已通过身份验证的用户是否可以使用资源。

ASP.NET 提供了Windows 和 Forms 身份验证。对于 Web 应用程序，最常用的身份验证技术是 Forms 身份验证。ASP.NET 还为 Forms 身份验证提供了一些新特性。Windows 身份验证利用 Windows 账户和 IIS 来验证用户的身份。

ASP.NET 为用户的身份验证提供了许多类。图 18-21 列出了新体系结构。在 ASP.NET 中，有许多新的安全控件，如 Login 和 PasswordRecovery。这些控件都利用了 Membership API。在 Membership API 中，可以创建和删除用户、验证登录信息或获取当前登录的用户信息。Membership API 本身又利用了成员提供程序。在 ASP.NET 4 中，存在不同的提供程序，用于访问 Access 数据库、SQL Server 数据库或 Active Directory 中的用户。还可以创建定制的提供程序，来访问 XML 文件或其他定制存储器。

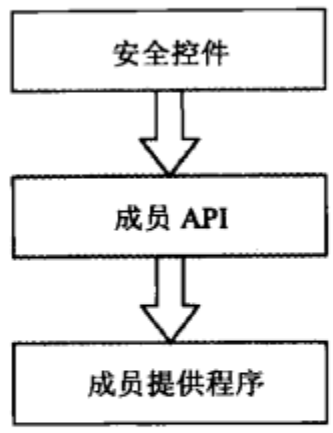


图　18-21

18.12.1　身份验证的配置

本章将演示 Forms 身份验证和一个成员提供程序。下面的示例会为 Web 应用程序配置安全性，给不同的文件夹赋予不同的访问列表。

试一试：安全配置

(1) 用 Visual Studio 打开前面创建的 Web 应用程序 EventRegistrationWeb。

(2) 在 Solution Explorer 中选择 Web 目录，再选择菜单项 Website | Add Folder | Regular Folder，创建一个新文件夹，把它命名为 Intro。这个文件夹要配置为由所有的用户访问，而主文件夹只能由授权的用户访问。前面创建的文件夹 Admin 只能由 Editors 角色中的用户访问。

(3) 选择 Visual Studio 2010 中的菜单项 Project | ASP.NET Configuration，启动 ASP.NET Web Application Administration。

(4) 选择 Security 选项卡，如图 18-22 所示。

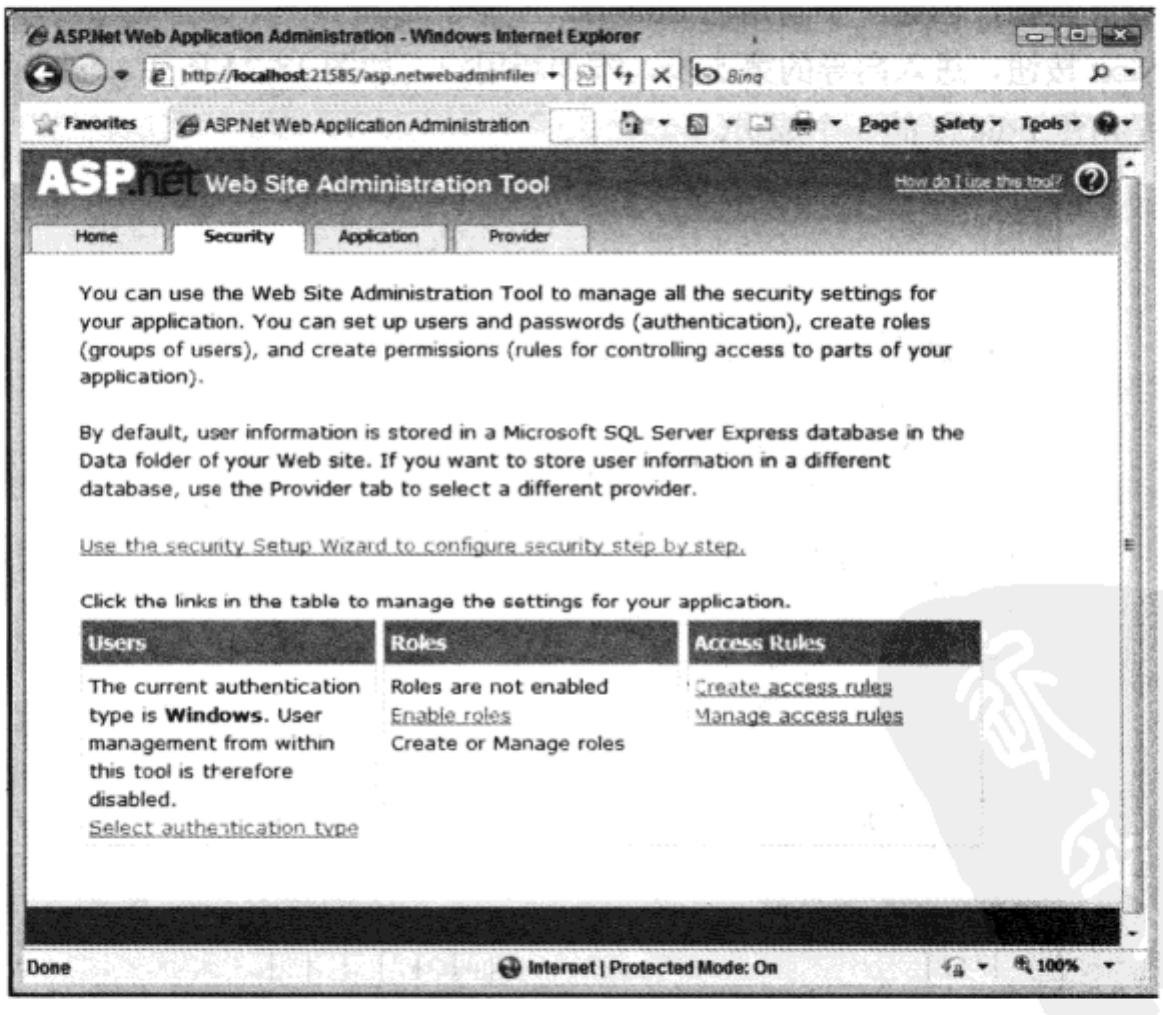


图　18-22

(5) 单击 Security Setup Wizard 链接。在欢迎屏幕中，单击 Next 按钮。在该向导的第 2 步中，选择访问方法 From the internet，如图 18-23 所示。

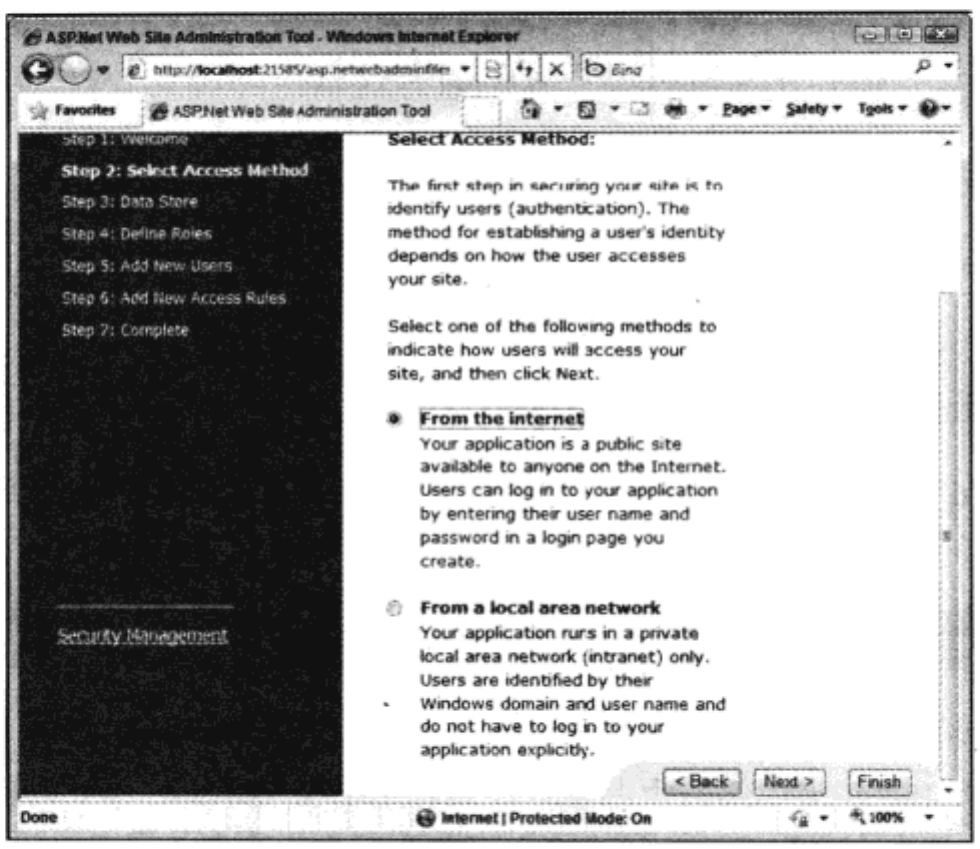


图 18-23

(6) 单击 Next 按钮，进入向导的第 3 步，其中显示了已配置的提供程序信息。默认的提供程序是 SQL Server 数据库提供程序。这个配置不能在向导模式下修改，但可以在稍后进行修改。

(7) 单击 Next 按钮，在 Define Roles 屏幕中，选中复选框 Enable roles for this Web site。

(8) 单击 Next 按钮，创建一个新角色 Editors。

(9) 单击 Next 按钮，进入向导的第 5 步，添加新用户，如图 18-24 所示。创建两个新账户。其中一个账户应是角色 Editors 的一个成员。

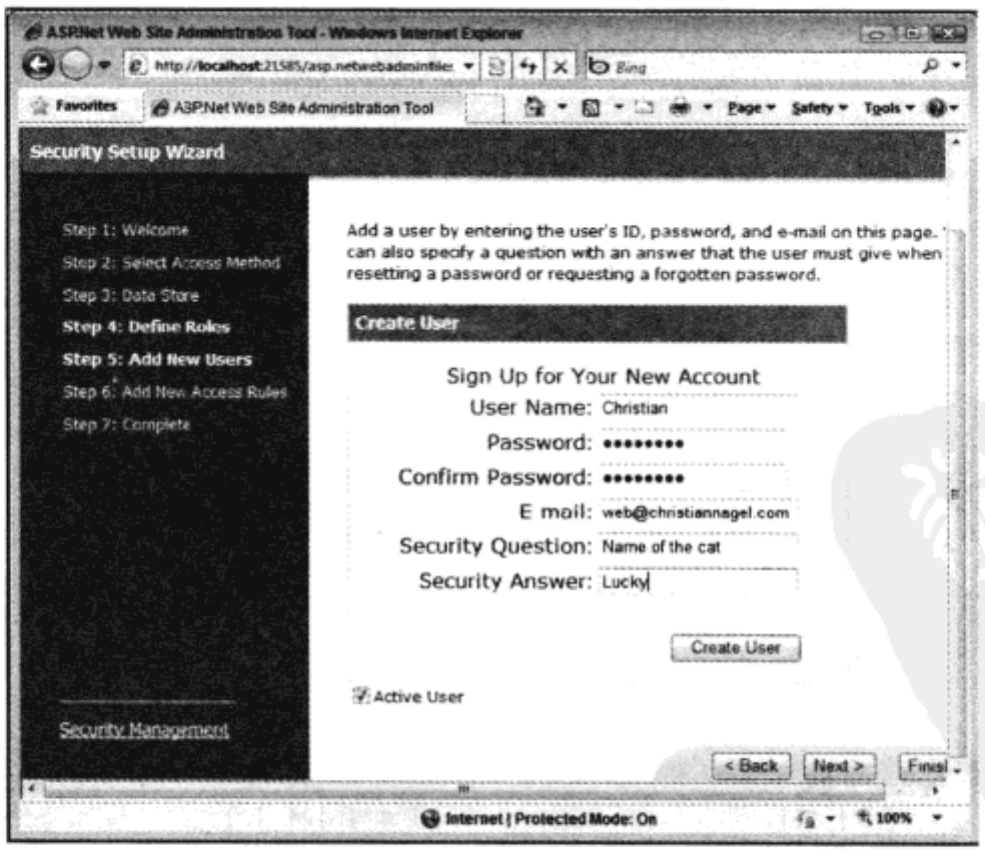


图 18-24

(10) 成功创建用户后，单击 Next 按钮，进入向导的第 6 步，如图 18-25 所示。在这里可以配置哪些用户可以访问网站或特定的目录。添加一个拒绝匿名用户的规则。接着选择目录 Intro，添加一个允许匿名用户访问的规则。选择 Admin 文件夹，拒绝通过身份验证的用户访问，但允许角色 Editors 的用户访问。之后单击 Next 按钮，最后单击 Finish 按钮。

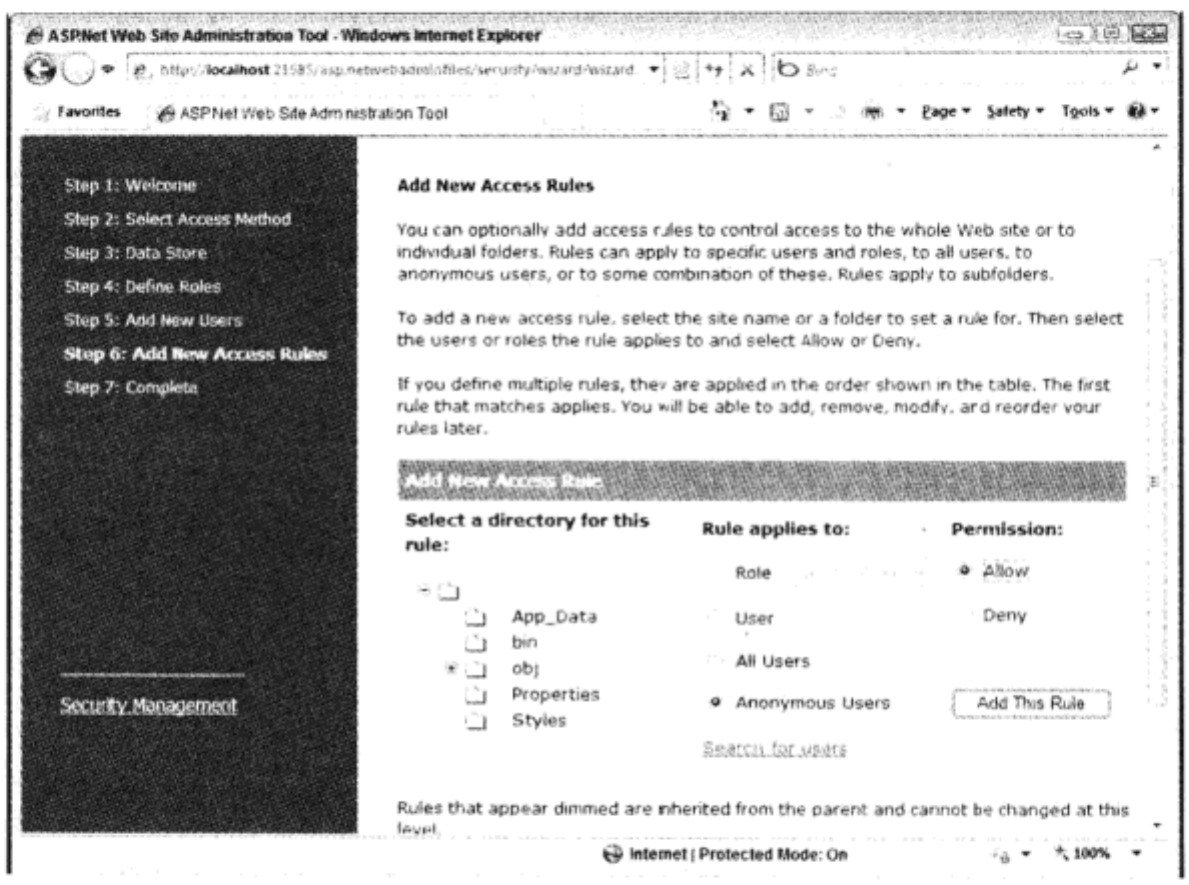


图　18-25

示例的说明

在完成安全配置后，就会创建一个新的 SQL Server 数据库。刷新 Solution Explorer 中的文件，就可以看到目录 App_Data 下的新 SQL Server Express 数据库 ASPNETDB.mdf。这个数据库包含的表由 SQL Membership 提供程序使用。

现在，除了 Web 应用程序之外，还有一个配置文件 web.config。这个文件包含 Forms 身份验证的配置，因为选择了通过 Internet 进行身份验证，<authorization>部分拒绝匿名用户的访问。如果 Membership 提供程序改变了，新的提供程序就会出现在配置文件中。SQL 提供程序是默认的，已经用计算机配置文件定义了，所以不需要列出来。

可从 wrox.com 下载源代码

```
<authorization>
  <deny users="?" />
</authorization>
<roleManager enabled="true" />
<authentication mode="Forms" />
```

代码段 Web.config

子文件夹 Intro 中包含另一个配置文件 web.config。在这个配置文件中没有身份验证部分，因为身份验证配置信息是从父目录中提取的。但是，授权部分是不同的，这里使用<allow users="?" />允许匿名用户的访问。

可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</configuration>
```

代码段 Intro/Web.config

Admin 子文件夹中包含另一个配置文件 web.config。授权部分允许 Editors 角色的访问，拒绝通过身份验证的用户的访问：

可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <system.web>
        <authorization>
            <allow roles="Editors" />
            <deny users="*" />
        </authorization>
    </system.web>
</configuration>
```

代码段 Admin/Web.config

18.12.2 使用安全控件

ASP.NET 包含许多安全控件，现在不用编写一个定制的表单，让用户输入用户名和密码，而可以使用预定义的 Login 控件。安全控件及其功能如表 18-9 所示。

表 18-9

安 全 控 件	说 明
Login	一个复合控件，包含要求用户输入用户名和密码的控件
LoginStatus	根据用户是否登录来决定包含登录或注销的超链接
LoginName	显示用户名
LoginView	根据用户是否登录显示不同的内容
PasswordRecovery	一个复合控件，用于重新设置被遗忘的密码。根据安全配置，要求用户回答以前设置的问题，或通过电子邮件发送密码
ChangePassword	一个复合控件，允许登录的用户改变其密码
CreateUserWizard	一个向导，可以创建新用户，把用户信息写入 Membership 提供程序

在下面的示例中，将给 Web 应用程序添加一个登录页面。

试一试：创建登录页面

如果在网站配置为拒绝匿名用户后启动它，就应接收到一个错误，因为该网站还没有 login.aspx

页面。如果没有用 Forms 身份验证配置某个登录页面，就使用默认的 login.aspx。下面创建一个 login.aspx 页面。

(1) 使用母版页添加一个新的 Web Form，命名为 login.aspx。

(2) 在窗体中添加 Login 控件。

(3) 这就是创建登录页面所需要做的工作。现在启动站点 default.aspx，就会重定向到 login.aspx，在这个窗体中可以为前面创建的用户输入用户凭据。

示例的说明

添加 Login 控件之后，就可以在源代码视图中查看到其代码：

可从 wrox.com 下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
          AutoEventWireup="true" CodeBehind="Login.aspx.cs"
          Inherits="EventRegistrationWeb.Login" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
   <asp:Login ID="Login1" runat="server">
   </asp:Login>
</asp:Content>
```

代码段 Login.aspx

可以通过这个控件的属性来配置标题、用户名标签、密码标签和登录按钮的文本。设置 DisplayRememberMe 属性还可以使复选框 Remember me next time 可见。

如果要更多地控制 Login 控件的外观和操作方式，可以把该控件转换为模板。为此，在设计视图中单击智能标记，选择 Convert to Template。之后，单击 Edit Templates，就得到一个视图，在这里可以添加和修改任何控件。

对于用户凭据的验证，在单击 Login 按钮时，控件会调用 Membership.ValidateUser()方法，不需要手工完成。

如果用户在 EventRegistration 网站上没有登录账户，就应创建自己的账户。为此可以使用 CreateUserWizard 控件，如下面的示例所示。

试一试：使用 CreateUser 向导

(1) 在前面创建的 Intro 文件夹中创建一个新的 Web 页面 RegisterUser.aspx，配置这个文件夹，使之可由匿名用户访问。

(2) 在这个 Web 页面上添加一个 CreateUserWizard 控件。

(3) 把 ContinueDestinationPageUrl 属性设置为~/Default.aspx。

(4) 在 Login.aspx 页面上添加 LinkButton 控件。把这个控件的内容设置为 Register User，PostBackUrl 属性设置为 Web 页面 Intro/RegisterUser.aspx。

(5) 启动应用程序。单击 Login.aspx 页面上的 Register User 链接，就会重定向到 RegisterUser. aspx 页面上，在这个页面上将用输入的数据创建一个新账户。

示例的说明

CreateUserWizard 控件类似于向导，包含多个向导步骤，它们是用<WizardSteps>元素定义的：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
    AutoEventWireup="true" CodeBehind="RegisterUser.aspx.cs"
    Inherits="EventRegistrationWeb.Intro.RegisterUser" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
    <asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
        <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
            </asp:CreateUserWizardStep>
            <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
            </asp:CompleteWizardStep>
        </WizardSteps>
    </asp:CreateUserWizard>
</asp:Content>
```

代码段 RegisterUser.aspx

可以在设计器中配置这些向导步骤。控件的智能标记可以分别配置每个步骤。图 18-26 显示了步骤 Sign Up for Your New Account 的配置。也可以用定制控件添加定制步骤以便添加特定的要求，例如，用户应在签署一个账户前接受合同。

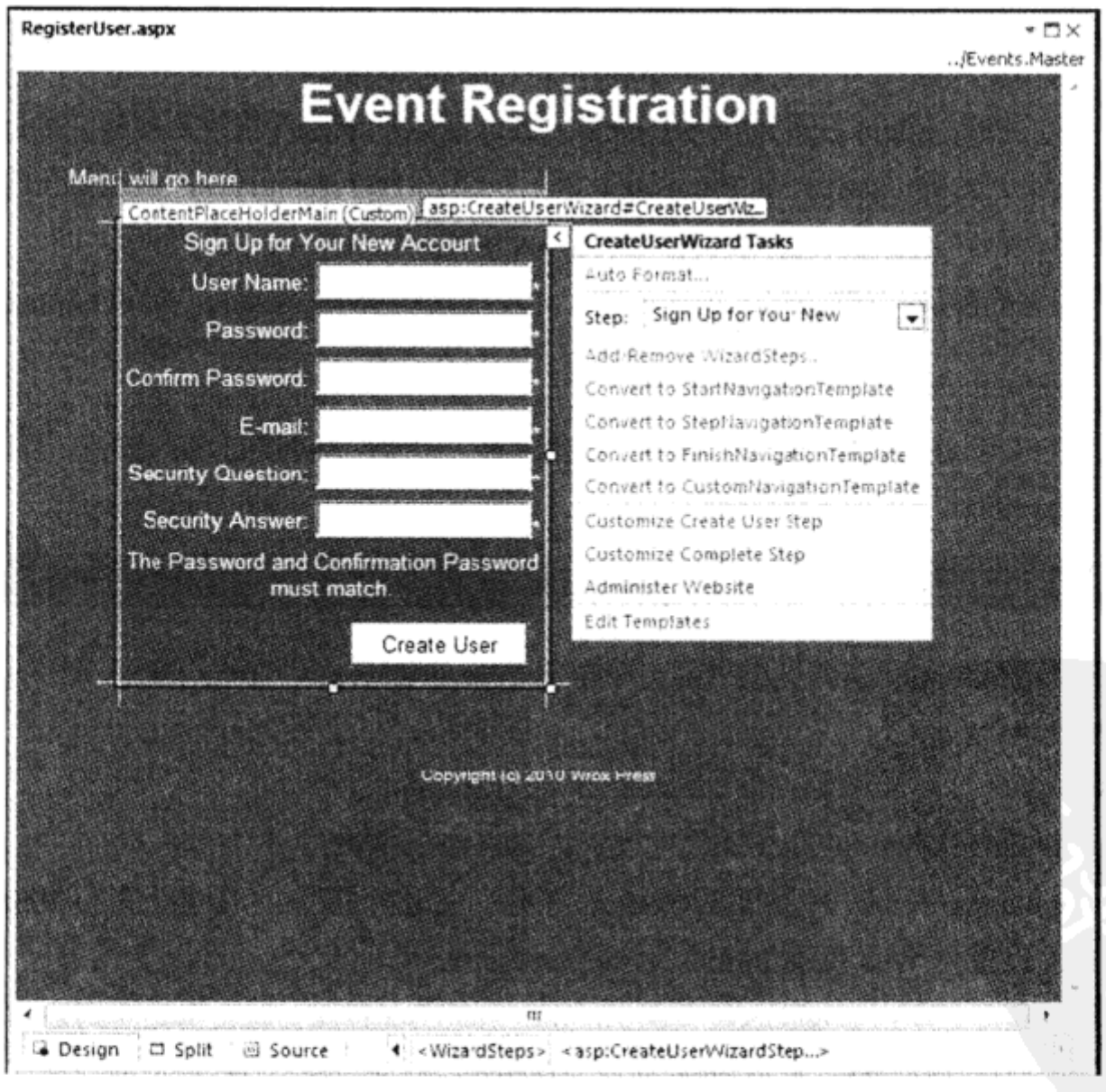


图 18-26

18.13　读写 SQL Server 数据库

大多数 Web 应用程序都需要访问数据库，在其中读写数据。本节将创建一个新数据库，以存储事件信息，说明如何在 ASP.NET 中使用这个数据库。首先在下面的示例中创建一个新的 SQL Server 数据库，这可以直接在 Visual Studio 2010 中完成。

试一试：创建新数据库

(1) 打开前面创建的 Web 应用程序 EventRegistrationWeb。

(2) 打开 Server Explorer。如果在 Visual Studio 中看不到它，可以使用菜单项 View | Other Windows | Server Explorer，打开该窗口。

(3) 在 Server Explorer 中，选择 Data Connections，右击打开关联菜单，选择 Create New SQL Server Database，打开如图 18-27 所示的对话框。

图　18-27

(4) 输入服务器名(local)\sqlexpress 和数据库名 BegVCSharpEvents。

(5) 创建数据库后，在 Server Explorer 中选择新数据库。

(6) 在数据库下选择 Tables 项，再选择 Visual Studio 中的菜单项 Data | Add New | Table。

(7) 现在输入列名和数据类型，如表 18-10 所示。

表　18-10

列　名	数 据 类 型
Id	int
Title	nvarchar(50)
Date	datetime
Location	nvarchar(50)

(8) 通过标识递增 1 和标识种子 1，把 ID 列配置为主键列。所有的列都配置为不允许使用 null 值。

(9) 保存该表，将其命名为 Events。

(10) 在表中添加一些事件，包含标题、日期和位置。

为了显示、编辑数据，工具箱提供了一个独立的 Data 区域，表示数据控件。数据控件可以分为两组：数据视图和数据源。数据源控件与数据源(如 XML 文件、SQL 数据库或.NET 类)关联在一起。数据视图与数据源相连接，就可以表示数据。表 18-11 中列出了所有数据控件。

表 18-11

数 据 控 件	说　明
GridView	用行和列来显示数据
DataList	在一列中显示所有的项
DetailsView	如果数据具有主从(master/detail)关系，DetailsView 控件就可以和 GridView 一起使用
FormView	显示数据源中的一行
Repeater	基于模板的控件，可用于定义从数据源中的数据生成什么 HTML 元素
ListView	基于模板，类似于 Repeater 控件

数据源控件及其功能如表 18-12 中所示。

表 18-12

数据源控件	说　明
SqlDataSource	访问 SQL Server 或其他 ADO.NET 提供程序(例如，Oracle、 ODBC 和 OLEDB)。在内部，它使用 DataSet 或 DataReader 类
AccessDataSource	可以使用 Access 数据库
EntityDataSource	.NET 4.0 中的新控件，允许把 ADO.NET Entity Framework 用作数据源
ObjectDataSource	允许把.NET 类用作数据源
XmlDataSource	允许访问 XML 文件。使用这个数据源可以显示层次结构
SiteMapDataSource	使用 XML 文件定义站点结构，创建到该网站的链接和引用。这个功能将在第 20 章讨论

下面的示例将使用 GridView 控件显示、编辑前面创建的数据库中的数据。

试一试：使用 GridView 控件显示数据

(1) 在 Admin 文件夹中打开前面创建的 Web 页面 EventsManagement.aspx。

(2) 在 Web 页面中添加一个 GridView 控件。

(3) 在控件智能标记的 Choose Data Source 组合框中，选择<New data source.>，打开如图 18-28 所示的对话框。

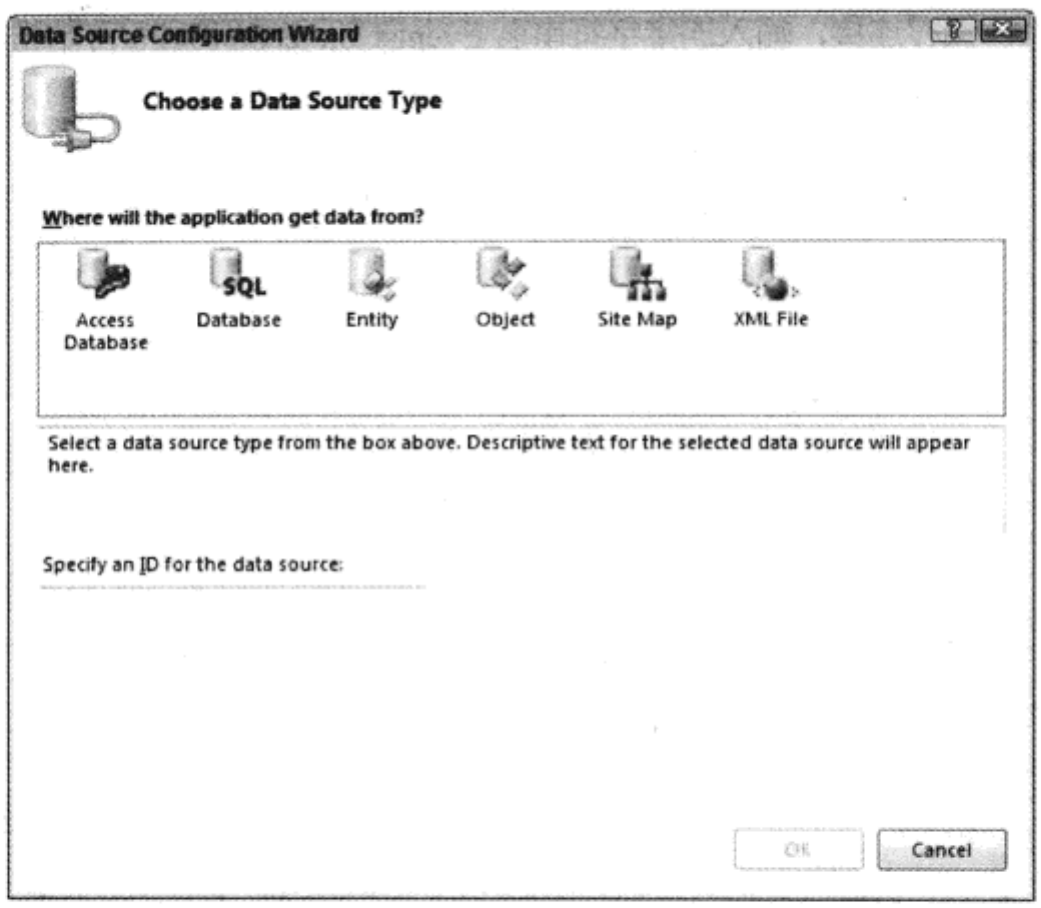


图　18-28

(4) 选择 Database，为这个新数据源输入名称 EventsDataSource。

(5) 单击 OK 按钮配置数据源，打开 Configure Data Source 对话框，单击 New Connection 按钮，创建一个新连接。

(6) 在 Add Connection 对话框中，输入(local)\sqlexpress 作为服务器名，选择前面创建的数据库 BegVCSharpEvents。单击 Test Connection 按钮，验证是否正确配置了连接。满意后单击 OK 按钮。打开下一个对话框(用于存储连接字符串，如图 18-29 所示)。

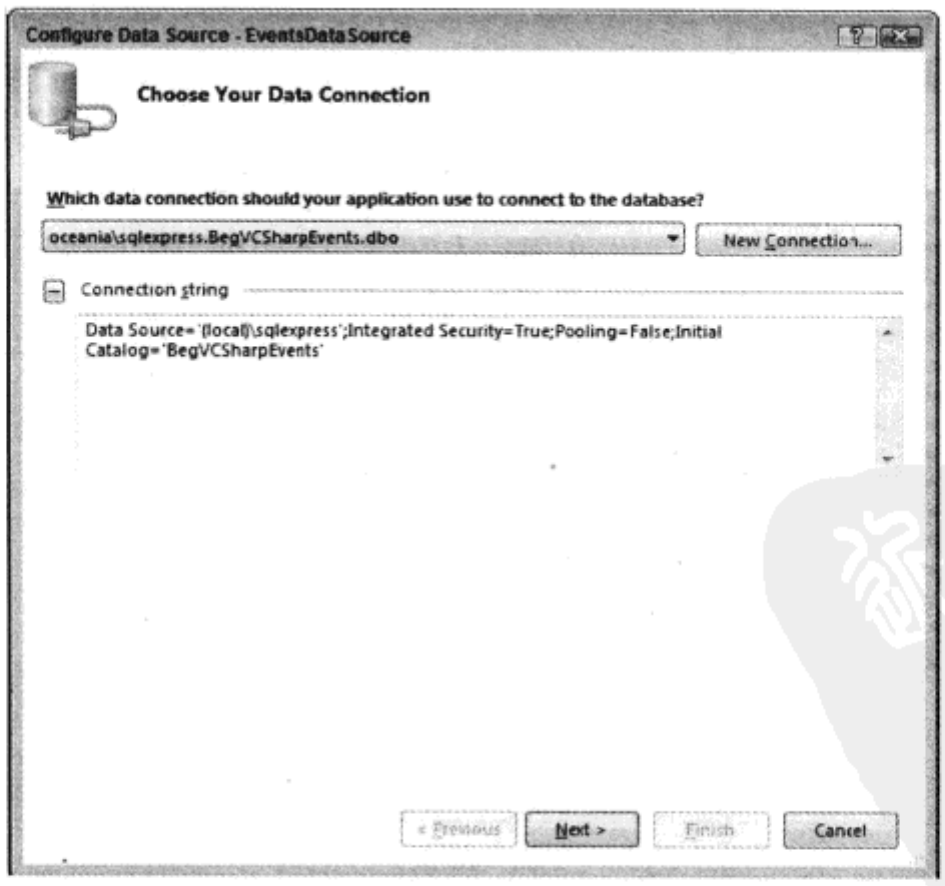


图　18-29

(7) 选中复选框，保存连接，输入连接字符串名 EventsConnectionString，单击 Next 按钮。

(8) 在下一个对话框中，选择 Events 表，以便从这个表中读取数据，如图 18-30 所示。选择 ID、Title、Date 和 Location 列，以定义图中显示的 SQL 命令。接着单击 Next 按钮。

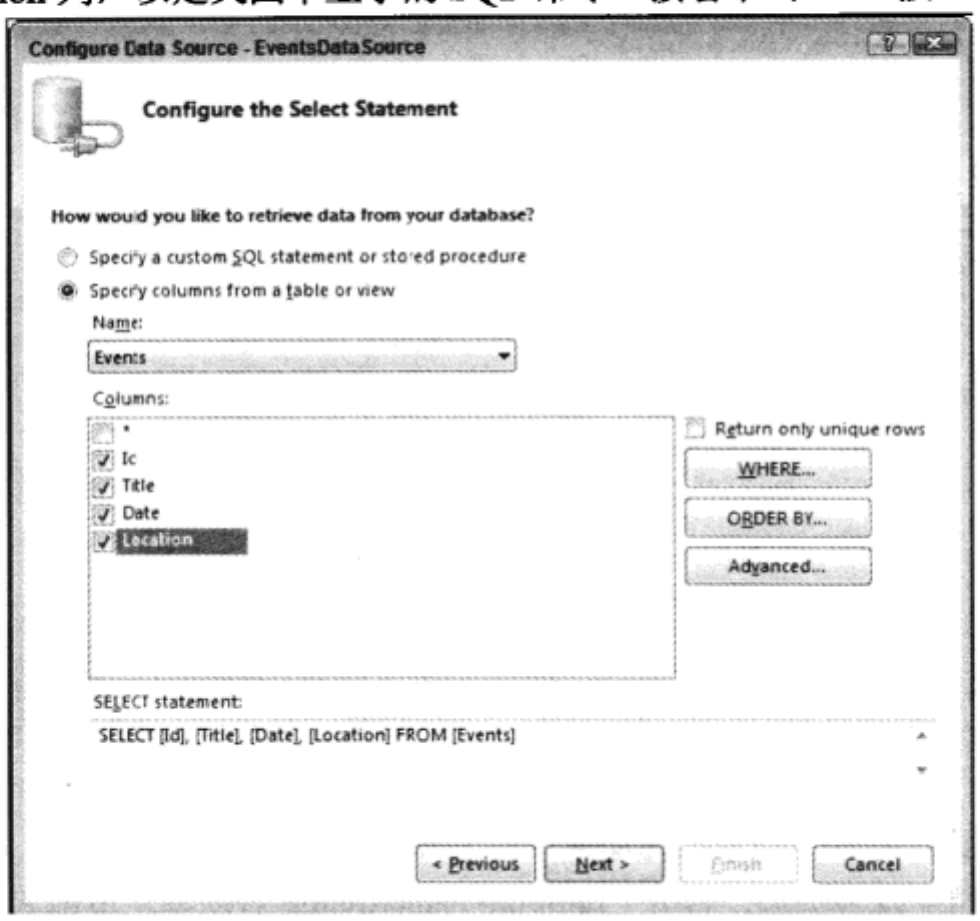


图 18-30

(9) 在 Configure Data Source 对话框的最后一个窗口中，可以测试查询。最后单击 Finish 按钮。

(10) 在设计器中，GridView 控件带有一些虚拟数据，SqlDataSource 的名称是 EventsDatasource。

(11) 为了使 GridView 控件的布局更漂亮，从智能标记中选择 AutoFormat，再选择模式 Mocha，如图 18-31 所示。

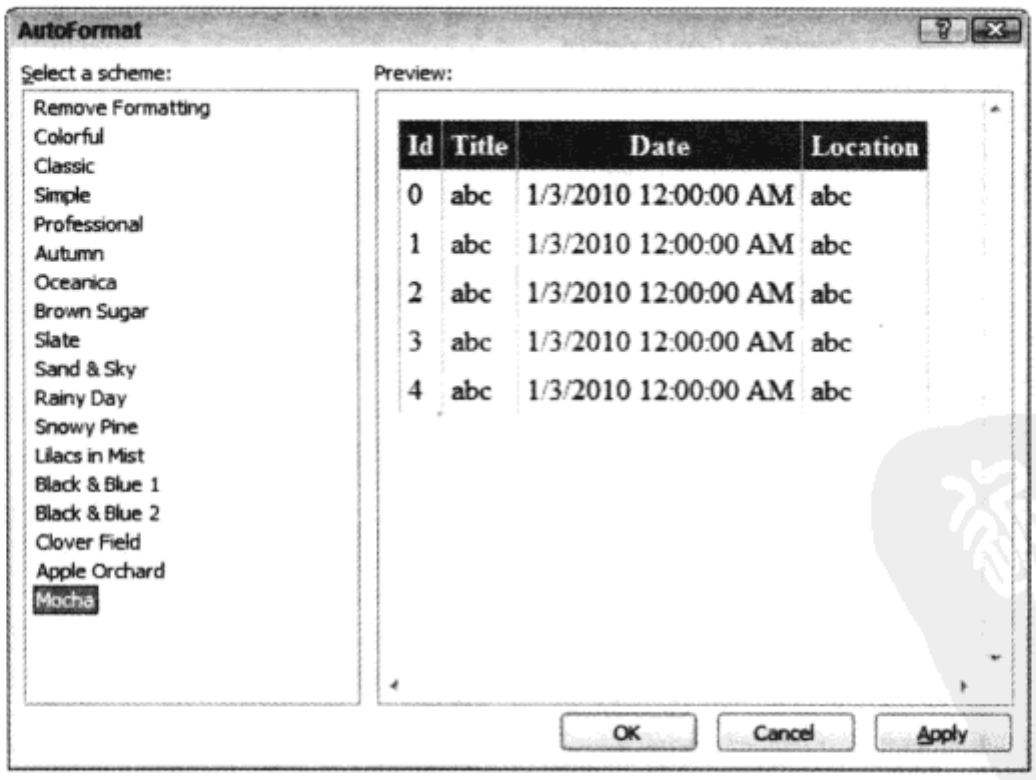


图 18-31

(12) 在 Visual Studio 中启动页面，事件存储在一个美观的表格中，如图 18-32 所示。

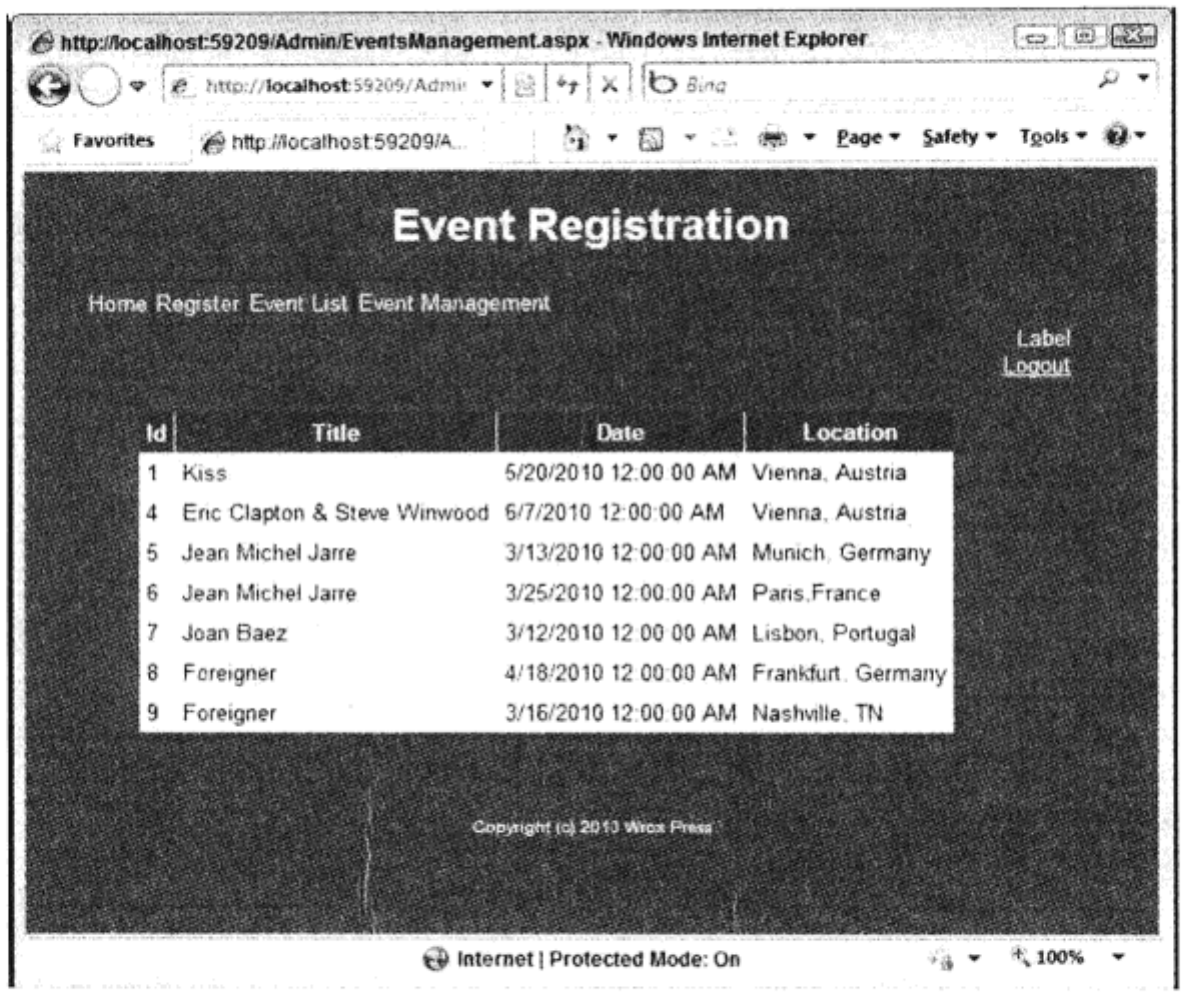


图　18-32

示例的说明

添加了 GridView 控件之后，就可以在源代码中看到它的配置。DataSourceID 特性定义了与数据源控件的关联，数据源控件在栅格控件的后面。在<Columns>元素中，列出了用于显示数据的所有绑定列。HeaderText 定义了标题的文本，DataField 定义了数据源中的字段名。

用<asp:SqlDataSource>元素定义数据源，其中 SelectCommand 定义了如何从数据库中读取数据，ConnectionString 定义了如何连接数据库。因为我们选择在配置文件中保存连接字符串，所以使用<%$建立与配置文件中动态生成的类之间的关联。

可从 wrox.com 下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
          AutoEventWireup="true" CodeBehind="EventsManagement.aspx.cs"
          Inherits="EventRegistrationWeb.Admin.EventsManagement" %>

<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
   <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
                 BackColor="White" BorderColor="#DEDFDE" BorderStyle="None"
                 BorderWidth="1px" CellPadding="4" DataKeyNames="Id"
                 DataSourceID="EventsDataSource" ForeColor="Black"
                 GridLines="Vertical" PageSize="5">
      <AlternatingRowStyle BackColor="White" />
      <Columns>
         <asp:BoundField DataField="Id" HeaderText="Id" InsertVisible="False"
            ReadOnly="True" SortExpression="Id" />
         <asp:BoundField DataField="Title" HeaderText="Title"
            SortExpression="Title" />
         <asp:BoundField DataField="Date" HeaderText="Date"
```

```
            SortExpression="Date" />
        <asp:BoundField DataField="Location" HeaderText="Location"
            SortExpression="Location" />
    </Columns>
    <FooterStyle BackColor="#CCCC99" />
    <HeaderStyle BackColor="#6B696B" Font-Bold="True" ForeColor="White" />
    <PagerStyle BackColor="#F7F7DE" ForeColor="Black"
                HorizontalAlign="Right" />
    <RowStyle BackColor="#F7F7DE" />
    <SelectedRowStyle BackColor="#CE5D5A" Font-Bold="True"
                      ForeColor="White" />
    <SortedAscendingCellStyle BackColor="#FBFBF2" />
    <SortedAscendingHeaderStyle BackColor="#848384" />
    <SortedDescendingCellStyle BackColor="#EAEAD3" />
    <SortedDescendingHeaderStyle BackColor="#575357" />
</asp:GridView>
<asp:SqlDataSource ID="EventsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:BegVCSharpEventsConnectionString %>"
    SelectCommand="SELECT [Id], [Title], [Date], [Location] FROM [Events]">
</asp:SqlDataSource>
</asp:Content>
```

代码段 EventsManagement.aspx

可以在配置文件 web.config 中找到数据库的连接字符串：

```
<connectionStrings>
    <add name="BegVCSharpEventsConnectionString"
        connectionString="Data Source=' (local)\sqlexpress' ;
        Integrated Security=True;Pooling=False;
        Initial Catalog=' BegVCSharpEvents' "
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

代码段 web.config

现在 GridView 控件的配置应有所不同。在下面的示例中，不再向用户显示 ID，只显示日期，不显示时间。

试一试：配置 GridView 控件

(1) 选择 GridView 控件的智能标记，再选择 Edit Columns 菜单。打开如图 18-33 所示的 Fields 对话框。选择 Id 字段，把 Visible 属性改为 False。也可以用这个对话框安排列的位置，改变颜色，定义标题文本。把 Date 列的 DataFormatString 设置为{0:D}，只显示日期，不显示时间。

(2) 要编辑 GridView，必须用数据源定义一个 update 命令。选择名为 EventsDataSource 的 SqlDataSource 控件，再从智能标记中选择 Configure Data Source。在 Configure Data Source 对话框中，单击 Next 按钮，直到看到前面配置的 SELECT 命令为止。单击 Advanced 按钮，选中复选框 Generate INSERT, UPDATE, and DELETE statements，如图 18-34 所示。单击 OK 按钮，再单击 Next 和 Finish 按钮。

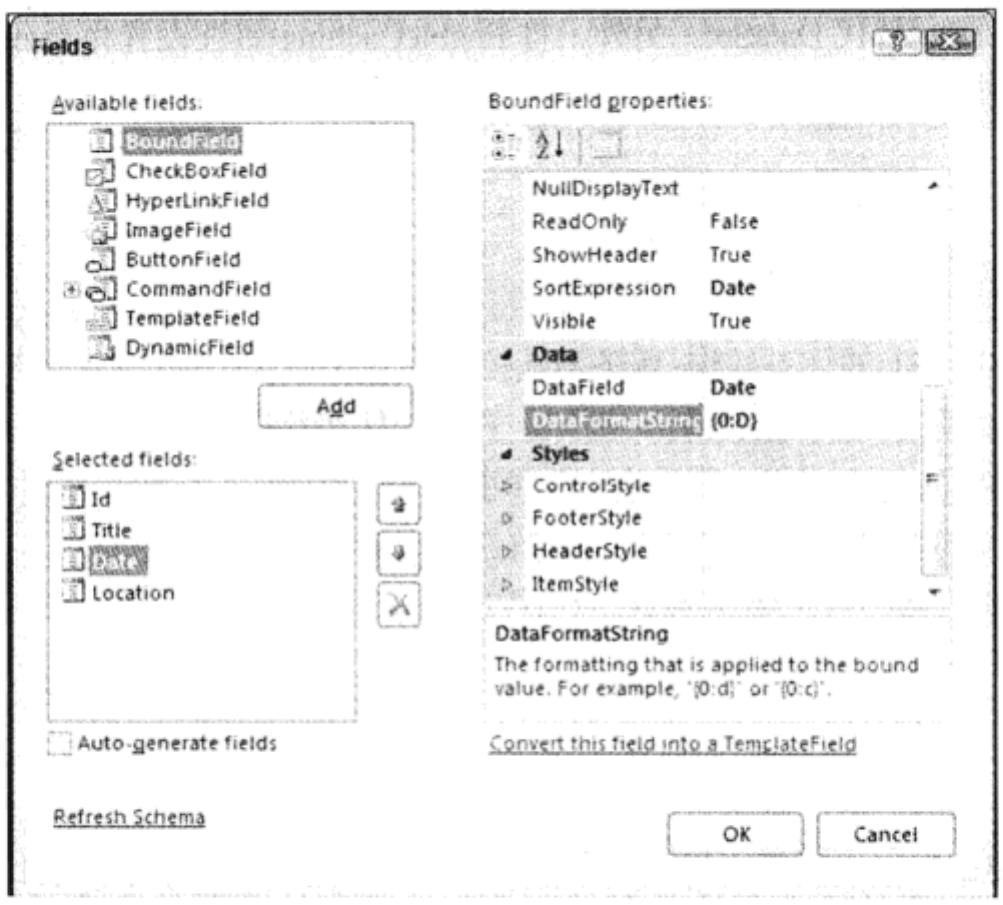


图　18-33

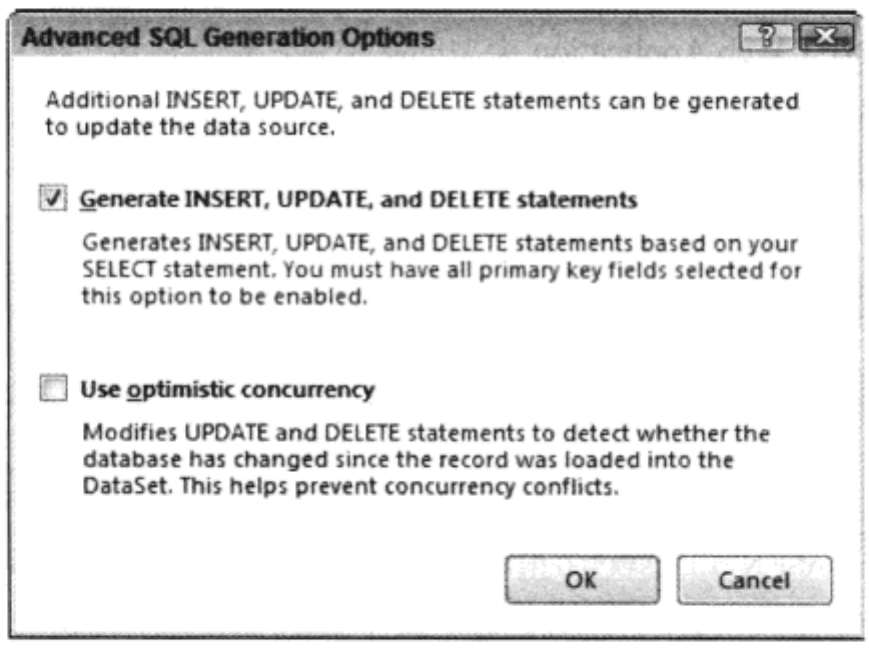


图　18-34

(3) 再次选择 GridView 控件的智能标记。在智能标记菜单中有一个菜单项 Enable Editing。在选中复选框启动编辑后，GridView 控件中会添加一个新列。也可以用智能标记菜单编辑列，以便安排新 Edit 按钮的位置。另外选择 Enable Paging、Enable Sorting 和 Enable Selection 选项。

(4) 启动应用程序，编辑已有的事件记录。单击一个标题，就会按该标题排序。

示例的说明

不必手工编写代码，使用 ASP.NET Web 控件就可以完成上述所有任务。这些控件在后台利用了许多功能。

例如，SqlDataSource 控件在 SqlDataAdapter 的帮助下，用数据库中的数据填充 DataSet。用于填充 DataSet 的数据是通过连接字符串和 SELECT 命令定义的。改变 SqlDataSource 的一个属性，就

可以使用 SqlDataReader 替代 DataSet。另外，将 EnableCaching 属性设置为 true，会自动使用 Cache 对象(本章前面讨论过)。

18.14 小结

本章学习了 ASP.NET 的体系结构、服务器端控件的用法和 ASP.NET 的一些基本功能。ASP.NET 提供了许多不需要编写太多代码的控件，例如，登录控件和数据控件。

学习了 ASP.NET 的基本功能、服务器控件和事件处理机制后，讨论了如何进行输入验证、状态管理的几种方法、身份验证和授权，以及显示数据库中的数据。

下面的练习帮助扩展本章开发的 Web 应用程序。

18.15 练习

(1) 在本章创建的母版页顶部添加用户名。可以使用 LoginName 控件。使用 LoginView，从而仅给通过身份验证的用户显示这个信息。

(2) 修改 Registration.aspx 页面的数据源，以使用 Events 数据库来显示事件。

(3) 创建一个 ASP.NET Web Application 类型的新项目。检查从这个项目模板中创建的所有文件和文件夹。它们看起来应很熟悉。

附录 A 给出了练习答案。

18.16 本章要点

主 题	重 要 概 念
使用 Web 服务器控件	Web 服务器控件是生成 HTML 代码的服务器端控件。这些控件的用法类似于 Windows 控件
使用 ASP.NET 回送	ASP.NET 回送模型在编写 ASP.NET Web 应用程序时是一个非常重要的概念。服务器端代码仅对服务器上的回送信息起作用。目前有了 ASP.NET Ajax，还可以定义 ASP.NET Ajax 回送，仅更新页面的一部分
通过验证控件验证用户的输入	ASP.NET 提供了几个验证控件，可以方便地验证客户端和服务器端的用户输入。客户端的验证是为了提高性能，但因为 Web 客户从来都不是可信任的，所以还必须在服务器端进行验证
状态管理	在 Web 应用程序中，必须考虑在何处存储状态。状态可通过 cookie 或视图状态用于客户端，在服务器端，则通过会话、缓存和应用程序对象来使用状态
母版页	母版页用于包含多个页面中的公共部分
导航	菜单控件可用于在网站的不同页面之间导航。可以把站点地图绑定到菜单上，而无需把页面的链接直接添加到菜单控件上
读写 SQL Server 数据库	通过 ASP.NET 控件抽象出了数据库的访问。GridView 很容易在设计器上定制。这个网格的数据源只需设置属性，就可以在数据库中读写数据，而无需编写 C#代码

第19章

Web 服务

本章内容：

- Web 服务概述
- 如何使用 ASP.NET 创建 Web 服务
- 如何通过 Windows 窗体应用程序使用 Web 服务
- 如何在 ASP.NET 客户程序中使用 Web 服务
- 如何异步调用 Web 服务
- 如何在 Web 服务之间传送数据

Web 应用程序是用户访问应用程序的功能的前端，而 Web 服务是应用程序访问应用程序的功能的前端。Web 服务是服务器端的程序，用以监听来自客户应用程序的消息，并返回特定的信息。这些信息可能来自 Web 服务本身，同一个域中的其他组件，或其他 Web 服务。

本章并不深入探讨 Web 服务的内部工作原理，但提供了足够多的通过 VS 开始创建和使用简单 ASP.NET Web 服务的信息。

19.1 使用 Web 服务的场合

为了从另一个角度来了解 Web 服务，需要区分用户—应用程序通信和应用程序—应用程序通信之间的区别。首先看看用户—应用程序通信，从 Web 上获得某些天气信息。weather.msn.com 和 www.weather.com 等多个网站都提供了天气信息，其格式用户都很容易理解。用户可以直接阅读这些页面。

如果要创建一个富客户应用程序来显示天气(应用程序—应用程序通信)，该应用程序必须连接到一个 Web 站点上，其 URL 字符串包含某个城市，我们想知道该城市的天气。接着要分析从 Web 站点返回的 HTML 信息，得到气温和天气信息，最后以合适的格式向富客户应用程序显示这些信息。

即使只想得到某个城市的气温信息，工作量也很大。而且，从 HTML 中获得数据的过程并不是很简单。这是因为 HTML 数据主要显示在 Web 浏览器上，并非供其他客户端商务应用程序使用。

因此，数据内嵌在文本中，不容易提取，需要重新编写或修改客户应用程序，从同一个 Web 页上提取不同的数据信息。而使用 Web 浏览器，用户可以立即得到需要的数据，忽略不需要的内容。

为了解决处理 HTML 数据的问题，Web 服务提供了一种有效的方式，可以只返回请求的数据。只要调用远程服务器上的一个方法，获得需要的信息，客户应用程序就可以直接使用这些信息了。我们不必处理对用户界面有意义的、预先格式化的文本，因为 Web 服务以 XML 格式显示信息，处理 XML 数据的工具早就有了。客户应用程序只需调用.NET Framework XML 类的一些方法，就可以得到需要的数据。更妙的是，如果用 C#编写.NET Web 服务的客户应用程序，甚至不需要编写执行这些任务的代码，有一些可以自动生成 C#代码的工具！

刚才讨论的天气应用程序是使用 Web 服务的一个示例，Web 服务还有其他许多用途。

19.1.1 宾馆旅行社代理应用程序

如何预订假日宾馆？不必求助于旅行社代理处，可以在 Internet 上预订假日宾馆。在某条航线的 Web 站点上，可以查找可能的航线，并预订它们。可以使用 Web 搜索引擎查找某个城市的宾馆。在许多情况下，还可以找到去宾馆的地图。在找到该宾馆的主页后，就可以导航到预订窗体页面，预订房间。接着可以搜索一个汽车租赁公司……

现在要做的是利用搜索引擎查找 Web 站点，接着浏览这些站点。另外，可以创建一个家庭旅行代理应用程序，它使用包含宾馆、航线和汽车租赁公司等信息的 Web 服务，给客户提供了一种方便的方式来处理所有的假日旅行问题，包括提前预订某场音乐会。在假日里使用移动设备，就可以使用相同的 Web 服务得到娱乐活动用的地图，文化活动或电影院的某些信息等。

19.1.2 图书发布应用程序

Web 服务还可以用于两个拥有合作关系的公司。假定图书出版商要得到书店销售图书的信息，就可以用一个 Web 服务来实现。可以建立一个使用该 Web 服务的 ASP.NET 应用程序，直接给用户提供这个服务。这个服务的另一个客户应用程序是书店的 Windows 应用程序，首先检查本地书店，再检查分销商的书店。销售员可以立即答出交货日期，无需用其他应用程序检查不同的书店。

19.1.3 客户应用程序的类型

Web 服务的客户应用程序可以是使用 Windows 窗体、WPF、Silverlight 创建的 Windows 应用程序，也可以是使用 Web 窗体创建的 ASP.NET 应用程序，Windows PC、UNIX 系统和移动设备都可以使用该 Web 服务。在.NET Framework 中，Web 服务可以用于各类应用程序中。

19.2 应用程序的体系结构

使用 Web 服务的应用程序究竟是什么样的？无论是开发 ASP.NET 或 Windows 应用程序，还是开发小设备的应用程序，从前面介绍的应用可以看出，Web 服务的调用是非常类似的。在这些类型的应用程序中，Web 服务都是非常重要的技术。

图 19-1 说明了使用 Web 服务的方式。设备和浏览器通过 Internet 连接到一个用 Web 窗体开发的 ASP.NET 应用程序上。这个 ASP.NET 应用程序使用某些本地 Web 服务，以及可以通过网络获得的其他远程 Web 服务：门户 Web 服务、用于某个应用程序的 Web 服务和构件 Web 服务。下面的列

表详细描述了这些服务类型的含义：

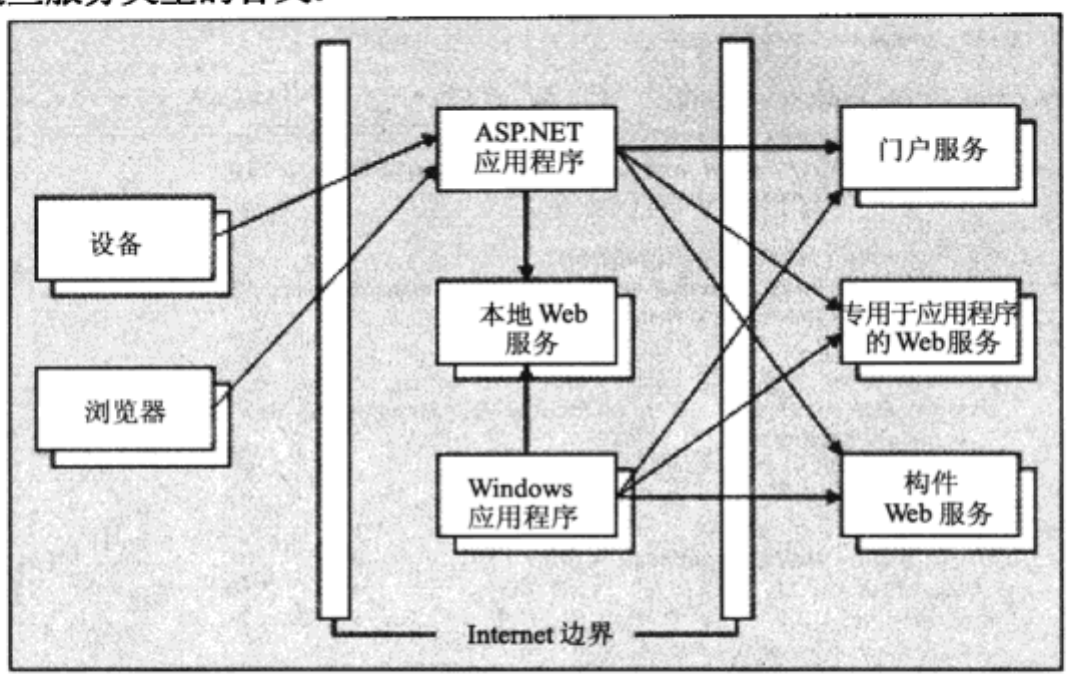


图　19-1

- **门户 Web 服务**：使用同一个主题提供不同公司的服务。这将便于使用多个服务的入口点。
- **用于某个应用程序的 Web 服务**：仅用于某个应用程序。
- **构件 Web 服务**：可以在多个应用程序中使用。

图 19-1 中的 Windows 应用程序可以直接使用 Web 服务，无需通过 ASP.NET 应用程序。

19.3　Web 服务的体系结构

Web 服务利用 SOAP 协议，SOAP 协议是许多公司定义的标准。Web 服务的一个主要优点在于它的平台独立性。当需要在多个平台上协作时，Web 服务是一种有用的技术。在客户端和服务器端开发.NET 应用程序时，Web 服务也是一种很有用的技术。其优点是客户端和服务器都可以独立存在。服务描述是用 WSDL(Web 服务描述语言，Web Service Description Language)文档定义的，它可以用独立于 Web 服务新版本的方式设计，因此无需改变客户端。

下面详细介绍这一系列步骤。

19.3.1　调用方法和 WSDL

WSDL 文档包含了下述信息：Web 服务支持什么方法，如何调用这些方法，给服务传送的参数类型，以及从服务返回的参数类型。图 19-2 显示了 ASP.NET 运行库生成的 WSDL。在.asmx 文件的最后加上字符串?wsdl，以便返回一个 WSDL 文档。

不必直接处理这些信息。WSDL 文档是用 WebMethod 特性动态生成的；本章的后面会介绍这个属性。使用 Visual Studio 添加对客户应用程序的 Web 引用，就可以请求 WSDL 文档。这个 WSDL 文档用于创建带有相同方法和参数的客户代理程序。而利用这个代理程序，客户应用程序就只需在服务器中执行时调用方法，因为代理程序会把它们转换为 SOAP 调用，通过网络进行调用。

WSDL 规范由 World Wide Web Consortium(W3C)维护，可以在 W3C 网站 www. w3.org/ TR/wsdl 上阅读这个规范。

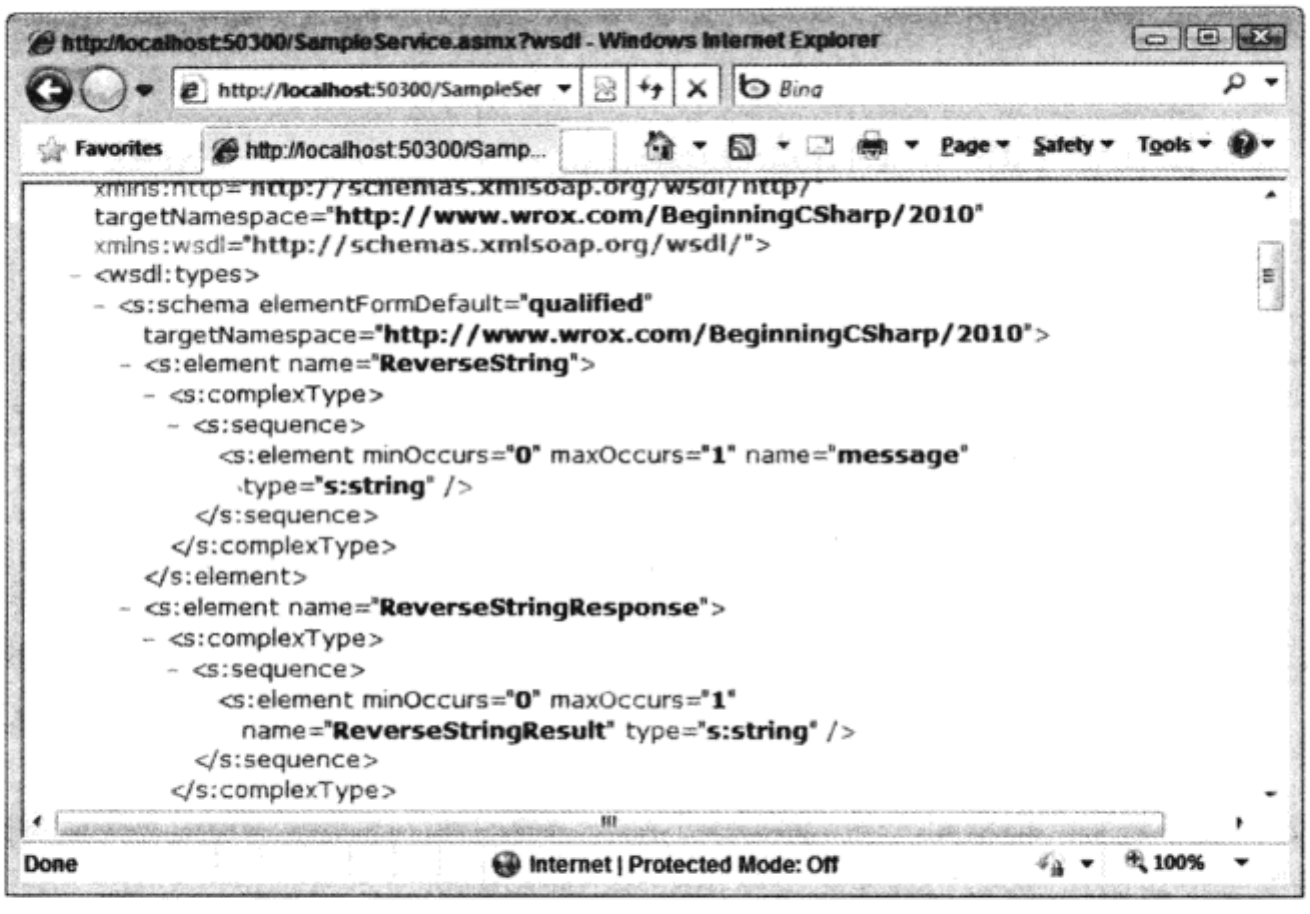


图 19-2

19.3.2 调用方法

SOAP 消息是客户机和服务器之间的基本通信单元。要调用 Web 服务上的一个方法，该调用必须转换为 SOAP 消息，因为它是在 WSDL 文档中定义的。图 19-3 显示了 SOAP 消息的一部分。SOAP 封套(envelop)把所有的 SOAP 消息封装在一个块中。SOAP 封套本身由两部分组成：SOAP 标题和 SOAP 体。标题是可选的，它定义了客户机和服务器应如何处理 SOAP 体。SOAP 体是必须有的，它包括发送的数据，通常 SOAP 体中的信息是要调用的方法和序列化的参数值。SOAP 服务器在 SOAP 消息的消息体中返回值。

图 19-3

在下面的示例中，说明了从客户机发送给服务器的 SOAP 消息是什么样的。客户机调用 Web 服务方法 ReverseString()。字符串 Hello World! 作为这个方法的参数传送。该方法调用位于 SOAP 体中，位于 XML 元素<soap:Body>中。消息体本身包含在封套<soap:Envelope>中。在 SOAP 消息的开头，有 HTTP 标题，因为 SOAP 消息通过 HTTP POST 请求来发送。

未必要创建这样一个消息，因为它可以由客户代理程序创建：

```
POST /Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 508
SOAPAction: "http://www.wrox.com/webservices/ReverseString"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseString xmlns="http://www.wrox.com/webservices">
      <message>Hello World!</message>
    </ReverseString>
  </soap:Body>
</soap:Envelope>
```

服务器用 SOAP 消息!dlroW olleH 作为响应，如 XML 元素 ReverseStringResult 所示：

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 446

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseStringResponse xmlns="http://www.wrox.com/webservices">
      <ReverseStringResult>!dlroW olleH</ReverseStringResult>
    </ReverseStringResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP 规范由 W3C 的 XML Protocol Working Group 维护(请访问 www.w3.org/TR/soap)。

19.3.3 WS-I 规范

SOAP 规范和其他基于 SOAP 的规范已经存在一段时间了。其间有很多变化，出现了多个版本，从而增大了交互难度。为了解决这个问题，成立了 Web 服务交互操作组织(Web Services Interoperability Organization (http://ws-i.org))。这个组织用 WS-I Basic Profile 规范定义了 Web 服务的需求。WS-I Basic Profile 详见 www.ws-i.org。用 ASP.NET 开发的 Web 服务遵循 Basic Profile 1.1，它在 http://www.ws-i.org/Profiles/BasicProfile-1.1.html 文档中定义。

19.4 Web 服务和.NET Framework

在.NET Framework 中，很容易创建和使用 Web 服务。下面列出的 3 个主要名称空间可以处理 Web 服务：

- System.Web.Services 名称空间中的类用于创建 Web 服务。
- 使用名称空间 System.Web.Services.Description，可以通过 WSDL 描述 Web 服务。

- 使用 System.Web.Services.Protocols，可以创建 SOAP 请求和响应。

可以使用 ASP.NET 或 WCF 创建 Web 服务。WCF 比 ASP.NET Web 服务灵活得多，因为它给 Web 服务的驻留提供了不同选项，而不仅限于 ASP.NET，还支持不同的协议，而不仅限于 HTTP。ASP.NET Web 服务的优点是更容易使用。另外，WCF 模板不能在 VS2008 的 Express 版本中使用，在编写本书时，WCF 模板能否在 VS2010 中使用还不清楚。第 27 章介绍 WCF。

19.4.1 创建 Web 服务

要实现 Web 服务，可以从 System.Web.Services.WebService 中派生 Web 服务类。WebService 类提供了对 ASP.NET Application 和 Session 对象的访问，这个类是可选的，只有需要访问该类提供的属性时，才需要从这个类中派生。

表 19-1 列出了 WebService 类的一些属性。

表 19-1

属 性	说 明
Application	为当前请求返回一个 HttpApplicationState 对象
Context	返回一个封装 HTTP 特定信息的 HttpContext 对象。在这里，可以读取 HTTP 标题信息
Server	返回一个 HttpServerUtility 对象。这个类有一些帮助方法，可以进行 URL 编码和解码
Session	返回一个 HttpSessionState 对象，以存储客户机的一些状态信息
User	返回一个实现接口 IPrincipal 的用户对象。使用这个接口可以得到用户名和身份验证类型
SoapVersion	返回 Web 服务使用的 SOAP 版本。SOAP 版本封装在 SoapProtocolVersion 枚举中

1. WebService 特性

用特性 WebService 来标记 WebService 的子类。WebServiceAttribute 类有下述属性，如表 19-2 所示。

表 19-2

属 性	说 明
Description	服务的描述信息，可用于 WSDL 文档
Name	获取或设置 Web 服务名称
Namespace	获取或设置 Web 服务的 XML 名称空间。其默认值是 http://tempuri.org，它用于测试，但在公开这个服务前，应修改该名称空间

2. WebMethod 特性

Web 服务中可供使用的所有方法都必须用 WebMethod 特性来标记。当然，服务还可以有其他

没有用 WebMethod 标记的方法。这些方法可以从 WebMethod 中调用，但不能在客户机上调用。使用特性类 WebMethodAttribute，就可以在远程客户机上调用方法，并可以定义是否缓存响应，缓存的有效使用时间，会话状态是否与指定的参数一起存储。表 19-3 列出了 WebMethod Attribute 类的属性。

表　19-3

属　　性	说　　明
BufferResponse	获取或设置表示是否应缓存响应的标志。默认值为 true。使用被缓存的响应，仅可以将已完成的数据包传递给客户机
CacheDuration	设置结果应缓存的时长。如果在这个属性设置的时间段中第二次发出了相同的请求，就返回缓存的值。默认值为 0，这表示不缓存结果
Description	用于给预期的用户生成服务帮助页面
EnableSession	布尔值，表示会话状态是否有效。默认值是 false，因此 WebService 类的 Session 属性不能用于存储会话状态
MessageName	默认状态下，把消息名设置为方法名
TransactionOption	这个属性表示方法的事务处理支持。默认值是 Disabled

3. WebServiceBinding 特性

特性 WebServiceBinding 用于把 Web 服务标记为可交互操作的一致性级别。表 19-4 中列出了 WebServiceBindingAttribute 类的一些属性。

表　19-4

属　　性	说　　明
ConformsTo	设置为 WsiProfile 枚举的一个值。WsiProfile 有两个值：Web 服务遵循 Basic Profile 1.1 时，其值为 BasicProfile1_1；没有定义任何一致性级别时，其值为 None
EmitConformanceClaims	一个布尔属性，定义了用 ConformanceClaims 属性指定的一致性级别是否应传送给生成的 WSDL 文档
Name	定义绑定的名称。该名称默认情况下与 Web 服务相同，但末尾加上了 Soap 字符串
Location	定义了绑定消息的位置，例如 http://www.wrox.com/DemoWebservice.asmx?wsdl
Namespace	定义了绑定的 XML 名称空间

19.4.2　客户程序

要调用一个方法，客户机必须创建一个与 Web 服务所在的服务器之间的 HTTP 连接，并发送一个 HTTP 请求，以便传送 SOAP 消息。方法调用必须转换为 SOAP 消息。这些都是由客户代理程序实现的。客户代理程序的实现代码在 SoapHttpClientProtocol 类中。

1. SoapHttpClientProtocol

类 System.Web.Services.Protocols.SoapHttpClientProtocol 是客户代理程序的基类。 Invoke()方法转换参数，建立一个 SOAP 消息，发送给 Web 服务。调用哪个 Web 服务由 Url 属性确定。

SoapHttpClientProtocol 类也支持使用 BeginInvoke()和 EndInvoke()方法的异步调用。

2. 其他客户协议

如果不使用 SoapHttpClientProtocol 类，还可以使用其他代理程序类，HttpGetClientProtocol 和 HttpPostClientProtocol 仅执行简单的 HTTP GET 或 HTTP POST 请求，没有 SOAP 调用的系统开销。

如果解决方案在客户机和服务器上使用.NET，就可以使用 HttpGetClientProtocol 和 HttpPostClientProtocol 类。如果要使用不同的技术，就必须使用 SOAP 协议。

比较 HTTP POST 请求和本章前面介绍的 SOAP 调用：

```
POST /WebServiceSample/Service1.asmx/ReverseString HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

message=string
```

HTTP GET 请求甚至更短，GET 请求的缺点是发送的参数长度是有限制的。如果其长度超过了 1K，就应考虑使用 POST：

```
GET /WebServiceSample/Service1.asmx/ReverseString?message=string HTTP/1.1
Host: localhost
```

HttpGetClientProtocol 和 HttpPostClientProtocol 的开销比 SOAP 方法小，但缺点是其他平台上的 Web 服务不支持它，且只能发送简单数据。

19.5 创建简单的 ASP.NET Web 服务

下面用 Visual Studio 创建一个简单的 Web 服务。

试一试：创建一个 Web 服务项目

(1) 选择 File | New | Project，再选择 ASP.NET Empty Web Application 模板，创建一个新的 Web 服务项目，如图 19-4 所示。把项目命名为 WebServiceSample，单击 OK 按钮。

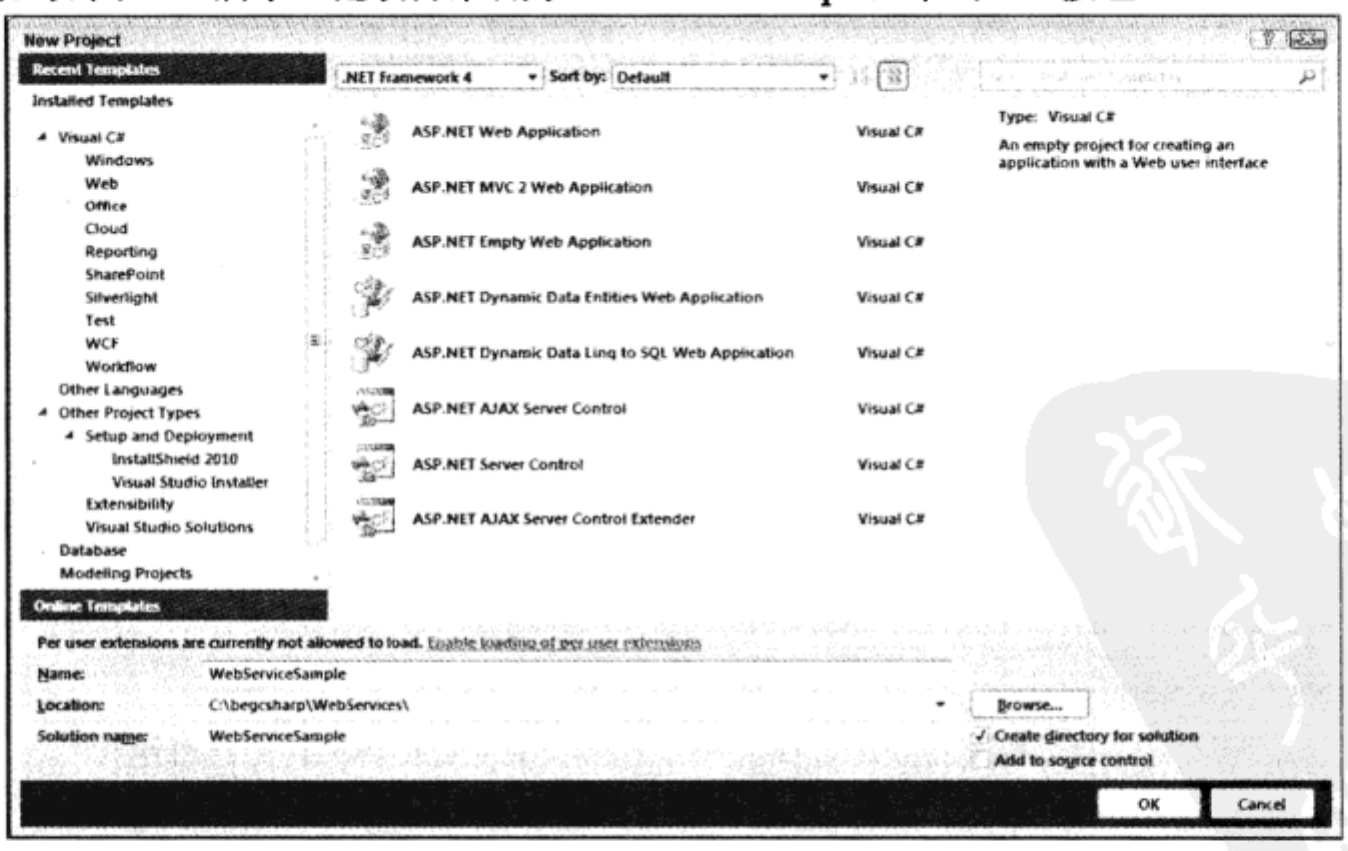


图 19-4

(2) 添加一个新项，选择 Web Service 模板，把所创建的文件命名为 SampleService.asmx，如图 19-5 所示。

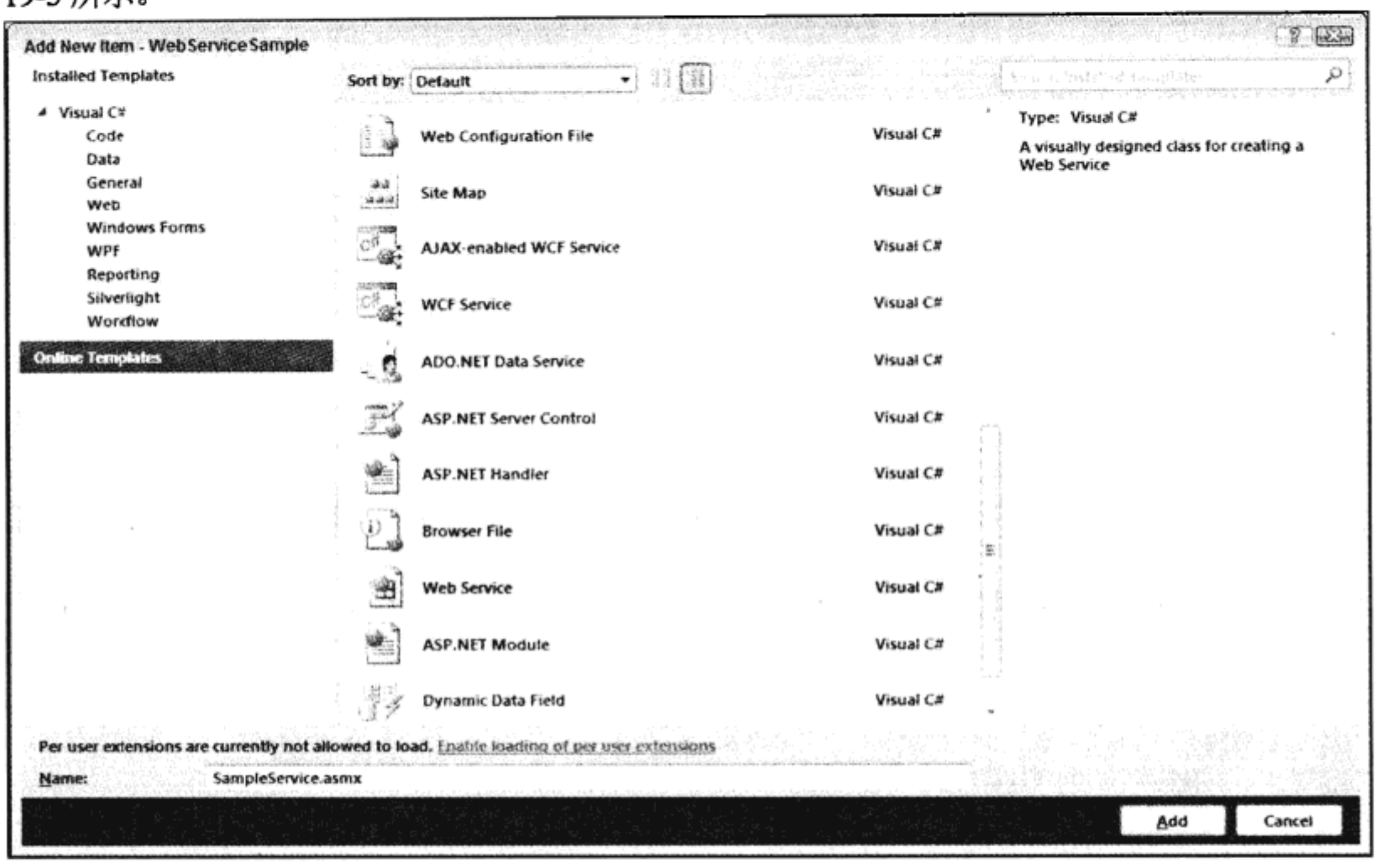


图　19-5

示例的说明

项目模板和项模板生成的文件如下所示：

- SampleService.asmx：保存 Web 服务类。所有的 ASP.NET Web 服务都用.asmx 扩展名来标识。包含源代码的文件是 SampleService.asmx.cs，因为在 Visual Studio 中可以使用后台编码功能。
- SampleService.asmx.cs：项模板在该文件中生成了一个派生自 System.Web.Services.WebService 的类 SampleService。在这个文件中，一些示例代码说明了 Web 服务的方法是如何编码的 ——它应是公共的，用 WebMethod 特性标记：

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

namespace WebServiceSample
{
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    public class SampleService : System.Web.Services.WebService
    {
        [WebMethod]
```

```
        public string HelloWorld()
        {
            return "Hello World";
        }
    }
}
```

代码段 WebServiceSample/SampleService.asmx.cs

添加 Web 方法

接着给 Web 服务添加一个定制方法。这里添加一个简单方法 ReverseString()，它接收一个字符串，颠倒该字符串中的字符顺序后，将其返回给客户机。

试一试：添加一个方法

(1) 删除方法 HelloWorld()及其全部实现代码。在 SampleService.asmx.cs 文件中添加下述代码。

可从 wrox.com 下载源代码

```
[WebMethod]
public string ReverseString(string message)
{
    return new string(message.Reverse().ToArray());
}
```

代码段 WebServiceSample/SampleService.asmx.cs

(2) 修改 SampleService.asmx.cs 文件中的示例代码，如下所示：

```
[WebService(Namespace = "http://www.wrox.com/BeginningCSharp/2010")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class SampleService : System.Web.Services.WebService
```

代码段 WebServiceSample/SampleService.asmx.cs

(3) 编译项目。

示例的说明

ASP.NET 运行库使用反射技术读取 Web 服务专用的一些特性，例如[WebMethod]，把方法提供为 Web 服务的操作。ASP.NET 运行库还提供了 WSDL 来描述服务。

要在为 Web 服务生成的描述信息中唯一地标识 XML 元素，应添加一个 XML 名称空间。这个示例在类 Service 中用[WebService]特性添加了名称空间 http://www.wrox.com/webservices。当然，可以使用其他字符串唯一地标识 XML 元素，例如，公司页面的 URL 链接。Web 链接不一定存在，它仅用于唯一标识。如果使用基于公司网址的名称空间，就可以确保其他公司不会使用这个名称空间。

如果没有改变 XML 名称空间，使用的默认名称空间是 http://tempuri.org。从学习的角度来看，这个默认的名称空间就已足够了，但不应部署一个使用它的 Web 服务产品。

19.6　测试 Web 服务

现在就可以测试服务了。在浏览器中打开文件 Service1.asmx(可以选择 Debug | Start Without Debugging，在 Visual Studio 2010 中启动它)，列出服务中的所有方法，如图 19-6 所示。在该服务中，只有一个方法 ReverseString()。

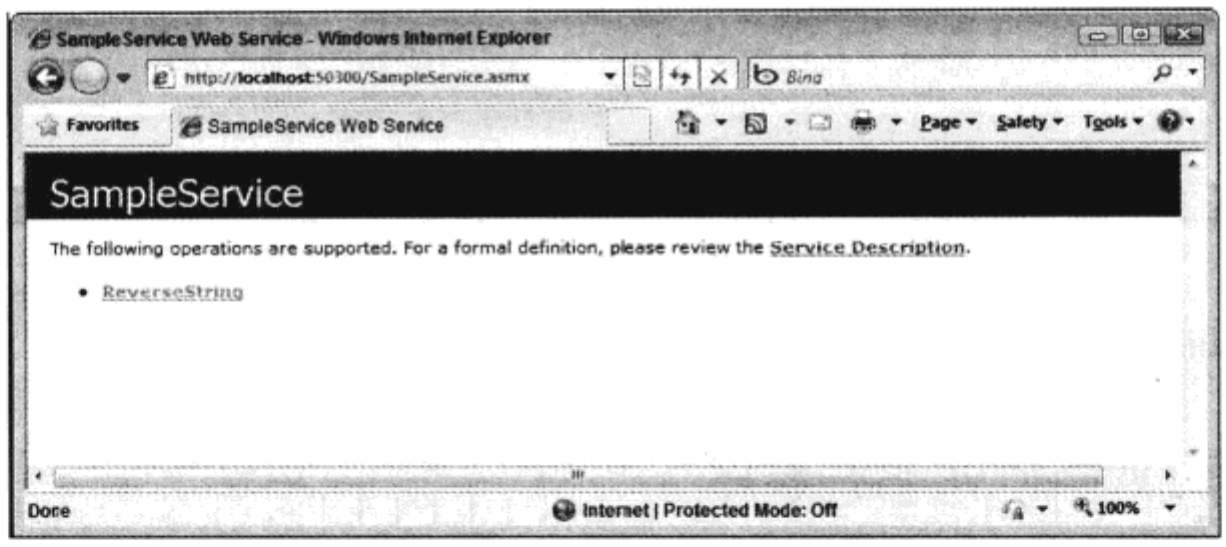


图　19-6

选择 ReverseString 方法的链接，显示一个对话框，以测试 Web 服务。该测试对话框已经编辑了传递给 ReverseString 方法的参数的字段，在本例中，该方法只有一个参数。

在这个页面中，还可以获得客户机中的 SOAP 调用信息，以及服务器返回的响应信息，如图 19-7 所示。本示例显示 SOAP 和 HTTP POST 请求。

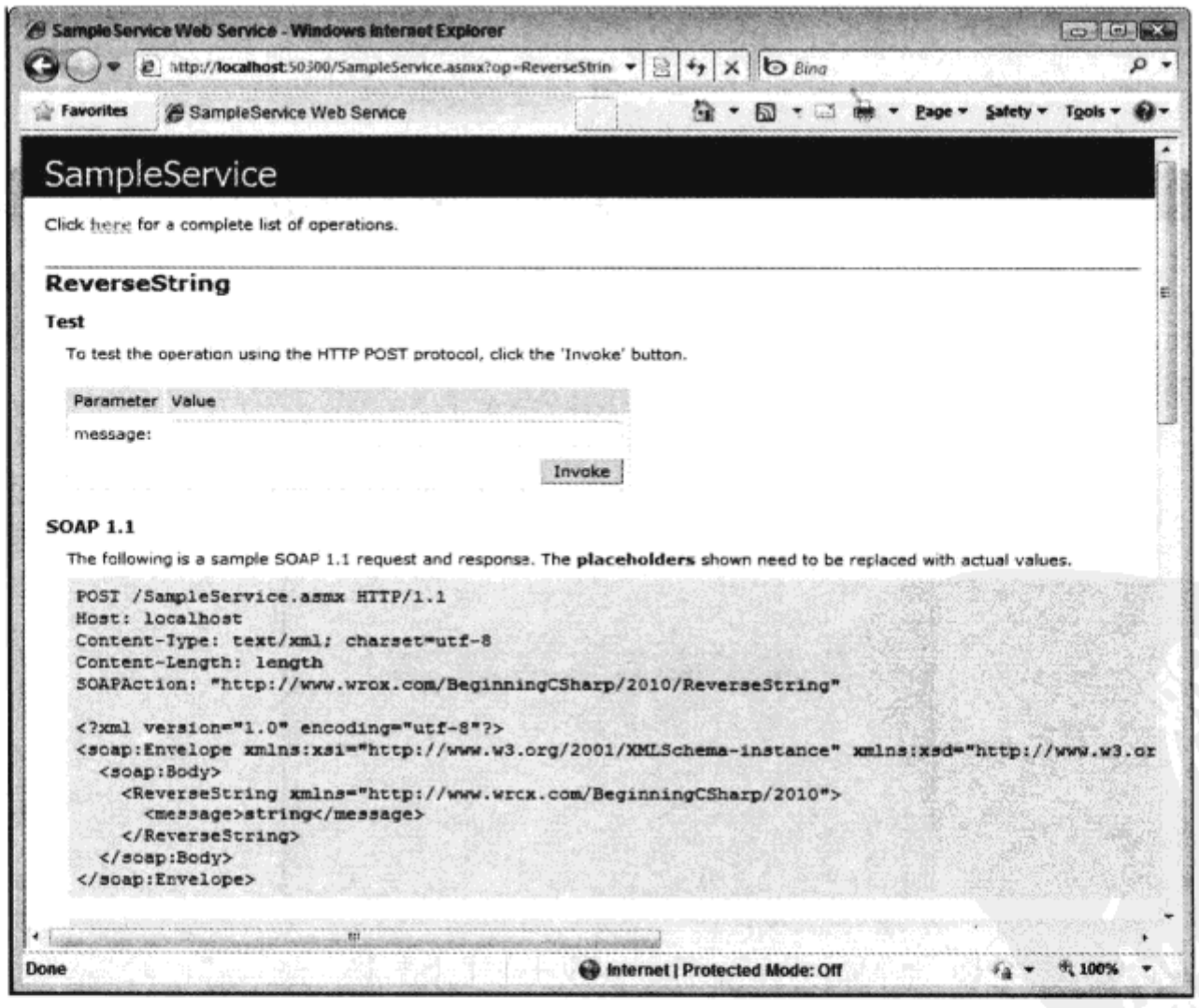


图　19-7

在把字符串 Hello Web Services!写到文本框中后，单击 Invoke 按钮，服务器就会发送如图 19-8 的结果。

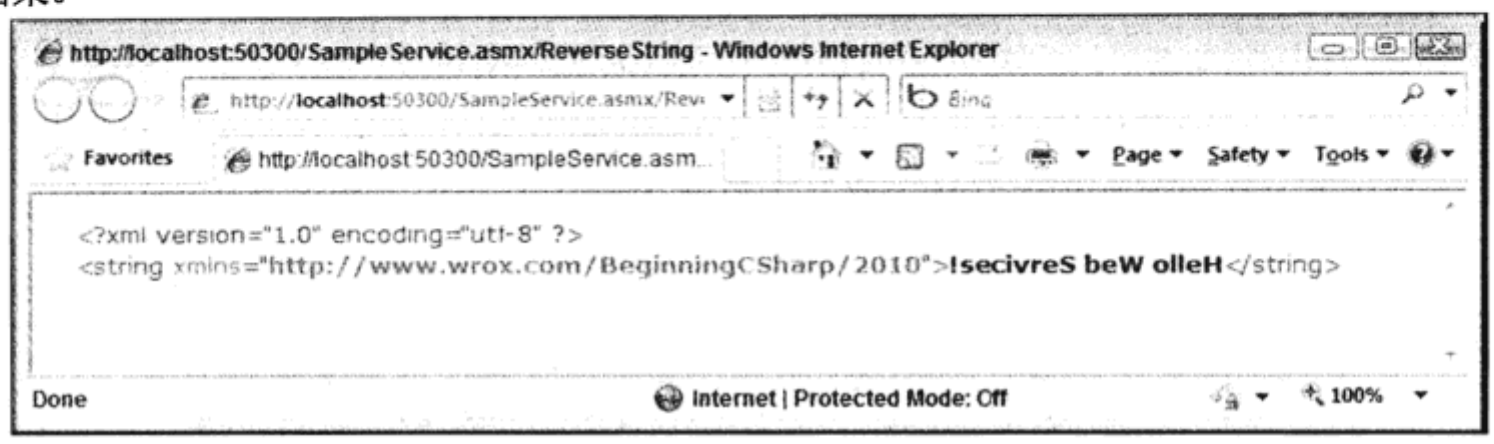


图 19-8

结果是 string 类型，与预期的一样，它的顺序与我们输入的字符串的顺序相反。

19.7 实现 Windows 客户程序

测试工作进展顺利，但我们希望创建一个使用 Web 服务的 Windows 客户程序。该客户程序必须创建一个 SOAP 消息，通过 HTTP 通道发送出去。这不一定要我们自己去完成。VS2010 创建了一个代理类，它使用 WCF 中的 HTTP 信道，在后台完成了所有的工作。

有关 WCF 的具体内容请参见第 26 章。

试一试：创建一个 Windows 客户程序

(1) 在已有的解决方案 WebServiceSample 中添加一个新的 C# Windows 窗体应用程序 WindowsFormsClient。在窗体上添加两个文本框和一个按钮，如图 19-9 所示。您将利用按钮的单击事件处理程序调用该 Web 服务。

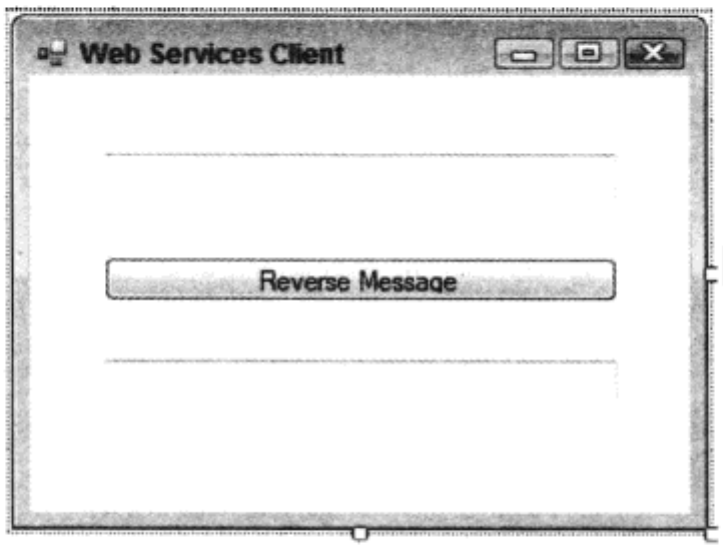


图 19-9

(2) 要添加一个服务引用，选择 Project | Add Service Reference 菜单项。在该对话框中，单

击 Discover 按钮箭头，选择 Services in Solution。前面创建的 Web 服务就会显示在左边的视图中。在左边的树形视图中选择 SampleServiceSoap。在单击 OK 按钮之前，将 Namespace 名称改成 WebServicesSample，如图 19-10 所示。

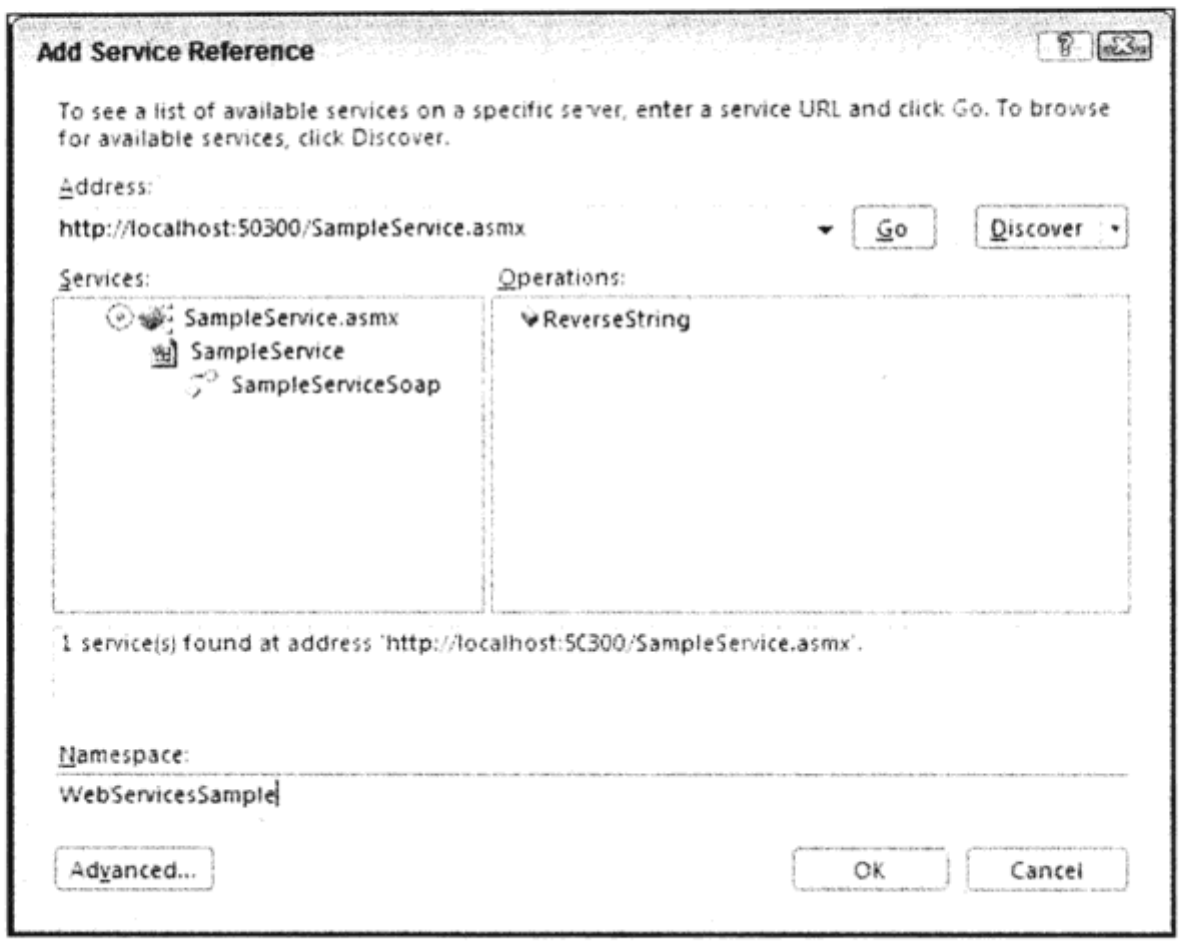


图　19-10

(3) 前面一直都没有为客户程序编写代码，而是设计了一个小用户界面，使用 Add Service Reference 菜单创建了一个代理类。现在要创建两者之间的链接。给按钮添加一个 Click 事件处理程序 button1_Click()，并添加如下两条语句：

可从 wrox.com 下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    var client = new WebServicesSample.SampleServiceSoapClient();
    textBox2.Text = client.ReverseString(textBox1.Text);
}
```

代码段 WindowsFormsClient/Form1.cs

示例的说明

在 Solution Explorer 中，可以看到一个新服务引用 WebServicesSample，如图 19-11 所示。单击 Show All Files 按钮后，可以看到 WSDL 文档和文件 Reference.cs，该文件包含代理程序的源代码。Show All Files 按钮是 Solution Explorer 工具栏中的第二个按钮。把鼠标指针移动到这些按钮上，就会显示工具提示，给出每个按钮的相关信息。

单击 Show All Files 按钮后在 Solution Explorer 中显示的内容，使用 Class View 更容易看到(实现客户代理程序的新类)。该类把方法调用转换为 SOAP 格式。在 Class View(如图 19-12 所示)中，有一个使用 Web Reference 名称定义的新名称空间。本例创建了 WebServicesSample。类 SampleServiceSoapClient 派生自 ClientBase<SampleServiceSoap>，实现 Web 服务的 ReverseString()

方法。

图 19-11

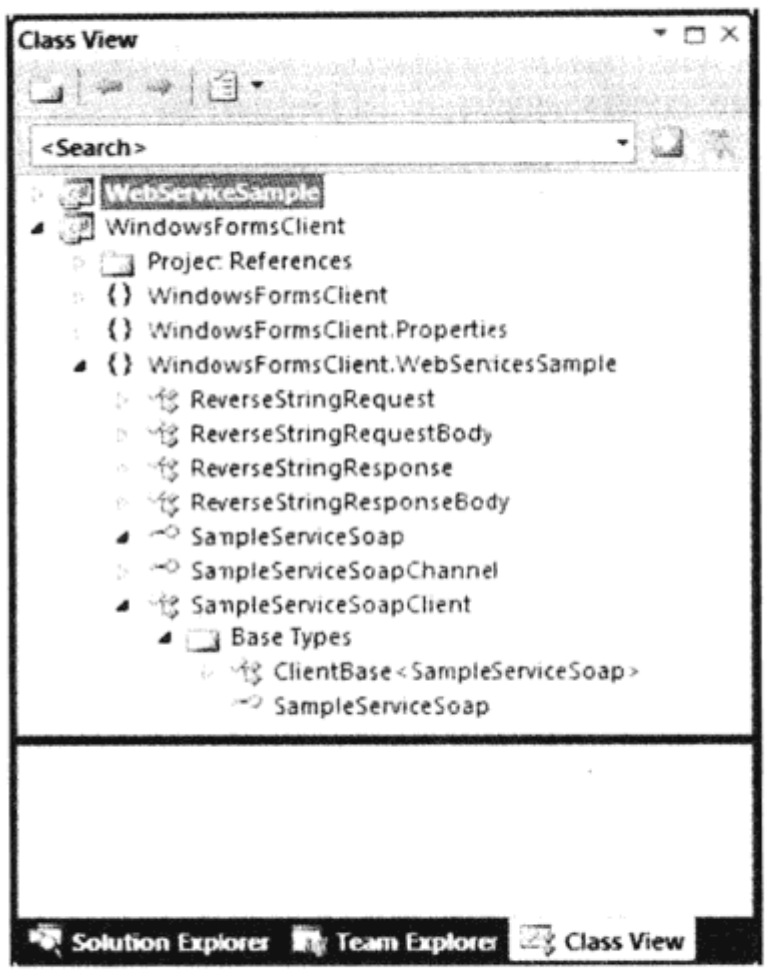


图 19-12

双击 SampleServiceSoapClient 类，打开自动生成的 Reference.cs 文件，下面看看生成的代码。

类 SampleServiceSoapClient 派生自 ClientBase<SampleServiceSoap>。这个基类在 Invoke()方法中创建了一个 SOAP 消息。SampleServiceSoap 是一个接口，定义了 Web 服务的所有操作。

可从 wrox.com 下载源代码

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
    "4.0.0.0")]
public partial class SampleServiceSoapClient : ClientBase<SampleServiceSoap>,
    SampleServiceSoap {
```

代码段 WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs

最重要的方法是 Web 服务提供的：ReverseString()。该方法与服务器上实现的方法有相同的参数。ReverseString()客户端版本的实现代码调用基类 SoapHttpClientProtocol 的 Invoke()方法。Invoke()使用方法名 ReverseString 和参数 message 创建一个 SOAP 消息。这个方法在文件 reference.cs 中：

可从 wrox.com 下载源代码

```
public string ReverseString(string message) {
    ReverseStringRequest inValue = new ReverseStringRequest();
    inValue.Body = new ReverseStringRequestBody();
    inValue.Body.message = message;
    ReverseStringResponse retVal =
        ((SampleServiceSoap)(this)).ReverseString(inValue);
    return retVal.Body.ReverseStringResult;
}
```

代码段 WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs

通过 HTTP 发送的 SOAP 请求是在自动创建的应用程序配置文件中定义的，它定义了 basicHttpBinding：

可从 wrox.com 下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="SampleServiceSoap" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
          maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
          useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
              realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:50300/SampleService.asmx"
        binding="basicHttpBinding" bindingConfiguration="SampleServiceSoap"
        contract="WebServicesSample.SampleServiceSoap" name="SampleServiceSoap" />
    </client>
  </system.serviceModel>
</configuration>
```

代码段 WindowsFormsClient/app.config

对服务的调用是在下面的语句中使用生成的代理类实现的。利用下面这行代码，创建代理类的一个新实例。在构造函数的实现代码中，Url 属性设置为 Web 服务：

可从 wrox.com 下载源代码

```
var client = new WebServicesSample.SampleServiceSoapClient();
```

代码段 WindowsFormsClient/Form1.cs

调用代理类的 ReverseString()方法，把 SOAP 消息发送给服务器，以调用 Web 服务：

可从 wrox.com 下载源代码

```
xtBox2.Text = client.ReverseString(textBox1.Text);
```

代码段 WindowsFormsClient/Form1.cs

运行程序，会得到如图 19-13 所示的输出结果。

图 19-13

19.8 异步调用服务

在通过网络发送消息时，总是要注意网络的延迟时间。如果 Web 服务是同步调用的，客户应用程序就会在调用返回后再继续执行。这在本地网络上速度比较快，但必须注意生产系统的网络体系架构。

可以给 Web 服务异步发送消息。客户代理程序不仅会创建同步方法，还可以创建异步方法。但是，Windows 应用程序存在一个特殊的问题，即每个 Windows 控件都会被绑定到一个线程上，而 Windows 控件的方法和属性只能在创建它的线程中调用。.NET 4 的代理类对此问题有特定的解决方法，如下面生成的代理代码所示。

试一试：异步调用服务

要使用代理类的异步实现代码，请执行以下步骤：

(1) 修改已生成的代理类：选择服务引用 WebServicesSample。打开关联菜单，选择 Configure Service Reference，打开的对话框如图 19-14 所示。

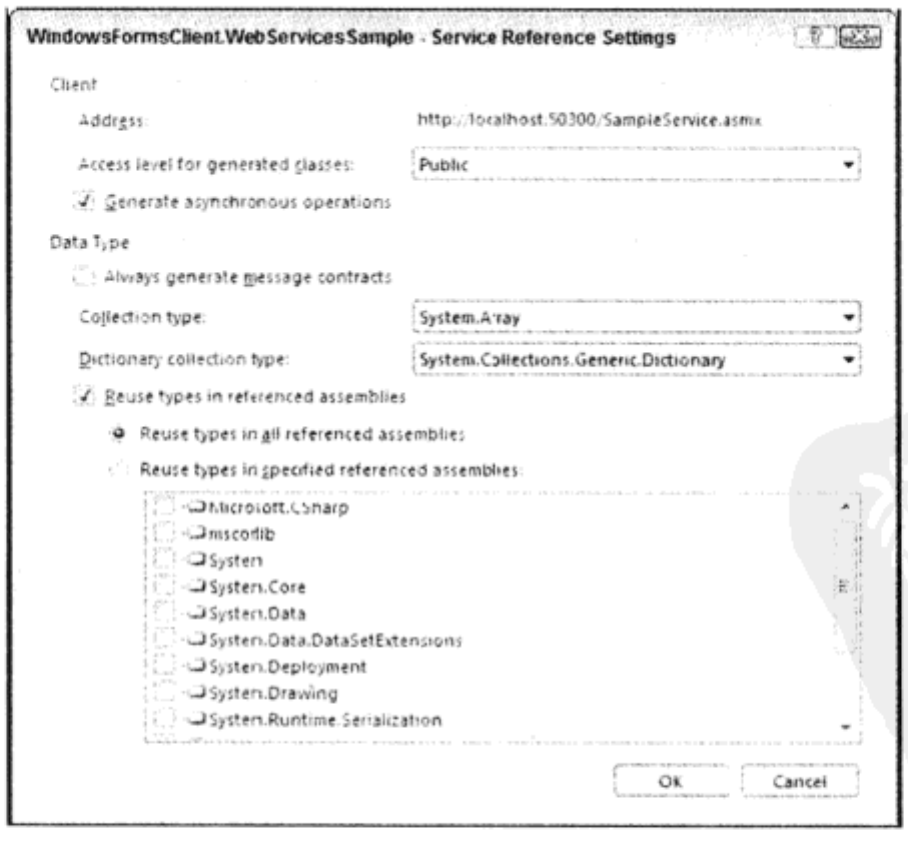


图 19-14

(2) 在 Service Reference Settings 对话框中选中 Generate asynchronous operations 复选框。

(3) 要异步调用 Web 服务，应修改 button1_Click 方法的实现代码(如下所示)。在初始化代理后，给事件 ReverseStringCompleted 添加事件处理程序 client_ReverseStringCompleted。接着调用代理 ReverseStringAsync 的 asnyc 方法，传送 textBox1 的 Text 属性。在 asnyc 方法中，会创建一个线程，以调用 Web 服务。

```
        private void button1_Click(object sender, EventArgs e)
        {
            var client = new WebServicesSample.SampleServiceSoapClient();
            client.ReverseStringCompleted += client_ReverseStringCompleted;
            client.ReverseStringAsync(textBox1.Text);
        }
}
```

代码段 WindowsFormsClient/Form1.cs

(4) 现在实现处理方法 client_ReverseStringCompleted：

```
void client_ReverseStringCompleted(object sender,
                                   ReverseStringCompletedEventArgs e)
{
    if (e.Error != null)
    {
        textBox2.Text = e.Result;
    }
    else
    {
        MessageBox.Show(e.Error.Message);
    }
}
```

代码段 WindowsFormsClient/Form1.cs

会在 Web 服务调用完成后再调用这个方法。在实现代码中，ReverseStringCompletedEventArgs 参数的 Result 属性被传送给 textBox2 的 Text 属性。

(5) 现在就可以再次运行客户程序，测试异步调用了。还可以在 Web 服务的实现中添加一段休眠时间，这样就可以看到，客户应用程序的 UI 在调用 Web 服务的过程中没有停止。

示例的说明

下面的代码是 ReverseString()方法的异步版本。在代理类的异步实现代码中，总是有一个方法可以异步调用，也总是有一个事件可以确定在完成 Web 服务方法时调用哪个方法。

ReverseStringAsync()方法只有发送给服务器的参数。从客户端接收的数据可以通过把事件处理程序赋予 ReverseStringCompleted 事件来读取。这个事件的类型是 EventHandler <ReverseString-CompletedEventArgs>。该类型是一个委托，其中第二个参数 ReverseStringCompletedEventArgs 是从 ReverseString()方法的输出参数中创建的。类 ReverseStringCompleted EventArgs 在 Result 属性中包含 Web 服务的返回数据。这段执行代码能正常工作是因为使用了 SendOrPostCallback 委托，这个委托把调用转发给 Windows Forms 控件的正确线程。

可从
wrox.com
下载源代码

```
public event System.EventHandler<ReverseStringCompletedEventArgs>
    ReverseStringCompleted;

public void ReverseStringAsync(string message) {
    this.ReverseStringAsync(message, null);
}

public void ReverseStringAsync(string message, object userState) {
    if ((this.onBeginReverseStringDelegate == null)) {
        this.onBeginReverseStringDelegate =
            new BeginOperationDelegate(this.OnBeginReverseString);
    }
    if ((this.onEndReverseStringDelegate == null)) {
        this.onEndReverseStringDelegate =
            new EndOperationDelegate(this.OnEndReverseString);
    }
    if ((this.onReverseStringCompletedDelegate == null)) {
        this.onReverseStringCompletedDelegate =
            new SendOrPostCallback(this.OnReverseStringCompleted);
    }
    base.InvokeAsync(this.onBeginReverseStringDelegate, new object[] {
                message}, this.onEndReverseStringDelegate,
                this.onReverseStringCompletedDelegate, userState);
}

private void OnReverseStringCompleted(object state) {
    if ((this.ReverseStringCompleted != null)) {
        InvokeAsyncCompletedEventArgs e =
            ((InvokeAsyncCompletedEventArgs)(state));
        this.ReverseStringCompleted(this,
            new ReverseStringCompletedEventArgs(e.Results, e.Error,
                e.Cancelled, e.UserState));
        }
    }
}
public partial class ReverseStringCompletedEventArgs :
    AsyncCompletedEventArgs {

    private object[] results;

    public ReverseStringCompletedEventArgs(object[] results,
        Exception exception, bool cancelled, object userState) :
            base(exception, cancelled, userState) {
        this.results = results;
    }

    public string Result {
        get {
            base.RaiseExceptionIfNecessary();
            return ((string)(this.results[0]));
        }
    }
}
```

代码段WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs

19.9　实现 ASP.NET 客户程序

现在可以在 ASP.NET 客户应用程序中使用相同的服务。引用 Web 服务的方式与处理 Windows Forms 应用程序的方式相同。

试一试：创建 ASP.NET 客户应用程序

(1) 打开前面创建的 Web 服务 WebServicesSample。

(2) 在解决方案中添加一个新 C# ASP.NET Empty Web Application，命名为 ASPNETClient。

(3) 添加一个新的 Web 窗体 Default.aspx，在 Web 窗体中添加两个文本框和一个按钮，如图 19-15 所示。

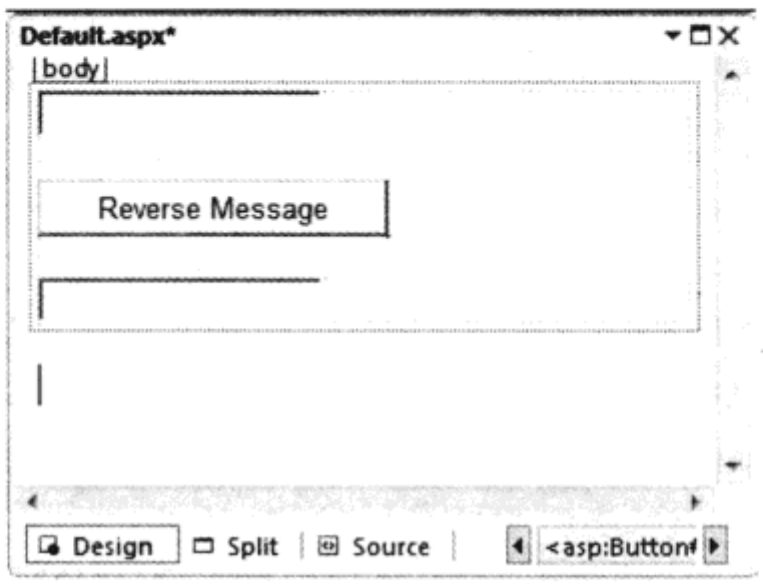


图　19-15

(4) 采用前面处理 Windows 应用程序的方式，添加一个对 http://localhost:50300/sampleservice.asmx 的服务引用。根据自己的配置，可能需要另一个端口号。

(5) 添加服务引用后，再次生成一个客户代理类。给按钮添加一个单击处理程序，在该处理程序中输入如下代码：

可从 wrox.com 下载源代码

```
protected void Button1_Click(object sender, EventArgs e)
{
    var client = new WebServicesSample.SampleServiceSoapClient();
    TextBox2.Text = client.ReverseString(TextBox1.Text);
}
```

代码段ASPNETClient/Default.aspx.cs

(6) 生成该项目。选择 Debug | Start Without Debugging，可以启动浏览器，在第一个文本框中输入一个测试消息。单击按钮时，会调用 Web 服务，在第二个文本框中得到逆序的消息，如图 19-16 所示。在多项目的解决方案中，必须把启动项目设置为想要启动的项目。

图 19-16

示例的说明

Windows 应用程序中代理类的功能与 Web 应用程序的一样。添加服务引用会创建基于 WSDL 文档的代理类。代理类向服务发送 SOAP 请求。

19.10 传送数据

在前面开发的简单 Web 服务中，只把一个简单的字符串传送给 Web 服务。下面要添加一个方法，从 Web 服务中请求天气信息。这个信息需要在 Web 服务中传送和接收较复杂的数据。

试一试：用 Web 服务传送数据

(1) 使用 Visual Studio 打开前面创建的 Web 服务项目 WebServicesSample。在这个 Web 服务中，用下面的代码定义类型。GetWeatherRequest 类和 GetWeatherResponse 类(参见下面的代码)定义了要在 Web 服务中传送和接收的文档。在这两个类中将使用枚举 TemperatureType 和 TemperatureCondition。

ASP.NET Web 服务使用 XML 序列化，把对象转换为 XML 表示。可以使用名称空间 System.Xml.Serialization 中的特性来决定生成的 XML 格式。

```
public enum TemperatureType
{
      Fahrenheit,
   Celsius
}
public class GetWeatherRequest
{
   public string City { get; set; };
   public TemperatureType TemperatureType { get; set; };
}
```

代码段 WebServiceSample/GetWeatherRequest.cs

```
public enum TemperatureCondition
{
```

```
        Rainy,
        Sunny,
        Cloudy,
        Thunderstorm
    }

    public class GetWeatherResponse
    {
        public TemperatureCondition Condition { get; set; };
        public int Temperature { get; set; };
    }
```

代码段 WebServiceSample/GetWeatherResponse.cs

(2) 添加 Web 服务方法 GetWeather():

```
[WebMethod]
public GetWeatherResponse GetWeather(GetWeatherRequest req)
{
    var resp = new GetWeatherResponse();
    var r = new Random();

    int celsius = r.Next(-20, 50);

    if (req.TemperatureType == TemperatureType.Celsius)
        resp.Temperature = celsius;
    else
        resp.Temperature = (212 - 32) / 100 * celsius + 32;

    if (req.City == "Redmond")
        resp.Condition = TemperatureCondition.Rainy;
    else
        resp.Condition = (TemperatureCondition)r.Next(0, 3);
    return resp;
}
```

代码段 WebServiceSample/SampleService.asmx.cs

这个方法接收用 GetWeatherRequest 定义的数据，返回用 GetWeatherResponse 定义的数据。在实现代码中，返回一个随机的天气条件——Microsoft 总部所在地 Redmond Washington 是例外，那里整个星期都在下雨。要生成随机的天气，可使用 System 名称空间中的 Random 类。

(3) 在生成 Web 服务之后，使用 Windows Forms Application 模板创建一个新项目，把应用程序命名为 WeatherClient。

(4) 修改主对话框，如图 19-17 所示。Temperature Type 控件中有两个单选按钮，用于选择温度类型(Celsius 或 Fahrenheit)，城市名可以输入。单击 Get Weather 按钮，会调用 Web 服务，Web 服务的结果显示在 Weather Condition 和 Temperature 文本框控件中。

表 19-5 中列出了控件、控件名称以及 Text 属性值。

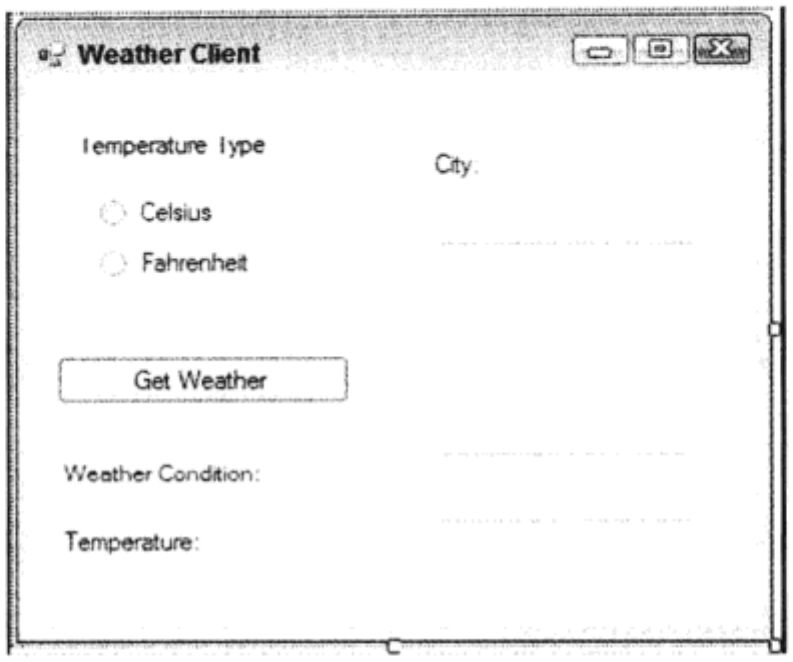


图 19-17

表 19-5

控 件	名 称	Text 属 性
GroupBox	groupBox1	Temperature Type
RadioButton	radioButtonCelsius	Celsius
RadioButton	radioButtonFahrenheit	Fahrenheit
Label	labelCity	City
TextBox	textCity	
Button	buttonGetWeather	Get Weather
Label	labelWeatherCondition	Weather Condition
Label	labelTemperature	Temperature
TextBox	textWeatherCondition	
TextBox	textTemperature	

(5) 添加对 Web 服务的引用，这类似于前面的客户应用程序项目。把引用命名为 Weather Service。

(6) 为客户应用程序导入 WeatherClient.WeatherService 名称空间。

(7) 使用 buttonGetWeather 按钮的 Property 对话框，给按钮添加 Click 事件处理程序 OnGetWeather。

(8) 给 OnGetWeather()方法添加实现代码，如下所示：

可从 wrox.com 下载源代码

```
private void OnGetWeather(object sender, EventArgs e)
{
    var req = new GetWeatherRequest();
    if (radioButtonCelsius.Checked)
       req.TemperatureType = TemperatureType.Celsius;
    else
       req.TemperatureType = TemperatureType.Fahrenheit;
```

```
        req.City = textCity.Text;

        var client = new SampleServiceSoapClient();
        GetWeatherResponse resp = client.GetWeather(req);
        textWeatherCondition.Text = resp.Condition.ToString();
        textTemperature.Text = resp.Temperature.ToString();
    }
```

代码段 WeatherClient/Form1.cs

首先创建一个 GetWeatherRequest 对象，它定义了发送给 Web 服务的请求。调用 GetWeather() 方法，就可以调用 Web 服务。这个方法返回一个 GetWeatherResponse 对象，其值显示在用户界面上。

(9) 启动客户应用程序。输入一个城市，单击 Get Weather 按钮。如果幸运，会显示真实天气，如图 19-18 所示。

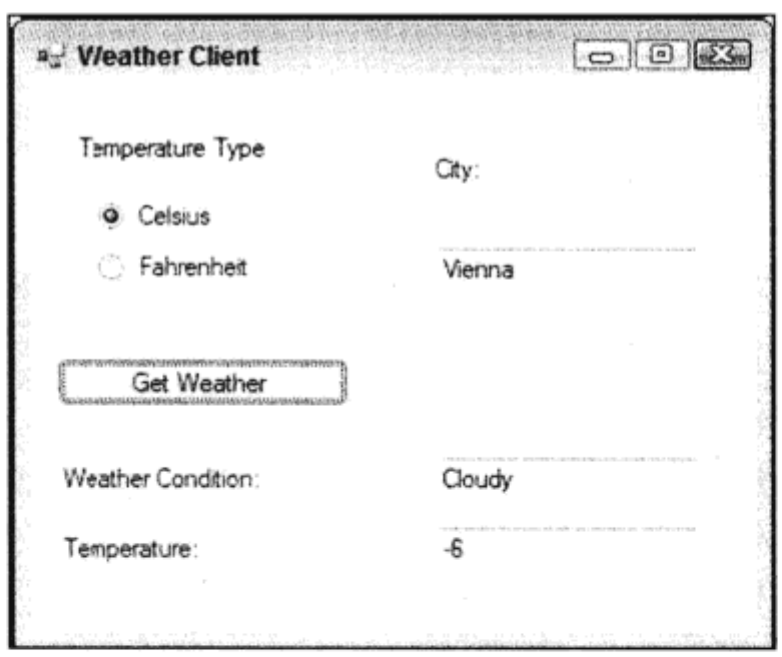


图　19-18

示例的说明

在 ASP.NET Web 服务中发送和接收数据需要使用 XML 序列化。通过 XML 序列化，会序列化所有公共属性和公共字段。类 GetWeatherRequest 和 GetWeatherResponse 使用了公共属性。为了进行 XML 序列化，类必须是公共的，有一个公共的默认构造函数。如果没有给类添加构造函数(与 GetWeatherRequest 和 GetWeatherResponse 一样)，编译器会默认创建一个公共的默认构造函数，来初始化类的所有成员字段。值类型初始化为 0，引用类型初始化为 null。

在 System.Xml.Serialization 名称空间中定义的特性类可以用于定制序列化的 XML 结果。XmlIgnoreAttribute 特性类用于忽略类成员。可以指定 XmlElementAttribute 特性类，来重命名序列化的 XML 元素，使用 XmlAttributeAttribute 类可以序列化 XML 特性，而不是元素。

使用带服务选项的 XML 可序列化类型，把 XML 模式信息添加到 WSDL 文档中，此类信息用于在添加服务引用时创建客户应用程序的类型。

19.11　小结

本章介绍了 Web 服务，并概述了 Web 服务使用的协议。要查找和运行 Web 服务，必须使用下

述一个或所有的方式：

- **描述** —— WSDL 描述了方法和参数
- **调用** —— 利用 SOAP 协议调用独立于平台的方法

利用 Visual Studio 创建 Web 服务是非常简单的，其中 WebService 类使用 WebMethod 特性定义了一些方法。创建使用 Web 服务的客户程序与创建 Web 服务一样简单，只需给客户项目添加一个 Web 引用，同时使用代理程序即可。客户程序的核心是 SoapHttpClientProtocol 类，它可以把方法调用转换为一个 SOAP 消息。所创建的客户代理提供了异步和同步方法。在使用异步方法时，客户接口不会在 Web 服务方法执行完毕之前停止运行。我们还学习了如何创建定制类，以便定义要传输的复杂数据。第 22 章将介绍如何部署 Web 应用程序和 Web 服务。

19.12 练习

下面的练习将使用本章的知识，创建一个新的 Web 服务，为电影院提供座位预定系统。

(1) 创建一个新的 Web 服务 CinemaReservation。

(2) 需要 ReserveSeatRequest 和 ReserveSeatResponse 类来定义在 Web 服务中发送和接收的数据。ReserveSeatRequest 类需要一个 string 类型的成员 Name 来发送名称，还需要两个 int 类型的成员，对使用 Row 和 Seat 定义的座位发送请求。ReserveSeatResponse 类定义了发回给客户端的数据，即预定的名称和实际预定的座位。

(3) 创建一个 Web 方法 ReserveSeat，它需要把 ReserveSeatRequest 作为参数，返回一个 ReserveSeatResponse。在 Web 服务的实现代码中，可以使用 Cache 对象(参见第 18 章)存储已预定的座位。如果请求的座位还未预定，就返回该座位，把它保存在 Cache 对象中。如果请求的座位已被预定，就使用下一个未预定的座位。对于 Cache 对象中的座位，可以使用一个二维数组，参见第 5 章。

(4) 创建一个 Windows 客户应用程序，它使用 Web 服务预定电影院中的座位。

附录 A 给出了练习答案。

19.13 本章要点

主 题	重 要 概 念
使用 ASP.NET 创建 Web 服务	可以在 ASP.NET Web 项目中创建 ASP.NET Web 服务。使用 WebService 和 WebMethod 特性来定义服务
调用 Web 服务	客户应用程序要调用 Web 服务的操作，可以在 Solution Explorer 中选择 Add | Service Reference，来创建一个代理。添加服务引用，就可以使用 WSDL，创建代理类
异步调用 Web 服务	使用服务引用中的高级选项，可以创建异步方法，以异步方式调用 Web 服务。带 Async 前缀的方法接受 Web 服务方法的输入参数。服务调用结束时，会引发一个事件，接收输出参数
在 Web 服务之间传送数据	要把非简单的数据类型传送给 Web 服务，可以创建一个定制类。XML 序列化用于把对象转换为在线路上传送的消息

第20章

部署 Web 应用程序

本章内容：

- 如何为 ASP.NET Web 应用程序配置 IIS
- 如何复制 Visual Studio Web 站点
- 如何发布 Web 应用程序
- 如何为 Web 应用程序创建 Windows 安装程序包

前面 2 章学习了如何用 ASP.NET 开发 Web 应用程序和 Web 服务。对于这些类型的应用程序，有不同的部署选项。可以复制 Web 页面，发布 Web 站点，或者创建安装程序。本章介绍这些不同选项的优缺点，以及如何完成这些任务。

20.1 Internet Information Services

对于用 Visual Studio 2010 开发的 Web 应用程序，不需要安装 Internet Information Services (IIS)。因为 Visual Studio 2010 有自己的 Web 服务器：Visual Web Development Server。这是一个简单的 Web 服务器，只运行在本地机器上。在产品系统上，必须有 IIS 才能运行 Web 应用程序。

IIS 不能用于 Windows 7 Home Edition。在其他版本上，可以安装 IIS，其方式与安装其他 Windows 组件相同。在控制面板上，单击 Programs。其中包含一个类别 Programs and Features 和一个链接 Turn Windows features on or off。单击这个链接。Windows 的一个功能是 Internet Information Services，需要选择它才能安装 Internet Information Services。还可以要求系统管理员在系统上安装 IIS。

只有用 IIS 配置 ASP.NET 运行库，才能运行 ASP.NET Web 应用程序。只要用 IIS 管理器工具检查处理程序映射(如图 20-1)，就可以验证 ASP.NET 运行库是否配置。如果安装了 IIS，这个工具就位于 Administrative Tools 中，其菜单项是 Internet Information Services (IIS) Manager。如果没有把 Administrative Tools 配置为可以直接从 Start 菜单中使用，就可以在控制面板上查找 Administrative Tools。

在 Internet Information Services (IIS) Manager 中，双击 Handler Mappings。在信息中滚动，可以

看到*.aspx 路径配置了多次。可以找到.NET Framework 的多个版本，还有本机配置和托管配置。.aspx 扩展名的 IsapiModule 配置定义了本机配置 System.Web.UI.PageHandlerFactory，这是处理请求的.NET 类。

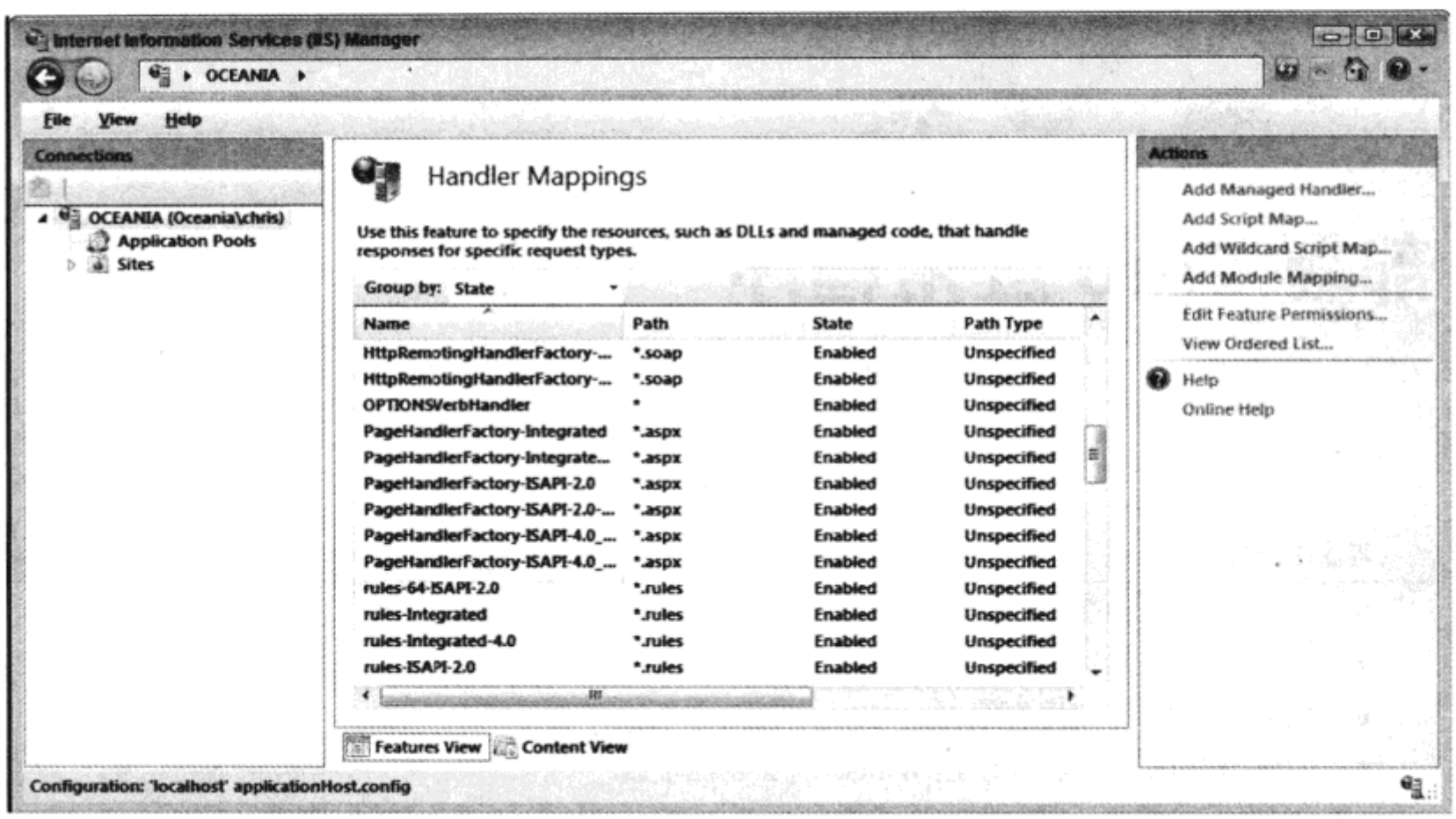


图 20-1

如果没有在系统上配置 ASP.NET 运行库的处理程序映射，可以启动程序 aspnet_regiis -i，使用 IIS 安装文件扩展名和模块。

IIS 的主进程是 inetinfo.exe，它运行在权限很高的 System 账户下。配置了 IsapiModule 后，就把对 ASPX 文件的请求发送给辅助进程(w3wp.exe)。不同的辅助进程可以配置为运行.NET 运行库的不同版本。也可以配置用户身份，以该用户身份运行这个进程，指定循环选项。

20.2 IIS 配置

必须在运行 Web 应用程序之前配置 IIS。下面的示例将用 Internet Information Services Manager 工具创建一个 Web 站点，首先，Web 站点需要一个虚拟目录，这是客户端用于访问 Web 应用程序的目录。例如 http://server/mydirectory，mydirectory 就是一个虚拟目录。虚拟目录完全独立于在磁盘上存储文件的物理目录。例如，mydirectory 的物理目录是 D:\someotherdirectory。

试一试：创建新的应用程序池

(1) 启动 Internet Information Services Manager 工具，这个工具位于控制面板的管理工具上。如图 20-2 所示。

(2) 在树型视图中选择 Application Pools，右击它，从关联菜单中选择 Add Application Pool 选项。

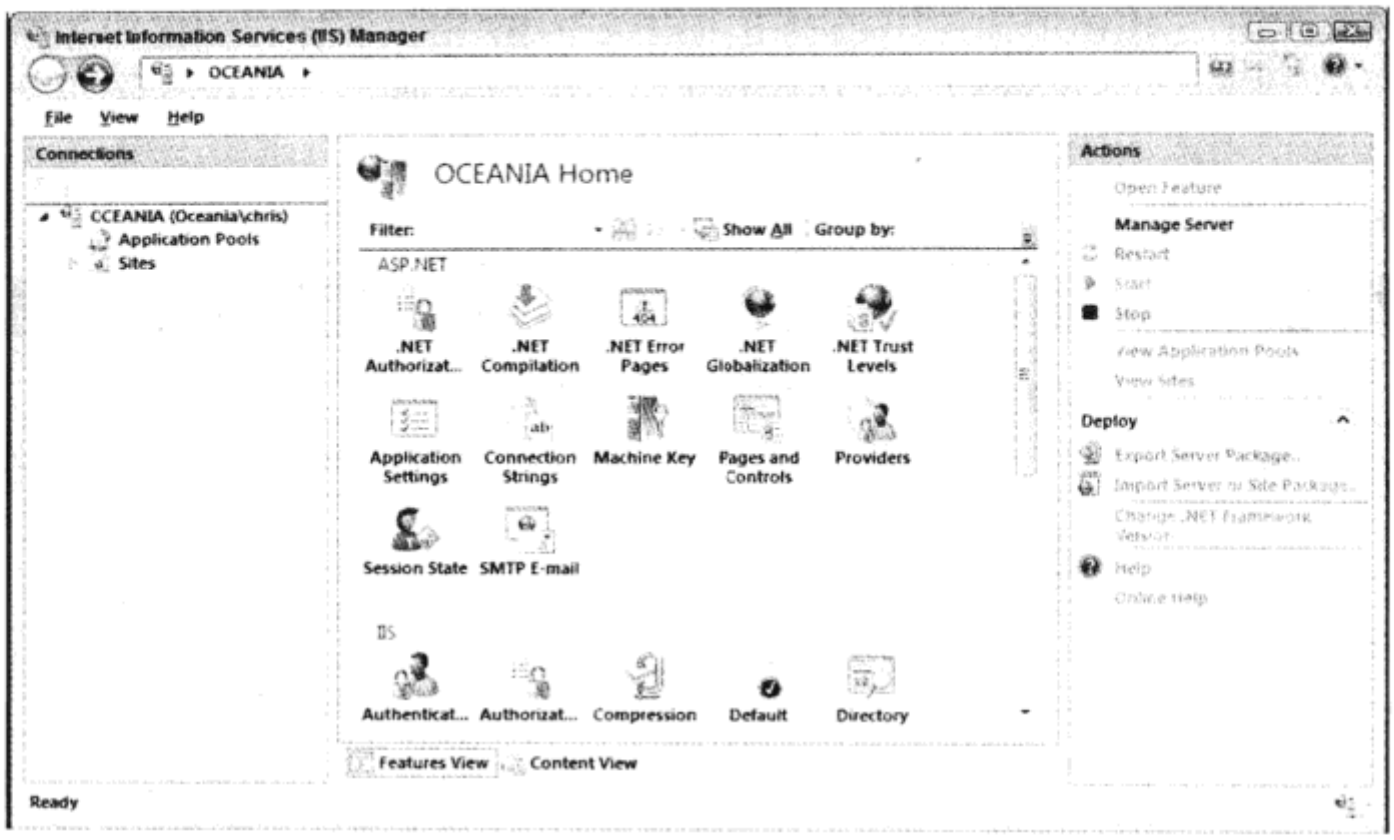


图　20-2

(3) 打开 Add Application Pool 对话框，如图 20-3 所示。在 Name 文本框中，输入 Beginning Visual C# App Pool，再选择.NET Framework version v4。单击 OK 按钮。实际上，我们配置了.NET 运行库的版本。对于 ASP.NET 2.0、3.0 和 3.5 应用程序，可以使用相同的版本号 2.0.50727。

(4) 创建应用程序池后，就可以配置高级设置了，如图 20-4 所示。定义进程在什么身份下运行，在多核或多 CPU 系统上还要指定应使用哪些 CPU，并定义在这个池中应运行的辅助进程数。从 Internet Information Services (IIS) Manager 右边的 Actions 类别中选择了应用程序池，或从关联菜单中选择了它后，就可以配置高级设置了。

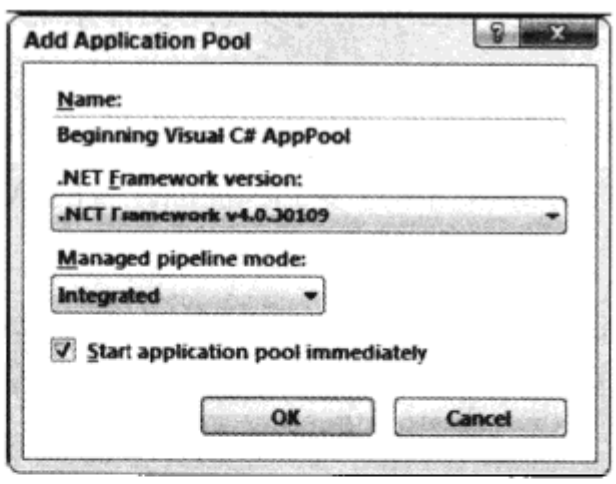


图　20-3

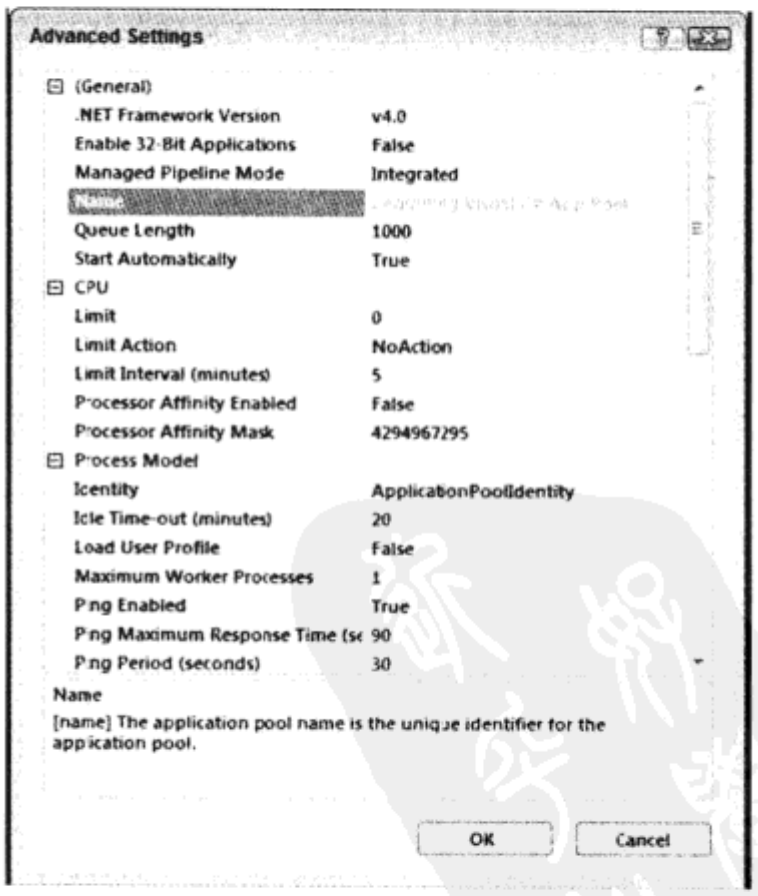


图　20-4

示例的说明

应用程序池允许不同的 Web 站点运行 ASP.NET 运行库的不同版本，可以有不同的用户账户和稳定性。

配置好应用程序池后，就可以创建新的 Web 应用程序了，如下面的示例所示。

试一试：创建新的 Web 应用程序

(1) 在 IIS Manager 的树型视图中，选择 Default Web Site。

(2) 右击后从关联菜单中选择 Add Application 选项。打开 Add Application 对话框，如图 20-5 所示。

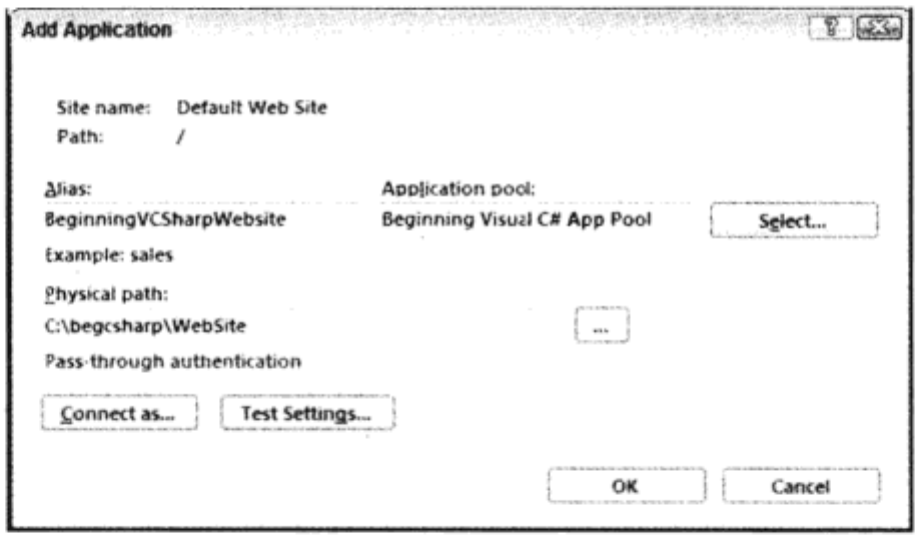


图 20-5

(3) 输入 Web 站点的物理路径和别名 BeginningVCSharpWebsite。选择刚才创建的应用程序池 Beginning Visual C# App Pool。

(4) 单击 OK 按钮。

配置好虚拟目录后，就可以在 VS 中将 Web 应用程序复制或发布到这个 Web 站点了。

20.3 复制 Web 站点

在 Visual Studio 2010 中，可以把文件从源 Web 站点复制到远程 Web 站点上。源 Web 站点就是用 Visual Studio 打开 Web 应用程序的 Web 站点，这个 Web 站点可以从本地文件系统或 IIS 上访问，具体取决于 Web 应用程序的创建方式。文件复制目的地的远程 Web 站点可以使用文件系统、FTP 协议或 IIS 上的 FrontPage Server Extensions 访问。

复制文件可以双向进行：从源 Web 站点复制到远程 Web 站点上，或者从远程 Web 站点复制到源 Web 站点上。下面的示例将使用 Visual Studio 把新建的 Web 应用程序复制到前面配置的 Web 站点上。

复制网站的 VS 菜单只能用于 Web Site，不能用于 Web Project。

试一试：复制 Web 站点

(1) 在 Windows 7 或 Windows Vista 中，用提升的管理权限启动 Visual Studio 2010。把 Web 站点复制到本地 IIS 上需要管理权限。在 Windows XP 上，如果用管理权限登录到一个账户上，就可以正常启动 Visual Studio。

(2) 选择 File | New Web Site 菜单，再选择 ASP.NET Web Site 模板，创建一个新的 Web 站点。把本地文件系统选择为这个 Web 站点的位置。这会创建包含几个页面和样式的示例站点。

(3) 选择菜单项 Website | Copy Web Site，打开如图 20-6 所示的对话框。

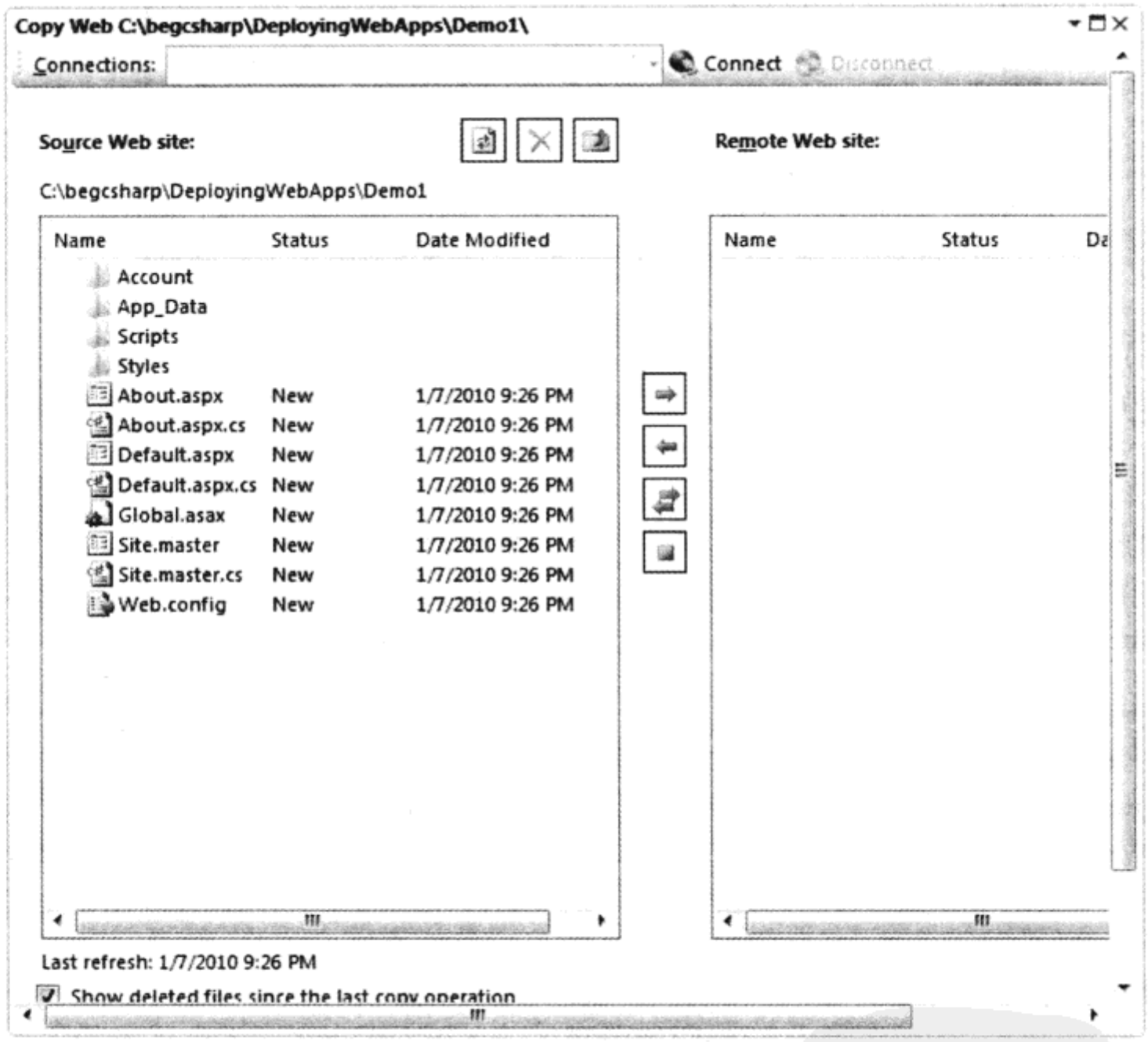


图　20-6

(4) 单击窗口顶部的 Connect 按钮，打开如图 20-7 所示的 Open Web Site 对话框。

(5) 在 Open Web Site 对话框中，可以选择要复制到本地文件系统、本地 IIS、FTP 站点和远程站点(已安装了 FrontPage Server Extensions)中的文件。选择 Local IIS，再选择前面创建的 Web 站点 BeginningVCSharpWebsite。如果运行的是 Windows Home Edition，就只能把文件复制到本地文件系统上，因为不能使用 IIS。

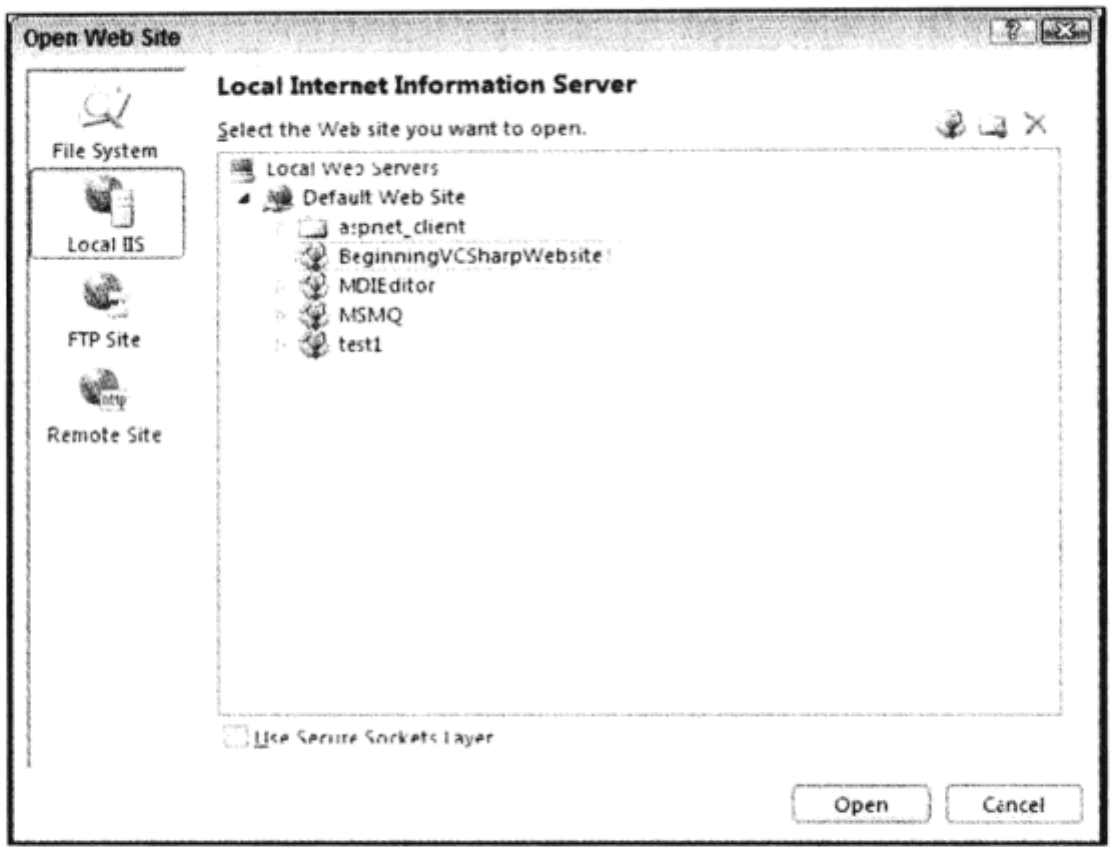


图 20-7

(6) 在 Source Web site 列表中，选择要从 Source Web Site 复制到 Remote Web Site 的文件。

(7) 单击 Copy Selected Files 按钮。这个按钮在 Source Web Site 视图和 Remote Web Site 视图的中间，有一个箭头。如果把鼠标停放在按钮上，就会显示一个工具提示，解释按钮的用途。箭头方向表示文件的复制方向，从源站点复制到目标站点，或者从目标站点复制到源站点。箭头指向两个方向的按钮验证哪些文件是新文件，把新文件复制到源站点和目标站点上。

(8) 现在所有选中的文件都已被复制到新的 Web 站点上。可以打开浏览器，输入链接 http://localhost/BeginningVCSharpWebsite，就会看到复制的 Web 站点。

示例的说明

使用 Copy Web Site 工具，还可以选择要从远程 Web 站点复制到源 Web 站点的文件。单击 Synchronize Selected Files 显示双向按钮，新文件就会从远程 Web 站点复制到源 Web 站点上，也会从源 Web 站点复制到远程 Web 站点上。如果其他开发人员在同一个公用 Web 服务器上同步文件，这就是一个很有用的选项。在两个方向上同步，会把新文件复制到公用 Web 服务器上，同时把同事的远程 Web 服务器上的文件复制到本地站点上。

复制文件时，不能确保是否能编译文件。编译应在浏览器访问文件时进行。使用命令行工具 aspnet_compiler.exe 就可以对 Web 站点进行预编译。

如果输入命令

```
aspnet_compiler -v /BegVCSharpWebsite
```

就预编译了 Web 站点 BeginningVCSharpWebsite。这样第一个用户就不必等到 ASPX 页面编译后才访问 Web 站点了。

这个工具位于.NET 运行库的目录下。

20.4　发布 Web 站点

使用 Visual Studio 2010 Web 项目，还可以发布 Web 应用程序。如果没有自包含的 IIS，且需要把 Web 应用程序发布给提供商，这就是最佳选项。

Visual Studio 2010 中的几个发布选项如下所示：

- 发布到文件系统上
- 发布到安装了 FrontPage Server Extensions 的服务器上
- 使用 FTP
- 使用 1-Click 发布选项，这是 Visual Studio 2010 的一个新特性。1-Click 选项只能用于支持该特性的主机上，但这样的主机很多，很容易就能找到

下面的示例使用 Visual Studio 的新发布特性发布 Web 应用程序。

试一试：发布 Web 应用程序

(1) 打开第 18 章创建的 Web 项目 EventRegistrationWeb。

(2) 打开 Package/Publish 项目设置，如图 20-8 所示。选择创建发布包的位置。单击 Include all Databases configured in Deploy SQL Tab 设置旁边的链接 Open Settings。

图　20-8

(3) 显示 Deploy SQL 设置，如图 20-9 所示。单击 Import from Web.config 按钮，导入数据库连接字符串，就可以部署该连接字符串引用的数据库。验证其他设置。可以定义指向目标数据库服务器的连接字符串，数据库数据和模式应写入数据库服务器中。

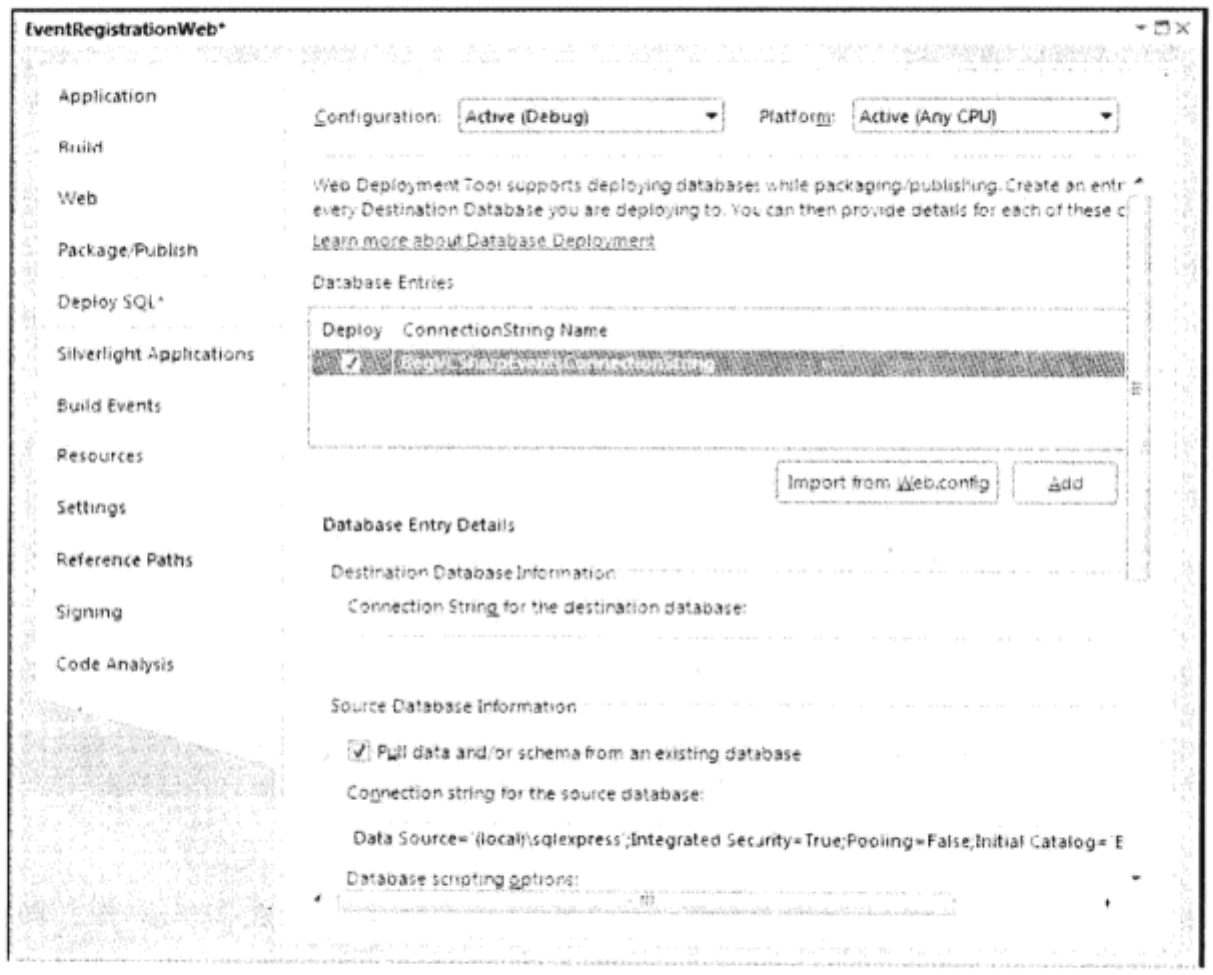


图 20-9

(4) 在 Visual Studio Build 菜单中，选择 Publish 菜单项，打开如图 20-10 所示的 Publish Web 对话框。选择发布方法 MSDeploy.Publish 的设置。这是一个 1-Click 发布选项，可用于几个主机提供商。单击链接 Click Here，在当地找到一个主机公司，如果不使用这个发布选项，可以把发布方法改为 File System。当然，如果使用支持这个发布选项的主机提供商，就可以使用它发布 Web 应用程序。

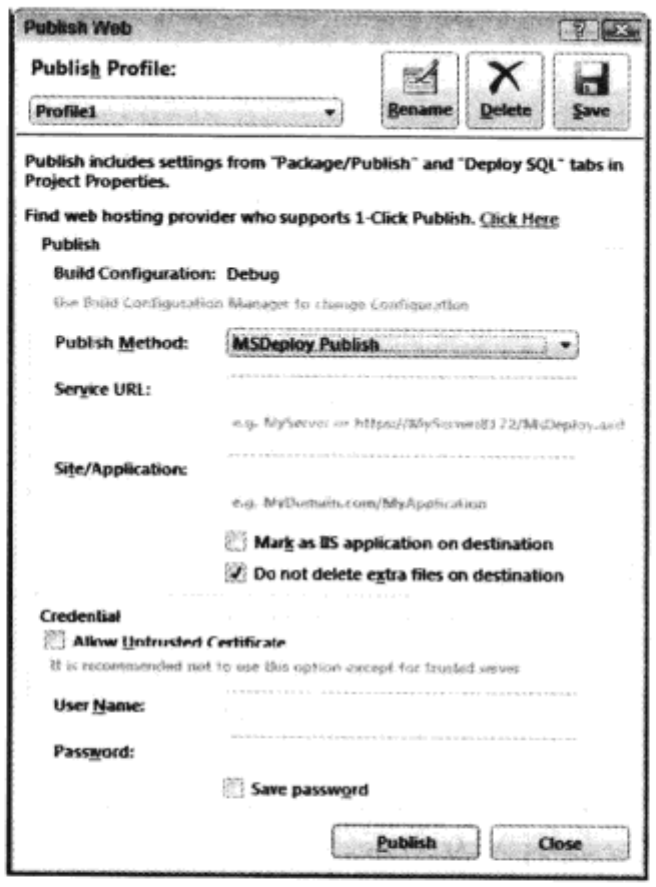


图 20-10

(5) 选择发布到文件系统上后，会打开如图 20-11 所示的对话框。在 Target Location 设置中输入一个本地目录，单击 Publish 按钮。

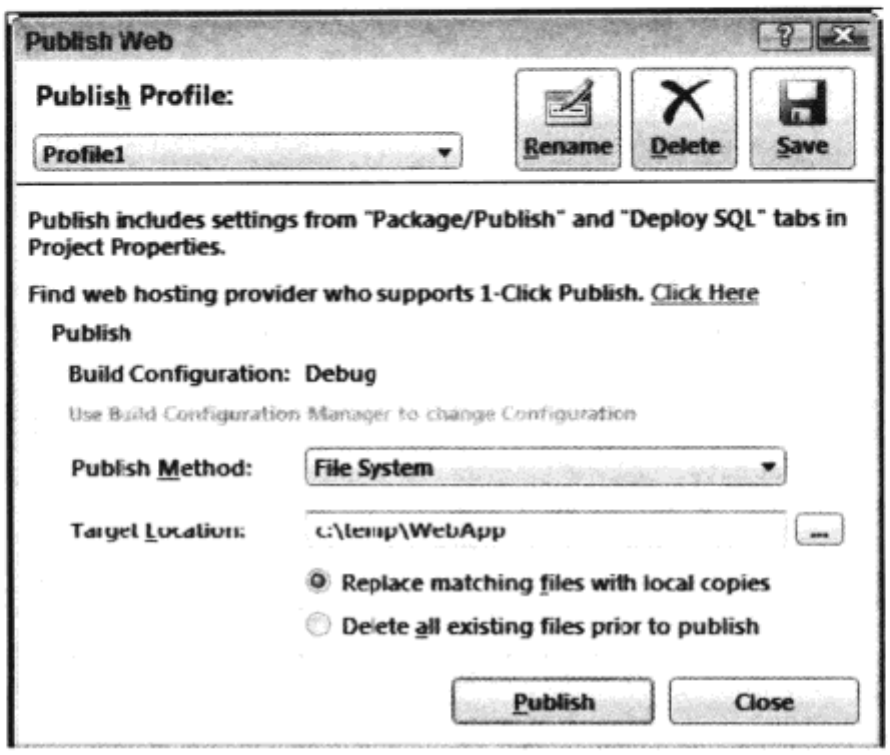


图　20-11

(6) 使用 Windows 资源管理器验证目标位置，检查已发布的文件。

20.5　Windows 安装程序

也可以创建一个 Windows 安装程序来安装 Web 应用程序。如果 Web 应用程序需要共享的程序集，就需要创建安装程序。使用安装程序的优点在于，虚拟目录用 IIS 配置，不需要手工创建虚拟目录。安装 Web 应用程序的用户可以启动 setup.exe 程序，安装过程会自动进行。当然，启动这个程序需要管理权限。

20.5.1　创建安装程序

Visual Studio 2010 提供了项目类型 Web Setup Project，用于为 Web 应用程序创建安装程序。通过 Web Setup Project，可以使用下列编辑器：File System、Registry、File Types、User Interface、Custom Action 和 Launch Conditions。第 17 章在讨论 Windows 应用程序时已讨论过它们。这里只介绍与 Web 应用程序相关的编辑器。

下面的示例将创建一个安装程序，可供安装 Web 应用程序。

试一试：创建安装程序

(1) 用 Visual Studio 2010 打开第 18 章的 Web 应用程序 EventRegistrationWeb。

(2) 在同一个解决方案中添加一个 Web Setup Project 类型的新项目，如图 20-12 所示，把该项目命名为 EventRegistrationWebSetup，单击 OK 按钮。

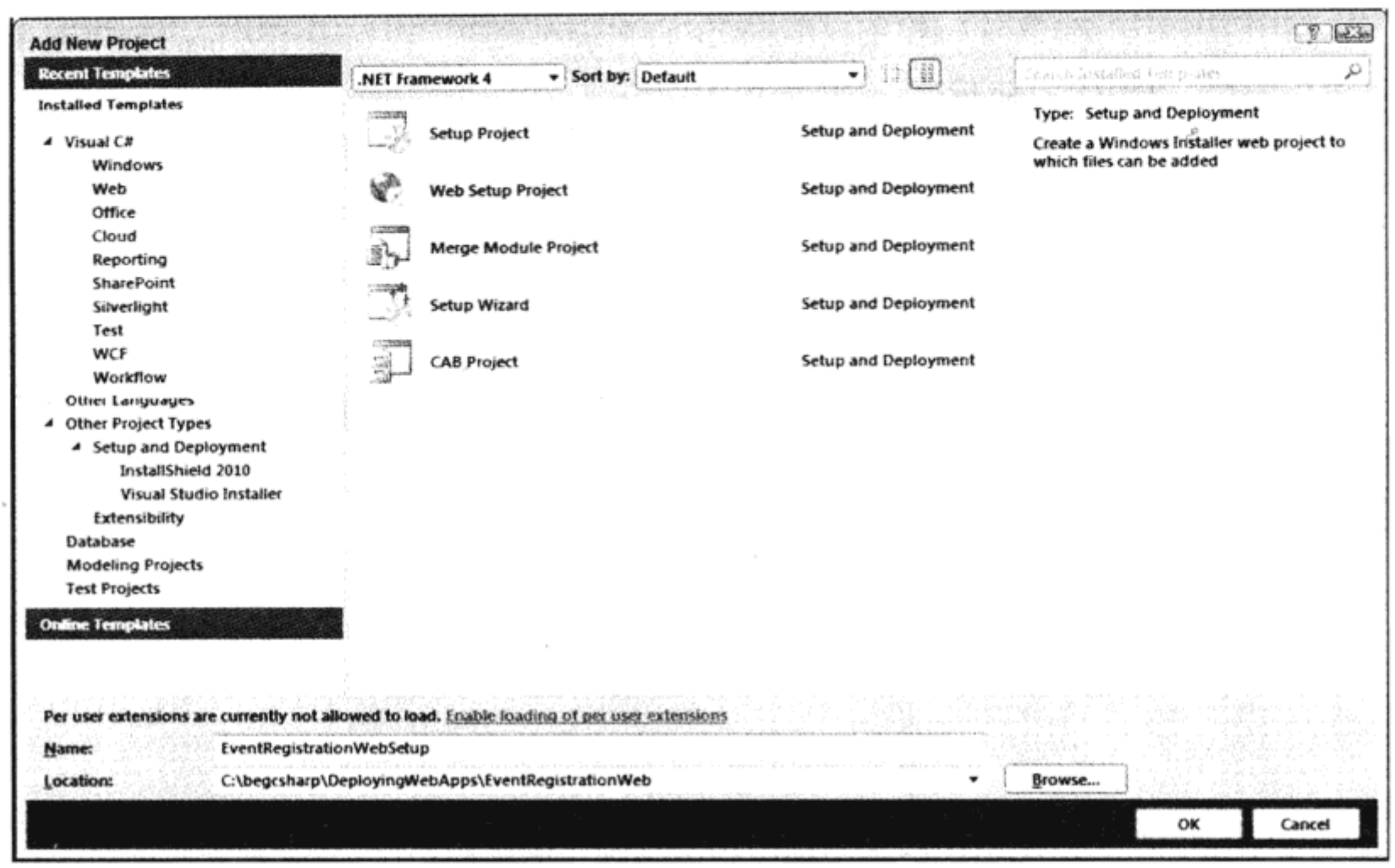


图 20-12

(3) 如果创建项目后没有打开 File System 编辑器，就打开它。选择 File System on Target Machine，再选择菜单项 Project | Add | Project Output，在 Project Output 对话框中选择 Content Files of the Web application，单击 OK 按钮。

(4) 在 File System 编辑器中，选择 Web Application Folder。现在可以用属性编辑器配置 Web 应用程序了。表 20-1 中列出了属性及其说明。

表 20-1

Web Application Folder 属性	说 明
AllowDirectoryBrowsing	一个 IIS 配置选项。把这个选项设置为 true，就可以浏览 Web 站点上的文件，其默认值是 false
AllowReadAccess	这个选项默认设置为 true。要访问 ASPX 页面，需要拥有读取访问权限
AllowScriptSourceAccess	默认情况下，把这个属性设置为 false，拒绝对脚本源文件的访问
AllowWriteAccess	默认为拒绝写入访问
DefaultDocument	设置 Web 站点的主页，如 Default.aspx
ExecutePermissions	默认设置为 vsdepScriptsOnly，它允许访问 ASP.NET 页面，但不允许在服务器上运行定制的可执行文件。如果允许在服务器上运行定制的可执行文件，这个选项就可以设置为 vsdepScriptsAndExecutables
LogVisits	设置为 true，就配置客户访问日志
VirtualDirectory	设置用 IIS 配置的虚拟目录名

(5) 用菜单项 View | Editor | Launch Conditions 打开 Launch Conditions 编辑器。安装条件定义了在安装进行之前必须安装在目标系统上的产品。

(6) 检查已配置的安装条件。Search for IIS 配置检查注册表项 SYSTEM\CurrentControlSet\Services\W3SVC\Parameters，获得 IIS 的版本，就可以验证 IIS 是否安装在目标系统上。使用默认添加的如下条件：

```
(IISMAJORVERSION >= "#5" AND IISMINORVERSION >= "#1") OR IISMAJORVERSION >= "#6"
```

安装条件 IIS Condition 确认 IIS 版本至少是 5.1。

(7) 选择 Build | Build EventRegistrationWebSetup 菜单项，生成安装应用程序。

(8) 在安装项目的目录下，有一个 setup.exe 文件和一个安装软件包 EventRegistrationWebSetup.msi。

20.5.2　安装 Web 应用程序

启动 setup.exe 程序，就可以安装 Web 应用程序。

试一试：安装 Web 应用程序

(1) 单击 setup.exe，开始安装 Web 应用程序。在 User Account Control 对话框中，单击 Yes，允许修改。打开的 Setup Wizard 对话框如图 20-13 所示。单击 Next 按钮。

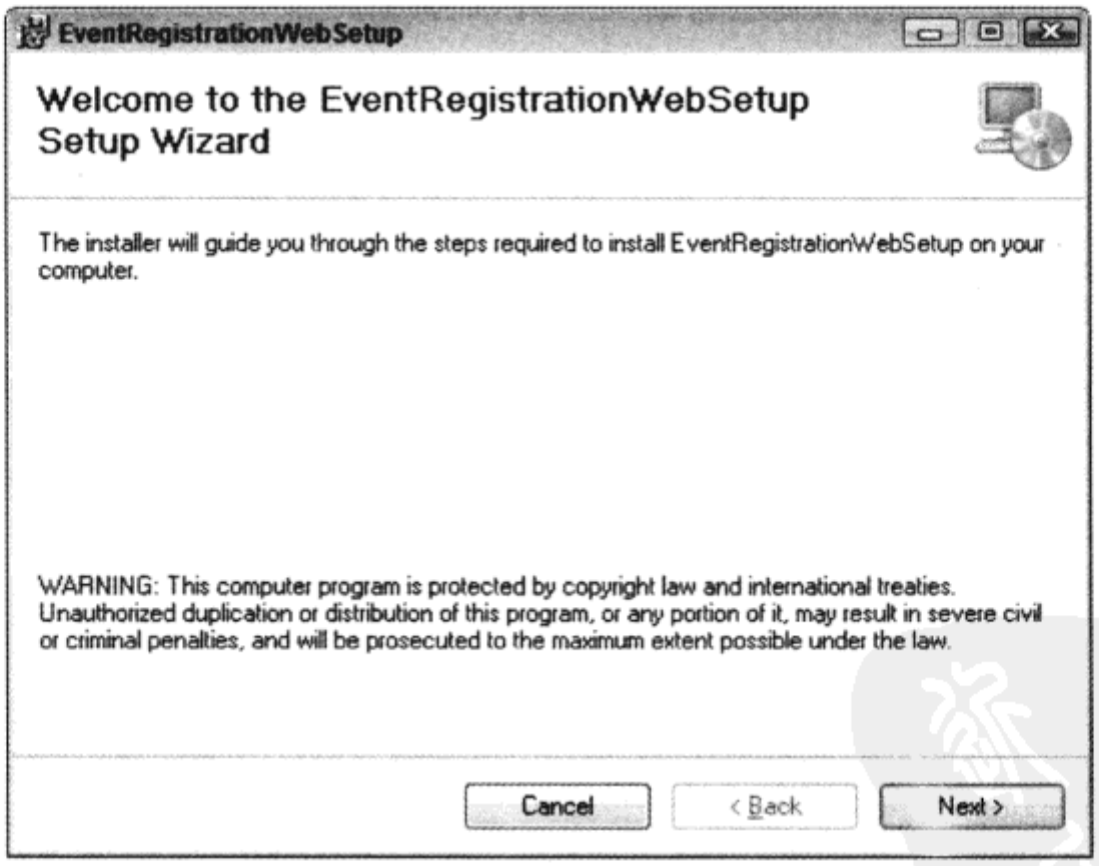


图　20-13

(2) 在 Select Installation Address 页面上，如图 20-14 所示，把虚拟目录重命名为没有用 IIS 配置的名称。接着选择前面创建的应用程序池(Beginning Visual C# App Pool)，单击 Next 按钮。

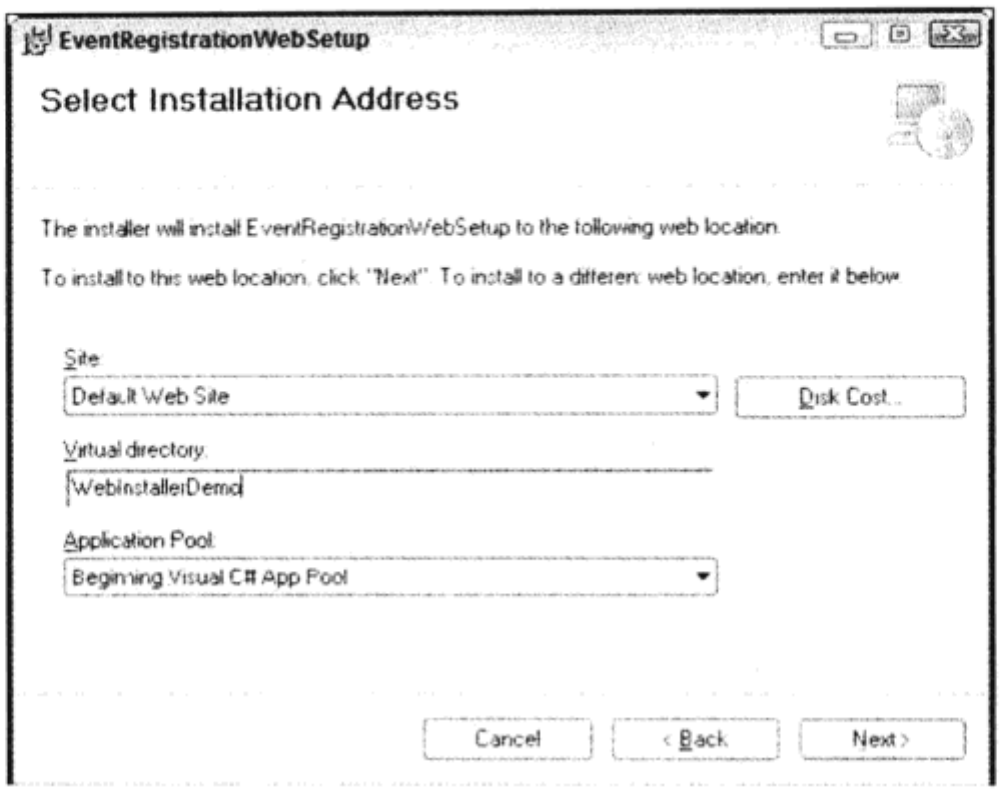


图 20-14

(3) 在 Confirm Installation 对话框中单击 Next 按钮，确认安装，下一个对话框会在安装程序运行过程中，显示出进度条。

(4) 成功安装后，会显示 Installation Complete 对话框，如图 20-15 所示。单击 Close 按钮。

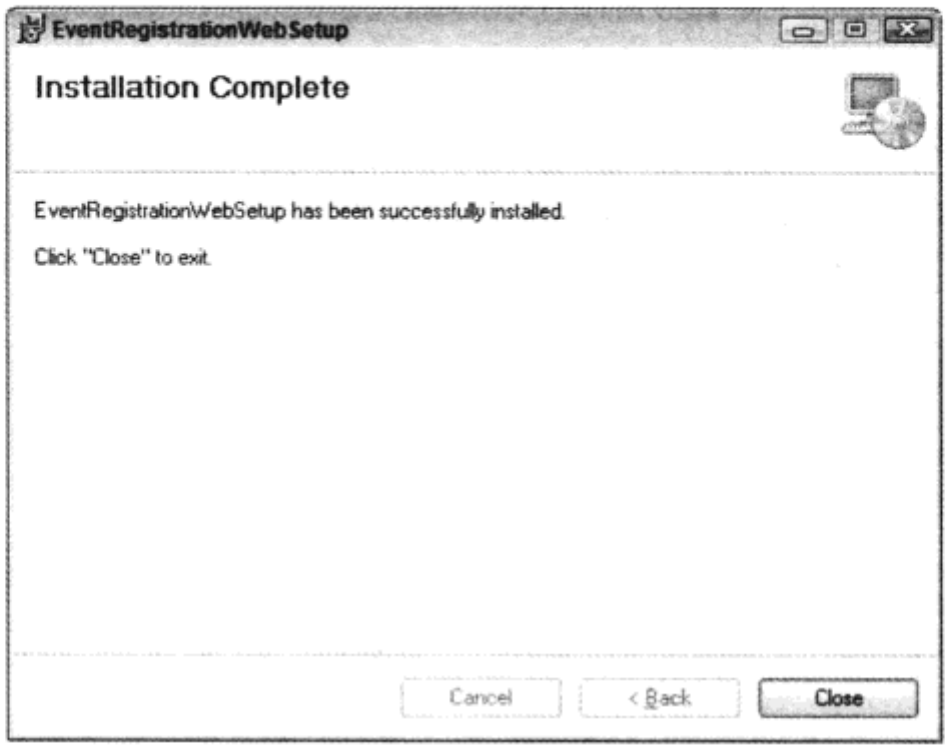


图 20-15

(5) 现在可以在新的虚拟目录下启动 Web 站点了。

20.6 小结

本章介绍了部署 Web 应用程序的不同选项。Copy Web Site 工具可以使用文件共享、FTP 或 FrontPage Server Extensions 把文件复制到 Web 服务器上。文件的同步可以双向进行。发布 Web 应用

程序是 VS2010 的一个新特性，可以将其与 1-Click 发布选项一起使用，发布到目录、FTP 服务器或安装了 FrontPage Server Extensions 的 IIS 上。如果有 IIS 上的管理权限进行发布，就可以在 IIS 中创建一个新的应用程序，来安装 Web 应用程序。安装项目不仅复制 ASP.NET 页面和程序集，还在 IIS 中创建了一个虚拟目录。

20.7　练习

(1) 复制和发布 Web 应用程序有什么区别？在什么场合下使用它们？
(2) 何时使用安装程序比复制站点更合适？
(3) 发布 Web 项目有哪些不同的选项？各个发布选项有什么要求？
(4) 把第 19 章的 Web 服务发布到用 IIS 定义的一个虚拟目录中。

附录 A 给出了练习答案。

20.8　本章要点

主　题	重 要 概 念
IIS 配置	为了在 IIS 上运行 ASP.NET Web 应用程序，必须配置 IIS。在请求带有特定扩展名(如.aspx)的文件时，处理程序映射会定义要调用的类。通过应用程序池配置，可以定义要使用的.NET 运行库版本
复制 Web 站点	发布 Web 应用程序的一个简单选项是使用复制功能。复制 Web 站点的菜单不能用于 Web 项目，只能用于 Visual Studio Web 站点。可以在开发计算机和服务器之间来回复制文件
发布 Web 应用程序	如果使用 Web 站点的主机，就可以使用 Visual Studio 2010 中发布 Web 应用程序的一个简单选项 1-Click。使用发布菜单，还可以把 Web 应用程序发布到 FTP 服务器和文件系统上。Web 应用程序使用的数据库也可以发布
Web 应用程序的 Windows 安装程序	如果需要在 IIS 中创建 Web 应用程序，就可以使用安装程序。Web Setup Project 模板会创建一个 Windows 安装程序文件，它不仅会复制 Web 应用程序的内容，还会创建一个虚拟目录

第Ⅳ部分

数 据 访 问

- 第 21 章　文件系统数据
- 第 22 章　XML
- 第 23 章　LINQ 简 介
- 第 24 章　应用 LINQ

PDG

第21章

文件系统数据

本章的主要内容：

- 流的含义，以及.NET 如何使用流类访问文件
- 如何使用 File 对象处理文件结构
- 如何读写文件
- 如何在文件中读写格式化的数据
- 如何读写压缩文件
- 如何序列化和反序列化对象
- 如何监控文件和目录的变化

本章将介绍如何读写文件，这是许多.NET 应用程序的一个基础性工作。我们要讨论用于创建、读写文件的主要类，支持在 C#代码中处理文件系统的类。本章不详细介绍全部的类，但将深入介绍一些类，以便读者理解概念和基本理论。

文件是在应用程序的实例之间存储数据的一种便利方式，也可以用于在应用程序之间传输数据。文件可以存储用户和应用程序配置，以便在下次运行应用程序时检索它们。定界的文本文件(例如用逗号分隔的文件)由许多旧系统使用；为了与这些旧系统进行交互，还需要了解如何处理定界数据。.NET Framework 提供的工具可以在应用程序中有效地使用文件。

21.1 流

在.NET Framework 中进行的所有输入和输出工作都要用到流(stream)。流是序列化设备(serial device)的抽象表示。序列化设备可以以线性方式存储数据，并可以按同样的方式访问：一次访问一个字节。此设备可以是磁盘文件、网络通道、内存位置或其他支持以线性方式读写的对象。把设备变成抽象的，就可以隐藏流的底层目标和源。这种抽象的级别支持代码重用，允许编写更通用的例程，因为不必担心数据传输方式的特性。因此，当应用程序从文件输入流、网络输入流或其他流中读取数据时，就可以转换并重用类似的代码。而且，使用文件流还可以忽略每种设备的物理机制，

无需担心硬盘头或内存分配问题。

有两种类型的流：

- **输出流**：当向某些外部目标写入数据时，就要用到输出流。这可以是物理磁盘文件、网络位置、打印机或另一个程序。理解流编程技术可以带来许多高级应用。本章仅讨论文件系统数据，所以只介绍写入磁盘文件。
- **输入流**：用于将数据读入程序可以访问的内存或变量中。到目前为止，我们使用的最常见的输入流形式是键盘。输入流可以来自任何源，在此主要关注读取磁盘文件。适用于读/写磁盘文件的概念也适用于大多数设备，因此通过本章的学习，我们就会对流有基本的认识，并学习可以应用于许多情况的有效方法。

21.2　用于输入和输出的类

System.IO 命名空间包含本章要介绍的几乎所有的类。System.IO 包含用于在文件中读写数据的类，必须在 C#应用程序中引用此名称空间才能访问这些类，而无需完全限定类型名。在 System.IO 命名空间中包含了不少类，在图 21-1 中就可以看到，但这里仅介绍用于文件输入和输出的主要类。

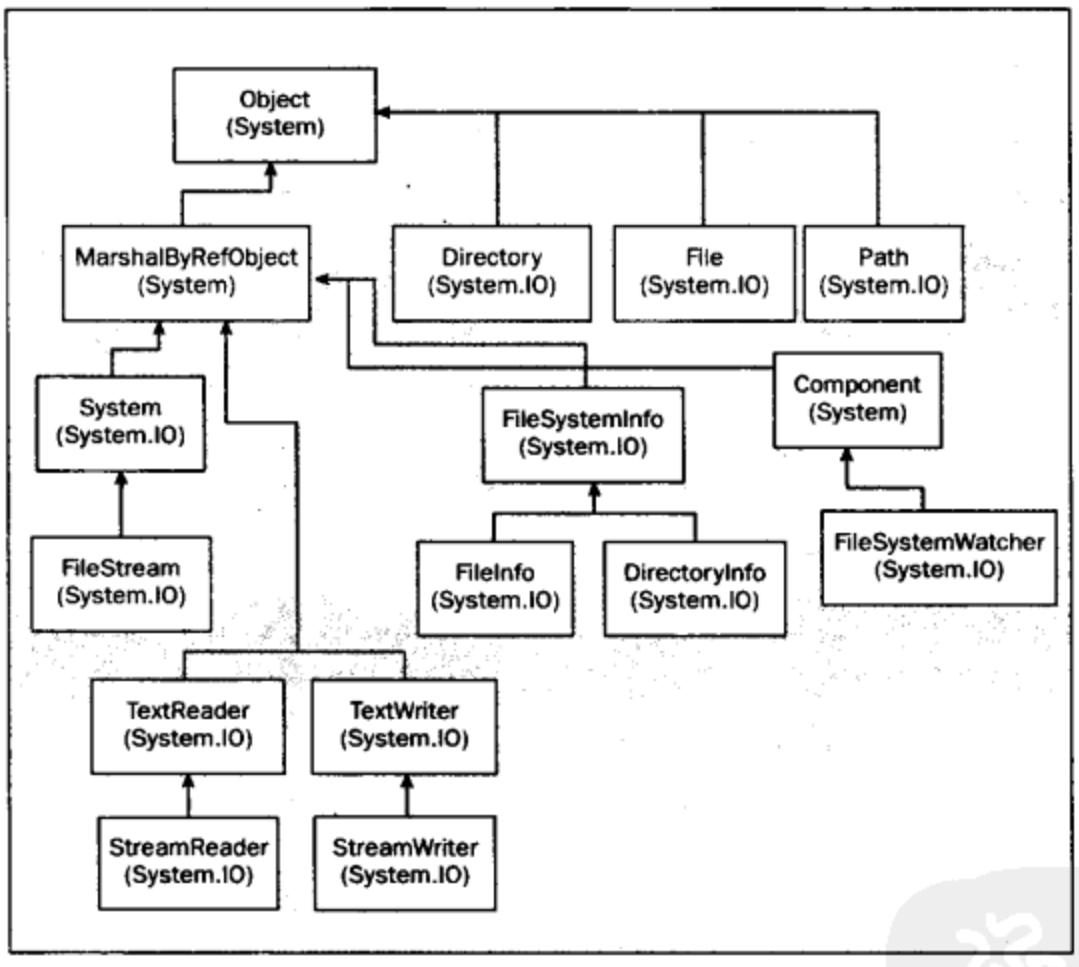


图　21-1

本章将介绍如表 21-1 中所示的一些类。

表　21-1

类	说　明
File	静态实用类，提供许多静态方法，用于移动、复制和删除文件
Directory	静态实用类，提供许多静态方法，用于移动、复制和删除目录

(续表)

类	说　明
Path	实用类，用于处理路径名称
FileInfo	表示磁盘上的物理文件，该类包含处理此文件的方法。要完成对文件的读写工作，就必须创建 Stream 对象
DirectoryInfo	表示磁盘上的物理目录，该类包含处理此目录的方法
FileStreamInfo	用作 FileInfo 和 DirectoryInfo 的基类，可以使用多态性同时处理文件和目录
FileStream	表示可写或可读，或二者均可的文件。此文件可以同步或异步地读写
StreamReader	从流中读取字符数据，可以使用 FileStream 将其创建为基类
StreamWriter	向流写入字符数据，可以使用 FileStream 将其创建为基类
FileSystemWatcher	FileSystemWatcher 是本章要介绍的最复杂的类。它用于监控文件和目录，提供了这些文件和目录发生变化时应用程序可以捕获的事件。在 Windows 编程技术中缺乏此功能，但是现在.NET Framework 很容易对文件系统事件作出响应

本章还将介绍 System.IO.Compression 名称空间，它允许使用 GZIP 压缩或 Deflate 压缩模式读写压缩文件：

- DeflateStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，使用 Deflate 算法来实现压缩。
- GZipStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，压缩使用 GZIP 算法来实现。

最后，学习使用 System.Runtime.Serialization 名称空间及其子名称空间进行对象的序列化，主要介绍 System.Runtime.Serialization.Formatters.Binary 名称空间中的 BinaryFormatter 类，它允许把对象序列化为二进制数据流，并可以反序列化这些数据。

21.2.1　File 类和 Directory 类

File 和 Directory 实用类提供了许多静态方法，用于处理文件和目录。这些方法可以移动文件、查询和更新特性，创建 FileStream 对象。如第 8 章所述，可以在类上调用静态方法，而无需创建它们的实例。

File 类的一些最常用的静态方法如表 21-2 所示。

表　21-2

方　法	说　明
Copy()	将文件从源位置复制到目标位置
Create()	在指定的路径上创建文件
Delete()	删除文件
Open()	返回指定路径上的 FileStream 对象
Move()	将指定的文件移动到新位置。可以在新位置为文件指定不同的名称

Directory 类的一些常用的静态方法如表 21-3 所示。

表 21-3

方 法	说 明
CreateDirectory()	创建具有指定路径的目录
Delete()	删除指定的目录及其中的所有文件
GetDirectories()	返回表示指定目录下的目录名的 string 对象数组
EnumerateDirectories()	与 GetDirectories()类似，但返回目录名的 IEnumerable<string>集合
GetFiles()	返回在指定目录中的文件名的 string 对象数组
EnumerateFiles()	与 GetFiles()类似，但返回文件名的 IEnumerable<string>集合
GetFilesSystemEntries()	返回在指定目录中的文件和目录名的 string 对象数组
Enumerate FilesSystemEntries()	与 GetFilesSystemEntries()类似，但返回文件和目录名的 IEnumerable<string>集合
Move()	将指定的目录移到新位置。可以在新位置为文件夹指定一个新名称

其中的 3 个 EnumerateXxx()方法是.NET 4 新增的，在存在大量文件或目录时，其性能比对应的 GetXxx()方法好。

21.2.2 FileInfo 类

FileInfo 类不像 File 类，它不是静态的，没有静态方法，仅可用于实例化的对象。FileInfo 对象表示磁盘或网络位置上的文件。提供文件路径，就可以创建一个 FileInfo 对象：

```
FileInfo aFile = new FileInfo(@"C:\Log.txt");
```

本章处理的是表示文件路径的字符串，该字符串中有许多“\”字符，所以上述字符串的前缀@表示这个字符串应逐个字符地解释，“\”解释为“\”，而不解释为转义字符。如果没有@前缀，就需要使用“\\”代替“\”，以免把这个字符解释为转义字符。本章总是在字符串前面加上前缀@。

也可以把目录名传送给 FileInfo 的构造函数，但实际上这并不是很有效。这么做会用所有的目录信息初始化 FileInfo 的基类 FilesSystemInfo，但 FileInfo 中与文件相关的专用方法或属性都不会工作。

FileInfo 类提供的许多方法类似于 File 类的方法，但是因为 File 是静态类，它需要一个字符串参数为每个方法调用指定文件位置。因此，下面的调用可以完成相同的工作：

```
FileInfo aFile = new FileInfo("Data.txt");

if (aFile.Exists)
   Console.WriteLine("File Exists");

if (File.Exists("Data.txt"))
   Console.WriteLine("File Exists");
```

这段代码检查文件 Data.txt 是否存在。注意，这里没有指定任何目录信息，这说明只检查当前的工作目录。这个目录包含调用此代码的应用程序。本章后面的“路径名和相对路径”一节会详细介绍这一内容。

大多数 FileInfo 方法采用这种方式反映 File 方法。在大多数情况下使用什么技术并不重要，但下面的规则有助于确定哪种技术更合适：

- 如果仅进行单一方法调用，则可以使用静态 File 类上的方法。在此，单一调用要快一些，因为.NET Framework 不必实例化新对象，再调用方法。
- 如果应用程序在文件上执行几种操作，则实例化 FileInfo 对象并使用其方法就更好一些。这节省时间，因为对象已在文件系统上引用正确的文件，而静态类必须每次都寻找文件。

FileInfo 类也提供了与底层文件相关的属性，其中一些属性可以用来更新文件，其中很多属性都继承于 FileSystemInfo，所以可应用于 File 和 Directory 类。FileSystemInfo 类的属性如表 21-4 所示。

表　21-4

属　性	说　明
Attributes	使用 FileAttributes 枚举，获取或者设置当前文件或目录的特性
CreationTime， CreationTimeUtc	获取当前文件的创建日期和时间，可以使用 UTC 和非 UTC 版本
Extension	提取文件的扩展名。这个属性是只读的
Exists	确定文件是否存在，这是一个只读抽象属性，在 FileInfo 和 DirectoryInfo 中进行了重写
FullName	检索文件的完整路径，这个属性是只读的
LastAccessTime， LastAccessTimeUtc	获取或设置上次访问当前文件的日期和时间，可以使用 UTC 和非 UTC 版本
LastWriteTime， LastWriteTimeUtc	获取或设置上次写入当前文件的日期和时间，可以使用 UTC 和非 UTC 版本
Name	检索文件的完整路径，这是一个只读抽象属性，在 FileInfo 和 DirectoryInfo 中进行了重写

FileInfo 的专用属性如表 21-5 所示。

表　21-5

属　性	说　明
Directory	检索一个 DirectoryInfo 对象，表示包含当前文件的目录。这个属性是只读的
DirectoryName	返回文件目录的路径。这个属性是只读的
IsReadOnly	文件只读特性的快捷方式。这个属性也可以通过 Attributes 来访问
Length	获取文件的容量(以字节为单位)，返回 long 值。这个属性是只读的

FileInfo 对象本身不表示流。要读写文件，必须创建 Stream 对象。FileInfo 对象提供了几个返回实例化 Stream 对象的方法来帮助做到这一点。

21.2.3 DirectoryInfo 类

DirectoryInfo 类的作用类似于FileInfo 类。它是一个实例化的对象，表示计算机上的单一目录。同 FileInfo 类一样，在 Directory 和 DirectoryInfo 之间可以复制许多方法调用。选择使用 File 或 FileInfo 方法的规则也适用于 DirectoryInfo 方法：

- 如果进行单一调用，就使用静态 Directory 类。
- 如果进行一系列调用，则使用实例化的 DirectoryInfo 对象。

DirectoryInfo 类的大多数属性继承自 FileSystemInfo，与 FileInfo 类一样，但这些属性作用于目录上，而不是文件上。还有两个 DirectoryInfo 专用属性，如表 21-6 所示。

表 21-6

属 性	说 明
Parent	检索一个 DirectoryInfo 对象，表示包含当前目录的目录。这个属性是只读的
Root	检索一个 DirectoryInfo 对象，表示包含当前目录的根目录，例如 C:\目录。这个属性是只读的

21.2.4 路径名和相对路径

在.NET 代码中规定路径名时，可以使用绝对路径名，也可以使用相对路径名。绝对路径名显式地规定文件或目录来自于哪一个已知的位置，比如 C:驱动器。它的一个示例是 C:\Work\ LogFile.txt。注意这个路径准确地定义了其位置。

相对路径名相对于一个起始位置。使用相对路径名时，无需规定驱动器或已知的位置；前面的当前工作目录就是起点，这是相对路径名的默认设置。例如，如果应用程序运行在 C:\Development\ FileDemo 目录上，并使用相对路径 LogFile.txt，该文件就是 C:\Development\ FileDemo\LogFile.txt。为了上移目录，要使用..字符串。这样，在同一个应用程序中，路径..\Log.txt 表示 C:\Development\ Log.txt 文件。

如前所述，工作目录起初设置为运行应用程序的目录。当使用 VS 或 VCE 开发程序时，这就表示应用程序是所创建的项目文件夹下的几个目录。它通常位于 ProjectName\bin\ Debug。要访问项目根文件夹中的文件，必须用..\..\上移两个目录，这在本章中很常见。

只要需要，就可以使用 Directory.GetCurrentDirectory()找出工作目录的当前设置，也可以使用 Directory.SetCurrentDirectory()设置新路径。

21.2.5 FileStream 对象

FileStream 对象表示在磁盘或网络路径上指向文件的流。这个类提供了在文件中读写字节的方法，但经常使用 StreamReader 或 StreamWriter 执行这些功能。这是因为 FileStream 类操作的是字节和字节数组，而 Stream 类操作的是字符数据。字符数据易于使用，但是有些操作，如随机文件访问(访问文件中间某点的数据)，就必须由 FileStream 对象执行，稍后对此进行介绍。

还有几种方法可以创建 FileStream 对象。构造函数具有许多不同的重载版本，最简单的构造函数仅有两个参数，即文件名和 FileMode 枚举值。

```
FileStream aFile = new FileStream(filename, FileMode.Member);
```

FileMode 枚举包含几个成员，指定了如何打开或创建文件。稍后介绍这些枚举成员。另一个常用的构造函数如下：

```
FileStream aFile = new FileStream(filename, FileMode.Member, FileAccess. Member);
```

第三个参数是 FileAccess 枚举的一个成员，它指定了流的作用。FileAccess 枚举的成员如表 21-7 所示。

表 21-7

成　员	说　明
Read	打开文件，用于只读
Write	打开文件，用于只写
ReadWrite	打开文件，用于读写

对文件进行非 FileAccess 枚举成员指定的操作会导致抛出异常。此属性的作用是，基于用户的身份验证级别改变用户对文件的访问权限。

在 FileStream 构造函数不使用 FileAccess 枚举参数的版本中，使用默认值 FileAccess. ReadWrite。

FileMode 枚举成员如表 21-8 所示。使用每个值会发生什么，取决于指定的文件名是否表示已有的文件。注意，这个表中的项表示创建流时该流指向文件中的位置，下一节将详细讨论这个主题。除非特别说明，否则流就指向文件的开头。

表 21-8

成　员	文 件 存 在	文件不存在
Append	打开文件，流指向文件的末尾，只能与枚举 FileAccess.Write 结合使用	创建一个新文件。只能与枚举 FileAccess.Write 结合使用
Create	删除该文件，然后创建新文件	创建新文件
CreateNew	抛出异常	创建新文件
Open	打开现有的文件，流指向文件开头	抛出异常
OpenOrCreate	打开文件，流指向文件开头	创建新文件
Truncate	打开现有文件，清除其内容。流指向文件开头，保留文件的初始创建日期	抛出异常

File 和 FileInfo 类都提供了 OpenRead()和 OpenWrite()方法，更易于创建 FileStream 对象。前者打开了只读访问的文件，后者只允许写入文件。这些都提供了快捷方式，因此不必以 FileStream 构造函数的参数形式提供前面所有的信息。例如，下面的代码行打开了用于只读访问的 Data.txt 文件：

```
FileStream aFile = File.OpenRead("Data.txt");
```

下面的代码执行同样的功能：

```
FileInfo aFileInfo = new FileInfo("Data.txt");
```

```
FileStream aFile = aFileInfo.OpenRead();
```

1. 文件位置

FileStream类维护内部文件指针，该指针指向文件中进行下一次读写操作的位置。在大多数情况下，当打开文件时，它就指向文件的开始位置，但是可以修改此指针。这允许应用程序在文件的任何位置读写，随机访问文件，或直接跳到文件的特定位置上。当处理大型文件时，这非常省时，因为马上可以找到正确的位置。

实现此功能的方法是 Seek()方法，它有两个参数：第一个参数规定文件指针移动距离(以字节为单位)。第二个参数规定开始计算的起始位置，用SeekOrigin 枚举的一个值表示。SeekOrigin 枚举包含 3 个值：Begin、Current 和 End。

例如，下面的代码行将文件指针移动到文件的第 8 个字节，其起始位置就是文件的第 1 个字节：

```
aFile.Seek(8,SeekOrigin.Begin);
```

下面的代码行将指针从当前位置开始向前移动 2 个字节。如果在上面的代码行之后执行下面的代码，文件指针就指向文件的第 10 个字节：

```
aFile.Seek(2,SeekOrigin.Current);
```

注意读写文件时，文件指针会随之改变。在读取了10 个字节之后，文件指针就指向被读取的第10 个字节之后的字节。

也可以规定反向查找位置，这可以与 SeekOrigin.End 枚举值一起使用，查找靠近文件末端的位置。下面的代码会查找文件中倒数第 5 个字节：

```
aFile.Seek(-5, SeekOrigin.End);
```

采用这种方式访问的文件有时称为随机访问文件，因为应用程序可以访问文件中的任何位置。稍后介绍的 Stream 类可以连续地访问文件，但不允许以这种方式操作文件指针。

.NET 4 引入了一个新名称空间 System.IO.MemoryMappedFiles，它包含的类型(例如 MemoryMappedFile)提供了另一种随机访问特大型文件的方式。这个名称空间在本章不介绍，但如果需要访问特大型文件，就可以考虑使用这种方式。

2. 读取数据

使用 FileStream 类读取数据不像使用本章后面介绍的 StreamReader 类读取数据那样容易。这是因为 FileStream 类只能处理原始字节(raw byte)。处理原始字节的功能使 FileStream 类可以用于任何数据文件，而不仅仅是文本文件。通过读取字节数据，FileStream 对象可以用于读取诸如图像和声音的文件。这种灵活性的代价是，不能使用 FileStream 类将数据直接读入字符串，而使用 StreamReader 类却可以这样处理。但是有几种转换类可以很轻易地将字节数组转换为字符数组，或者将字符数组转换为字节数组。

FileStream.Read()方法是从 FileStream 对象所指向的文件中访问数据的主要手段。这个方法从文件中读取数据，再把数据写入一个字节数组。它有三个参数：第一个参数是传入的字节数组，用来接受 FileStream 对象中的数据。第二个参数是字节数组中开始写入数据的位置；它通常是 0，表示从数组开端向文件中写入数据。最后一个参数指定从文件中读出多少字节。

下面的示例演示了从随机访问文件中读取数据。要读取的文件实际是为此示例创建的类文件。

试一试：从随机访问文件中读取数据

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 ReadFile。

(2) 在 Program.cs 文件的顶部添加下面的 using 指令：

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

代码段 ReadFile\Program.cs

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
   byte[] byData = new byte[200];
   char[] charData = new Char[200];

   try
   {
      FileStream aFile = new FileStream("../../Program.cs",FileMode.Open);
      aFile.Seek(113,SeekOrigin.Begin);
      aFile.Read(byData,0,200);
   }
   catch(IOException e)
   {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WriteLine(e.ToString());
      Console.ReadKey();
      return;
   }

   Decoder d = Encoding.UTF8.GetDecoder();
   d.GetChars(byData, 0, byData.Length, charData, 0);

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

(4) 运行应用程序。结果如图 21-2 所示。

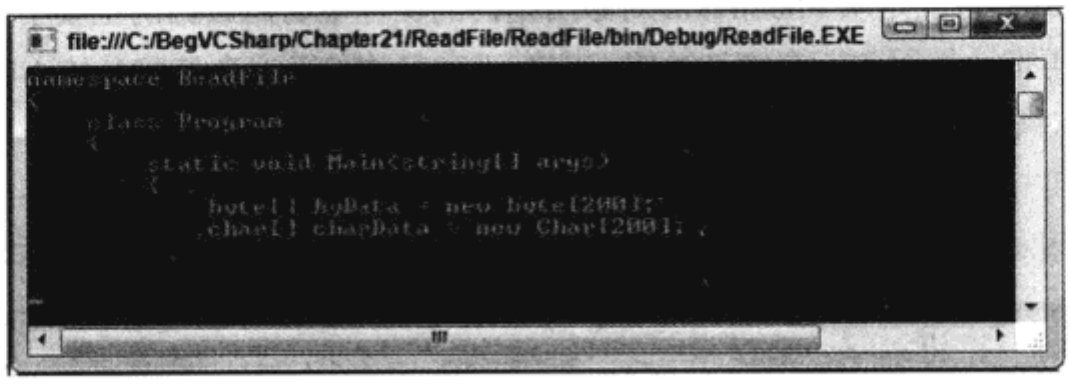


图 21-2

示例的说明

此应用程序打开自己的.cs 文件，用于从中读取。它在下面的代码行中使用..字符串向上逐级导航两个目录，找到该文件：

```
FileStream aFile = new FileStream("../../Program.cs",FileMode.Open);
```

下面两行代码执行查找工作，并从文件的具体位置读取字节：

```
aFile.Seek(113,SeekOrigin.Begin);
aFile.Read(byData,0,200);
```

第一行代码将文件指针移到文件的第 113 个字节。在 Program.cs 中，是指 namespace 中的 “n”；其前面的 113 个字符是 using 指令。第二行将接下来的 200 个字节读入到 byData 字节数组中。

注意，这两行代码封装在 try...catch 块中，以处理可能抛出的异常。

```
try
{
   aFile.Seek(113,SeekOrigin.Begin);
   aFile.Read(byData,0,100);
}
catch(IOException e)
{
   Console.WriteLine("An IO exception has been thrown!");
   Console.WriteLine(e.ToString());
   Console.ReadKey();
   return;
}
```

文件 I/O 涉及到的所有操作都可以抛出 IOException 类型的异常。所有产品代码都必须包含错误处理，尤其是处理文件系统时更是如此。本章的所有示例都具有错误处理的基本形式。

从文件中获取了字节数组后，就需要将其转换为字符数组，以便在控制台显示它。为此，使用 System.Text 名称空间的 Decoder 类。此类用于将原始字节转换为更有用的项，比如字符：

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
```

这些代码基于 UTF-8 编码模式创建了 Decoder 对象。这就是 Unicode 编码模式。然后调用 GetChars()方法，此方法提取字节数组，将它转换为字符数组。完成之后，就可以将字符数组输出到控制台。

3. 写入数据

向随机访问文件中写入数据的过程与从中读取数据非常类似。首先需要创建一个字节数组；最简单的办法是首先构建要写入文件的字符数组。然后使用 Encoder 对象将其转换为字节数组，其用法非常类似于 Decoder 对象。最后调用 Write()方法，将字节数组传送到文件中。

下面构建一个简单的示例演示其过程。

试一试：将数据写入随机访问文件

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 WriteFile。

(2) 在 Program.cs 文件顶部添加下面的 using 指令：

可从 wrox.com 下载源代码

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

代码段 WriteFile\Program.cs

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
   byte[] byData;
   char[] charData;
   try
   {
      FileStream aFile = new FileStream("Temp.txt", FileMode.Create);
      charData = "My pink half of the drainpipe.".ToCharArray();
      byData = new byte[charData.Length];
      Encoder e = Encoding.UTF8.GetEncoder();
      e.GetBytes(charData, 0, charData.Length, byData, 0, true);

      // Move file pointer to beginning of file.
      aFile.Seek(0, SeekOrigin.Begin);
      aFile.Write(byData, 0, byData.Length);
   }
   catch (IOException ex)
   {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WriteLine(ex.ToString());
      Console.ReadKey();
      return;
   }
}
```

(4) 运行该应用程序。稍后将其关闭。

(5) 导航到应用程序目录 ——在目录中已经保存了文件，因为我们使用了相对路径。目录位于 WriteFile\bin\Debug 文件夹中。打开 Temp.txt 文件。可以在文件中看到如图 21-3 所示的文本。

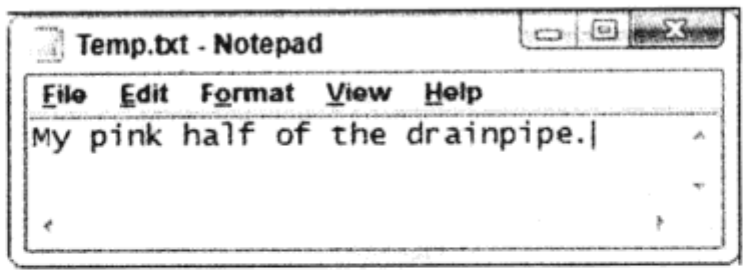


图 21-3

示例的说明

此应用程序在自己的目录中打开文件，并在文件中写入了一个简单的字符串。这个示例的结构非常类似于前面的示例，只是用 Write()代替了 Read()，用 Encoder 代替了 Decoder。

下面的代码行使用 String 类的 ToCharArray()静态方法，创建了字符数组。因为 C#中的所有事物都是对象，文本“My pink half of the drainpipe.”实际上是一个 String 对象(尽管有点古怪)，所以甚至可以在字符串上调用这些静态方法。

```
CharData = " My pink half of the drainpipe. ".ToCharArray();
```

下面的代码行显示了如何将字符数组转换为 FileStream 对象需要的正确字节数组。

```
Encoder e = Encoding.UTF8.GetEncoder();
e.GetBytes(charData,0,charData.Length, byData,0,true);
```

这次，要基于 UTF-8 编码方法来创建 Encoder 对象。也可以将 Unicode 用于解码。这里在写入流之前，需要将字符数据编码为正确的字节格式。在 GetBytes()方法中可以完成这些工作，它可以将字符数组转换为字节数组，并将字符数组作为第一个参数(本例中的 charData)，将该数组中起始位置的下标作为第二个参数(0 表示数组的开头)。第三个参数是要转换的字符数量(charData.Length，charData 数组中的元素个数)。第四个参数是在其中放入数据的字节数组(byData)，第五个参数是在字节数组中开始写入位置的索引(0 表示 byData 数组的开头)。

最后一个参数决定在结束后 Encoder 对象是否应该更新其状态，即 Encoder 对象是否仍然保留它原来在字节数组中的内存位置。这有助于以后调用 Encoder 对象，但是当只进行单一调用时，这就没有什么意义。最后对 Encoder 的调用必须将此参数设置为 true，以清空其内存，释放对象，用于垃圾回收。

此后使用 Write()方法向 FileStream 写入字节数组就变得非常简单：

```
aFile.Seek(0,SeekOrigin.Begin);
aFile.Write(byData,0,byData.Length);
```

与 Read()方法一样，Write()方法也有三个参数：要写入的数组，开始写入的数组索引和要写入的字节数。

21.2.6 StreamWriter 对象

操作字节数组比较麻烦，因为使用 FileStream 对象非常困难，那么，还有简单一些的方法吗？恐怕是没有了，因为有了 FileStream 对象，通常都会把它包装在 StreamWriter 或 Stream Reader 中，并使用它们的方法来处理文件。如果不需要将文件指针改变到任意位置，这些类就很容易操作文件。

StreamWriter 类允许将字符和字符串写入到文件中，它处理底层的转换，向 FileStream 对象写入

数据。

还可以通过许多方法创建 StreamWriter 对象。如果已经有了 FileStream 对象，则可以使用此对象来创建 StreamWriter 对象：

```
FileStream aFile = new FileStream("Log.txt",FileMode.CreateNew);
StreamWriter sw = new StreamWriter(aFile);
```

也可以直接从文件中创建 StreamWriter 对象：

```
StreamWriter sw = new StreamWriter("Log.txt",true);
```

这个构造函数的参数是文件名和一个 Boolean 值，这个 Boolean 值规定是追加文件，还是创建新文件：

- 如果此值设置为 false，则创建一个新文件，或者截取现有文件并打开它。
- 如果此值设置为 true，则打开文件，保留原来的数据。如果找不到文件，则创建一个新文件。

与创建 FileStream 对象不同，创建 StreamWriter 对象不会提供一组类似的选项：除了使用 Boolean 值追加文件或创建新文件之外，根本没有像 FileStream 类那样指定 FileMode 属性的选项。而且，没有设置 FileAccess 属性的选项，因此总是拥有对文件的读/写权限。为了使用高级参数，必须先在 FileStream 构造函数中指定这些参数，然后在 FileStream 对象中创建 StreamWriter，如下面的示例所示。

试一试：把数据写入输出流

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 StreamWrite。

(2) 再次使用 System.IO 名称空间，因此在 Program.cs 文件靠近顶部的位置添加下面的 using 指令：

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

代码段 StreamWrite\Program.cs

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
   try
   {
      FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
      StreamWriter sw = new StreamWriter(aFile);

      bool truth = true;
      // Write data to file.
      sw.WriteLine("Hello to you.");
      sw.WriteLine("It is now {0} and things are looking good.",
                   DateTime.Now.ToLongDateString());
```

```
            sw.Write("More than that,");
            sw.Write(" it's {0} that C# is fun.", truth);
            sw.Close();
        }
        catch(IOException e)
        {
            Console.WriteLine("An IO exception has been thrown!");
            Console.WriteLine(e.ToString());
            Console.ReadLine();
            return;
        }
    }
```

(4) 生成并运行该项目。如果找不到错误，则项目会很快运行，并关闭。因为我们在控制台上没有显示任何内容，所以在控制台中无法看到程序的执行情况。

(5) 进入应用程序目录，找到 Log.txt 文件，它位于 StreamWrite\bin\Debug 文件夹，这是因为我们使用了相对路径。

(6) 打开文件，可以看到图 21-4 所示的文本。

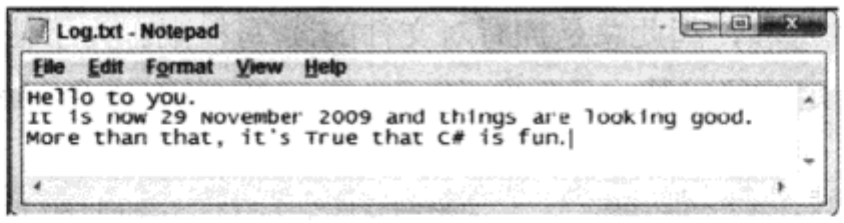


图 21-4

示例的说明

这个简单的应用程序演示了 StreamWriter 类的两个最重要的方法：Write()和 WriteLine()。这两个方法具有许多重载的版本，可以完成更高级的文件输出，但是本示例只使用基本的字符串输出。

WriteLine()方法会写入传递给它的字符串，其后跟有换行符。在此示例中，下一个写入操作在新行上开始。

如同可以向控制台写入格式化数据一样，也可以向文件写入格式化数据。例如，可以使用标准格式化参数把变量的值写入文件：

```
sw.WriteLine("It is now {0} and things are looking good.",
             DateTime.Now.ToLongDateString());
```

DateTime.Now 存储当前日期，ToLongDateString()方法用于把这个日期转换为易于读取的格式。

Write()方法只是把传送给它的字符串写入文件，但不追加换行符，因此可以使用多个 Write()语句写入完整的句子或段落。

```
sw.Write("More than that,");
sw.Write(" it's {0} that C# is fun.", truth);
```

这里也使用了格式化参数，这次是使用 Write()显示布尔值 truth。前面把这个变量设置为 true，其值会自动格式化，转换为字符串“True”。

可以使用 Write()和格式化参数写入用逗号分隔的文件：

```
[StreamWriter object].Write("{0},{1},{2}", 100, "A nice product", 10.50);
```

在更复杂的示例中，这些数据还可以来自数据库或其他数据源。

21.2.7　StreamReader 对象

输入流用于从外部源中读取数据。很多情况下，数据源是磁盘上的文件或网络的某些位置。任何可以发送数据的位置都可以是数据源，比如网络应用程序、Web 服务甚至是控制台。

用来从文件中读取数据的类是 StreamReader。同 StreamWriter 一样，这是一个通用类，可以用于任何流。下面的示例会再次围绕 FileStream 对象构造 StreamReader 类，使它指向正确的文件。

StreamReader 对象的创建方式与 StreamWriter 对象非常类似。创建它的最常见方式是使用前面创建的 FileStream 对象：

```
FileStream aFile = new FileStream("Log.txt",FileMode.Open);
StreamReader sr = new StreamReader(aFile);
```

同 StreamWriter 一样，StreamReader 类可以直接在包含具体文件路径的字符串中创建：

```
StreamReader sr = new StreamReader("Log.txt");
```

试一试：从输入流中读取数据

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新控制台应用程序 StreamRead。

(2) 必须再次输入 System.IO 名称空间，因此将下面的代码放在 Program.cs 文件的靠近顶部的位置：

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

代码段 StreamRead\Program.cs

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
   string strLine;

   try
   {
      FileStream aFile = new FileStream("Log.txt", FileMode.Open);
      StreamReader sr = new StreamReader(aFile);
      Line = sr.ReadLine();
      // Read data in line by line.
      while(strLine != null)
      {
         Console.WriteLine(strLine);
         Line = sr.ReadLine();
      }
      sr.Close();
   }
```

```
    catch(IOException e)
    {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WriteLine(e.ToString());
      return;
    }
    Console.ReadKey();
  }
```

(4) 把前面示例中创建的Log.txt 文件复制到 StreamRead\bin\Debug 目录下。如果没有 Log.txt 文件，FileStream 构造函数找不到该文件，就会抛出异常。

(5) 运行该应用程序，可以看到写入到控制台的文件文本，如图 21-5 所示。

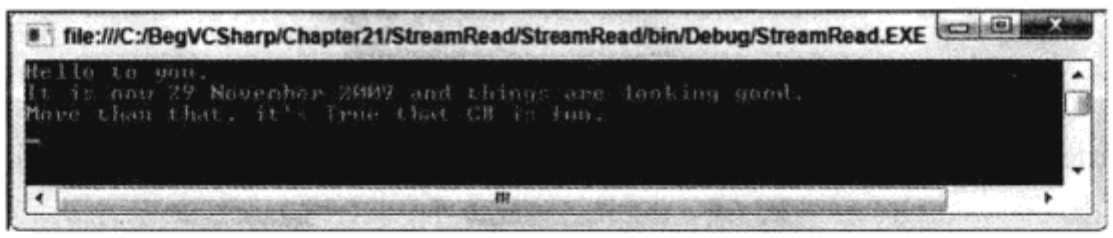


图 21-5

示例的说明

这个应用程序与前面的应用程序非常类似。其明显的区别就是，它是在读取数据，而不是写入数据。同前面一样，只有导入 System.IO 名称空间，才能访问需要的类。

使用 ReadLine()方法从文件中读取文本。这个方法读取换行符之前的文本，并以字符串的形式返回结果文本。当到达文件尾时，该方法就返回空值，通过这种方法可以测试文件是否已到达了尾部。注意使用了 while 循环，以便确保在执行循环体的代码之前读取的行不为空，这样就只显示文件的有效内容：

```
strLine = sr.ReadLine();
while(strLine != null)
{
  Console.WriteLine(strLine);
  strLine = sr.ReadLine();
}
```

1. 读取数据

ReadLine()方法不是在文件中访问数据的唯一方法。StreamReader 类还包含许多读取数据的方法。

读取数据最简单的方法是 Read()。此方法将流的下一个字符作为正整数值返回，如果到达了流的结尾处，则返回 - 1。使用 Convert 实用类可以把这个值转换为字符。在上面的示例中，程序的主体可以按如下方式重新编写：

```
StreamReader sr = new StreamReader(aFile);
int nChar;
nChar = sr.Read();
while(nChar != -1)
{
  Console.Write(Convert.ToChar(nChar));
  nChar = sr.Read();
}
sr.Close();
```

对于小型文件，可以使用一个非常方便的方法 ReadToEnd()。此方法读取整个文件，并将其作为字符串返回。在此，前面的应用程序可以简化为：

```
StreamReader sr = new StreamReader(aFile);
Line = sr.ReadToEnd();
Console.WriteLine(strLine);
sr.Close();
```

这似乎非常容易和方便，但必须小心。将所有的数据读取到字符串对象中，会迫使文件中的数据放到内存中。应根据数据文件的大小禁止这样处理。如果数据文件非常大，最好将数据留在文件中，并使用 StreamReader 的方法访问文件。

处理大型文件的另一种方式是.NET 4 中新增的静态方法 File.ReadLines()。实际上，File 的几个静态方法可以用于简化文件数据的读写，但这个方法特别有趣，因为它返回 IEnumerable<string>集合。可以迭代这个集合中的字符串，一次读取文件中的一行。使用这个方法，可以把前面的示例重写为：

```
foreach (string alternativeLine in File.ReadLines("Log.txt"))
   Console.WriteLine(alternativeLine);
```

可以看出，在.NET 中，可以通过多种不同的方式获得相同的结果——读取文件中的数据。您可以选择其中最适当的技术。

2. 用分隔符分隔的文件

用分隔符分隔的文件是一种常见的数据存储形式，由许多旧系统使用，如果应用程序必须与这种系统协作，就会经常遇到用分隔符分隔的数据格式。最常见的分隔符是逗号，例如 Excel 电子表格、Access 数据库或 SQL Server 数据库中的数据都可以导出为用逗号分隔的值(CSV)文件。

前面介绍了如何使用 StreamWriter 类编写使用这种方法存储数据的文件。使用逗号分隔的文件也易于读取。如第 5 章所述，String 类的 Split()方法可以用于将字符串转换为基于所提供的分隔符的数组。如果规定逗号为分隔符，它就会创建尺度合理的字符串数组，其中包含在初始逗号分隔的字符串中的全部数据。

下面的示例将说明其用途。这个示例处理用逗号分隔的值，把它们加载到 List<Dictionary<string, string>>对象中。这个示例非常通用，如果需要处理用逗号分隔的值，就可以在自己的应用程序中使用这项技术。

试一试：逗号分隔的值

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新控制台应用程序 CommaValues。

(2) 将下面的代码放在 Program.cs 文件的顶部。只有导入 System.IO 名称空间，才能进行文件处理：

可从 wrox.com 下载源代码

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

代码段 CommaValues\Program.cs

(3) 在 Program.cs 主体的 Main()方法之前，添加下面的 GetData()方法：

```
private static List<Dictionary<string, string>> GetData(
    out List<string> columns)
{
    string line;
    string[] stringArray;
    char[] charArray = new char[] {','};
    List<Dictionary<string, string>> data =
        new List<Dictionary<string, string>>();
    columns = new List<string>();
    try
    {
        FileStream aFile = new FileStream(@"..\..\SomeData.txt", FileMode.Open);
        StreamReader sr = new StreamReader(aFile);

        // Obtain the columns from the first line.
        // Split row of data into string array
        line = sr.ReadLine();
        stringArray = line.Split(charArray);

        for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
        {
            columns.Add(stringArray[x]);
        }

        line = sr.ReadLine();
        while (line != null)
        {
            // Split row of data into string array
            stringArray = line.Split(charArray);
            Dictionary<string, string> dataRow = new Dictionary<string, string>();

            for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
            {
                dataRow.Add(columns[x], stringArray[x]);
            }

            data.Add(dataRow);

            line = sr.ReadLine();
        }

        sr.Close();
        return data;
    }
    catch (IOException ex)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(ex.ToString());
        Console.ReadLine();
        return data;
    }
}
```

(4) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
   List<string> columns;
   List<Dictionary<string, string>> myData = GetData(out columns);

   foreach (string column in columns)
   {
      Console.Write("{0,-20}", column);
   }
   Console.WriteLine();

   foreach (Dictionary<string, string> row in myData)
   {
      foreach (string column in columns)
      {
         Console.Write("{0,-20}", row[column]);
      }
      Console.WriteLine();
   }
   Console.ReadKey();
}
```

(5) 从 Project | Add New Item 对话框中选择 Text File，添加一个新文本文件 SomeData.txt。

(6) 在新文本文件中输入下面的文本：

可从 wrox.com 下载源代码

```
ProductID,Name,Price
1,Spiky Pung,1000
2,Gloop Galloop Soup,25
4,Hat Sauce,12
```

代码段 CommaValues\SomeData.txt

(7) 运行该应用程序——可以看到写入到控制台的文件文本，如图 21-6 所示。

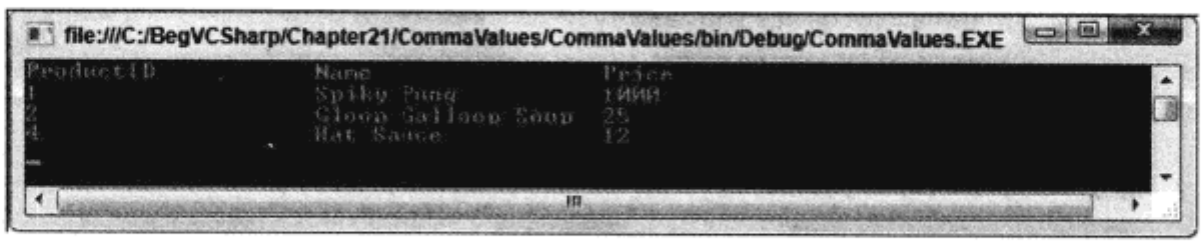


图　21-6

示例的说明

与前面的示例相同，这个应用程序也逐行地把文件的数据读入字符串。但是，因为这是一个包含用逗号分隔的文本值的文件，所以按不同的方式处理它。而且，还把读取的值存储在一些数据结构中。

首先需要查看一些用逗号分隔的数据：

```
ProductID,Name,Price
1,Spiky Pung,1000
```

注意第一行包含数据列的名称，其他行包含数据。所以，过程应从文件的第一行中获取列名，再从其他的行中获取数据。

GetData()方法声明为 static，所以可以在不创建类实例的情况下调用它。这个方法返回一个 List<Dictionary<string, string>>对象，下面要创建这个对象，并使用逗号分隔的文本文件中的数据填充它。该方法还返回一个包含标题名的 List<string>对象。下面的代码初始化了这些对象：

```
List<Dictionary<string, string>> data = new List<Dictionary<string, string>>();
columns = new List<string>();
```

columns 包含逗号分隔的文本文件中第一行的列名，data 包含后续各行的值。

首先创建一个 FileStream 对象，再构建一个 StreamReader 对象，就像前面的示例那样。现在就可以读取文件的第一行，从这个字符串中创建一个字符串数组：

```
line = sr.ReadLine();
stringArray = line.Split(charArray);
```

第 5 章介绍了 Split()方法，它的参数是一个字符数组，在这里该数组只包含“,”，所以 stringArray 包含的字符串数组是在 line 的每个“,”实例处进行分隔而构建的。当前是从文件的第一行中读取数据，这一行包含数据列的名称，所以需要迭代 stringArray 中的每个字符串，把它们添加到 columns 中：

```
for (int x = 0; x <= strArray.GetUpperBound(0); x++)
{
   columns.Add(strArray[x]);
}
```

数据有了列名后，就可以读取数据了。其代码与前面的 StreamRead 示例的代码相同，只是需要把 Dictionary<string, string>对象添加到 data 中：

```
line = sr.ReadLine();
while (line != null)
{
     // Split row of data into string array.
     stringArray = line.Split(charArray);
     Dictionary<string, string> dataRow = new Dictionary<string, string>();

     for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
     {
        dataRow.Add(columns[x], stringArray[x]);
     }

     data.Add(dataRow);

     line = sr.ReadLine();
}
```

对于文件中的每一行，都创建一个新的 Dictionary<string, string>对象，用一行数据填充它。这个集合中的每一项都有一个对应于列名的键和一个值，这个值对应于该行的列值。键从前面创建的 columns 对象中提取，值从使用 Split()获得的字符串中提取，Split()方法用于从数据文件中提取文本行。

读取完文件中的所有数据，就关闭 StreamReader，然后返回数据。Main()方法中的代码把从 GetData()方法获得的数据放在变量 myData 和 columns 中，并在控制台上显示这些信息。首先显示每一列的名称：

```
foreach (string column in columns)
{
    Console.Write("{0,-20}", column);
}
Console.WriteLine();
```

格式化字符串{0, - 20}中的 - 20 部分确保显示的名称在 20 字符长的列中左对齐，这有助于格式化显示结果。

最后迭代 myData 集合中的每个Dictionary<string, string>对象，显示该行中的值，这里再次使用格式化字符串来格式化输出结果。

```
foreach (Dictionary<string, string> row in myData)
{
  foreach (string column in columns)
  {
     Console.Write("{0,-20}", row[column]);
  }
  Console.WriteLine();
}
```

可以看到，使用.NET Framework 从(CSV)文件中提取有意义的数据非常容易。这种技术也很容易与后续章节中介绍的数据访问技术合并使用，即 CSV 文件中的数据可以像其他数据源(如数据库)中的数据那样操作。但从CSV 文件中提取的数据没有数据类型信息。当前是把所有的数据看作字符串，但对于企业级的商务应用程序来说，就需要更进一步，给提取的数据添加类型信息。这可能来自存储在CSV 文件中的附加信息，可以手工进行配置，也可以从文件的字符串中推断，具体取决于特定的应用程序。

第 22 章介绍的 XML 也是一种存储和传输数据的优秀方法，而 CSV 文件仍然非常普遍，还会使用相当长的时间。逗号分隔文件的优点是非常简洁，因此要比 XML 文件小些。

21.2.8　读写压缩文件

在处理文件时，常常会发现文件中有许多空格，耗尽了硬盘空间。图形和声音文件尤其如此。读者可能使用过能压缩文件和解压文件的工具，当希望带着文件到其他地方去或者把文件邮寄给朋友时，使用这些工具是很方便的。System.IO.Compression 名称空间就包含能在代码中压缩文件的类，这些类使用 GZIP 或 Deflate 算法，这两种算法都是公开的、免费的，任何人都可以使用。

但压缩文件并不只是把它们压缩一下就完事了。商业应用程序允许把多个文件放在一个压缩文件中，本节介绍的内容要简单得多：只是把文本数据保存在压缩文件中。不能在外部实用程序中访问这个文件，但这个文件比未压缩版本要小得多。

System.IO.Compression 名称空间中有两个压缩流类 DeflateStream 和 GZipStream，它们的工作方式非常类似。对于这两个类，都要用已有的流初始化它们，对于文件，流就是 FileStream 对象。此后就可以把它们用于 StreamReader 和 StreamWriter 了，就像使用其他流一样。除此之外，只需指定流是用于压缩(保存文件)还是解压缩(加载文件)，类就知道要对传送给它的数据执行什么操作。这最

好用一个示例来加以说明。

试一试：读写压缩数据

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 Compressor。

(2) 把下面的代码放在 Program.cs 靠近顶部的位置。需要导入System.IO 名称空间才能处理文件，导入 System.IO.Compression 才能使用压缩类：

可从 wrox.com 下载源代码

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

代码段 Compressor\Program.cs

(3) 把下面的方法添加到 Program.cs 中 Main()方法的前面：

```
static void SaveCompressedFile(string filename, string data)
{
   FileStream fileStream =
      new FileStream(filename, FileMode.Create, FileAccess.Write);
   GZipStream compressionStream =
      new GZipStream(fileStream, CompressionMode.Compress);
   StreamWriter writer = new StreamWriter(compressionStream);
   writer.Write(data);
   writer.Close();
}

static string LoadCompressedFile(string filename)
{
   FileStream fileStream =
      new FileStream(filename, FileMode.Open, FileAccess.Read);
   GZipStream compressionStream =
      new GZipStream(fileStream, CompressionMode.Decompress);
   StreamReader reader = new StreamReader(compressionStream);
   string data = reader.ReadToEnd();
   reader.Close();
   return data;
}
```

(4) 给 Main()方法添加如下代码：

```
static void Main(string[] args)
{
   try
   {
      string filename = "compressedFile.txt";

      Console.WriteLine(
         "Enter a string to compress (will be repeated 1000 times):");
      string sourceString = Console.ReadLine();
      StringBuilder sourceStringMultiplier =
```

```
            new StringBuilder(sourceString.Length * 100);
        for (int i = 0; i < 100; i++)
        {
            sourceStringMultiplier.Append(sourceString);
        }
        sourceString = sourceStringMultiplier.ToString();
        Console.WriteLine("Source data is {0} bytes long.", sourceString.Length);

        SaveCompressedFile(filename, sourceString);
        Console.WriteLine("\nData saved to {0}.", filename);

        FileInfo compressedFileData = new FileInfo(filename);
        Console.WriteLine("Compressed file is {0} bytes long.",
                          compressedFileData.Length);

        string recoveredString = LoadCompressedFile(filename);
        recoveredString = recoveredString.Substring(0, recoveredString.Length / 100);
        Console.WriteLine("\nRecovered data: {0}", recoveredString);

        Console.ReadKey();
    }
    catch (IOException ex)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(ex.ToString());
        Console.ReadKey();
    }
}
```

(5) 运行应用程序，输入一个长度合理的长字符串，结果如图 21-7 所示。

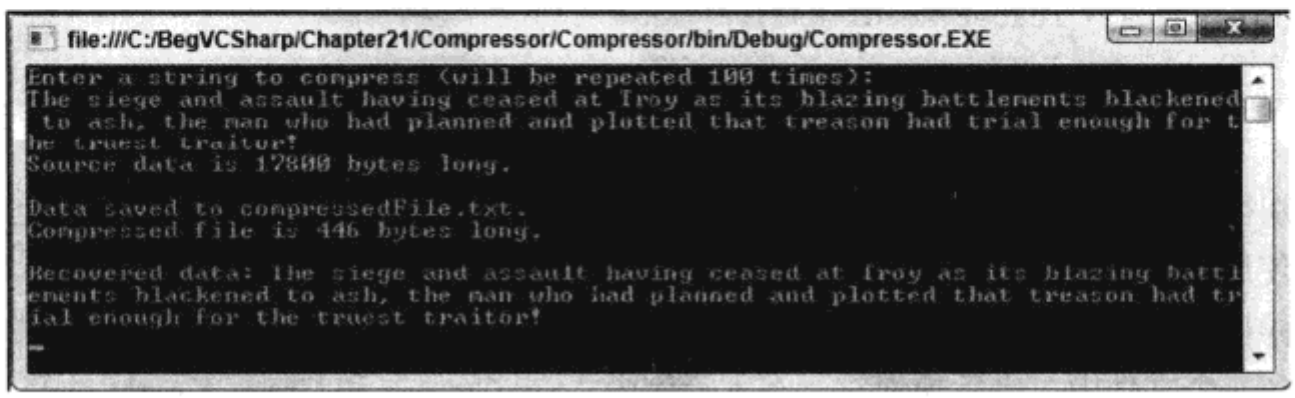


图　21-7

(6) 在记事本中打开 compressedFile.txt，文本如图 21-8 所示。

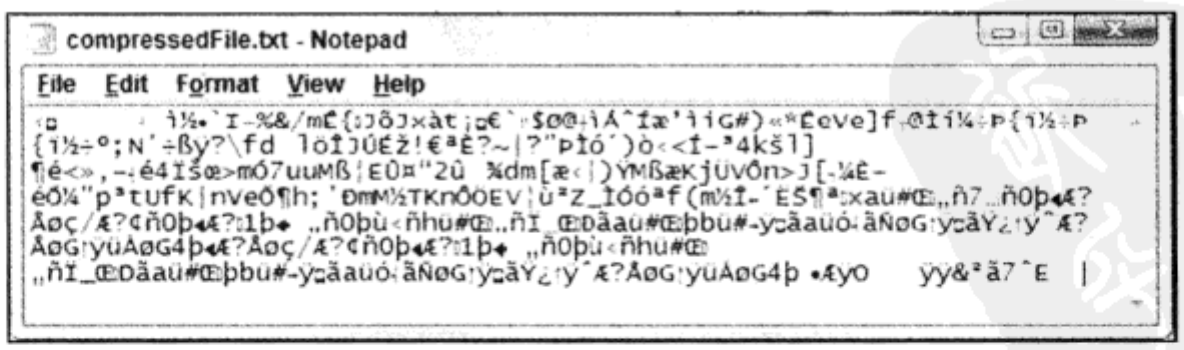


图　21-8

示例的说明

在这个示例中，定义了两个方法，用于保存和加载已压缩的文本文件。第一个方法是 SaveCompressedFile()，如下所示：

```
static void SaveCompressedFile(string filename, string data)
{
   FileStream fileStream =
      new FileStream(filename, FileMode.Create, FileAccess.Write);
   GZipStream compressionStream =
      new GZipStream(fileStream, CompressionMode.Compress);
   StreamWriter writer = new StreamWriter(compressionStream);
   writer.Write(data);
   writer.Close();
}
```

代码首先创建一个 FileStream 对象，然后使用它创建一个 GZipStream 对象。注意，可以用 DeflateStream 替换这段代码中的所有 GZipStream——这两个类的工作方式相同。使用 CompressionMode.Compress 枚举值指定数据要进行压缩，然后使用 StreamWriter 把数据写入文件。

LoadCompressedFile()方法与 SaveCompressedFile()方法正好相对，它不是保存到文件名中，而是把压缩的文件加载到字符串中：

```
static string LoadCompressedFile(string filename)
{
   FileStream fileStream =
      new FileStream(filename, FileMode.Open, FileAccess.Read);
   GZipStream compressionStream =
      new GZipStream(fileStream, CompressionMode.Decompress);
   StreamReader reader = new StreamReader(compressionStream);
   string data = reader.ReadToEnd();
   reader.Close();
   return data;
}
```

其区别很显然：使用了不同的 FileMode、FileAccess 和 CompressionMode 枚举值来加载和解压数据，使用 StreamReader 从文件中提取出未压缩的文本。

Main()中的代码是这些方法的一个简单测试。它请求一个字符串，把字符串复制 100 次，再把它压缩到一个文件中，之后检索它。在本示例中，Sir Gawain 和 Green Knight 的开头部分重复 100 次就有 17 800 个字符，但压缩后只占用 446 个字节，压缩率是 40:1。应该承认，这有以偏盖全之嫌，GZIP 算法很适合重复数据，但这里只是演示压缩过程而已。

我们还查看了存储在压缩文件中的文本。显然，它是不可读的。这说明可以在应用程序之间共享数据。因为是用已知的算法压缩文件，应用程序肯定知道如何解压缩。

21.3 序列化对象

应用程序常常需要在硬盘上存储数据。本章前面介绍了逐段构建文本和数据文件，但这常常不是最简便的方式。有时最好以对象形式存储数据。

.NET Framework 在 System.Runtime.Serialization 和 System.Runtime.Serialization.Formatters 名称

空间中提供了序列化对象的基础架构，这两个名称空间中的一些类实现了这个基础架构。Framework 中有两个可用的实现方式：

- System.Runtime.Serialization.Formatters.Binary：这个名称空间包含了 BinaryFormatter 类，它能把对象序列化为二进制数据，把二进制数据序列化为对象。
- System.Runtime.Serialization.Formatters.Soap：这个名称空间包含了 SoapFormatter 类，它能把对象序列化为 SOAP 格式的 XML 数据，把 SOAP 格式的 XML 数据序列化为对象。

本章只介绍 BinaryFormatter，因为还没有学习 XML 数据。实际上不鼓励使用 SoapFormatter 格式化器，但希望进行可读的序列化时，仍可以使用它。这两个类都实现了 IFormatter 接口，这里讨论的很多内容适用于这两个类。

.NET Framework 中的另外两个类也实现了 IFormatter 接口，第一个是 ObjectStateFormatter，用于在 ASP.NET 中序列化 viewstate，另一个是 NetDataContractSerializer，用于序列化 WCF 数据合同。

IFormatter 接口提供了如表 21-9 中所示的方法。

表 21-9

方　法	说　明
void Serialize(Stream stream, object source)	把 source 序列化为 stream
object Deserialize(Stream stream)	反序列化 stream 中的数据，返回得到的对象

重要的是，为了便于本章的讨论，这些方法都处理流，以便把这些方法与本章前面介绍的文件访问技术联系起来——可以使用 FileStream 对象。

所以，使用 BinaryFormatter 进行序列化非常简单：

```
IFormatter serializer = new BinaryFormatter();
serializer.Serialize(myStream, myObject);
```

反序列化同样也很简单：

```
IFormatter serializer = new BinaryFormatter();
MyObjectType myNewObject = serializer.Deserialize(myStream) as MyObjectType;
```

显然，需要流和对象才能进行处理，但上面的代码对于几乎所有的情况都是正确的。下面的示例将把这些代码用于实际。

试一试：对象的序列化和反序列化

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 ObjectStore。

(2) 给项目添加一个新类 Product，修改代码，如下所示：

可从 wrox.com 下载源代码

```
namespace ObjectStore
{
   public class Product
```

```
    {
        public long Id;
        public string Name;
        public double Price;

        [NonSerialized]
        string Notes;

        public Product(long id, string name, double price, string notes)
        {
            Id = id;
            Name = name;
            Price = price;
            Notes = notes;
        }

        public override string ToString()
        {
            return string.Format("{0}: {1} (${2:F2}) {3}", Id, Name, Price, Notes);
        }
    }
}
```

代码段 ObjectStore\Product.cs

(3) 把下面的代码放在 Program.cs 的顶部。需要导入 System.IO 名称空间，才能处理文件，导入其他名称空间，才能进行序列化：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```

代码段 ObjectStore\Program.cs

(4) 把下面的代码添加到 Program.cs 的 Main()方法中：

```
static void Main(string[] args)
{
    try
    {
        // Create products.
        List<Product> products = new List<Product>();
        products.Add(new Product(1, "Spiky Pung", 1000.0, "Good stuff."));
        products.Add(new Product(2, "Gloop Galloop Soup", 25.0, "Tasty."));
        products.Add(new Product(4, "Hat Sauce", 12.0, "One for the kids."));

        Console.WriteLine("Products to save:");
        foreach (Product product in products)
        {
            Console.WriteLine(product);
```

```
    }
    Console.WriteLine();

    // Get serializer.
    IFormatter serializer = new BinaryFormatter();

    // Serialize products.
    FileStream saveFile =
      new FileStream("Products.bin", FileMode.Create, FileAccess.Write);
    serializer.Serialize(saveFile, products);
    saveFile.Close();

    // Deserialize products.
    FileStream loadFile =
      new FileStream("Products.bin", FileMode.Open, FileAccess.Read);
    List<Product> savedProducts =
       serializer.Deserialize(loadFile) as List<Product>;
    loadFile.Close();

    Console.WriteLine("Products loaded:");
    foreach (Product product in savedProducts)
    {
       Console.WriteLine(product);
    }
}
catch (SerializationException e)
{
    Console.WriteLine("A serialization exception has been thrown!");
    Console.WriteLine(e.Message);
}
catch (IOException e)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString());
}

Console.ReadKey();
}
```

(5) 运行应用程序，结果如图 21-9 所示。

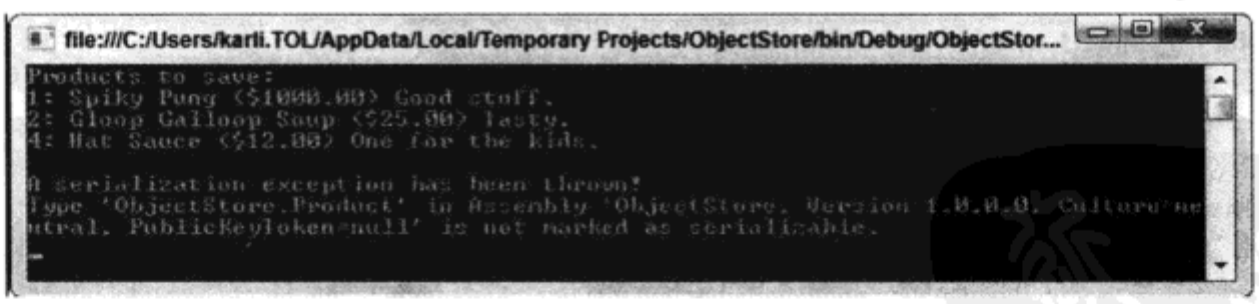


图　21-9

(6) 修改 Product.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
namespace ObjectStore
{
    [Serializable]
    public class Product
```

```
{
    ...
}
```

代码段 ObjectStore\Product.cs

(7) 再次运行应用程序，结果如图 21-10 所示。

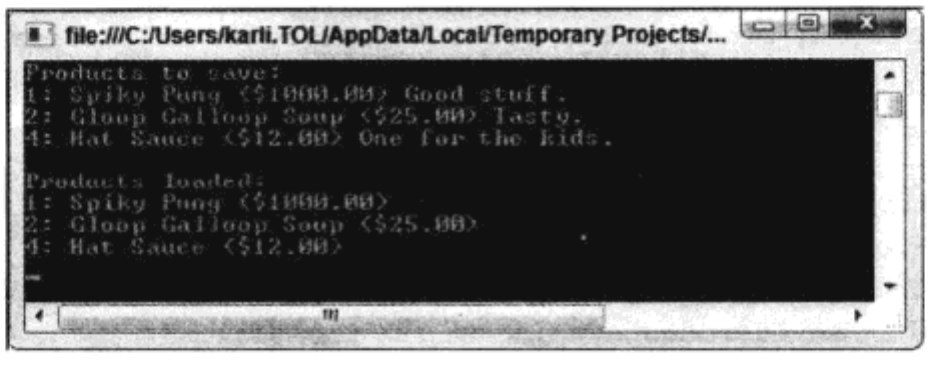


图 21-10

(8) 在记事本中打开 Products.bin，文本如图 21-11 所示。

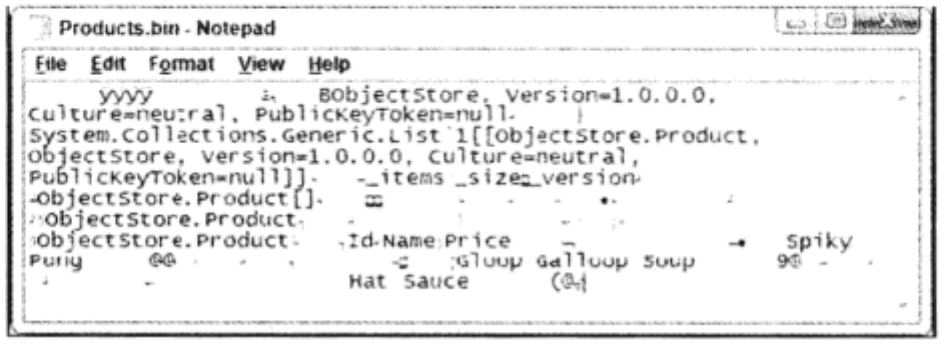


图 21-11

示例的说明

本示例创建了一个 Product 对象集合，把集合保存到磁盘上，然后重新加载它。但第一次运行应用程序时，抛出了一个异常，因为 Product 对象没有标记为“可序列化”。

.NET Framework 要求把对象标记为可序列化，才能序列化它们。这有许多原因，包括：

- 一些对象序列化的效果不佳。例如，它们需要引用只有它们本身位于内存中时才存在的本地数据。
- 一些对象包含敏感的数据，这些数据不应以不安全的方式保存或传送到另一个进程中。

如本例所示，使用 Serializable 特性就可以把对象标记为可序列化：

```
namespace ObjectStore
{
    [Serializable]
    public class Product
    {
        ...
    }
```

这里要注意，这个特性并没有由派生类继承，它必须应用于要进行序列化的每个类。另外，用于生成 Product 对象集合的 List<T>类有这个特性，否则，把它应用于 Product 就无益于使集合可序列化。

products 集合成功序列化和反序列化(在第二次尝试中)时，要注意另一个重要的问题。只重新构

建了 Id、Name 和 Price 字段，这是因为使用了另一个特性 NonSerialized。

```
[NonSerialized]
string Notes;
```

任何成员都可以用这个特性标记，但不能和其他成员一起存储。如果只有一个字段或属性包含敏感数据，这就是很有用的。

示例中还介绍了最后保存的数据。注意这里的一些数据是人们可以看懂的，这可能与我们期望的不同。BinaryFormatter 类并没有试图把数据隐藏起来。当然，这里使用了流，在把它保存到磁盘上，或者加载并应用自己的加密算法时，很容易截取其中的数据。压缩也是这样——使用上一节中的技术，很容易在把对象数据保存到磁盘上时压缩它们。

序列化涉及许多内容，但前面介绍的基础知识已经足够了。下面要研究的一个较高级技术是使用 ISerializable 接口的定制序列化，它允许定制序列化的数据。例如，在发布后升级类时，这是很重要的。修改可序列化的成员，会使所保存的已有数据不再可读，除非提供自己的逻辑来保存和获取数据。

21.4　监控文件系统

有时，应用程序所需要完成的工作不仅仅限于从文件系统中读写文件。例如，知道修改文件或目录的时间非常重要。.NET Framework 允许方便地创建完成这些任务的定制应用程序。

帮助完成这些任务的类是 FileSystemWatcher。这个类提供了几个应用程序可以捕获的事件。应用程序可以对文件系统事件作出响应。

使用 FileSystemWatcher 的基本过程非常简单。首先必须设置一些属性，指定监控的位置、内容以及引发应用程序要处理的事件的时间。然后给 FileSystemWatcher 提供定制事件处理程序的地址，当发生重要事件时，FileSystemWatcher 就可以进行调用。最后打开 FileSystemWatcher，等待事件。

在启用 FileSystemWatcher 对象之前必须设置的属性如表 21-10 所示。

表　21-10

属　性	说　明
Path	设置要监控的文件位置或目录
NotifyFilter	这是 NotifyFilters 枚举值的组合，NotifyFilters 枚举值规定在被监控的文件内要监控哪些内容。这些表示要监控的文件或文件夹的属性。如果规定的属性发生了变化，就引发事件。可能的枚举值是 Attributes、CreationTime、DirectoryName、FileName、LastAccess、LastWrite、Security 和 Size。注意，可以通过二元 OR 运算符来合并这些枚举值
Filter	指定要监控哪些文件的过滤器，例如，*.txt

设置之后，就必须为 4 个事件 Changed、Created、Deleted 和 Renamed 编写事件处理程序。如第 13 章所述，这需要创建自己的方法，并将方法赋给对象的事件。将自己的事件处理程序赋给这些方法，就可以在引发事件时调用方法。当修改与 Path、NotifyFilter 和 Filter 属性匹配的文件或目录

时，就引发每个事件。

在设置了属性和事件之后，将 EnableRaisingEvents 属性设置为 true，就可以开始监控工作。下面的示例将在一个简单的客户应用程序中使用 FileSystemWatcher，来监控所选目录。

试一试：监控文件系统

这是一个较复杂的示例，使用了本章介绍的许多内容。

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的 Windows 应用程序 FileWatch。

(2) 使用表 21-11 中所示的属性设置各个窗体属性。

表 21-11

属 性	设 置
FormBorderStyle	FixedDialog
MaximizeBox	False
MinimizeBox	False
Size	302, 160
StartPosition	CenterScreen
Text	File Monitor

(3) 使用表 21-12 中所示的属性，在窗体中添加所需的控件，并设置正确的属性。

表 21-12

控 件	名 称	位 置	大 小	文 本
TextBox	txtLocation	8，26	184，20	
Button	cmdBrowse	208，24	64，24	Browse...
Button	cmdWatch	88，56	80，32	Watch!
Label	lblWatch	8，104	0，13	

把 cmdWatch 按钮的 Enabled 属性设置为 false，因为不能在指定文件前就监控它，lblWatch 的 AutoSize 属性设置为 true，以查看其内容。再在窗体上添加一个 OpenFileDialog 控件，将其 Name 设置为 FileDialog，Filter 设置为 All Files|*.*。结束时，窗体应如图 21-12 所示。

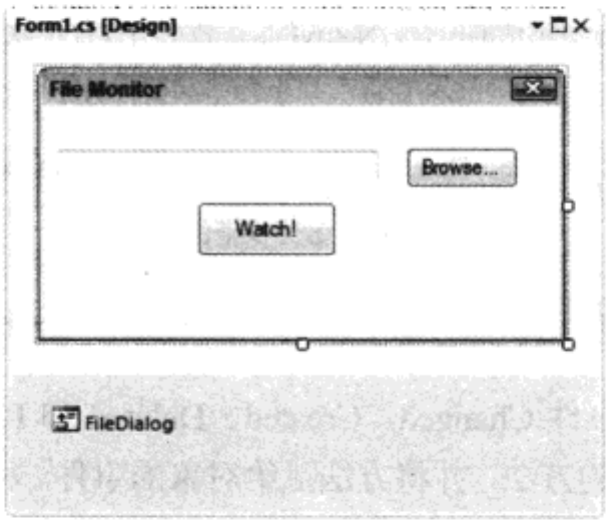


图 21-12

(4) 下面添加一些代码，使窗体可以正常使用。首先需要使用 using 指令将 System.IO 名称空间添加到已有的 using 指令列表中：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

代码段 FileWatch\Form1.cs

(5) 在 Form1 类中添加 FileSystemWatcher 类和一个委托，以便在不同的线程中修改 lblWatch 的文本。为此，在 Form1.cs 中添加下面的代码：

```
namespace FileWatch
{
  partial class Form1 : Form
  {
    //File System Watcher object.
    private FileSystemWatcher watcher;
    private delegate void UpdateWatchTextDelegate(string newText);
```

(6) 在窗体构造函数中调用 InitializeComponent()方法之后要添加如下代码。这段代码用于初始化 FileSystemWatcher 对象，把事件关联到后面要创建的方法上：

```
public Form1()
{
  InitializeComponent();

  this.watcher = new FileSystemWatcher();
  this.watcher.Deleted +=
    new FileSystemEventHandler(this.OnDelete);
  this.watcher.Renamed +=
    new RenamedEventHandler(this.OnRenamed);
  this.watcher.Changed +=
    new FileSystemEventHandler(this.OnChanged);
  this.watcher.Created +=
    new FileSystemEventHandler(this.OnCreate);
}
```

(7) 在 Form1 类中添加下面 5 个方法。第一个方法用于在运行 FileSystemWatcher 事件处理程序的线程中异步更新 lblWatch 中的文本，其他方法是事件处理程序。

```
// Utility method to update watch text.
public void UpdateWatchText(string newText)
{
  lblWatch.Text = newText;
}
```

```
// Define the event handlers.
public void OnChanged(object source, FileSystemEventArgs e)
{
   try
   {
      StreamWriter sw =
         new StreamWriter("C:\FileLogs\Log.txt", true);
      sw.WriteLine("File: {0} {1}", e.FullPath,
                   e.ChangeType.ToString());
      sw.Close();
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Wrote change event to log");
   }
   catch (IOException)
   {
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Error Writing to log");
   }
}
public void OnRenamed(object source, RenamedEventArgs e)
{
   try
   {
      StreamWriter sw =
         new StreamWriter("C:\FileLogs\Log.txt", true);
      sw.WriteLine("File renamed from {0} to {1}", e.OldName,
                   e.FullPath);
      sw.Close();
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Wrote renamed event to log");
   }
   catch (IOException)
   {
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Error Writing to log");
   }
}
public void OnDelete(object source, FileSystemEventArgs e)
{
   try
   {
      StreamWriter sw =
         new StreamWriter("C:\FileLogs\Log.txt", true);
      sw.WriteLine("File: {0} Deleted", e.FullPath);
      sw.Close();
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Wrote delete event to log");
   }
   catch (IOException)
   {
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Error Writing to log");
   }
}
```

```
public void OnCreate(object source, FileSystemEventArgs e)
{
   try
   {
      StreamWriter sw =
         new StreamWriter("C:\FileLogs\Log.txt", true);
      sw.WriteLine("File: {0} Created", e.FullPath);
      sw.Close();
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Wrote create event to log");
   }
   catch (IOException)
   {
      this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
         "Error Writing to log");
   }
}
```

(8) 现在为 Browse 按钮添加 Click 事件处理程序。这个事件处理程序中的代码会打开 Open File 对话框，允许用户选择要监控的文件。双击 Browse 按钮，输入如下代码：

```
private void cmdBrowse_Click(object sender, EventArgs e)
{
   if (FileDialog.ShowDialog() != DialogResult.Cancel )
   {
      txtLocation.Text = FileDialog.FileName;
      cmdWatch.Enabled = true;
   }
}
```

ShowDialog()方法返回表示用户如何退出 File Open 对话框的 DialogResult 枚举值(用户可以单击 OK 按钮或 Cancel 按钮)。我们需要检查是否用户未单击 Cancel 按钮，因此在将用户选择的文件保存到 TextBox 之前，将方法调用的结果与 DialogResult.Cancel 枚举值作比较。最后，将 Watch 按钮的 Enabled 属性设置为 true，以便监控文件。

(9) 双击 Watch 按钮。给 Click 事件处理程序添加下面的代码，以便启动 FileSystemWatcher：

```
private void cmdWatch_Click(object sender, EventArgs e)
{
   watcher.Path =Path.GetDirectoryName(txtLocation.Text);
   watcher.Filter = Path.GetFileName(txtLocation.Text);
   watcher.NotifyFilter = NotifyFilters.LastWrite |
      NotifyFilters.FileName | NotifyFilters.Size;
   lblWatch.Text = "Watching " + txtLocation.Text;
   // Begin watching.
   watcher.EnableRaisingEvents = true;
}
```

(10) 还需要确保 FileLogs 目录是存在的，以便在其中写入数据。在 Form1 构造函数中添加下面的代码，检查该目录是否存在；如果不存在，则创建它。

```
public Form1()
{
   ...
```

```
        DirectoryInfo aDir = new DirectoryInfo(@"C:\\FileLogs");
        if (!aDir.Exists)
          aDir.Create();
    }
```

(11) 创建目录 C:\TempWatch，在该目录中创建文件 temp.txt。

(12) 运行应用程序。如果成功地构建了所有内容，则单击 Browse 按钮，并选择 C:\TempWatch\temp.txt。

(13) 单击 Watch 按钮，开始监控文件。在应用程序中可以见到的唯一变化是，标签控件显示正在监控文件。

(14) 使用 Windows Explorer 导航到 C:\TempWatch。在记事本中打开 temp.txt，并在文件中添加一些文本。保存此文件。

(15) 重命名该文件。

(16) 现在可以检查日志文件，以查看其变化。导航到 C:\FileLogs\Log.txt 文件，在记事本中打开它。可以看到对所监控文件的变动说明，如图 21-13 所示。

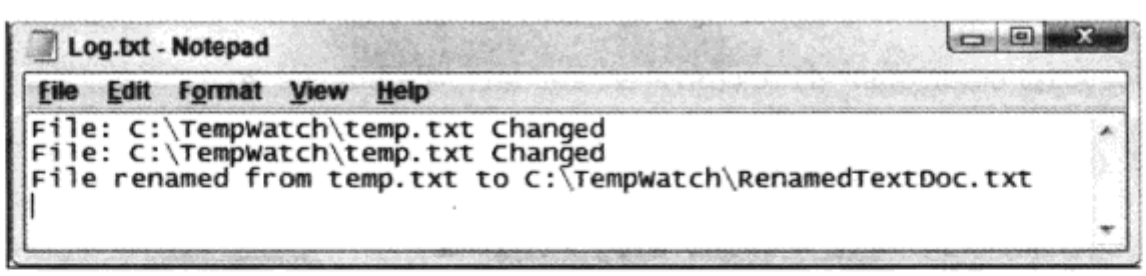


图 21-13

示例的说明

此应用程序非常简单，但它演示了 FileSystemWatcher 的工作原理。尝试在监控文本框中输入不同的字符串。如果在目录中指定*.*，程序就会监控目录中的所有变化。

应用程序中的大多数代码都用于建立 FileSystemWatcher 对象，以监控正确的位置：

```
watcher.Path =Path.GetDirectoryName(txtLocation.Text);
watcher.Filter = Path.GetFileName(txtLocation.Text);
watcher.NotifyFilter = NotifyFilters.LastWrite |
  NotifyFilters.FileName | NotifyFilters.Size;
lblWatch.Text = "Watching " + txtLocation.Text;
// Begin watching.
watcher.EnableRaisingEvents = true;
```

代码首先设置要监控的目录的路径。这使用了尚未介绍的一个新对象 System.IO.Path。这是一个静态类，非常类似于静态 File 对象。它给出了许多静态方法，以处理和提取文件位置字符串中的信息。这里首先使用它通过 GetDirectoryName()方法提取用户在文本框中键入的目录名称。

下一行代码设置对象的过滤器，过滤器可以是一个实际文件，表示仅监控该文件。过滤器也可以是*.txt，表示要监控指定目录中的所有.txt 文件。我们也可以使用 Path 静态对象从所提供的文件位置中提取信息。

NotifyFilter 是 NotifyFilters 枚举值的组合，规定组成变化的内容。在此，如果最后写入的时间信息、文件名称或文件大小发生了变化，它就将此变化通知应用程序。在更新了 UI 之后，将

EnableRaisingEvents 属性设置为 true，开始监控。

但在此之前，还要创建对象，设置事件处理程序。

```
this.watcher = new FileSystemWatcher();
this.watcher.Deleted+=
   new FileSystemEventHandler(this.OnDelete);
this.watcher.Renamed +=
   new RenamedEventHandler(this.OnRenamed);
this.watcher.Changed+=
   new FileSystemEventHandler(this.OnChanged);
this.watcher.Created+=
   new FileSystemEventHandler(this.OnCreate);
```

这就是使用前面创建的私有方法为监控器对象挂接事件处理程序的方式。当删除、重命名、修改或创建文件时，监控器对象就触发事件，调用事件处理程序。用户在自己的方法中可以决定如何处理实际发生的事件。注意，在事件发生之后我们才会得到通知。

在实际的事件处理方法中，只是将事件写入日志文件。显然，根据应用程序的不同，还可以有更复杂的响应。在目录中添加文件时，可以将其移动到别处，或读取其内容，引发新的进程。其可能的用法是无穷无尽的！

21.5　小结

本章学习了流和在.NET Framework 中使用流访问文件和其他序列化设备的原因。我们介绍了 System.IO 名称空间中的基类，包括：

- File
- FileInfo
- FileStream

File 类提供了许多静态方法，用于移动、复制和删除文件，FileInfo 表示磁盘上的一个物理文件，其方法可以处理该文件。FileStream 对象表示只读、只写或读写的文件。我们还介绍了 StreamReader 和 StreamWriter 类，以及它们在写入流时的作用。学习了使用 FileStream 类读写随机文件的方法。在此基础上，使用 System.IO.Compression 名称空间中的类在把流写入磁盘时压缩流，以及把对象序列化到文件中。最后构建了一个完整的应用程序，使用 FileSystemWatcher 类监控文件和目录。

概括地讲，本章学习了：

- 打开文件，读取文件，写入文件
- StreamWriter 和 StreamReader 类与 FileStream 类的区别
- 使用带分隔符的文件填充数据结构
- 压缩和解压缩流
- 序列化和反序列化对象
- 使用 FileSystemWatcher 类监控文件系统

21.6 练习

(1) 只有导入哪个名称空间才允许应用程序使用文件?

(2) 何时使用 FileStream 对象，而不是使用 StreamWriter 对象写入文件?

(3) StreamReader 类的哪些方法允许从文件中读取数据，每个方法的具体作用是什么?

(4) 哪个类可使用 Deflate 算法压缩流?

(5) 如何阻止所创建的类序列化?

(6) FileSystemWatcher 类提供了哪些事件，其作用是什么?

(7) 修改本章构建的 FileWatch 应用程序。无需退出应用程序就可以打开和关闭文件系统监控功能。

附录 A 给出了练习答案。

21.7 本章要点

主 题	重 要 概 念
流	流是序列化设备的一种抽象表示，可以一次从序列化设备中读取或写入一个字节。文件就是这种设备的一个例子。流有两种类型：输入和输出，分别用于读取和写入设备
文件访问类	.NET Framework 中有大量抽象了文件系统访问的类，包括通过静态方法处理文件和目录的 File 和 Directory，可实例化为表示特定文件和目录的 FileInfo 和 DirectoryInfo。后两个类在对文件和目录执行多个操作时使用，因为这两个类不要求为每个方法调用指定路径。可以在文件和目录上执行的典型操作包括查看和修改属性，创建、删除和复制操作
文件路径	文件和目录路径可以是绝对的或相对的。绝对路径给出了某位置的完整描述，从包含它的驱动器的根目录开始。所有的父目录都与子目录用反斜杠隔开。相对路径与之类似，但从文件系统的指定点开始，例如执行应用程序的目录(工作目录)。浏览文件系统时，常常使用父目录的别名..
FileStream 对象	FileStream 对象允许访问文件的内容，以进行读写。它以字节为单位访问文件数据，所以并不总是访问文件数据的最佳选项。FileStream 实例维护着文件内部的一个位置字节索引，这样就可以浏览文件的内容了。以这种方式访问文件的任意位置称为随机访问
读写流	一种读写文件数据的更简单方法是使用 StreamReader 和 StreamWriter 类，以及 FileStream。它们允许读写字符和字符串数据，而不是处理字节。这些类型提供了我们熟悉的处理字符串的方法，包括 ReadLine()和 WriteLine()。因为它们处理的是字符串数据，所以使用这些类，可以更加方便地处理逗号分隔的文件(这是表示结构化数据的常见方式)
压缩文件	可以使用 DeflateStream 和 GZipStream 压缩流类读写文件中的压缩数据。这些类与 FileStream 一样，也处理字节数据，但可以通过 StreamReader 和 StreamWriter 类访问数据，以简化代码
对象序列化	我们常常需要存储和检索表示对象状态的数据。可以使用序列化技术来自动保存和加载对象状态，而无需编写代码来保存和加载属性值。为此，必须使用 Serializable 特性把对象类型标记为可序列化。还可以使用其他属性控制如何序列化成员，例如 NonSerialized 特性禁止序列化给定的成员
监控文件系统	可以使用 FileSystemWatcher 类监控文件系统数据的变化。可以监控文件和目录，如果需要，还可以提供一个过滤器，根据需要仅修改有特定扩展名的文件。FileSystemWatcher 实例通过触发事件，来通知我们发生了变化，这些变化可以在代码中处理

第22章

XML

本章内容：

- 如何读写 XML
- 应用于格式良好的 XML 的规则
- 如何根据两种模式 XSD 和 XDR 验证 XML 文档
- 如何在应用程序中使用 XML
- 如何在程序中通过.NET 使用 XML
- 如何使用 XPath 查询搜索 XML 文档

Extensible Markup Language(可扩展标记语言，XML)近几年得到了广泛的关注。XML 并不是新技术，肯定不是 Microsoft 为了在.NET 环境中使用而开发的，但 Microsoft 在 XML 的开发初期就认识到了它的潜力，所以，XML 在.NET 中执行大量的任务，包括描述应用程序的配置、在 Web 服务之间传输信息等。

XML 是一种以简单文本格式存储数据的方式，这意味着它可以被任何计算机读取。在前面章节介绍 Web 编程技术时可以看到，XML 是在 Internet 上传输数据的绝佳格式。甚至对于人们来说，它也很容易理解！

从 VS 的第一版开始，Microsoft 就在开发使用 XML 的解决方案上投入了大量的精力。目前.NET 中的大多数应用程序都以某种形式使用 XML，例如存储配置细节的.config 文件，WPF 中使用的 XAML 文件等。甚至 Office 2007 引入的新文档格式都基于 XML，虽然 Office 应用程序本身并不是.NET 应用程序。

XML 的细节非常复杂，因此在此不介绍其所有细节。但其基本格式非常简单，大多数情况下，不需要了解 XML 的详细知识，因为 VS 通常会处理其中大多数工作 ——我们基本上不必手动编写 XML 文档。前面已经说过，XML 在.NET 领域非常重要，因为它是传输数据的默认格式，所以理解其基本知识至关重要。

22.1 XML 文档

XML 中的完整数据集就是 XML 文档。XML 文档可以是计算机上的物理文件，或是内存中的字符串。但是其本身必须是完整的，必须遵循某些规则(稍后介绍)。XML 文档由许多不同的部分组成。其中最重要的部分是 XML 元素，它包含文档的实际数据。

22.1.1 XML 元素

XML 元素包含一个开始标记(放在尖括号中的元素名称，如<myElement>)、元素中的数据和结束标记(与开始标记相同，但是在左括号后有一个斜线：</myElement>)。

例如，定义一个存储书名的元素：

```
<book>Tristram Shandy</book>
```

如果您了解一些 HTML 知识，就可能认为这非常类似——非常正确！事实上，HTML 和 XML 有许多相同的语法。最大的不同是，XML 没有任何预定义元素——我们选择自定义元素的名称，因此元素数量不受限制。最重要的是 XML——不管它使用什么样的名称——实际上不是语言，而是定义语言的标准(称为 XML 应用)。每种语言都有自己独特的词汇库——一组可以用于文档的特定元素和允许采用这些元素的结构。稍后会介绍，我们可以显式地限制 XML 文档中允许使用的元素。另外，也可以使用任何元素，允许使用文档的程序给出其结构。

元素名称区分大小写，因此<book>和<BOOK>是不同的元素。这表示，如果试图通过使用大小写不同的结束标记(例如</BOOK>)关闭<book>元素，XML 文档就是非法的。XML 分析程序(parser)读取 XML 文档，并分析其中各个元素，它们会拒绝任何包含非法 XML 的文档。

元素也可以包含其他元素，因此可以修改此<book>元素，添加两个子元素，使之包括作者和标题：

```
<book>
   <title>Tristram Shandy</title>
   <author>Lawrence Sterne</author>
</book>
```

但是元素不允许重叠，因此在父元素的结束标记之前必须关闭所有子元素。例如，不能编写如下代码：

```
<book>
   <title>Tristram Shandy
   <author>Lawrence Sterne
   </title></author>
</book>
```

这是非法的，因为在<title>元素内打开了<author>元素，但是结束标记</title>位于结束标记</author>之前。

所有元素必须有结束元素的规定有一个例外。可以有“空”元素，其中没有内嵌的数据或文本。在此，可以直接在开始元素之后添加结束标记，如下所示：

```
<book></book>
```

或者使用简短语法，在开始元素末尾添加结束元素的斜线：

```
<book />
```

22.1.2 特性

除了在元素体内存储数据之外，也可以在特性内存储数据，将特性添加到元素的开始标记内。特性的形式为：

```
name="value"
```

其中特性值必须包含在单引号或双引号内。例如：

```
<book title="Tristram Shandy"></book>
```

或者：

```
<book title='Tristram Shandy'></book>
```

这都是合法的，但是下面的语句不合法：

```
<book title=Tristram Shandy></book>
```

为什么在 XML 中需要两种方式来存储数据？下面二者的区别是什么：

```
<book>
   <title>Tristram Shandy</title>
</book>
```

和

```
<book title="Tristram Shandy"></book>
```

实际上，二者并没有太大的区别。使用其中任何一个都没有什么优势可言。如果以后需要为数据添加更多信息，最好选择使用元素 ——总是可以给元素添加子元素或特性，但是对特性就不能进行这样的操作。有争议的是，元素是否更易于读取，更简洁(这只能根据个人不同的爱好来决定)。另一方面，如果未经压缩就在网络上传输文档，则特性会占用更少的带宽(即使压缩了，区别也不大)，更便于保存对文档的每一位用户而言无关紧要的信息。也许最好的选择是同时使用二者，可以根据自己的爱好选择使用某种方式来存储特定的数据项。但是确实没有硬性规定。

22.1.3 XML 声明

除了元素和特性之外，XML 文档还可以包含许多组成部分。XML 文档的各个组成部分称为节点 ——因此元素、元素内的文本和特性都是 XML 文档的节点。如果需要深入研究 XML，这些都是非常重要的。但是只有一种类型的节点存在于几乎所有的 XML 文档中。这就是 XML 声明，如果包括了它，它就必须是文档的第一个节点。

XML 声明的格式类似于元素，但是在尖括号内有问号。它的名称始终都是 xml，并总是有 version 特性；当前，它只有两个值：1.0(第一版)和 1.1(第二版)，但 VS 不支持第二版。第二版并没有给 Windows 平台上使用的 XML 做什么改进，而且 World Wide Web Consortium (www.w3c.org)建议尽可能使用第一版。因此最简单的 XML 声明形式为：

```
<?xml version="1.0"?>
```

另外，它还可以包含特性 encoding(其值表示用于读取文档的字符集，比如 UTF-16 表示文档使用 16 位 Unicode 字符集)和 standalone(其值是 yes 或 no，表示 XML 文档是否依赖于其他文件)。但是这些特性并不是必需的，可以仅在 XML 文件中包括 version 特性。

22.1.4 XML 文档的结构

XML 最重要的一点就是它提供了一种组织数据的结构化方式，它非常不同于关系数据库。最新型的数据库系统在表中存储数据，表通过各列中的值而相互关联。每个表在行和列中存储数据——每一行代表一个记录，每一列代表该记录的特定数据项。相反，XML 数据是分层组织的，有点类似于 Windows Explorer 中的文件夹和文件。每个文档必须有一个根元素，其中包含所有元素和文本数据。如果在文档的顶级中有多个元素，该文档就是不合法的 XML 文档。但是可以在顶级包括其他 XML 节点——通常是 XML 声明。所以下面的 XML 文档是合法的：

```
<?xml version="1.0"?>
<books>
   <book>Tristram Shandy</book>
   <book>Moby Dick</book>
   <book>Ulysses</book>
</books>
```

但是下面的文档就不合法：

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

在此根元素之下，组织数据时具有极大的灵活性。在关系数据中，每一行都有相同的列数，而元素所包含的子元素数量不受限制。尽管 XML 文档经常以与关系数据类似的方式组织，每一个记录具有一个元素，但XML 文档不需要任何预定义的结构。这是传统关系数据库和 XML 的主要区别之一。关系数据库总是在添加数据之前定义信息的结构，而信息存储在 XML 中时不需要这个初始设置，所以 XML 是存储小块数据的非常便利的方式。稍后就会看到，可以为 XML 提供结构，但与关系数据库不同，这个结构不是强制的，除非明确要求有这个结构。

22.1.5 XML 名称空间

如第 9 章所述，每个人都可以定义自己的 C#类，也可以定义自己的 XML 元素；这就导致了一个问题 ——如何知道哪个元素属于哪个词汇表？注意本节的标题，可以按类似的方式回答这个问题。如同定义名称空间以组织 C#类型一样，可以使用 XML 名称空间定义 XML 词汇表。这就可以将不同词汇表中的元素包含到一个 XML 文档中，而没有因为(例如)两个不同的词汇表定义了一个<customer>元素而误解元素的风险。

XML 名称空间非常复杂，因此在此不详细介绍它，但其基本语法非常简单。使用前缀，后跟冒号，就可以将具体的元素或特性关联到特定的名称空间。例如，<wrox:book>表示 wrox 名称空间中的<book>元素。但是如何知道名称空间 wrox 所表示的内容呢？为了使此方法有效，必须保证每个名称空间都是唯一的。最简单的办法是将前缀映射到某些独特事物上。如下所示：在 XML 文档

的某个位置，需要将名称空间前缀关联到 Uniform Resource Identifier(唯一资源标识符，URI)上。URI 包含几种类型，最常见的类型是 Web 地址，如 www.wrox.com。

为了用具体的名称空间标识前缀，可以在元素内使用 xmlns:prefix 特性，将其值设置为标识名称空间的唯一 URI。然后前缀就可以用于该元素的任何位置，包括任何内嵌的子元素。例如：

```
<?xml version="1.0"?>
<books>
  <book xmlns:wrox="http://www.wrox.com">
    <wrox:title>Beginning C#</wrox:title>
    <wrox:author>Karli Watson</wrox:author>
  </book>
</books>
```

在此，<title>和<author>元素使用 wrox:前缀，因为它们位于<book>元素内，其中定义了该前缀。但是如果试图将此前缀添加到<books>元素上，XML 就是非法的，因为并没有为此元素定义前缀。

也可以使用 xmlns 特性为元素定义默认的名称空间：

```
<?xml version="1.0"?>
<books>
  <book xmlns="http://www.wrox.com">
    <title>Beginning Visual C#</title>
    <author>Karli Watson</author>
    <html:img alt="Cover Image" src="begcsharp.gif"
              xmlns:html="http://www.w3.org/1999/xhtml" />
  </book>
</books>
```

这里将<book>元素的默认名称空间定义为"http://www.wrox.com"。因此此元素内的所有内容都属于此名称空间，除非添加不同的名称空间前缀显式指定其他内容，比如对<img>元素所做的处理(将它设置到与 XML 兼容的 HTML 文档所使用的名称空间中)。

22.1.6 格式良好并有效的 XML

前面介绍的都是“合法”的 XML。事实上，有两种合法形式的 XML：格式良好和有效。遵循 XML 标准的所有规则的文档就是“格式良好(well-formed)”的 XML。如果 XML 文档的格式有误，分析程序就不能正确地解释它，并拒绝该文档。为了有一个格式良好的 XML，对文档的要求如下：

- 有且只有一个根元素
- 每一个元素都有结束标记(上面提到的简短语法除外)
- 没有重叠元素 ——所有子元素必须完全嵌套在父元素内
- 所有特性必须放在引号内

这些还不够全面，但强调了 XML 新手容易犯的一些错误。完全遵循这些规则的 XML 文档仍然可能是无效的。前面说过，XML 本身不是语言，而是定义 XML 应用程序的标准。格式良好的 XML 文档仅仅符合 XML 标准；为了使之有效，它们还必须符合对 XML 应用程序所规定的规则。并不是所有的分析程序都检查文档是否有效；进行这种检查的分析程序就是验证分析程序。但是为了检查一个文档是否符合应用程序的规则，首先需要一种指定规则的方式。

22.1.7 验证 XML 文档

XML 支持通过两种方法，来定义在文档中可以放置那些元素和特性，及其放置顺序——文档类型定义(Document Type Definitions，DTD)和模式。

1. DTD

DTD 使用从 XML 的父文档继承的非 XML 语法，并逐渐被模式(schema)所代替。DTD 不允许规定元素和属性的数据类型，因此不太灵活，在.NET Framework 的环境中用得不多。另一方面，模式使用得非常多——它们允许规定数据类型，是用 XML 兼容的语法编写的。但是，模式非常复杂，有不同的格式定义它们——即使在.NET 中也是如此！

2. 模式

.NET 支持的模式具有两种不同的格式——XML Schema Definition 语言(XSD)和 XML-Data Reduced 模式(XDR)。

XDR 模式

XDR 模式定义是一个旧标准，专用于 Microsoft，除了 Microsoft 分析器外，其他分析器通常不能识别它，也很少使用它。XSD 是一个开放标准，W3C 推荐这个模式，所以这里介绍其定义。模式可以包括在 XML 文档内，也可以放在单独文件中。在编写 XML 文档之前需要非常熟悉 XML。但是，在模式中识别主要元素时它非常有用，因此我们要解释一下基本原则。为此，介绍这个简单 XML 文档的示例 XSD 模式，它包含了两本 Wrox 的 C#图书的详情。这个 XML 在本书的下载代码中，其文件名是 book.xml：

```
<?xml version="1.0"?>
<books>
   <book>
      <title>Beginning Visual C# 2010</title>
      <author>Karli Watson</author>
      <code>7582</code>
   </book>
   <book>
      <title>Professional C# 2010</title>
      <author>Simon Robinson</author>
      <code>7043</code>
   </book>
</books>
```

Code snippet book.xml

XSD 模式

XSD 模式中的元素必须属于名称空间 http://www.w3.org/2001/XMLSchema。如果未包括此名称空间，就不能识别这些模式元素。

为了将 XML 文档与另一文件中的 XSD 模式关联，需要在根元素中添加 schemalocation 元素：

```
<?xml version="1.0"?>
```

```
<books schemalocation="file://C:\Beginning Visual C#\Chapter 22\books.xsd">
   .
</books>
```

下面分析一个 XSD 模式示例：

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="books">
      <complexType>
         <choice maxOccurs="unbounded">
            <element name="book">
               <complexType>
                  <sequence>
                     <element name="title" />
                     <element name="author" />
                     <element name="code" />
                  </sequence>
               </complexType>
            </element>
         </choice>
         <attribute name="schemalocation" />
      </complexType>
   </element>
</schema>
```

在此首先要注意，默认名称空间设置为 XSD 名称空间。这就告诉分析器，文档中的所有元素属于模式。如果不指定此名称空间，分析器就认为元素仅是普通的 XML 元素，认识不到需要使用它们进行验证。

完整的模式包含在<schema>元素内(使用小写 s——大小写非常重要！)。文档中的每个元素都必须由<element>元素来表示。此元素具有指示元素名称的 name 特性。如果元素将要包含嵌套的子元素，就必须在<complexType>元素内为这些子元素包含<element>标记。可以在其中指定子元素的操作方式。

例如，使用<choice>元素指定对子元素进行选择操作，或使用<sequence>规定子元素必须以它们在模式中列举的顺序出现。如果一个元素可能多次出现(如<book>元素)，就需要在其父元素内包括 maxOccurs 特性。将其设置为 unbounded，表示元素可以出现无限次。最后，特性必须由<attribute>元素表示，包括 schemalocation 特性，它告诉分析器模式所在的位置。它放在子元素列表的结尾处。

前面介绍了 XML 的基本理论，接下来的示例创建 XML 文档。VS 为我们完成了大部分繁重工作，甚至基于 XML 文档创建了 XSD 模式，我们无需编写任何代码！

试一试：在 VS 中创建 XML 文档

按照下面的步骤，创建 XML 文档。

(1) 打开 VS，从菜单中选择 File | New | File。如果没有看到这个选项，请创建一个新项目。在 Solution Explorer 中右击项目，选择添加一个新项。然后从对话框中选择 XML File。

(2) 在 New File 对话框中，选择 XML File，单击 Add。VS 会自动创建一个新的 XML 文档。如图 22-1 所示，VS 添加了 XML 声明，以 encoding 特性结束(其特性和元素也是彩色的，但是在黑白纸张中其效果不太好)。

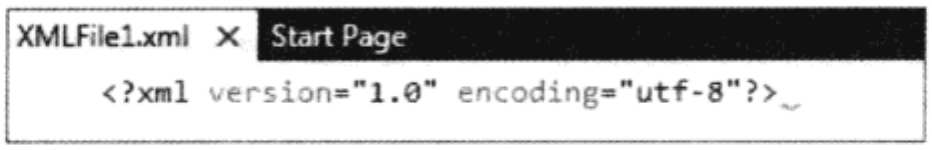


图 22-1

(3) 按下 Ctrl+S 组合键，或者在菜单中选择 File | Save XMLFile1.xml，保存文件。VS 会询问文件的保存位置，以及文件的名称。将其保存在 BegVCSharp\Chapter22 文件夹中，取名为 GhostStories.xml。

(4) 将光标移动到 XML 声明下面的代码行，键入文本<stories>。注意，输入大于号关闭开始标记时，VS 会自动置入结束标记。

(5) 输入下面的 XML 文件，单击 Save：

可从 wrox.com 下载源代码

```
<stories>
  <story>
    <title>A House in Aungier Street</title>
    <author>
      <name>Sheridan Le Fanu</name>
      <nationality>Irish</nationality>
    </author>
    <rating>eerie</rating>
  </story>
  <story>
    <title>The Signalman</title>
    <author>
      <name>Charles Dickens</name>
      <nationality>English</nationality>
    </author>
    <rating>atmospheric</rating>
  </story>
  <story>
    <title>The Turn of the Screw</title>
    <author>
      <name>Henry James</name>
      <nationality>American</nationality>
    </author>
    <rating>a bit dull</rating>
  </story>
</stories>
```

Code snippet Chapter22\GhostStories.xml

(6) 现在可以让 Visual Studio 为刚才编写的 XML 文件创建相应的模式。为此，从 XML 菜单中选择 Create Schema 菜单项，单击 Save as GhostStories.xsd，保存得到的 XSD 文件。

(7) 返回到 XML 文件，在结束标记</stories>之前键入如下 XML：

```
<story>
  <title>Number 13</title>
  <author>
    <name>M.R. James</name>
    <nationality>English</nationality>
```

```
    </author>
    <rating>mysterious</rating>
</story>
```

注意，开始键入开始标记时，会显示 IntelliSense 提示。这是因为 Visual Studio 知道把新建的 XSD 模式连接到正在键入的 XML 文件上。

(8) 可以在 Visual Studio 中创建 XML 和一个或多个模式之间的链接。选择 XML | Schemas，会打开如图 22-2 所示的对话框。在 Visual Studio 可识别的长长模式列表顶部，会看到 GhostStories.xsd。在它的左边是一个绿色的复选标记，表示这个模式用于当前的 XML 文档。

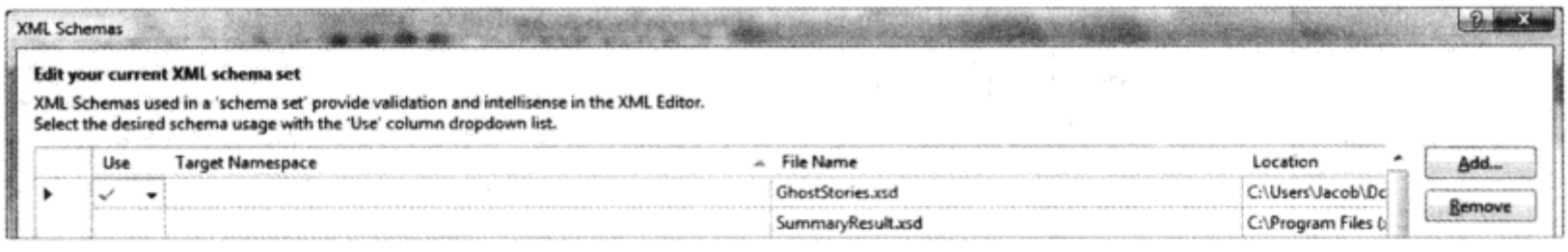


图　22-2

图 22-2 显示的 XSD 对话框包含 VS 可识别的一个很长的模式列表，但它不会自动存储以前使用的模式。如果重复使用一个模式，不希望每次在需要它时都指定它，就可以把它复制到如下位置 C:\Program Files\Microsoft Visual Studio 10.0\Xml\Schemas。复制到这个位置的所有模式都会显示在 Schemas 对话框中。

22.2　在应用程序中使用 XML

了解如何创建 XML 文档后，就应把这些知识用于实践。.NET Framework 提供了许多名称空间和类，使 XML 的读取、处理和写入非常简单。下面几节将介绍这些类，说明如何使用它们以编程方式创建和处理 XML。

22.2.1　XML 文档对象模型

XML 文档对象模型(Document Object Model，DOM)是一组以非常直观的方式访问和处理 XML 的类。DOM 不是读取 XML 数据的最快捷的方式，但只要理解了类和 XML 文档中元素之间的关系，DOM 就很容易使用。

构成 DOM 的类在名称空间 System.Xml 中。在这个名称空间中有几个类和子名称空间。但本章只介绍几个易于操作 XML 的类。要学习和使用的类如表 22-1 所示。

表　22-1

类　名	说　明
XmlNode	这个类表示文档树中的一个节点，它是本章许多类的基类。如果这个节点表示 XML 文档的根，就可以从它导航到文档的任意位置

(续表)

类 名	说 明
XmlDocument	扩展了 XmlNode 类，但常常是使用 XML 的第一个对象，因为这个类用于加载磁盘或其他地方的数据并在这些位置保存数据
XmlElement	表示 XML 文档中的一个元素。XmlElement 派生于 XmlLinkedNode，XmlLinkedNode 派生于 XmlNode
XmlAttribute	表示一个特性，与 XmlDocument 类一样，它也派生于 XmlNode 类
XmlText	表示开始标记和结束标记之间的文本
XmlComment	表示一种特殊类型的节点，这种节点不是文档的一部分，但为阅读器提供文档各部分的信息
XmlNodeList	表示一个节点集合

1. XmlDocument 类

通常，要处理 XML 的应用程序，首先应从磁盘中读取它。如表 22-1 中所示，这是 XmlDocument 类的工作。可以将 XmlDocument 看作磁盘上文件的内存表示。使用 XmlDocument 类把文件加载到内存中后，就可以从中获得文档的根节点，开始读取和处理 XML 了：

```
using System.Xml;
.
.
.

XmlDocument document = new XmlDocument();
document.Load(@"C:\Beginning Visual C#\Chapter 22\books.xml");
```

这两行代码创建了 XmlDocument 类的一个新实例，并在其中加载 books.xml 文件。Xml Document 类位于 System.Xml 名称空间中，所以应在代码开头的 using 部分插入 using System.Xml;语句。

除了加载和保存 XML 之外，XmlDocument 类还负责维护 XML 结构。所以，这个类有许多方法可以用于创建、修改和删除树中的节点。稍后将介绍其中一些方法，但为了正确理解这些方法，还需要了解另一个类：XmlElement。

2. XmlElement 类

文档加载到内存中后，就要对它执行一些操作。上面代码创建的 XmlDocument 实例的 DocumentElement 属性会返回一个 XmlElement 实例(表示 XmlDocument 的根节点)。这个元素非常重要，因为有了它，就可以访问文档中的所有信息。

```
XmlDocument document = new XmlDocument();
document.Load(@"C:\Beginning Visual C#\Chapter 22\books.xml");
XmlElement element = document.DocumentElement;
```

获得文档的根节点后，就可以使用信息了。XmlElement 类包含的方法和属性可以处理树的节点和特性。下面首先看看用于导航 XML 元素的属性，如表 22-2 所示。

表 22-2

属 性	说 明

FirstChild	这个属性返回当前节点之后的第一个子节点。在本章前面的 books.xml 文件中，文档的根节点是 books，根节点之后的节点是 book，在该文档中，根节点 books 的第一个子节点是 book。
	`<books>  @@la Root node`
	`  <book>  @@la Firstchild`
	nodeFirstChild 返回一个 XmlNode 对象，应测试返回节点的类型，因为它不总是一个 XmlElement 实例。在 books 示例中，Title 元素的子元素是表示文本 Beginning Visual C# 的 XmlText 节点

LastChild	这个属性的操作与 FirstChild 属性十分类似，但返回当前节点的最后一个子节点。在 books 示例中，books 节点的最后一个子节点仍是 book，但它表示 Professional C# 2010。
	`<books> @@la Root node`
	`  <book> @@la FirstChild`
	`  <title>Beginning Visual C# 2010</title>`
	`  <author>Karli Watson</author>`
	`  <code>7582</code>`
	`  </book>`
	`  <book> @@la LastChild`
	`  <title>Professional C# 2010</title>`
	`  <author>Simon Robinson</author>`
	`  <code>7043</code>`
	`  </book>`
	`</books>`

ParentNode	这个属性返回当前节点的父节点。在 books 示例中，books 节点是 book 节点的父节点
NextSibling	FirstChild 和 LastChild 属性返回当前节点的叶子节点，而 NextSibling 节点返回有相同父节点的下一个节点。在 books 示例中，title 元素的 NextSibling 属性返回 author 元素，在 author 元素上调用 NextSibling，会返回 code 元素
HasChildNodes	检查当前元素是否有子元素，而无需获取 FirstChild 的值并检查它是否为 null

使用表 22-2 中的 4 个属性，可以遍历整个 XmlDocument，如下面的示例所示。

试一试：迭代 XML 文档中的所有节点

在这个示例中，要创建一个小型 Windows Forms 应用程序，迭代 XML 文档中的所有节点，打印出元素的名称，如果是 XmlText 元素，就打印出包含在元素中的文本。这段代码使用了 book.xml，如前面的“模式”一节所述。如果没有创建这个文件，可在本书的下载代码中找到它。

(1) 首先创建一个新的 Windows 窗体项目。选择 File | New | Project 菜单项，在打开的对话框中，选择 Windows | Windows Forms Application。将项目命名为 LoopThroughXmlDocument，按下回车键。

(2) 将一个 TextBox 和一个 Button 控件拖放到窗体上，按照图 22-3 所示设计窗体。

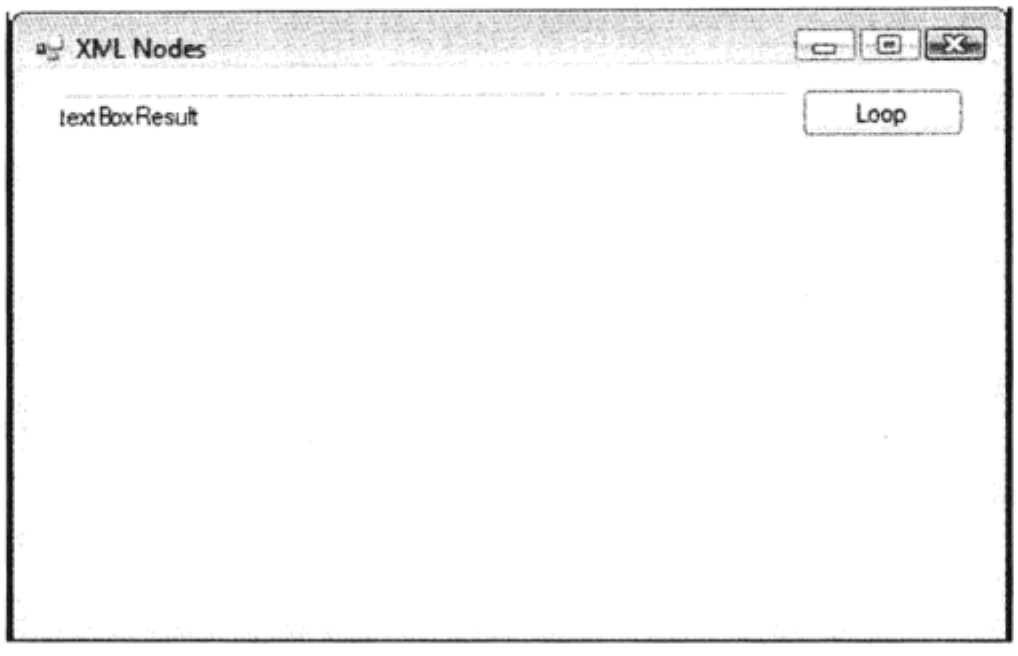


图 22-3

(3) 将文本框命名为 testBoxResult，按钮命名为 buttonLoopThroughDocument。把文本框的 Multiline 属性设置为 true，Scrollbars 属性设置为 Vertical。

(4) 双击按钮，输入下面的代码。注意，要在文件顶部的 using 部分添加 using System.Xml;：

```
private void buttonLoopThroughDocument_Click(object sender, EventArgs e)
{
  XmlDocument document = new XmlDocument();
  document.Load(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");
  textBoxResult.Text = FormatText(document.DocumentElement as XmlNode, "", "");
}

private string FormatText(XmlNode node, string text, string indent)
{
  if (node is XmlText)
  {
    text += node.Value;
    return text;
  }

  if (string.IsNullOrEmpty(indent))
    indent = "";
  else
  {
    text += "\r\n" + indent;
  }

  if (node is XmlComment)
  {
    text += node.OuterXml;
    return text;
  }

  text += "<" + node.Name;
  if (node.Attributes.Count > 0)
  {
    AddAttributes(node, ref text);
  }
```

```
    if (node.HasChildNodes)
    {
      text += ">";
      foreach (XmlNode child in node.ChildNodes)
      {
        text = FormatText(child, text, indent + " ");
      }
      if (node.ChildNodes.Count == 1 &&
          (node.FirstChild is XmlText || node.FirstChild is XmlComment))
        text += "</" + node.Name + ">";
      else
        text += "\r\n" + indent + "</" + node.Name + ">";
    }
    else
      text += " />";
    return text;
  }

  private void AddAttributes(XmlNode node, ref string text)
  {
    foreach (XmlAttribute xa in node.Attributes)
    {
      text += " " + xa.Name + "='" + xa.Value + "'";
    }
  }
```

Code snippet Chapter22\LoopThroughXmlDocument\Form1.cs

(5) 运行该应用程序，单击 Loop 按钮。结果如图 22-4 所示。

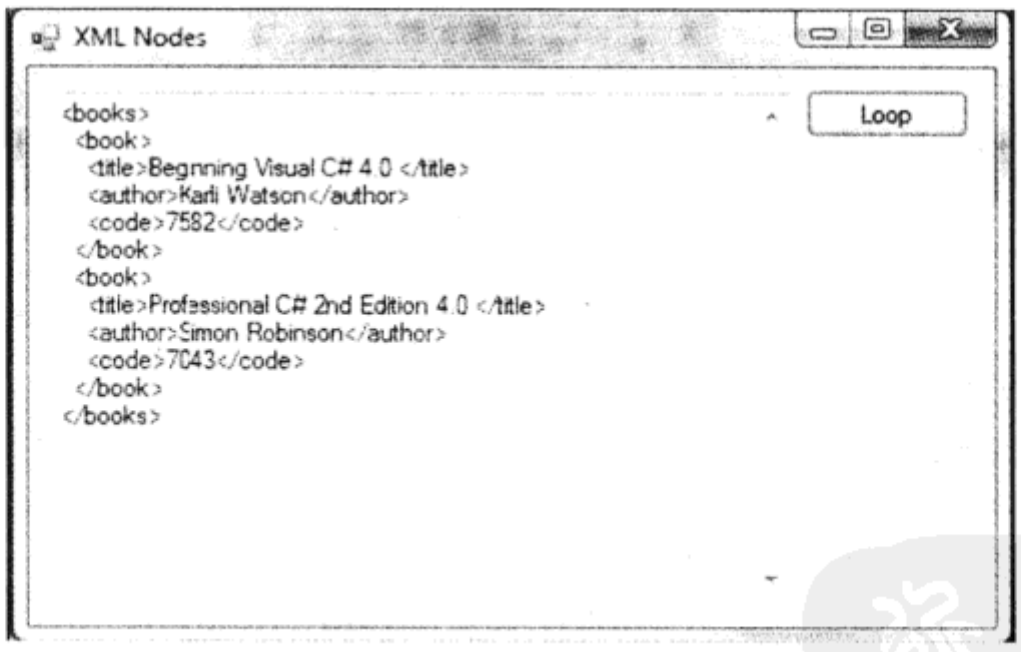


图 22-4

示例的说明

单击按钮时，首先会调用 XmlDocument 方法 Load。这个方法把文件中的 XML 加载到 XmlDocument 实例中，XmlDocument 实例用于访问 XML 的元素。接着调用一个方法来递归迭代 XML，并把 XML 文档的根节点传送给方法。根元素是使用 XmlDocument 类的属性 DocumentElement 获得的。除了在传送给 RecurseXmlDocument 方法的根参数上检查 null 之外，还要注意 if 语句：

```
if (node is XmlText)
{
    .
}
```

is 运算符可以检查对象的类型，如果实例是指定的类型，就返回 true。即使根节点声明为 Xmlnode，这也只是要操作的对象的基本类型。使用 is 运算符除了可以测试对象的类型之外，还可以在运行期间确定对象类型，并根据该类型选择要执行的操作。

在 FormatText 方法中给文本框生成文本。注意必须知道根节点的当前实例的类型，因为要显示的信息的获取方式对于不同的元素来说是不同的。我们要显示 XmlElements 的名称和 XmlText 元素的值。

3. 修改节点的值

在了解如何改变节点值之前，先要明白，节点值一般比较复杂。实际上，即使派生于 XmlNode 的所有类都包含 Value 属性，它也很少返回有用的信息。初看起来它可能令人失望，但实际上是十分合理的。看看前面的 books 示例：

```
<books>
  <book>
    <title>Beginning Visual C# 4.0</title>
    <author>Karli Watson</author>
    <code>7582</code>
  </book>
  <book>
</books>
```

文档中的每对标记都解析为 DOM 中的一个节点。在迭代文档中的所有节点时，会遇到许多 XmlElement 节点和三个 XmlText 节点。上述 XML 中的 XmlElement 节点是<books>、<book>、<title>、<author>和<code>。XmlText 节点是 title、author 和 code 开始标记和结束标记之间的文本。title、author 和 code 的值是标记之间的文本，但文本本身就是一个节点，节点包含了值。其他标记都没有相关的值。

在上述 FormatText 方法的代码靠近顶部的位置，if 块中的下述代码在当前节点是 XmlText 时执行：

```
text += node.Value;
```

XmlText 节点实例的 Value 属性用于获取节点的值。

如果使用 XmlElement 类型的节点的 Value 属性，就返回 null，但如果使用另外两个方法 InnerText 和 InnerXml 中的一个，就可以获取 XmlElement 开标记和闭标记之间的信息。也就是说，可以使用两个方法和一个属性来操作节点的值，如表 22-3 所示。

表 22-3

属 性	说 明
InnerText	这个属性获取当前节点中所有子节点的文本，把它作为一个串联字符串返回。也就是说，在上面的 XML 中，如果获取 book 节点的 InnerText 值，就返回字符串 Beginning Visual C 2010#Karli Watson7582。如果获取 title 节点的 InnerText，就只返回 Beginnning Visual C# 2010。 可以使用这个方法设置文本，但要小心，因为如果设置了错误节点的文本，就很可能改写不想改变的信息

(续表)

属　性	说　明
InnerXml	InnerXml 属性返回类似于 InnerText 的文本，也返回所有的标记。如果获取 book 节点上的 InnerXml 值，结果是如下字符串： <title>Beginning Visual C# 2010</title><author>Karli Watson</author><code>7582</code> 可以看出，如果字符串包含要直接插入 XML 文档的内容，这是很有用的。但是要对该字符串负全责，如果插入格式错误的 XML，应用程序就会生成异常
Value	Value 属性是操作文档中信息的最精练方式，但如前所述，在获取值时，只有几个类会返回有用的信息。返回所需文本的类如下所示： XmlText XmlComment XmlAttribute

插入新节点

了解了如何遍历 XML 文档，如何获取元素的值后，下面学习如何给前面使用的 books 文档添加节点，改变文档的结构。

要在列表中插入新元素，需要使用 XmlDocument 和 XmlNode 类中的新方法，如表 22-4 所示。XmlDocument 类的方法可以创建新的 XmlNode 和 XmlElement 实例，这非常不错，因为这两个类都只有一个受保护的构造函数，不能直接使用 new 创建它们的实例。

表 22-4

方　法	说　明
CreateNode	创建任意类型的节点。该方法有三个重载版本，其中两个允许创建 XmlNodeType 枚举中所列出的类型的节点，另一个允许把要使用的节点类型指定为字符串。除非对指定的不是枚举中的节点类型有完全的把握，否则强烈推荐使用枚举的两个重载版本。该方法返回的一个 XmlNode 实例，该实例可以显式转换为合适的类型
CreateElement	这只是 CreateNode 的一个版本，只能创建 XmlDocument 类型的节点
CreateAttribute	这也只是 CreateNode 的一个版本，只能创建 XmlAttribute 类型的节点
CreateTextNode	创建 XmlTextNode 类型的节点
CreateComment	在这个列表中包含这个方法，是为了说明可以创建的节点类型的多样性。这个方法并不创建由 XML 文档表示的数据节点，而是创建注释，以便人们读取数据。在应用程序中读取文档时，就可以读取注释

表 22-4 中的方法都用于创建节点，在调用其中一个方法后，就必须执行一些操作。在创建节点后，节点并没有包含其他信息，节点也没有插入到文档中。为此，应使用派生于 XmlNode 的类(包括 XmlDocument 和 XmlElement)中的方法。表 22-5 描述了这些方法。

表 22-5

方 法	说 明
AppendChild	把一个子节点追加到 XmlNode 类型或其派生类型的节点上。在调用该方法之后，追加的节点显示在相应节点的子节点列表的最后。如果不关心子节点的顺序，这就不重要，但如果子节点的顺序很重要，就应以正确的顺序追加节点
InsertAfter	使用 InsertAfter 方法，可以控制插入新节点的位置。该方法带有两个参数，第一个是新节点，第二个是在其后插入新节点的节点
InsertBefore	这个方法与 InsertAfter 类似，但新节点插到参考节点之前

下面的示例以前面的示例为基础，在 books.xml 文档中插入一个 book 节点。该示例中没有清理文档的代码，所以如果该示例运行几次，文档中就会包含许多相同的节点。

试一试：创建节点

按照下面的步骤给 books.xml 文档添加一个节点。

(1) 在窗体的已有按钮下面添加一个按钮，命名为 buttonCreateNode。将其Text 属性改为 Create Node。

(2) 双击新按钮，输入下面的代码：

```
private void buttonCreateNode_Click(object sender, EventArgs e)
{
  // Load the XML document.
  XmlDocument document = new XmlDocument();
  document.Load(@"C:\Beginning Visual C#\Chapter 22\Books.xml");

  // Get the root element.
  XmlElement root = document.DocumentElement;

  // Create the new nodes.
  XmlElement newBook = document.CreateElement("book");
  XmlElement newTitle = document.CreateElement("title");
  XmlElement newAuthor = document.CreateElement("author");
  XmlElement newCode = document.CreateElement("code");
  XmlText title = document.CreateTextNode("Beginning Visual C# 2008");
  XmlText author = document.CreateTextNode("Karli Watson et al");
  XmlText code = document.CreateTextNode("1234567890");
  XmlComment comment = document.CreateComment("The previous edition");

  // Insert the elements.
  newBook.AppendChild(comment);
  newBook.AppendChild(newTitle);
  newBook.AppendChild(newAuthor);
  newBook.AppendChild(newCode);
  newTitle.AppendChild(title);
  newAuthor.AppendChild(author);
  newCode.AppendChild(code);
  root.InsertAfter(newBook, root.FirstChild);
```

```
        document.Save(@"C:\Beginning Visual C#\Chapter 22\Books.xml");
    }
```

Code snippet Chapter22\InsertingNodes\Form1.cs

(3) 运行应用程序，单击 Create Node 按钮。再单击 Loop 按钮，对话框如图 22-5 所示。

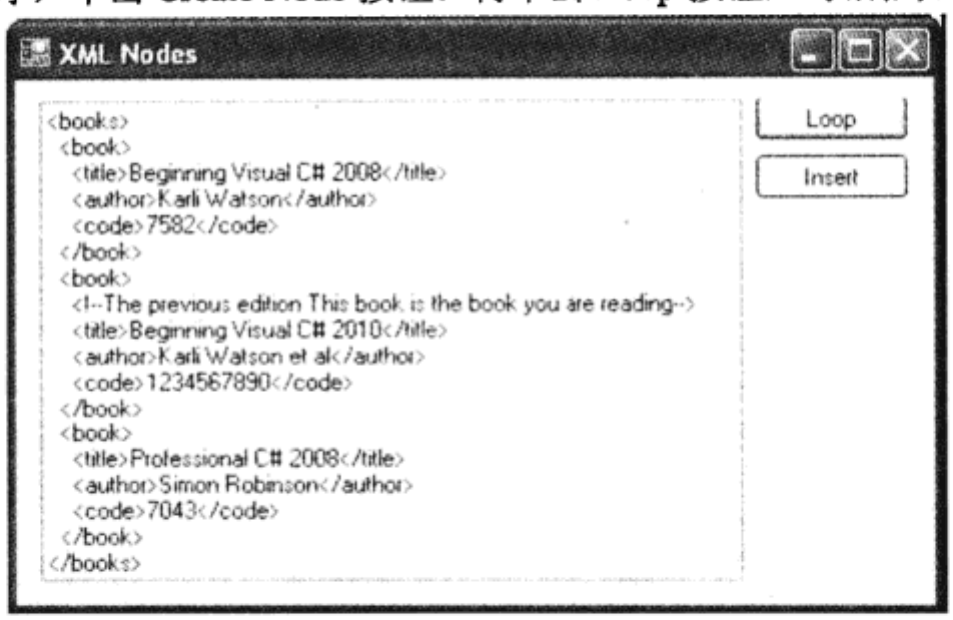


图　22-5

在上面的示例中没有创建一个重要类型的节点：XmlAttribute，这里把它留作本章的练习。

示例的说明

buttonCreateNode_Click 方法中的代码创建了所有节点。它创建了 8 个新节点，其中 4 个节点的类型是 XmlElement，3 个节点的类型是 XmlText，最后一个节点的类型是 XmlComment。

所有节点都是使用封装 XmlDocument 实例的方法创建的。XmlElement 节点是用 CreateElement 方法创建的，XmlText 节点是用 CreateTextNode 方法创建的，XmlComment 节点是用 CreateComment 方法创建的。

创建完节点后，还需要把它们插入 XML 树。这是使用元素上的 AppendChild 方法实现的，新节点将成为该元素的一个子节点。唯一的例外是book 节点，它是所有新节点的根节点。这个节点使用根对象的 InsertAfter 方法插入到树中。使用 AppendChild 方法插入的所有节点总是成为子节点列表的最后一项，而 InsertAfter 允许指定节点的位置。

删除节点

学习了如何创建新节点，剩下的就是如何删除节点了。派生于XmlNode 的所有类都包含允许从文档中删除节点的两个方法，如表 22-6 所示。

表　22-6

方 法 名	说　明
RemoveAll	这个方法删除节点上的所有子节点。不太明显的是，它还会删除节点上的所有特性，因为也将它们看作子节点
RemoveChild	这个方法删除节点上的一个子节点，返回从文档中删除的节点。如果改变主意，还可以把它重新插回到文档

下面的示例将扩展前面两个示例创建的 Windows 窗体应用程序，使之包含删除节点的功能。只需找出 book 节点的最后一个实例，并删除它。

试一试：删除节点

下面的步骤将找出 book 节点的最后一个实例，并删除它。

(1) 在前面两个已有的按钮下面添加一个新按钮，命名为buttonDeleteNode，将其 Text 属性设置为 Delete Node。

(2) 双击新按钮，输入下面的代码：

可从 wrox.com 下载源代码

```
private void buttonDeleteNode_Click(object sender, EventArgs e)
{
  // Load the XML document.
  XmlDocument document = new XmlDocument();
  document.Load(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");

  // Get the root element.
  XmlElement root = document.DocumentElement;

  // Find the node. root is the <books> tag, so its last child
  // which will be the last <book> node.
  if (root.HasChildNodes)
  {
    XmlNode book = root.LastChild;

    // Delete the child.
    root.RemoveChild(book);

    // Save the document back to disk.
    document.Save(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");
  }
}
```

代码段 Chapter22\RemovingNodes\Form1.cs

(3) 运行应用程序。单击 Delete Node 按钮，再单击 Loop 按钮，树中的最后一个节点就会消失。

示例的说明

把 XML 加载到 XmlDocument 对象上后，就检查根元素，确定在加载的 XML 中是否有子元素。如果有，就使用 XmlElement 类的 LastChild 属性获取最后一个子元素。之后，调用 RemoveChild，给它传送要删除的元素实例(在本例中是根元素的最后一个子元素)，就删除了该元素。

22.2.2 选择节点

前面介绍了如何浏览 XML 文档、如何操作文档的值、如何创建新节点以及如何删除它们。剩下的就是在不遍历整个树的情况下选择节点。

XmlNode 类包含的两个方法常用于从文档中选择节点，且不遍历其中的每个节点，如表 22-7 所示。这两个方法是 SelectSingleNode 和 SelectNodes，它们都使用一种特殊的查询语言 XPath 来选择节点。稍后介绍该语言。

表 22-7

方 法	说 明
SelectSingleNode	选择一个节点。如果创建一个查找多个节点的查询，就只返回第一个节点
SelectNodes	以 XmlNodesList 类的形式返回一个节点集合

22.2.3 XPath

XPath 是 XML 文档的查询语言，就像 SQL 是关系数据库的查询语言一样。它由表 22-7 中的两个方法使用，以免遍历 XML 文档的整个树。但是需要花一定的时间才能熟悉它，因为其语法与 SQL 或 C#完全不同。

XPath 相当复杂，这里只介绍其中的一小部分，就足以选择节点了。如果想了解 XPath 的更多内容，可参阅 www.w3.org/TR/xpath 和 Visual Studio 帮助页面。

为了正确使用 XPath，下面要使用 XML 文件 Elements.xml。该文档包含元素周期表中的部分化学元素。这个 XML 文件的部分内容列在本章后面“选择节点”示例中，其完整内容可以在本书网站的本章下载代码 Elements.xml 中找到。

表 22-8 列出了 XPath 执行的最常见操作。如果未特别说明，XPath 查询示例就根据它操作的节点来选择。必须有一个节点名称，可以假定当前节点是上述 XML 文档中的<book>节点。

表 22-8

目 的	XPath 查询示例
选择当前节点	.
选择当前节点的父节点	..
选择当前节点的所有子节点	*
选择带有特定名称的所有子节点，这里是 title	title
选择当前节点的一个特性	@Type
选择当前节点的所有特性	@*
按照索引选择一个子节点，这里是第二个元素节点	element[2]
选择当前节点的所有文本节点	text()
选择当前节点的一个或多个孙子节点	element/text()
在文档中选择带有特定名称的所有节点，在这里是所有的 mass 节点	//mass
在文档中选择带有特定名称和特定父节点名称的所有节点，在这里父节点名称是 element，节点名称是 name	//element/name
选择值满足条件的节点，在这里，选择元素名为 Hydrogen 的元素	//element[name='Hydrogen']
选择特性值满足条件的节点，在此，Type 特性的值是 Noble Gas	//element[@Type='Noble Gas']

下面的“试一试”示例要创建一个小型应用程序，以执行许多预定义的查询，查看结果，并可以输入自己的查询。

试一试：选择节点

如前所示，这个示例使用新的 XML 文件 Element.xml。可以从本书的网站上下载这个文件，或者键入下面的代码：

可从 wrox.com 下载源代码

```
<?xml version="1.0"?>
<elements>
  <!--First Non-Metal-->
  <element Type="Non-Metal">
    <name>Hydrogen</name>
    <symbol>H</symbol>
    <number>1</number>
    <specification>
      <mass>1.007825</mass>
      <density>0.0899 g/cm3</density>
    </specification>
  </element>
  <!--First Noble Gas-->
  <element Type="Noble Gas">
    <name>Helium</name>
    <symbol>He</symbol>
    <number>2</number>
    <specification>
      <mass>4.002602</mass>
      <density>0.1785 g/cm3</density>
    </specification>
  </element>
  <!--First Halogen-->
  <element Type="Halogen">
    <name>Fluorine</name>
    <symbol>F</symbol>
    <number>9</number>
    <specification>
      <mass>18.998404</mass>
      <density>1.696 g/cm3</density>
    </specification>
  </element>
  <element Type="Noble Gas">
    <name>Neon</name>
    <symbol>Ne</symbol>
    <number>10</number>
    <specification>
      <mass>20.1797</mass>
      <density>0.901 g/cm3</density>
    </specification>
  </element>
</elements>
```

代码段 Chapter22\XpathQuery\Elements.xml

把这个 XML 文件保存为 Elements.xml。一定要修改下述代码中文件的路径。这个示例是一个小型查询工具，可用于测试通过代码提供的 XML 上的不同查询。

按照下面的步骤创建一个带有查询功能的 Windows 窗体应用程序。

(1) 创建一个新的 Windows 窗体应用程序，命名为 XPathQuery。

(2) 创建如图 22-6 所示的对话框。按照图中所示给控件命名，但按钮除外，它应命名为 buttonExecute，把 textBoxResult 控件的 Scrollbars 属性设置为 Vertical。

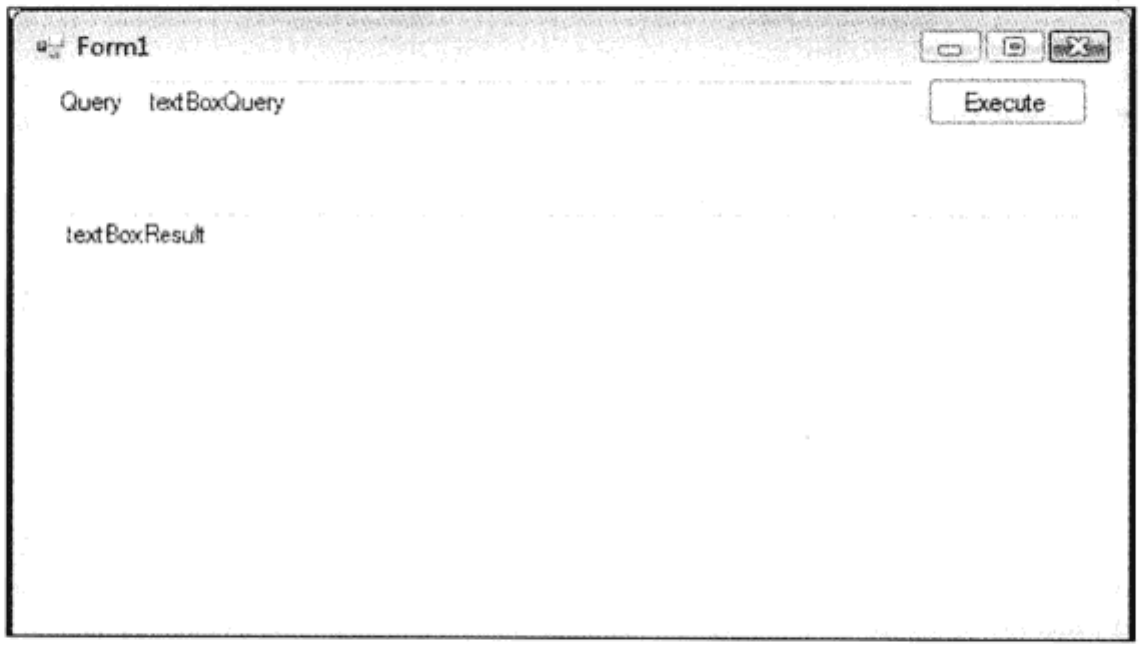


图　22-6

(3) 右击窗体，选择 View Code。首先包含 using 指令：

```
using System.Xml;
```

(4) 接着，添加一个私有字段，来保存文档，在构造函数中初始化它：

```
private XmlDocument document;

public Form1()
{
    InitializeComponent();

    document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C#\Chapter 22\Elements Subset.xml");
}
```

(5) 在此需要使用几个帮助方法，以便在文本框 textBoxResult 中显示查询结果：

```
private void Update(XmlNodeList nodes)
{
  if (nodes == null || nodes.Count == 0)
  {
    textBoxResult.Text = "The query yielded no results";
    return;
  }
  string text = "";
  foreach (XmlNode node in nodes)
  {
    text = FormatText(node, text, "") + "\r\n";
  }
  textBoxResult.Text = text;
}
```

(6) 更新构造函数，以便在应用程序启动时显示 XML 文件的全部内容：

```
public Form1()
{
  InitializeComponent();

  document = new XmlDocument();
  document.Load(@"C:\Beginning Visual C#\Chapter 22\Elements Subset.xml");

Update(document.DocumentElement.SelectNodes("."));
}
```

(7) 把上一个“试一试”示例中的 FormatText 和 AddAttributes 方法复制粘贴到新项目中。

(8) 最后插入代码，执行用户在文本框中输入的内容：

```
private void buttonExecute_Click(object sender, EventArgs e)
{
  try
  {
    XmlNodeList nodes = document.DocumentElement.SelectNodes(textBoxQuery.Text);
    Update(nodes);
  }
  catch (Exception err)
  {
    textBoxResult.Text = err.Message;
  }
}
```

(9) 运行应用程序，在 textBoxQuery 文本框中输入下面的查询，以选择一个元素节点，其中包含文本为 Hydrogen 的节点。

```
element[name=' Hydrogen' ]
```

示例的说明

buttonExecute_Click 方法是执行查询的方法。因为我们不可能事先知道键入到 textBoxQuery 中的查询会生成一个还是多个节点，所以必须使用 SelectNodes 方法。该方法返回一个 XmlNodeList 对象，但如果所使用的查询是非法的，该方法就抛出一个与 XPath 相关的异常。

Update 方法负责遍历 SelectNodes 选择出来的 XmlNodeList 的内容。它对每个节点调用前面示例中的 FormatText，FormatText 负责递归遍历节点树，创建可以在 textBoxResult 控件中读取的文本。

在本章最后的练习中，读者需要执行许多其他的 XPath 查询。在把它们输入 XPathQuery 应用程序中查看结果时，尝试自己确定查询结果。

22.3 小结

本章学习了 Extensible Markup Language (XML)，XML 是一种存储和提取数据的文本格式。我们讨论了需要遵循的规则，以确保 XML 文档是格式良好的。还论述了如何根据 XSD 和 XDR 模式验证它们。

学习了 XML 的基础知识后，探讨了如何使用 C#和 Visual Studio 在代码中使用 XML，最后阐述了如何使用 XPath 在 XML 中进行查询。

第 23 章将学习如何使用一种非常有趣的查询语言 LINQ。这种语言也可以用于查询 XML。但这超

出了本书的范围。在此之前，请先完成下面的练习。

22.4　练习

(1) 修改“创建节点”示例中的插入方式，把值为 1000 的特性 Pages 插入 book 节点。

(2) 确定下述 XPath 查询的结果，再把查询键入“选择节点”示例中的 XPathQuery 应用程序，来验证自己的结果。注意所有的查询都在元素节点 DocumentElement 上执行。

```
//elements
element
element[@Type=' Noble Gas' ]
//mass
//mass/..
element/specification[mass=' 20.1797' ]
element/name[text()=' Neon' ]
```

(3) 在许多 Windows 系统中，XML 的默认查看器都是 Web 浏览器。如果使用 Internet Explorer，在其中加载 Elements.xml 文件，就会看到美观的 XML 的格式化视图。在浏览器控件(而不是文本框)中显示查询的 XML 的效果为什么不理想？

附录 A 给出了练习答案。

22.5　本章要点

主　题	重 要 概 念
XML 语法	XML 文档从 XML 声明、XML 名称空间、XML 元素和特性中创建。XML 声明定义了 XML 版本，XML 名称空间用于定义词汇表，XML 元素和特性用于定义 XML 文档的内容
格式良好的 XML	格式良好的 XML 是遵循 XML 基本语法规则的 XML。如果文档正好有一个根元素，每个元素都有一个结束标记，没有相互重叠的元素(所有的子元素都必须完全嵌套在父元素中)，所有的特性都放在引号中，该文档就是格式良好的。所有 XML 读取器都可以读取格式良好的 XML，很少允许读取格式有误的 XML。严格地说，如果结束标记的文档是格式有误的，它就不是 XML 文档
有效的 XML	有效的 XML 是格式良好的 XML，且通过检查 XML 是从模式中生成的，就可以验证 XML 的有效性。确保 XML 有效是非常重要的，因为这样可以对 XML 文档的内容做一些假定，如果知道文档中的结构和名称与期望的一致，就可以处理利用这些结构和名称生成的 XML
XML 模式	XML 模式用于定义 XML 文档的结构。需要与第三方交换信息时，模式特别有用。对所交换的数据的模式达成一致后，我们和第三方就可以检查该文档是否有效
程序中的 XML	目前 XML 在.NET 领域中广泛使用，.NET Framework 为创建和处理 XML 提供了一系列类。可以使用 XML 存储应用程序配置，把数据保存到磁盘上，在线上传送信息等
XPath	XPath 是在 XML 文档中查询数据的一种方式之一。要使用 XPath，就必须熟悉 XML 文档的结构，才能从中选择各个元素。尽管 XPath 可以用于任何格式良好的 XML 文档，但创建查询时必须了解文档的结构，这意味着确保文档有效，也就确保了只要文档对于同一个模式而言是有效的，查询就可以用于多个文档

第23章

LINQ 简 介

本章内容：

- 编写 LINQ 查询，LINQ 查询语言的组成部分
- 使用 LINQ 方法语法和 LINQ 查询语法
- 对查询结果排序，包括多级排序
- 使用 LINQ 聚合运算符的方式和场合
- 使用投射(Projection)在查询中创建新对象
- 使用 Distinct()、Any()、All()、First()、FirstOrDefault()、Take()和 Skip()运算符
- 组合查询
- Set 运算符和联合

本章介绍 Language Integrated Query(LINQ)，这是 C# 3.0 语言引入的一个扩展，C#3.0 是 Visual Studio 2010 中支持的 C# 4 语言的上一版。LINQ 可以处理非常大的数据集合，这一般需要选择集合的一个子集，来完成执行程序的任务。

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

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

使用 LINQ 可以查询 C#中许多不同的数据源，包括对象、SQL 数据库、XML 文档、实体数据模型和外部应用程序(如 Amazon Web 服务和公司目录)。本章介绍的 LINQ 语法和方法适用于所有不同的数据源。查询不同数据源的 LINQ 提供程序在下一章介绍。

LINQ非常大，完整地介绍它的所有特性和方法已超出了本书的讨论范围。但本章将列举 LINQ 用户需要的所有运算符和语句类型的示例，并酌情给出深入探讨这些主题的资料。

23.1 第一个 LINQ 查询

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

试一试：第一个 LINQ 查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-1-FirstLINQquery，然后打开主源文件 Program.cs。

(2) 注意，Visual C# 2010 默认在 Program.cs 中包含 Linq 名称空间：

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

(3) 在 Program.cs 的 Main()方法中添加如下代码：

可从 wrox.com 下载源代码

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

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

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

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

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

代码段 23-1-FirstLINQquery\Program.cs

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

```
Names beginning with S:
Smith
Smythe
Small
```

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

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

示例的说明

第一步是引用 System.Linq 名称空间，这在创建项目时由 Visual C# 2010 自动完成：

```
using System.Linq;
```

所有的基本底层系统都支持 System.Linq 名称空间中用于 LINQ 的类。如果在 Visual C# 2010 外部创建 C#源文件或编辑以前的 Visual C# 2005 项目，就必须手动添加 using System.Linq 语句。

下一步创建一些数据，在本例中就是声明并初始化 names 数组：

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

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

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

这是一个看起来比较古怪的语句。它不像是 C#语言，实际上 from..where..select 语法类似于 SQL 数据库查询语言。但这个语句不是 SQL，而是 C#，在 Visual C# 2010 中输入这些代码时，from、where 和 select 会突出显示为关键字，这个古怪的语法对编译器而言是完全正确的。

这个程序中的 LINQ 查询语句使用了 LINQ 声明性查询语法：

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

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

23.1.1　用 var 关键字声明结果变量

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

```
var queryResults =
```

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

如果希望了解细节，查询结果将是实现了IEnumerable<>接口的类型。IEnumerable后面的尖括号表示这是一个泛型类型。泛型详见第 12 章。

在上例中，编译器创建了System.Linq.OrderedSequence<string, string>的一个实例，这个特殊的 LINQ 数据类型提供了字符串的有序列表(因为数据源是一个字符串集合)。

另外，queryResult 名称是随意指定的，可以把结果命名为任何名称，例如，namesBeginningWithS 或者在程序中有意义的其他名称。

23.1.2 指定数据源：from 子句

LINQ 查询的下一部分是 from 子句，它指定了要查询的数据：

```
from n in names
```

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

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

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

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

23.1.3 指定条件：where 子句

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

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

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

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

23.1.4 指定元素：select 子句

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

```
select n;
```

select 子句是必须的，因为必须指定结果集中有哪些元素。这个结果集并不是很有趣，因为在结

果集的每个元素中都只有一项 name。如果结果集中有比较复杂的对象，使用 select 子句的有效性就比较明显，不过我们还是先完成这个示例。

23.1.5　完成：使用 foreach 循环

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

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

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

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

23.1.6　延迟执行的查询

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

现在回过头来看看代码。由于输出了结果，所以程序结束：

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

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

23.2　使用 LINQ 方法语法

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

23.2.1　LINQ 扩展方法

LINQ 实现为一系列扩展方法，用于集合、数组、查询结果和其他实现了 IEnumerable 接口的对象。在 Visual Studio IntelliSense 特性中可以看到这些方法。例如，在 Visual C# 2010 中打开 FirstLINQquery 程序中的 Program.cs 文件，在 name 数组的下面键入对该数组的一个新引用：

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

names.
```

键入 names 后面的句点后，就会看到 Visual Studio IntelliSense 列出的可用于 names 的方法。

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

23.2.2 查询语法和方法语法

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

Visual C# 2010 联机帮助建议尽量使用查询语法，仅在需要时使用方法语法。

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

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

LINQ 很容易完成这个传送任务。使用 Lambda 表达式就可以创建方法/函数。参见第 14 章。下一个示例将更清楚地说明这一点。

试一试：使用 LINQ 方法语法

按照下面的步骤在 VC# 2010 中创建示例：

(1) 可以修改 FirstLINQQuery 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-2-LINQMethodSyntax。打开主源文件 Program.cs。

(2) Visual C# 2010 会自动在 Program.cs 中包含 Linq 名称空间：

```
using System.Linq;
```

(3) 在 Program.cs 的 Main()方法中添加如下代码：

可从 wrox.com 下载源代码

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

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

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

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

代码段 23-2-LINQMethodSyntax\Program.cs

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

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

示例的说明

与前面一样，Visual C# 2010 会自动引用 System.Linq 名称空间：

```
using System.Linq;
```

再次声明和初始化 names 数组，创建相同的源数据：

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

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

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

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

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

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

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

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

这里不重复说明这些代码行，因为本章第一个示例后面的“示例的说明”已经解释过了。下面继续研究如何使用 LINQ 的更多功能。

23.3　排序查询结果

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

排序。

试一试：给查询结果排序

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 可以修改 FirstLINQQuery 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序项目 23-3-OrderQueryResults。

(2) 打开主源文件 Program.cs。与以前一样，Visual C# 2010 会自动在 Program.cs 中包含 System.Linq 名称空间。

(3) 在 Program.cs 的 Main()方法中添加如下代码：

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

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

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

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

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

代码段 23-3-OrderQueryResults\Program.cs

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

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

示例的说明

这个程序与前一个程序几乎相同，只是在查询语句中增加了一行代码：

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

23.4　orderby 子句

orderby 子句如下所示：

```
orderby n
```

与 where 子句一样，orderby 子句也是可选的。只要添加一行，就可以给任意查询的结果排序，如果不使用LINQ，就需要添加至少几行代码和几个方法或集合，来存储根据选择实现的排序算法重新排序的结果。如果有多个需要排序的类型，就需要为每个类型实现一系列排序方法。而使用 LINQ 不需要做这些工作，只需在查询语句中添加一个子句即可。

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

```
orderby n descending
```

这会使示例的结果变成：

```
Smythe
Smith
Small
Singh
Samba
```

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

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

结果如下：

```
Samba
Smythe
Smith
Singh
Small
```

最后一个字母按字母顺序排序(a，e，h，h，l)。但这个执行过程依赖于实现方式。即无法保证除了 orderby 子句中指定的内容之外的其他字母的顺序。由于仅考虑最后一个字母，所以在本例中，Smith 在 Singh 的前面。

23.5　用方法语法排序

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

试一试：使用方法语法排序

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 可以修改 23-2-LINQMethodSyntax 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序项目 23-4-OrderMethodSyntax。

(2) 在 Program.cs 的 Main()方法中添加如下代码，在所有的示例中，Visual C# 2010 都会自动包含对 System.Linq 名称空间的引用。

可从 wrox.com 下载源代码

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

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

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

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

代码段 23-4-OrderMethodSyntax\Program.cs

(3) 编译并执行程序。结果是以 S 开头的 names 列表，且按字母顺序排序，与前一个示例的输出结果相同。

示例的说明

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

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

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

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

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

为了给元素逆序排序，可以调用 OrderByDescending()方法：

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

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

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

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

这会生成与上一个示例相同的结果，但按照每个姓名的最后一个字母排序。

23.6　查询大型数据集

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

试一试：查询大型数据集

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-5-LargeNumberQuery。与以前一样，创建项目时，Visual C# 2010 会自动在 Program.cs 中包含 Linq 名称空间。

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

(2) 在 Main()方法中添加如下代码：

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

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

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

代码段 23-5-LargeNumberQuery\Program.cs

(3) 添加如下方法，生成一个随机数列表：

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

(4) 编译并执行程序。结果是一个小于 1000 的数字列表，如下所示：

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

示例的说明

与前面一样，第一步是引用 System.Linq 名称空间，这是在创建项目时由 Visual C# 2010 自动引用的：

```
using System.Linq;
```

接着创建一些数据，本例中是创建并调用 generateLotsOfNumbers()方法：

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

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

这不是一个小数据集，数组中有 120 万个数字！在本章最后的一个练习中，需要修改传送给 generateLotsOfNumbers()方法的 size 参数，生成数量不同的随机数，看看这会对查询结果有什么影响。在做练习时会看到，这里的 size 参数 12 345 678 非常大，足以生成一些小于 1 000 的随机数，从而获得为第一个查询显示的结果。

数值应随机分布在有符号的整数范围内(从 0 到超过 20 亿)。用种子值 0 创建随机数生成器，可以确保每次创建相同的随机数集合，这是可以重复的，所以会获得与此处相同的查询结果，但在尝

试一些查询之前，并不知道查询结果是什么。而 LINQ 使这些查询很容易编写。

查询语句本身类似于前面用于 names 数组的查询，也是选择某些满足条件的数字(这里是条件的数字小于 1 000)：

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

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

用 foreach 语句输出查询的结果，与前面的示例相同：

```
Console.WriteLine("Numbers less than 1000:");

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

同样，将结果输出到控制台上，并读取一个字符以暂停输出：

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

后面所有的示例都有暂停代码，但不再列出，因为每个示例的暂停代码都相同。

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

23.7　聚合运算符

查询给出的结果常常超出了我们的期望。如果要修改大数查询程序的条件，只需列出大于 1 000 的数字，而不是小于 1 000 的数字，这会得到非常多的查询结果，数字会不停地显示出来。

LINQ 提供了一组聚合运算符，可用于分析查询的结果，而无需迭代所有结果。表 23-1 中列出的聚合运算符是数字结果集最常用的运算符，例如，大数查询的结果就常用这些聚合运算符，如果读者使用过数据库查询语言如 SQL，就会很熟悉这些运算符。

表　23-1

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

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

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

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

试一试：数字聚合运算符

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 这个示例可以修改前面的 LargeNumberQuery 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台项目 23-6-NumericAggregates。

(2) 与以前一样，创建项目时，Visual C# 2010 会自动在 Program.cs 中包含 Linq 名称空间。只需修改 Main()方法，如下面的代码和本示例其余部分所示。与上一个例子一样，这个查询也不使用 orderby 子句。但是 where 子句中的条件与前一个例子相反(数字应大于 1 000(n>1000)，而不是小于 1 000)。

可从 wrox.com 下载源代码

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

    Console.WriteLine("Numeric Aggregates");

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

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

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

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

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

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

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

代码段 23-6-NumericAggregates\Program.cs

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

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

(4) 编译并执行，显示个数、最小值、最大值和平均值，如下所示:

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

这个查询生成的结果数量超过上一个例子(超过 120 万)。在这个结果集上使用 orderby，对性能会有很显著的影响。结果集中的最大值超过 20 亿，最小值刚刚大于 1 000。平均值大约是 10 亿，接近数字范围的中间值。看来，Rand()函数可以生成均匀分布的数字。

示例的说明

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

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

查询也与上一个例子相同，只是把 where 条件从小于改为大于:

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

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

下面看看聚合运算符的用法:

- Count() 返回查询结果中的行数，在这个例子中是 12 345 671 行:

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

```
Console.WriteLine(queryResults.Count());
```

- Max() 返回查询结果中的最大值，在这个例子中是大于 20 亿的一个数 2 147 483 591，它非常接近 int 的最大值(int.MaxValue 或 2 147 483 647)。

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

- Min() 返回查询结果中的最小值，在这个例子中是 1 034。

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

- Average() 返回查询结果中的平均值，在这个例子中是 1 073 643 807.502 98，它非常接近 1 000 到 20 亿的值范围的中间值。对于随机的大数而言，这个中间值没有什么意义，但说明了可以对查询结果进行分析。本章最后一部分将使用这些运算符对面向业务的数据进行更切合实际的分析。

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

- Sum()，在此给 Sum()方法调用传送了 Lambda 表达式 n=>(long) n，以获得所有数字的总和。与 Count()、Min()、Max()等相同，Sum()有一个无参数的重载版本，但使用 Sum()方法的这个版本会导致溢出错误，因为数据集中的数字太多，它们的总和太大，不能放在 Sum()方法的无参数重载版本返回的标准的 32 位 int 中。Lambda 表达式允许把 Sum()方法的结果转换为 64 位长整数，它可以保存超过 13^{10} 的数字 13 254 853 218 619 179，而不出现溢出。Lambda 表达式允许执行这个转换。

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

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

23.8 查询复杂的对象

前面的例子说明了 LINQ 查询可以处理各种简单类型，例如，数字和字符串。下面看看如何使用 LINQ 查询较复杂的对象。我们要创建一个简单的 Customer 类，它拥有的信息足以创建一些有趣的查询。

试一试：查询复杂的对象

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-7-QueryComplex Objects。

(2) 在 Program.cs 的 Program 类开头，给 Customer 类添加如下的简短类定义：

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

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

代码段 23-7-QueryComplexObjects\Program.cs

(3) 在 Program.cs 的 Program 类的 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
        List < Customer > customers = new List < Customer > {
            new Customer { ID="A", City="New York", Country="USA",
                                                Region="North America", Sales=9999},
            new Customer { ID="B", City="Mumbai", Country="India",
                                                Region="Asia", Sales=8888 },
            new Customer { ID="C", City="Karachi", Country="Pakistan",
                                                Region="Asia", Sales=7777 },
            new Customer { ID="D", City="Delhi", Country="India",
                                                Region="Asia", Sales=6666 },
            new Customer { ID="E", City="S    o Paulo", Country="Brazil",
                                                Region="South America", Sales=5555 },
            new Customer { ID="F", City="Moscow", Country="Russia",
                                                Region="Europe", Sales=4444 },
            new Customer { ID="G", City="Seoul", Country="Korea", Region="Asia",
                                                Sales=3333 },
            new Customer { ID="H", City="Istanbul", Country="Turkey",
                                                Region="Asia", Sales=2222 },
            new Customer { ID="I", City="Shanghai", Country="China", Region="Asia",
                                                Sales=1111 },
            new Customer { ID="J", City="Lagos", Country="Nigeria",
                                                Region="Africa", Sales=1000 },
            new Customer { ID="K", City="Mexico City", Country="Mexico",
                                                Region="North America", Sales=2000 },
            new Customer { ID="L", City="Jakarta", Country="Indonesia",
                                                Region="Asia", Sales=3000 },
            new Customer { ID="M", City="Tokyo", Country="Japan",
                                                Region="Asia", Sales=4000 },
            new Customer { ID="N", City="Los Angeles", Country="USA",
```

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

代码段 23-7-QueryComplexObjects\Program.cs

(4) 编译并执行程序，结果是来自亚洲的顾客列表：

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

示例的说明

在 Customer 类的定义中使用了 C#的自动属性功能，来声明 Customer 类的公共属性(ID、City、Country、Region、Sales)，而不必为每个属性显式编写私有实例变量和 get/set 代码：

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

```
    public string City { get; set; }
    …
```

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

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

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

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

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

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

在 Main()方法中，创建查询语句，本例选择来自亚洲的顾客：

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

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

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

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

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

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

在循环中，仅编写了一条语句：

```
{
```

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

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

```
Customers in Asia:
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
Program finished, press Enter/Return to continue:
```

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

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

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

23.9 投影：在查询中创建新对象

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

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

这会生成如下结果：

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

```
Beijing
```

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

```
select n + 1
```

对于字符串数据类型查询，应如下所示：

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

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

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

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

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

试一试：投射——在查询中创建新对象

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 修改 23-7-QueryComplexObjects，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-8-ProjectionCreateNewObjects。

(2) 如果选择创建新项目，就从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

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

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

代码段 23-8-ProjectionCreateNewObjects\Program.cs

(4) Main()方法的其余代码与前面示例的相同。

(5) 编译并执行程序，将列出来自北美的顾客的所选字段，如下所示：

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

示例的说明

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

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

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

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

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

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

23.10 投影：方法语法

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

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

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

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

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

Region 属性未包含在 Select()投影创建的匿名类型(c.City，c.Country，c.Sales)中，所以程序会在 Where()方法上得到一个编译错误，表示匿名类型不包含 Region 的定义。

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

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

23.11 单值选择查询

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

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

试一试：投影——单值选择查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 修改前面的示例 23-8-ProjectionCreateNewObjects，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-9-SelectDistinctQuery。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

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

可从 wrox.com 下载源代码

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

代码段 23-9-SelectDistinctQuery\Program.cs

(4) Main()方法的其余代码与前面例子的相同。

(5) 编译并执行程序，结果显示的是顾客所在的唯一区域，如下所示：

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

示例的说明

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

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

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

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

查询语法由 C#编译器转换为方法语法中的同系列 LINQ 方法调用，所以如果可以改进可读性和样式，可以混合和匹配它们。

23.12　Any 和 All

我们常常需要的另一类查询是确定数据是否满足某个条件，或者确保所有数据都满足某个条件。例如，需要确定某个产品是否已经脱销(库存为 0)，或者是否发生了某个交易。

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

试一试：使用 Any()和 All()

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 修改前面的示例 23-9-SelectDistinctQuery，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-10-AnyAndAll。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

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

可从 wrox.com 下载源代码

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

代码段 23-10-AnyAndAll\Program.cs

(4) Main()方法的其余代码与前面例子的相同。

(5) 编译并执行程序，将看到一些消息，指出一些顾客来自美国，并不是所有的顾客都来自亚洲：

```
Some customers are in the USA
Not all customers are in Asia
Program finished, press Enter/Return to continue:
```

示例的说明

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

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

LINQ 方法 Any()把传送给它的 Lambda 表达式 c=>c.Country == "USA"应用于 customers 列表中

的所有数据，如果对于列表中的任意顾客，Lambda 表达式是 true，就返回 true。

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

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

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

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

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

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

23.13　多级排序

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

试一试：多级排序

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 修改前面的示例 23-8-ProjectionCreateNewObjects，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-11-MultiLevelOrdering。

(2) 如 23-7-QueryComplexObjects 示例所示，创建 Customer 类并初始化顾客列表(List<Customer> customers)，这些代码与前面示例中的代码完全相同。

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

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

代码段 23-11-MultiLevelOrdering\Program.cs

(4) 结果处理循环和 Main()方法的其余代码与前面例子中的相同。

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

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

示例的说明

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

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

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

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

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

添加了 descending 关键字后，结果如下：

```
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
```

```
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = T, Region = South America, Country = Peru, City = Lima }
{ ID = S, Region = South America, Country = Colombia, City = Bogotá }
{ ID = E, Region = South America, Country = Brazil, City = São Paulo }
Program finished, press Enter/Return to continue:
```

注意，即使国家的顺序被反转了，印度和中国的城市仍按升序排序。

23.14 多级排序方法语法：ThenBy

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

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

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

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

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

23.15 组合查询

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

试一试：组合查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-12-GroupQuery。

(2) 如 23-7-QueryComplexObjects 示例所示，创建 Customer 类并初始化顾客列表(List<Customer> customers)，这些代码与前面示例中的代码完全相同。

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

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

代码段 23-12-GroupQuery\Program.cs

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

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

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

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

示例的说明

Customer 类和 customers 列表的初始化与前面例子的相同。

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

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

要计算每个组的总和，应生成一个新的结果集 cg：

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

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

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

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

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

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

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

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

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

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

23.16　Take 和 Skip

假定需要数据集中销售量位于前 5 名的顾客。我们事先并不知道跻身前 5 名需要达到多高的销量，所以不能使用 where 条件查找他们。

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

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

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

下面的示例对顾客列表数据使用了 Take()和 Skip()。

试一试：使用 Take 和 Skip

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-13-TakeAndSkip。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码。

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

可从 wrox.com 下载源代码

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

代码段 23-13-TakeAndSkip\Program.cs

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

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

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

(5) 编译并执行程序，结果显示的是前 5 名顾客和剩余的顾客：

```
Top Five Customers by Sales
{ ID = A, City = New York, Country = USA, Sales = 9999 }
{ ID = R, City = Beijing, Country = China, Sales = 9000 }
{ ID = B, City = Mumbai, Country = India, Sales = 8888 }
{ ID = Q, City = London, Country = UK, Sales = 8000 }
{ ID = C, City = Karachi, Country = Pakistan, Sales = 7777 }
Customers Not In Top Five
{ ID = P, City = Tehran, Country = Iran, Sales = 7000 }
{ ID = D, City = Delhi, Country = India, Sales = 6666 }
{ ID = O, City = Cairo, Country = Egypt, Sales = 6000 }
{ ID = E, City = Sao Paulo, Country = Brazil, Sales = 5555 }
{ ID = N, City = Los Angeles, Country = USA, Sales = 5000 }
{ ID = F, City = Moscow, Country = Russia, Sales = 4444 }
{ ID = M, City = Tokyo, Country = Japan, Sales = 4000 }
{ ID = G, City = Seoul, Country = Korea, Sales = 3333 }
{ ID = L, City = Jakarta, Country = Indonesia, Sales = 3000 }
{ ID = H, City = Istanbul, Country = Turkey, Sales = 2222 }
{ ID = T, City = Lima, Country = Peru, Sales = 2002 }
{ ID = K, City = Mexico City, Country = Mexico, Sales = 2000 }
{ ID = I, City = Shanghai, Country = China, Sales = 1111 }
```

```
{ ID = S, City = Bogot á , Country = Colombia, Sales = 1001 }
{ ID = J, City = Lagos, Country = Nigeria, Sales = 1000 }
Program finished, press Enter/Return to continue:
```

示例的说明

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

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

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

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

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

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

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

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

23.17 First 和 FirstOrDefault

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

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

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

试一试：使用 First 和 FirstOrDefault

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-14-FirstOrDefault。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer>customers)的代码。

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

可从 wrox.com 下载源代码

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

代码段 23-14-FirstOrDefault\Program.cs

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

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

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

(5) 编译并执行程序，结果如下：

```
A customer in Africa
{ City = Lagos, Country = Nigeria, Region = Africa }
A customer in Antarctica

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

示例的说明

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

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

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

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

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

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

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

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

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

使用 FirstOrDefault()时，当查询条件不满足时，将为 customers 列表返回默认元素，而使用 First()时则是返回 null(因为这些元素的类型未知)。因而这里对南极洲地区查询时，您会收到一条异常消息。

输出结果和暂停屏幕显示的代码与前面示例中的相同，只是消息有一点儿变化。

23.18 集运算符

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

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

试一试：集运算符

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-15-SetOperators。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer>customers)的代码。

(3) 在 Customer 类的后面添加如下 Order 类：

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

代码段 23-15-SetOperators\Program.cs

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

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

(5) 在 orders 列表初始化后，输入如下查询：

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

(6) 用 Intersect()输入如下查询：

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

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

(7) 接着，用 Except()输入如下查询：

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

(8) 最后，用 Union()输入如下查询：

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

(9) 编译并执行程序，结果如下：

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

示例的说明

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

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

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

使用两个简单的 from...select 查询来获取 Customer 类和 Order 类的 ID 字段：

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

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

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

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

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

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

在其余的 foreach 循环中也使用了这个输出逻辑。

接着使用 Except()运算符查找没有匹配顾客的订单 ID：

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

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

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

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

暂停屏幕显示的代码与前面示例中的相同。

集运算符非常有用，但它们的实际用处因为受到所有对象的类型都必须相同的限制而缩小了。在需要处理类型类似的结果集等少数情况下，这些运算符是很有用的。但在更常见的需要处理不同相关对象类型的情况下，应采用专门为处理不同对象类型而设计的机制，例如，join 语句。

23.19　Join 查询

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

试一试：Join 查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-16-JoinQuery。

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

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

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

代码段 23-16-JoinQuery\Program.cs

(4) 用前面例子中使用的标准 foreach 查询处理循环结束程序：

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

(5) 编译并执行程序，结果如下：

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

示例的说明

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

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

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

on 关键字在关键字段 ID 的后面，equals 关键字指定另一个集合中的对应字段。查询结果仅包含两个集中 ID 字段值相同的对象数据。

select 语句投影了一个带指定属性的新数据类型，因此可以清楚地看到最初的总销售量、新订单和最终的新总销售量：

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

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

foreach 循环的逻辑和查询中值的显示与本章前面示例中的相同。

23.20　小结

LINQ 使 C#中编写的查询非常简单和强大。第 24 章将学习如何使用 LINQ 查询关系数据库，以高效地处理大型数据集。

方法语法中有太多的 LINQ 方法，不可能在一本入门图书中包含所有这些方法。更多的细节和示例可参阅 LINQ 的 Microsoft 联机文档。要获得每个LINQ方法的小示例，可参阅 MSDN 步骤中的 101LINQ Samples 主题(或联机查看 http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx)。

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

23.21　练习

(1) 修改第一个示例程序 23-1-FirstLINQquery，将结果降序排列。

(2) 在大数程序示例 23-5-LargeNumberQuery 中修改传送给 generateLotsOfNumbers()方法的数字，创建不同规模的结果集，看看查询结果所受的影响。

(3) 给大数程序示例 23-5-LargeNumberQuery 中的查询添加一个 orderby 子句，看看这会如何影响性能。

(4) 修改大数程序示例 23-5-LargeNumberQuery 中的查询条件，选择数字列表中的较大和较小子集，看看这会如何影响性能？

(5) 修改方法语法示例 23-2-LINQMethodSyntax，完全删除 where 子句，输出量会有多少？

(6) 修改查询复杂对象程序示例 23-7-QueryComplexObjects，用对应于字段的条件选择查询字段的不同子集。

(7) 给第一个示例程序 23-1-FirstLINQquery 添加聚合运算符，哪些简单的聚合运算符能适用于这种非数字的结果集？

附录 A 给出了练习答案。

23.22　本章要点

主　题	重 要 概 念
LINQ 的概念和使用场合	LINQ 是内置于 C#中的一种查询语言。使用 LINQ 可以在大型的对象集合、XML 或数据库中查询数据
LINQ 查询的组成部分	LINQ 查询包含 from、where、select 和 orderby 子句
获取 LINQ 查询的结果的方式	使用 foreach 语句迭代 LINQ 查询的结果
延迟执行	LINQ 查询会延迟到执行 foreach 语句时执行

(续表)

主 题	重 要 概 念
方法语法和查询语法	简单的 LINQ 查询使用查询语法，较高级的查询使用方法查询。对于任意给定的查询，查询语法和方法语法的结果相同
聚合运算符	使用 LINQ 聚合运算符获得大型数据集的信息，而无需迭代每个结果
投影	使用投影技术改变数据类型，在查询中创建新对象
组合查询	使用组合查询给数据分组，再按照组进行排序、计算聚合值以及进行比较
排序	使用 orderby 运算符给查询的结果排序
Set 运算符	使用 set 运算符 Union()、Intersect()和 Distinct()在多个结果集中查找匹配的数据
Join 运算符	使用 Join 运算符在一个查询中查找多个集合中的相关数据

第24章

应用 LINQ

本章内容：

- LINQ 变体
- 对数据库使用 LINQ
- 导航数据库关系
- 对 XML 使用 LINQ
- 使用 LINQ to XML 构造函数
- 从数据库中生成 XML
- 使用 XML 片段

第 26 章介绍了 LINQ，说明了如何使用 LINQ 处理对象。本章研究如何把 LINQ 应用于查询，处理不同数据源(例如数据库和 XML)中的数据。

24.1 LINQ 的变体

Visual Studio 2010 和.NET Framework 4 带有几个内置的 LINQ 功能，它们为不同类型的数据提供了查询解决方案：

- **LINQ to Objects**：为任意类型的 C#内存对象提供查询，例如数组、列表和其他集合类型。上一章的所有示例都使用 LINQ to Objects。也可以把本章介绍的技巧应用于 LINQ 的所有变体。
- **LINQ to XML**：提供了 XML 文档的创建和处理功能，其语法与一般查询机制和其他 LINQ 变体相同。
- **LINQ to ADO.NET**：.NET 的 ADO.NET 或 Active Data Objects 是一个统称，囊括了.NET 中访问数据库(例如 Microsoft SQL Server、Oracle 等)中的数据的所有不同的类和库。LINQ to ADO.NET 包括 LINQ to Entities、LINQ to DataSet 和 LINQ to SQL。

- **LINQ to Entities**：ADO.NET Entity Framework 是.NET 4 中数据接口类的最新集合，Microsoft 推荐使用它进行新的开发工作。本章将给 Visual C#项目添加 ADO.NET Entity Framework 数据源，再使用 LINQ to Entities 查询它。
- **LINQ to DataSet**：DataSet 对象在.NET Framework 的第一版中引入。这个 LINQ 的变体允许使用 LINQ 轻松地查询旧.NET 数据。
- **LINQ to SQL**：这是用于.NET 3.5 的备用 LINQ 接口，主要用于 Microsoft SQL Server，在.NET 4 中被 LINQ to Entities 替代。
- **PLINQ**：PLINQ 或 Parallel LINQ 用并行编程库扩展了 LINQ to Objects，它可以使查询同时在多核处理器上执行。

有这么多 LINQ 的变体，根本不可能在一本入门图书中全部介绍，所以本章介绍如何把 LINQ 应用于 XML 最常见的数据源和关系数据库实体上。LINQ 处理所有数据源的方式都是非常类似的，所以一旦学会了如何使用两三个 LINQ 变体，就可以很容易把 LINQ 应用于新数据源。

24.2 给数据库使用 LINQ

SQL 数据库(如 Microsoft SQL Server 和 Oracle)也称为关系数据库。关系数据库建立在实体-关系模型的基础上，其中实体是数据对象(如顾客)的抽象概念，它与其他实体(如订单和产品，如顾客给产品下了订单)相关。

关系数据库使用 SQL 数据库语言(SQL 表示 Structured Query Language，结构化查询语言)查询和操作数据。传统上，处理数据库需要至少了解一些 SQL 知识，才能把 SQL 语句嵌入编程语言，或者把包含 SQL 语句的字符串传送给在面向 SQL 的数据库类库中的 API 调用或方法。

这听起来很复杂。现在，Visual Studio 2010 和 ADO.NET Entity Framework 可以创建 C#对象来表示数据库模型中的实体，再自动处理与 SQL 数据库通信的所有细节。它会把 LINQ 查询自动转换为 SQL 语句，使程序能很容易地处理 C#对象。

创建代码，建立一系列匹配已有关系表结构的类和集合是很费时费力的，但使用 LINQ to SQL 对象关系映射功能，可以从数据库中自动创建匹配数据库表的类，这样就不必自己创建它，而可以直接开始使用类。

24.3 安装 SQL Server 和 Northwind 示例数据

要运行本章的示例，必须安装 Microsoft SQL Server Express，这是 Microsoft SQL Server 的轻型版本。

如果很熟悉SQL Server，而且能访问安装了Northwind 示例数据库的 Microsoft SQL Server Express 2005 标准版或企业版，就可以跳过这个安装过程，但必须修改连接信息，以匹配自己的SQL Server 实例。如果没有使用过 SQL Server，就需要安装 SQL Server Express。

24.3.1　安装 SQL Server Express 2008

Visual Studio 2010 和 Visual C# 2010 Express 版本都包含 SQL Server Express 副本，这是 SQL Server 2008 的轻型桌面引擎版本。

如果已经安装了 Visual Studio 2010 或 Visual C# 2010 Express，但没有安装 SQL Server 2008 Express Edition，就可以使用如下 URL 下载并安装它：

```
http://microsoft.com/express/sql/default.aspx
```

不能使用 Microsoft SQL Server 压缩版和 LINQ to Entities，必须使用 SQL Server 2008 Express Edition。

24.3.2　安装 Northwind 示例数据库

本章的示例需要 SQL Server 的 Northwind 示例数据库，它未包含在 Visual C# 2010 或 SQL Server 2008 Express 中，但可以从 Microsoft 获得一个独立的下载软件包。在 Google 或类似的搜索站点上搜索“northwind sample database download”，或者直接进入如下 URL，就可以找到这个软件包。

```
www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a6803
4&displaylang=en
```

该链接可以下载安装文件 SQL2000SampleDb.msi。

单击 Run 按钮，执行.msi 文件，将会安装 Northwind 示例数据库文件。接受各个安装屏幕中的默认选项。完成时，数据库文件就安装在 C:\SQL Server 2000 Sample Databases\NORTHWND.MDF 中。

Northwind MDF 的文件名是 NORTHWND.MDF，没有 I。

保留这个文件名和路径比较方便，因为在以后创建与数据库的连接时要引用它。这就完成了本章需要的 SQL Express 和示例数据的安装。现在就可以使用 LINQ to Entities 了。

24.4　第一个 LINQ 数据库查询

在下面的示例中，要创建一个简单的查询，用 LINQ to SQL 找出 Northwind SQL Server 示例数据中的一个顾客对象子集，再输出到控制台上。

试一试：第一个 LINQ 数据库查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序项目 BegVCSharp_24_1_FirstLINQtoDatabaseQuery。

(2) 单击 OK 按钮创建项目。

(3) 为了给 Northwind 数据库添加 LINQ to Entities 数据源，可进入 Solution Explorer 窗格，右击 C#项目 BegVCSharp_24_1_FirstLINQtoDataQuery，选择 Data | Add New Data Source 菜单项。

(4) 在 Choose a Data Source Type 对话框中，选择 Database。

(5) 在 Choose a Database Model 对话框中，选择 Entity Data Model。

(6) 在 Choose Model Contents 对话框中，选择 Generate From Database。

(7) 在 Choose Your Data Connection 对话框中，选择 New Connection。

(8) 在 Connection Properties 对话框中，单击 Database File Name(new or existing)文本框右边的 Browse 按钮，找到安装 Northwind 数据的目录 C:\SQL Server 2000 Sample Databases\，选择 NORTHWND.MDF 作为数据库文件。单击 OK 关闭对话框，完成导入实体模型的操作。

(9) 在 Choose Your Database Objects 对话框中，展开 Tables 控件，选择 Customers、Orders 和 Order Details。单击 Finish，就可以在 labeledModel1.edmx 窗口中看到实体数据对象的图形，如图 24-1 所示。

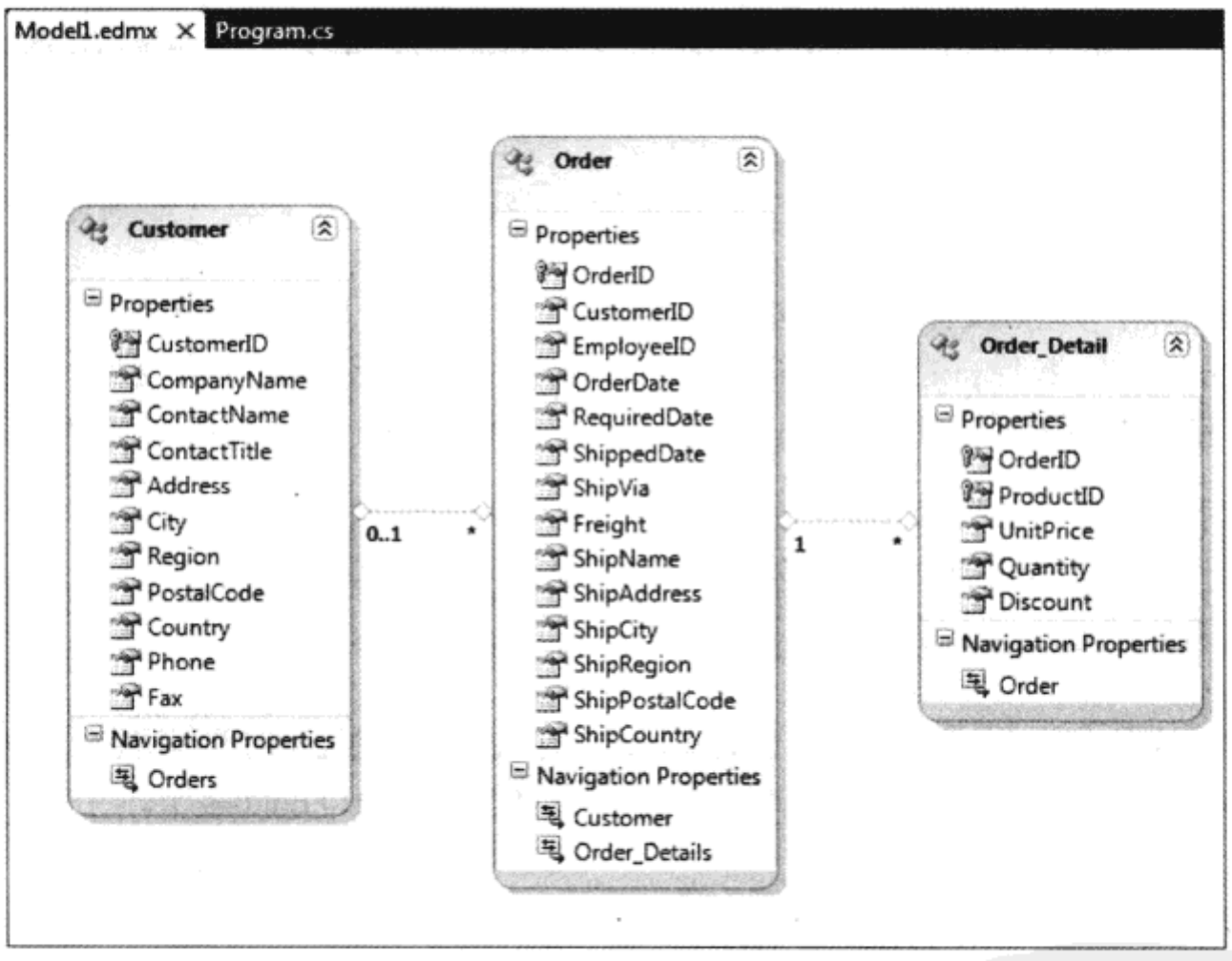


图 24-1

(10) 现在编译项目，这样在下一步开始输入代码时，就可以使用 Customer 对象了。

在 Model1.designer.cs 文件中可以看到为实体模型生成的类的代码，这些代码显示在 Solution Explorer 的 Model1.edmx 源文件的下面，类似于放在<formname>.designer.cs 中的窗体的生成代码。与窗体的生成代码相同，不应修改设计器生成的代码，最好不要在编辑器中打开这些代码，除非要验证类的名称或选择某个生成的数据类型。

(11) 打开主源文件 Program.cs，并在 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
    NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

    var queryResults = from c in northWindEntities.Customers
                       where c.Country == "USA"
                       select new {
                                   ID=c.CustomerID,
                                   Name=c.CompanyName,
                                   City=c.City,
                                   State=c.Region
                                  };
        foreach (var item in queryResults) {
           Console.WriteLine(item);
        };

        Console.WriteLine("Press Enter/Return to continue..");
        Console.ReadLine();
}
```

代码段BegVCSharp\Chapter24\BegVCSharp24_1_FirstLINQtoDatabaseQuery\Program.cs

(12) 编译并运行程序(按下 F5 键即可开始调试)，会看到来自美国的顾客信息，如下所示：

```
{ ID = GREAL, Name = Great Lakes Food Market, City = Eugene, State = OR }
{ ID = HUNGC, Name = Hungry Coyote Import Store, City = Elgin, State = OR }
{ ID = LAZYK, Name = Lazy K Kountry Store, City = Walla Walla, State = WA }
{ ID = LETSS, Name = Let's Stop N Shop, City = San Francisco, State = CA }
{ ID = LONEP, Name = Lonesome Pine Restaurant, Ci.ty = Portland, State = OR }
{ ID = OLDWO, Name = Old World Delicatessen, City = Anchorage, State = AK }
{ ID = RATTC, Name = Rattlesnake Canyon Grocery, City = Albuquerque, State = NM}
{ ID = SAVEA, Name = Save-a-lot Markets, City = Boise, State = ID }
{ ID = SPLIR, Name = Split Rail Beer & Ale, City = Lander, State = WY }
{ ID = THEBI, Name = The Big Cheese, City = Portland, State = OR }
{ ID = THECR, Name = The Cracker Box, City = Butte, State = MT }
{ ID = TRAIH, Name = Trail's Head Gourmet Provisioners, City = Kirkland,
  State = WA }
{ ID = WHITC, Name = White Clover Markets, City = Seattle, State = WA }
Press Enter/Return to continue...
```

按下回车键，结束程序，关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能)，就需要按下回车键两次，这会结束程序的运行。下面详细解释一下该示例。

示例的说明

这个示例和本章的所有其他示例的代码都类似于第 23 章介绍 LINQ 时使用的示例。其代码都使用了 System.Linq 名称空间中的扩展类，在创建项目时，Visual C# 2010 会自动插入一个 using 语句，以引用这个名称空间：

```
using System.Linq;
```

使用 LINQ to Entities 类的第一步是为要访问的数据库创建 ObjectContext 对象的一个实例，ObjectContext 是在数据源中创建的.edmx 文件的编译类。这个对象是数据库的网关，提供了在程序

中控制它的所有方法。它还是创建业务对象的工厂，业务对象对应于数据库中存储的概念化实体(例如，顾客和产品)。

在项目中，数据上下文类称为 NORTHWNDEntities，从 Model1.edmx 文件中编译。Main()方法的第一个操作是创建 NORTHWNDEntities 的一个实例，如下所示：

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

在 Choose Your Database Objects 窗格中选择 Customers 表时，会把 Customer 对象添加到 Model1.edmx 的 LINQ to Entities 类中，并把 Customers 成员添加到 northWindDataEntities 对象中，以查询 Northwind 数据库中的 Customer 对象。

实际的 LINQ 查询语句把 northWindEntities 的 Customers 成员用作数据源，建立查询：

```
var queryResults = from c in northWindEntities.Customers
                   where c.Country == "USA"
                   select new {
                              ID=c.CustomerID,
                               Name=c.CompanyName,
                               City=c.City,
                               State=c.Region
                          };
```

Customers 是一个类型化的 LINQ 表(System.Data.Linq.Table<Customer>)，类似于 Customer 对象的类型化集合(类似于 List<Table>)，但它是为 LINQ to SQL 实现的，并从数据库中自动填充。它实现了 IEnumerable/IQueryable 接口，所以可以用作 from 子句中的 LINQ 数据源，这与其他集合或数组一样。

where 子句把结果限制为来自美国的顾客。select 子句是一个投影，类似于第 23 章开发的示例，它也创建了一个新对象，其成员有 ID、Name、City 和 State。由于该结果是来自美国的顾客，所以可以把 Region 重命名为 State，以便更准确地显示结果。最后创建一个标准的 foreach 循环，这与第 23 章编写的 foreach 循环类似：

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

这段代码为每个 item 使用默认生成的 ToString()方法，以便格式化 Console.WriteLine(item)的输出，因此每个投影的成员示例都在花括号中显示其值：

```
{ ID=WHITC, Name=White Clover Markets, City=Seattle, State=WA }
```

最后，示例暂停显示，以便查看结果：

```
        Console.WriteLine("Press Enter/Return to continue... ");
        Console.ReadLine();
 };
```

我们创建了一个基本的 LINQ to SQL 查询，可以将它为基础，来创建更复杂的查询。

24.5　浏览数据库关系

ADO.NET Entity Framework 最强大的一个方面是可以自动创建 LINQ to SQL 对象，以便浏览数据库中相关表之间的关系。下面的示例要给 LINQ to Entities 类添加一个相关表，再添加代码，以浏览数据库中的相关数据对象，并输出它们的值。

试一试：浏览 LINQ to Entities 关系

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中修改上一个示例的项目 BegVCSharp_24_1_FirstLINQtoDataQuery，如下面的步骤所示。

(2) 打开主源文件 Program.cs。在 Main()方法中把 Orders 字段添加到 LINQ 查询的 select 子句中(别忘了在 c.Region 的后面添加一个逗号，把所添加的字段与列表的其他内容分开)：

```
static void Main(string[] args)
{
   NorthwindDataContext northWindDataContext = new NorthwindDataContext();

   var queryResults = from c in northWindDataContext.Customers
                      where c.Country == "USA"
                      select new {
                                 ID=c.CustomerID,
                                 Name=c.CompanyName,
                                 City=c.City,
                                 State=c.Region,
                                 Orders=c.Orders
                      };
```

(3) 修改 foreach 子句，输出查询结果，如下所示：

可从 wrox.com 下载源代码

```
        foreach (var item in queryResults) {

           Console.WriteLine(
              "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
                 item.Name, item.City, item.State, item.Orders.Count
           );
           foreach (Order o in item.Orders) {
              Console.WriteLine("\t\t{0}\t{1}", o.OrderID, o.OrderDate);
           }

        };

        Console.WriteLine("Press Enter/Return to continue... ");
        Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp_24_2_NavigatingDatabaseRelationships\Program.cs

(4) 编译并执行程序(按下 F5 键即可开始调试)，会看到来自美国的顾客及其订单信息，如下所

示(这是输出的最后一部分，第一部分已滚动出了控制台窗口的顶部)。

```
Customer: Trail's Head Gourmet Provisioners Kirkland, WA
3 orders:     Order ID          Order Date
              10574 6/19/1997   12:00:00 AM
              10577 6/23/1997   12:00:00 AM
              10822 1/8/1998    12:00:00 AM
Customer: White Clover Markets Seattle, WA
14 orders:    Order ID          Order Date
              10269 7/31/1996   12:00:00 AM
              10344 11/1/1996   12:00:00 AM
              10469 3/10/1997   12:00:00 AM
              10483 3/24/1997   12:00:00 AM
              10504 4/11/1997   12:00:00 AM
              10596 7/11/1997   12:00:00 AM
              10693 10/6/1997   12:00:00 AM
              10696 10/8/1997   12:00:00 AM
              10723 10/30/1997  12:00:00 AM
              10740 11/13/1997  12:00:00 AM
              10861 1/30/1998   12:00:00 AM
              10904 2/24/1998   12:00:00 AM
              11032 4/17/1998   12:00:00 AM
              11066 5/1/1998    12:00:00 AM
Press Enter/Return to continue...
```

与前面一样，按下回车键，结束程序，关闭控制台屏幕。

示例的说明

我们修改了前面的示例，而不是从头开始创建新程序，因此不需要重复创建数据源文件Model1.edmx 的步骤(注意示例代码中包含几个项目，每个项目都有自己的 Model1.edmx 实例)。

在 Choose Your Data Objects 对话框中选择 Orders 表，把 Order 类添加到 Model1.edmx 源文件中，以表示 Northwind 数据库映射中的 Orders 表。

Visual Studio 2010 检测到数据库中 Customers 和 Orders 表之间的关系，所以在 Customer 类中还添加了一个 Orders 集合成员，来表示该关系。所有这些都是自动完成的，这与给窗体添加新控件一样。

接着，在查询的 select 子句中添加新的 Orders 成员：

```
select new {
           ID=c.CustomerID,
           Name=c.CompanyName,
           City=c.City,
           State=c.Region,
           Orders=c.Orders
           };
```

Orders 是一种特殊的类型化 LINQ 集(System.Data.Linq.EntitySet<Order>)，表示关系数据库中两个表之间的关系。它实现了 IEnumerable/IQueryable 接口，所以可以用作 LINQ 数据源，或者用 foreach 语句迭代，这与其他集合或数组一样。

与上一个示例中的 Table 对象一样，EntitySet 类似于类型化的 Order 对象集合(类似于 List<Order>)，但只有顾客提交的订单才会显示在该 Customer 实例的 EntitySet 成员中。

Customer 的 EntitySet 成员中的 Order 对象对应于数据库中顾客 ID 与该 Customer 实例 ID 相同的订单行。

浏览关系只需建立一个嵌套的 foreach 语句，迭代每个顾客以及每个顾客的订单即可：

```
foreach (var item in queryResults) {

            Console.WriteLine(
               "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
                  item.Name, item.City, item.State, item.Orders.Count
            );
            foreach (Order o in item.Orders) {
                Console.WriteLine("\t\t{0}\t{1}", o.OrderID, o.OrderDate);
            }
};
```

这里不使用默认的 ToString()进行格式化，而是把输出结果格式化为更容易理解的形式，这样可以在每个顾客的下面显示订单列表，正确列出其层次。格式化字符串“Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date”为每个顾客在第一行设置了姓名、城市和州等占位符，接着在下一行为该顾客的订单输出一个列标题。这里使用 LINQ 聚合方法 Count()输出顾客的订单数，再在嵌套的 foreach 语句中为每一行输出订单 ID 和订单日期。

```
Customer: White Clover Markets Seattle, WA
14 orders:     Order ID Order Date
              10269 7/31/1996 12:00:00 AM
              10344 11/1/1996 12:00:00 AM
```

格式仍有待改进，因为我们希望仅显示订单的日期，而这里显示了时间。

成功查询了数据库后，下面尝试另一种数据源 XML。

24.6　使用 LINQ to XML

LINQ to XML 并不打算替代标准的 XML API，例如，XML DOM(Document Object Model)、XPath、Xquery 和 XSLT 等。如果熟悉这些 API 或当前需要使用或学习它们，可以继续使用或学习。

LINQ to XML 补充了这些标准 XML 类，更便于使用 XML。LINQ to XML 为创建和查询 XML 数据提供了额外的选项，代码更简单，开发许多常见的情形时速度更快，如果已经在其他程序中使用了 LINQ，开发速度将会更快。

24.7　LINQ to XML 函数构造方法

如前面章节所述，C#中的一个主题是利用对象初始化器和匿名类型等功能更方便地构建对象。LINQ to XML 针对这个主题引入了一种更便捷的新方式，叫作函数构建方式(functional construction)，通过这种方式可以创建 XML 文档。在这种方式中，构造函数的调用可以用反映 XML 文档结构的方式嵌套。下面的示例就使用函数构建方式建立了一个包含顾客和订单的简单 XML 文档。

试一试：LINQ to XML 构造函数

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp_24_3_Linq-ToXmlConstructors。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 的开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

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

(4) 在 Program.cs 的 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
            XDocument xdoc = new XDocument(
                new XElement("customers",
                     new XElement("customer",
                          new XAttribute("ID" ,"A"),
                          new XAttribute("City","New York"),
                          new XAttribute("Region","North America"),
                          new XElement("order",
                               new XAttribute("Item","Widget"),
                               new XAttribute("Price",100)
                          ),
                          new XElement("order",
                               new XAttribute("Item","Tire"),
                               new XAttribute("Price",200)
                          )
                     ),
                     new XElement("customer",
                          new XAttribute("ID","B"),
                          new XAttribute("City","Mumbai"),
                          new XAttribute("Region", "Asia"),
                          new XElement("order",
                               new XAttribute("Item","Oven"),
                               new XAttribute("Price",501)
                          )
                     )
                )
            );
            Console.WriteLine(xdoc);

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

代码段 BegVCSharp\Chapter24\BegVCSharp_24_3_ LinqToXmlConstructors\Program.cs

(5) 编译并执行程序(按下 F5 键即可开始调试)，输出结果如下所示：

```
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
<customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven"Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

输出屏幕上显示的 XML 文档包含前面示例中顾客/订单数据的一个简化版本。注意，XML 文档的根元素是<customers>，它包含两个嵌套的<customer>元素，这两个元素又包含许多嵌套的<order>元素。<customer>元素有两个特性：<City>和<Region>，<order>元素也有两个特性：<Item>和<Price>。

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

示例的说明

第一步是引用 System.Xml.Linq 名称空间。本章的所有示例都要求把这行代码添加到程序中：

```
using System.Xml.Linq;
```

在创建项目时，System.Linq 名称空间是默认包含的，但不包含 System.Xml.Linq 名称空间，所以必须显式地添加这行代码。

接着调用 LINQ to XML 构造函数 XDocument()、XElement()和 XAttribute()，它们彼此嵌套，如下所示：

```
XDocument xdoc = new XDocument(
   new XElement("customers",
       new XElement("customer",
           new XAttribute("ID","A"),
           .
```

注意，这些代码看起来类似于 XML 本身，即文档包含元素，每个元素又包含特性和其他元素。下面依次分析这些构造函数：

- XDocument()：在 LINQ to XML 构造函数层次结构中，最高层的对象是 XDocument()，它表示完整的 XML 文档，在代码中如下所示：

```
static void Main(string[] args)
{
          XDocument xdoc = new XDocument(
.
          );
```

在前面的代码中，省略了 XDocument()的参数列表，因此可以看到 XDocument()调用在何处开头和结尾。与所有 LINQ to XML 构造函数一样，XDocument()也把对象数组(object[])作为它的参数，以便给它传送其他构造函数创建的其他对象。在这个程序中调用的所有其他构造函数都是

XDocument()构造函数的参数。这个程序传送的第一个也是唯一一个参数是 XElement()构造函数。

- XElement()：XML 文档必须有一个根元素，所以大多数情况下，XDocument()的参数列表都以一个 XElement 对象开头。XElement()构造函数把元素名作为字符串，其后是包含在该元素中的一个XML 对象列表。本例中的根元素是 customers，它又包含一个 customer 元素列表：

```
new XElement("customers",
      new XElement("customer",

       ),

)
```

customer 元素不包含其他 XML 元素，只包含 3 个 XML 特性，它们用 XAttribute()构造函数构建。

- XAttribute()：这里给 customer 元素添加了 3 个 XML 特性：ID、City 和 Region：

```
new XAttribute("ID", "A"),
new XAttribute("City","New York"),
new XAttribute("Region","North America"),
```

根据定义，XML 特性是一个 XML 叶节点，它不包含其他 XML 节点，所以 XAttribute()构造函数的参数只有特性的名称和值。本例中生成的 3 个特性是 ID="A"、City="New York"和 Region="North America"。

- 其他 LINQ to XML 构造函数：这个程序中没有调用它们，但所有的 XML 节点类型都有其他 LINQ to XML 构造函数，例如，XDeclaration()用于 XML 文档开头的 XML 声明，XComment()用于 XML 注释等。这些构造函数不太常用，但如果需要它们来精确控制 XML 文档的格式化，就可以使用它们。

下面继续解释第一个示例：在 customer 元素的 ID、City 和 Region 特性后面再添加两个子 order 元素：

```
new XElement("order",
    new XAttribute("Item", "Widget"),
    new XAttribute("Price", 100)
),
new XElement("order",
    new XAttribute("Item", "Tire"),
    new XAttribute("Price", 200)
)
```

这些 order 元素都有两个特性：Item 和 Price，但没有子元素。

接着将 XDocument 的内容显示在控制台屏幕上：

```
Console.WriteLine(xdoc);
```

这行代码使用 XDocument()的默认方法 ToString()输出 XML 文档的文本。

最后暂停屏幕，以查看控制台输出，再等待用户按下回车键：

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

程序退出 Main()方法后，就结束程序。

用字符串构建 XML 元素的文本

刚才执行的示例格式化了 XML，其元素中没有文本内容。XML 常常包含文本内容，使用 LINQ to XML 构造函数 XElement()很容易格式化 XML 元素。例如，为了不把<customer>元素的 ID 文本建立为特性，只需把一个字符串作为构造函数 XElement()的参数传入，而不是传入嵌套的 XAttribute：

```
XDocument xdoc = new XDocument(
     new XElement("customers",
         new XElement("customer",
         "AAAAAA",
         new XAttribute("City","New York"),
         new XAttribute("Region","North America")
     ),
     new XElement("customer",
          "BBBBBB",
          new XAttribute("City","Mumbai"),
          new XAttribute("Region","Asia")
     )
);
```

这会生成如下所示的 XML 文档：

```
<customers>
  <customer City="New York"Region="North America"> AAAAAA </customer>
  <customer City="Mumbai"Region="Asia"> BBBBBB </customer>
</customers>
```

构造函数 XElement()把参数列表中的所有字符串联起来，放在元素的文本部分。

24.8　保存和加载 XML 文档

注意，用 Console.WriteLine()把 XML 文档显示到控制台屏幕上时，不会显示以<?xml version="1.0" 开头的 XML 声明。可以用 XDeclaration()构造函数显式地创建这样一个声明，但一般不需要这么做，因为在用 LINQ to XML 方法 Save()把 XML 文档保存到文件中时，会自动创建该声明。

另外，在程序中构建 XML 文档有助于理解构造函数的工作方式，但我们不经常这么做，而是从外部的数据源(如文件)中加载 XML 文档。

下面的示例将试验这两个操作。

试一试：保存和加载 XML 文档

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中修改上一个示例或者创建一个新的控制台应用程序 BegVCSharp_24_4_SaveLoadXML。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 的开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

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

```
using System.Xml.Linq;
using System.Text;
```

如果正在修改上一个示例，则已经引用了这个名称空间。

(4) 把上一个示例中的 XML 文档构造函数及其嵌套的 XML 元素和特性调用添加到 Program.cs 的 Main()方法中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
            XDocument xdoc = new XDocument(
                new XElement("customers",
                    new XElement("customer",
                        new XAttribute("ID","A"),
                        new XAttribute("City","New York"),
                        new XAttribute("Region","North America"),
                        new XElement("order",
                            new XAttribute("Item","Widget"),
                            new XAttribute("Price", 100)
                        ),
                        new XElement("order",
                            new XAttribute("Item","Tire"),
                            new XAttribute("Price", 200)
                        )
                    ),

                    new XElement("customer",
                        new XAttribute("ID","B"),
                        new XAttribute("City","Mumbai"),
                        new XAttribute("Region","Asia"),
                        new XElement("order",
                            new XAttribute("Item","Oven"),
                            new XAttribute("Price", 501)
                        )
                    )
                )
            );
```

代码段 BegVCSharp\Chapter24\BegVCSharp_24_4_ SaveLoadXml\Program.cs

(5) 在上一步中添加了 XML 文档构造函数代码后，在 Program.cs 的 Main()方法最后添加下面的代码，以便保存、加载和显示 XML 文档：

```
    string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";

    xdoc.Save(xmlFileName);

    XDocument xdoc2 = XDocument.Load(xmlFileName);

    Console.WriteLine("Contents of xdoc2: ");
    Console.WriteLine(xdoc2);

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

(6) 编译并执行程序(按下 F5 键即可开始调试)，控制台窗口中的输出结果如下所示：

```
Contents of xdoc2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget"Price="100"/>
    <order Item="Tire"Price="200"/>
  </customer>
  <customer ID="B"City="Mumbai"Region="Asia">
   <order Item="Oven"Price="501"/>
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

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

示例的说明

与前面一样，第(1)步也是引用 System.Xml.Linq 名称空间。接着是对 LINQ to XML 构造函数 XDocument()、XElement()和 XAttribute()的嵌套调用。请参阅第一个示例中对这部分和其他重复代码的说明。

创建了 XDocument()对象后，把文件名指定为一个字符串，调用 Save()方法把 XML 文档保存到文件中：

```
string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";
xdoc.Save(xmlFileName);
```

在这个例子中，XML 文档保存到指定的文件中，Save()方法还有一些重载版本，可以保存到 System.IO.TextWriter 或 System.Xml.XmlWriter 中，如果要编写另一个程序，且该程序已经使用其中的一个类写入到文件，就可以使用这些重载版本。

Save()方法的一个重载版本还可以指定 SaveOptions，以便禁用格式化功能(XML 文档保存时，默认带有缩进和空白，使文档看起来很“美观”)。

把文档保存到文件中后，就把它加载到一个新的 XDocument 实例 xdoc2 中：

```
XDocument xdoc2 = XDocument.Load(xmlFileName);
```

XDocument.Load()方法是静态的，因为它是一个工厂类型的方法，可以创建 XDocument 的新实例；使用它可以加载由另一个完全不同的程序创建的文档。

之后显示文档，这与前面一样，只是这次使用从文件中加载的 xdoc2 实例。程序的剩余部分与上一个示例中的相同：

```
Console.WriteLine("Contents of xdoc2: ");
Console.WriteLine(xdoc2);

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

24.8.1 从字符串中加载 XML

有时并不是从文件中加载XML，而是通过一个或多个方法接收从另一个应用程序发送的字符串形式的 XML。在 LINQ to XML 中，可以使用 Parse()方法通过字符串创建 XML 文档：

```
XDocument xdoc = XDocument.Parse(@"
          <customers>
               <customer ID=" "A" " City=" "New York" " Region=" "North America" ">
                      <order Item=" "Widget" " Price=" "100" " />
                      <order Item=" "Tire" " Price=" "200" " />
                    </customer >
                   <customer ID=" "B" " City=" "Mumbai" "Region=" "Asia" ">
                     <order Item=" "Oven" " Price=" "501" " />
                   </customer>
          </customers>
          ");
```

这段代码生成的结果与从文件中加载文档一样。Parse()方法与 Load()方法一样，也是一个类级方法，可以创建 XDocument 的新实例。在调用 Parse()方法之前，不需要创建新的 XDocument 对象。

在前面的示例中，XML 的字符串字面量带双引号，但在字符串的实际内容中，并没有使用双引号。双引号只能用于在@引用的字符串字面量中包含引号。

24.8.2 已保存的 XML 文档内容

用 IE 打开刚才在上一个示例中保存的文档。在地址栏中指定完整的路径名 C:\BegVCSharp\Chapter24\Xml\example2.xml。

注意，XML 文档声明<?xml version="1.0" encoding="utf-8"?>位于所保存文档的开头，但用 Console.Writeline()把 XDocument 对象显示在屏幕上时，该声明不会显示。使用 LINQ to XML 提供的默认功能时，不需要担心声明和许多其他 XML 细节。

在 Windows 中，XML 文档的默认编码方式是 UTF-8(8 位 Unicode Transformation Format)。不应改变这种编码方式，除非在特别不寻常的情况下，例如，创建一个 ASCII 编码的 XML 文档，由不理解 UTF-8 的旧程序使用。此时，可以添加一个 XDeclaration()对象，在 XDocument()构造函数的参数列表开头，把编码方式设置为 ASCII，或者设置 XDocument 的 Declaration 属性：

```
xdoc.Declaration = new XDeclaration("1.0","us-ascii","yes");
```

24.9 处理 XML 片段

与一些 XML API 不同，LINQ to XML 处理 XML 片段(部分或不完整的 XML 文档)的方式与处理完整的 XML 文档几乎完全相同。在处理片段时，只是把 XElement(而不是 XDocument)当作顶级

XML 对象。

唯一的限制是不能添加某些比较深奥的 XML 节点类型，XML 文档只能把这些节点类型应用于 XML 文档或 XML 片断，例如，XComment 应用于 XML 注释，XDeclaration 应用于 XML 文档声明，XProcessingInstruction 用于 XML 处理指令。

下面的示例会加载、保存和处理 XML 元素及其子节点，这与处理 XML 文档一样。

试一试：处理 XML 片段

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中修改上一个示例或者创建一个新的控制台应用程序 BegVCSharp_24_5_XMLFragments。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

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

如果正在修改上一个示例，则已经引用了这个名称空间。

(4) 把上一个示例中的 XML 元素(不包含 XML 文档构造函数)添加到 Program.cs 的 Main()方法中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
            XElement xcust =
                new XElement("customers",
                    new XElement("customer",
                        new XAttribute("ID", "A"),
                        new XAttribute("City", "New York"),
                        new XAttribute("Region", "North America"),
                        new XElement("order",
                            new XAttribute("Item", "Widget"),
                            new XAttribute("Price", 100)
                      ),
                      new XElement("order",
                        new XAttribute("Item", "Tire"),
                        new XAttribute("Price", 200)
                      )
                    ),
                    new XElement("customer",
                        new XAttribute("ID", "B"),
                        new XAttribute("City", "Mumbai"),
                        new XAttribute("Region", "Asia"),
                        new XElement("order",
                            new XAttribute("Item", "Oven"),
```

```
                    new XAttribute("Price", 501)
                )
            )
        )
    ;
```

代码段 BegVCSharp\Chapter24\BegVCSharp_24_5_ XmlFragments\Program.cs

(5) 在上一步添加了XML 元素构造函数代码后，添加下面的代码，以便保存、加载和显示 XML 元素：

```
        string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example3.xml";
        xcust.Save(xmlFileName);

        XElement xcust2 = XElement.Load(xmlFileName);

        Console.WriteLine("Contents of xcust: ");
        Console.WriteLine(xcust);

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

(6) 编译并执行程序(按下 F5 键即可开始调试)，控制台窗口中的输出结果如下所示：

```
Contents of XElement xcust2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

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

示例的说明

XElement 和 Xdocument 都继承自 LINQ to XML 类 XContainer，它实现了一个可以包含其他 XML 节点的 XML 节点。这两个类都实现了Load()和 Save()方法，因此，可以在 LINQ to XML 的 XDocument 上执行的大多数操作都可以在 XElement 实例及其子元素上执行。

这里只创建了一个 XElement 实例，它的结构与前面示例中的 XDocument 相同，但不包含 XDocument。这个程序的所有操作处理的都是 XElement 片段。

XElement 还支持 Load()和 Parse()方法，可以分别从文件和字符串中加载 XML。

24.10　从数据库中生成 XML

XML 常常用于在客户机和服务器之间交流数据，或者在多层应用程序的不同层之间交流数据。在数据库中查询某些数据，再根据这些数据生成 XML 文档或片段，传送给另一层是很常见的。下面的示例就创建一个查询，在 Northwind 示例数据库中查找某些数据，用 LINQ to SQL 查询数据，再用 LINQ to XML 类把数据转换为 XML。

试一试：从数据库生成 XML

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp_24_6_XML-fromDatabase。

(2) 按照本章开头的“第一个 LINQ 数据库查询”示例，在项目中添加一个新数据源 Model1.edmx，再添加与 Northwind 示例数据库的连接。

(3) 编译项目，这样在 Model1.edmx 中定义的类和属性就可以在后面编辑代码时通过 IntelliSense 使用了。

(4) 打开主源文件 Program.cs。

(5) 在 Program.cs 开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

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

(6) 把下面的代码添加到 Program.cs 的 Main()方法中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            XElement northwindCustomerOrders =
              new XElement("customers",
                from c in northWindEntities.Customers.AsEnumerable()
                select new XElement("customer",
                   new XAttribute("ID", c.CustomerID),
                   new XAttribute("City", c.City),
                   new XAttribute("Company", c.CompanyName),
                    from o in c.Orders
                    select new XElement("order",
                       new XAttribute("orderID", o.OrderID),
                       new XAttribute("orderDay",
                              o.OrderDate.Value.Day),
                       new XAttribute("orderMonth",
                              o.OrderDate.Value.Month),
                       new XAttribute("orderYear",
                              o.OrderDate.Value.Year),
                       new XAttribute("orderTotal",
                            o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
                     ) //end order
```

```
        ) // end customer
); // end customers

string xmlFileName =
          @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
northwindCustomerOrders.Save(xmlFileName);

Console.WriteLine("Successfully saved Northwind customer orders to:");
Console.WriteLine(xmlFileName);
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp_24_6_ XMLfromDatabase\Program.cs

(7) 编译并执行程序(按下 F5 键即可开始调试)，结果如下所示：

```
Successfully saved Northwind customer orders to:
C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml
Program finished, press Enter/Return to continue:
```

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

示例的说明

在 Program.cs 中添加了对 System.Xml.Linq 名称空间的引用，所以可以调用 LINQ to XML 构造函数类。

如本章第一部分所述，我们为 Northwind 示例数据库创建了一个数据源，再使用 VS2010 为 Northwind 数据创建一个 LINQ to Entities 对象模型。在主程序中，创建了一个 Northwind 数据上下文类的实例，以使用如下映射：

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

LINQ to Entities 查询把 Northwind 数据上下文对象的 Customers 成员作为数据源，通过 Customers、Orders 和 Order Details 表生成包含了所有顾客订单的列表。但是，由于 LINQ to Entities 查询的延迟执行，我们使用 Customer 对象上的 AsEnumerable()方法把中间结果转换为内存中的 LINQ to Object 可枚举类型。最后，查询结果被投影到查询的 Select 子句中，成为一组嵌套的 LINQ to XML 元素和特性：

```
XElement northwindCustomerOrders =
    new XElement("customers",
      from c in northWindDataContext.Customers.AsEnumerable()
      select new XElement("customer",
         new XAttribute("ID", c.CustomerID),
         new XAttribute("City", c.City),
         new XAttribute("Company", c.CompanyName),
         from o in c.Orders
           select new XElement("order",
              new XAttribute("orderID", o.OrderID),
              new XAttribute("orderDay",
                      o.OrderDate.Value.Day),
```

```
                new XAttribute("orderMonth",
                        o.OrderDate.Value.Month),
                new XAttribute("orderYear",
                        o.OrderDate.Value.Year),
                new XAttribute("orderTotal",
                    o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
            ) //end order
        ) // end customer
    ); // end customers
```

为了提取顾客的所有订单，在第一个 LINQ 查询(from c in northwindDataContext. Customers...)中嵌套使用了第二个查询(from o in c.Orders...)。

把 OrderDate 字段分为月份、日期和年份部分，可以使 XML 更容易查询。下一个示例将说明如何完成这项任务。

最后，把生成的 XML 保存到文件中，这与上一个示例相同：

```
string xmlFileName =
            @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
northwindCustomerOrders.Save(xmlFileName);

Console.WriteLine("Successfully saved Northwind customer orders to:");
Console.WriteLine(xmlFileName);
```

下面准备针对刚才写入磁盘的 XML 文件编写一个查询。

24.11　查询 XML 文档的方法

为什么需要在 XML 文档上执行 LINQ 查询？如果程序接收由另一个程序生成的 XML，就需要在该 XML 中查找特定的 XML 元素或特性，来确定如何处理它。程序可能只关注 XML 元素的一个子集，也可能需要对文档中的元素计数，或者搜索满足某个条件的元素或特性。LINQ 查询是针对这类情况的强大解决方案。

为了查询 XML 文档，LINQ to XML 类(如 XDocument 和 XElement)提供了成员属性和方法，它们返回 XML 文档或片段中可以使用 LINQ 查询的、由该 LINQ to XML 类表示的 LINQ to XML 对象集合。

下面的示例在前面示例创建的文档上使用了这些可查询的成员方法和属性。

试一试：查询 XML 文档

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp_24_7_QueryXML。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 的开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

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

```
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

(4) 把下面的代码添加到 Program.cs 的 Main()方法中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   string xmlFileName= @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
   XDocument customers = XDocument.Load(xmlFileName);

   Console.WriteLine("Elements in loaded document: ");
   var queryResult = from c in customers.Elements()
                     select c.Name;
   foreach (var item in queryResult)
   {
      Console.WriteLine(item);
   }
   Console.Write("Press Enter/Return to continue: ");
   Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp_24_7_QueryXML\Program.cs

(5) 编译并执行程序(按下 F5 键即可开始调试)，结果如下所示：

```
Elements in loaded document:
customers
Press Enter/Return to continue:
```

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

示例的说明

在读取每个查询方法的说明时，修改了刚才创建的 LINQ to XML 查询示例，以使用它。每个查询都返回包含 Name 属性的LINQ to XML 元素或特性对象集合，所以 select 子句仅返回这个 Name 属性，并在 foreach 循环中输出它：

```
var queryResult = from c in customers.Elements()
                  select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

在开发或调试程序时，可以首先使用此类代码来查看查询返回了什么结果。后面将修改输出，以便显示对终端用户更有意义的业务结果。

24.12 使用 LINQ to XML 查询成员

本节介绍可用的LINQ to XML 查询成员，然后把 NorthwindCustomerOrders.xml 文件作为数据源，

依次试验它们。

24.12.1　Elements()

第一个 LINQ to XML 查询方法是 XDocument 类的 Elements()成员，它也可以用于 XElement 类。

Elements()返回 XML 文档或片段中的所有第一级元素。对于有效的 XML 文档，例如，刚才创建的 NorthwindCustomerOrders.xml 文件，只有一个第一级元素，即根元素 customers：

```
< ?xml version="1.0" encoding="utf-8" ? >
< customers >
  .
< /customers >
```

其他元素都是 customers 的子元素，所以 Elements()只返回一个元素：

```
Elements in loaded document:
customers
```

XML 片段可能包含多个第一级元素，但查询子元素通常更有用，这需要使用下一节介绍的 Descendants()成员。

24.12.2　Descendants()

下一个 LINQ to XML 查询方法是 XDocument 类的 Descendants()成员，它也可以用于 XElement 类。

Descendants()返回 XML 文档或片段中的所有子元素(所有级别的子元素)。修改示例 BegVCSharp_24_7-QueryXML，如下所示：

```
Console.WriteLine("All descendants in document: ");
queryResult =
   from c in customers.Descendants()
   select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

编译并执行上述代码，会发现 customer 和 order 元素名以文档中的顺序重复出现：

```
All descendants in document:
customer
order
order
.
customer
order
.
customer
order
.
order
order
Press Enter/Return to continue:
```

这个输出滚动出了屏幕，所以可能看不到第一部分。这说明，NorthwindCustomerOrders.xml 文件只在根元素 customers 下包含 customer 和 order 元素：

```
<?xml version="1.0" encoding="utf-8" ?>
<customers >
 <customer . [{[SPACE]}]>
   <order . />
   <order . />
   .
 </customer>
 <customer ...>
   <order . />
   .
```

在结果上添加 LINQ 运算符 Distinct()，这样会更便于管理输出：

```
Console.WriteLine("All distinct descendants in document: ");
var queryResult =
   from c in customers.Descendants()
   select c.Name;
foreach (var item in queryResult.Distinct())
```

结果，列表中只包含不同的元素名：

```
All distinct descendants in document:
customers
customer
order
Press Enter/Return to continue:
```

第一次处理文档时，这非常有助于研究文档的结构。但查找所有元素并不是常常需要在已完成的应用程序中解决的问题。

一种常见的情形是查找具有指定名称的子级元素。Descendants()方法的一个重载版本把需要的元素名作为字符串参数，如下所示：

```
Console.WriteLine("Descendants named 'customer': ");
 var queryResult =
    from c in customers.Descendants("customer")
    select c.Name;
 foreach (var item in queryResult) // remove Distinct()
 {
    Console.WriteLine(item);
 }
```

这会返回 customer 元素：

```
Descendants named 'customer':
customer
customer
customer
.
customer
customer
Press Enter/Return to continue:
```

显然，这是一个更加通用的查询。查询指定类型的一组元素后，就可以搜索特定的特性，如下

所示。

为了完整起见，LINQ to XML 还提供了一个Ancestors()方法，它与Descendants()方法相反，在 XML 文档的树形结构中返回比源元素级别高的所有元素。它不如Descendants()方法常用，因为开发人员常常从根元素开始处理 XML 文档，沿着元素和属性的树形结构向下查找，一直到叶节点一级。Parent 特性指向直接上级节点，比较常用。

24.12.3　Attributes()

下一个 LINQ to XML 查询方法是 Attributes()成员，它返回当前选中元素的所有特性。为了说明它的工作方式，修改 BegVCSharp_24_7_QueryXML 示例，如下所示：

```
Console.WriteLine("Attributes of descendants named 'customer': ");
var queryResult =
   from c in customers.Descendants("customer").Attributes()
   select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

编译并执行代码，会看到 customer 元素的特性名：

```
Attributes of descendants named 'customer':
ID
City
Company
ID
City
Company
.
ID
City
Company
ID
City
Company
Press Enter/Return to continue:
```

这个输出再次滚动出了屏幕。这个查询找出了customer 元素的特性名：

```
< customer ID= . City= . Company= . >
< customer ID= . City= . Company= . >
< customer ID= . City= . Company= . >
.
```

与 Descendants()方法一样，也可以给Attributes()传送要搜索的具体名称。另外，不仅可以显示名称，还可以显示特性本身。下面的查询显示顾客 Company 的特性：

```
Console.WriteLine("customer attributes named 'Company': ");
```

```
var queryResult =
   from c in customers.Descendants("customer").Attributes("Company")
   select c;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

编译并运行代码，就会看到 customer 元素的 Company 特性：

```
.
Company="Toms Spezialitäten"
Company="Tortuga Restaurante"
Company="Tradiçao Hipermercados"
Company="Trail's Head Gourmet Provisioners"
Company="Vaffeljernet"
Company="Victuailles en stock"
Company="Vins et alcools Chevalier"
Company="Die Wandernde Kuh"
Company="Wartian Herkku"
Company="Wellington Importadora"
Company="White Clover Markets"
Company="Wilman Kala"
Company="Wolski Zajazd"
Press Enter/Return to continue:
```

下面是另一个示例，这次显示 orders 元素和 orderYear 特性：

```
Console.WriteLine("order attributes named 'orderYear': ");
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
    select c;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

编译并执行，结果如下：

```
.
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1996"
orderYear="1997"
orderYear="1997"
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1998"
Press Enter/Return to continue:
```

还可以用 Value 属性获得特性的值(这里显示年份)：

```
Console.WriteLine("Values of order attributes named 'orderYear': ");
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
```

```
    select c.Value;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}
```

编译并运行，结果如下：

```
.
1996
1997
1997
1998
1998
1998
1998
Press Enter/Return to continue:
```

现在可以回答一个问题：订单最早是在哪一年下的？这可以使用相同的查询，但需要在结果上应用 Min()聚合运算符，而不使用常用的 foreach 循环：

```
var queryResult =
      from c in customers.Descendants("order").Attributes("orderYear")
      select c.Value;
Console.WriteLine("Earliest year in which orders were placed: {0}",
                                                    queryResults.Min());
```

编译并执行，结果是 1996 年：

```
Earliest year in which orders were placed: 1996
Press Enter/Return to continue:
```

本章的练习要求回答指定的问题。

24.13　小结

至此，就完成了对 LINQ to XML 的介绍。LINQ to XML 集成了 LINQ 的概念和易用的 XML API，把 XML 快速集成到其他使用 LINQ 的其他程序中。因此，对于熟悉 LINQ 的其他形式的程序员而言，XML 文档上的查询非常简单、自然。

本章学习了如何用 LINQ to XML 函数构建方式构建 XML 文档，如何用 LINQ to XML 加载和保存 XML 文档。

我们掌握了用 LINQ to XML 处理不完整的 XML 文档(片段)，学习了如何从 LINQ to SQL 或 LINQ to Objects 查询中生成 XML 文档。

最后探讨了如何用 LINQ to XML 查询已有的 XML 文档，如何在 LINQ to XML 中使用高级 LINQ 特性，如 LINQ 聚合运算符。

24.14　练习

(1) 用 LINQ to XML 构造函数创建下面的 XML 文档：

```
<employees>
 <employee ID="1001" FirstName="Fred" LastName="Lancelot">
  <Skills>
   <Language> C# </Language>
   <Math> Calculus </Math>
  </Skills>
 </employee>
 <employee ID="2002" FirstName="Jerry" LastName="Garcia">
  <Skills>
   <Language> French </Language>
   <Math> Business </Math>
  </Skills>
 </employee>
</employees>
```

(2) 为自己创建的 NorthwindCustomerOrders.xml 文件编写一个查询，查找最早购买产品的顾客(在 Northwind 运行的第一年即 1996 年下订单的顾客)。

(3) 为 NorthwindCustomerOrders.xml 文件编写一个查询，查找单份订单金额超过 10 000 美元的顾客。

(4) 为 NorthwindCustomerOrders.xml 文件编写一个查询，查找购买量最大的顾客，例如，订单总额超过 100 000 美元的公司。

(5) 使用 LINQ to Entities 显示 Northwind 数据库中 Products 和 Employees 表的详细信息。

(6) 创建一个 LINQ to Entities 查询，显示 Northwind 数据库中最畅销的产品。

(7) 创建一个组合查询，按国家显示最畅销的产品。

附录 A 给出了练习答案。

24.15　本章要点

主　题	重 要 概 念
不同的 LINQ 变体	.NET 中每个不同的数据源都有一个 LINQ 变体或“风格”，可用于查询其数据
如何使用 LINQ 查询数据库	使用 Visual C# 2010 中的 Data Source Configuration Wizard(选择 Data | Add New Data Source)，可以为数据库生成 LINQ to Entities 类
如何使用 LINQ 导航数据库关系	LINQ to Entities 类为每个添加到数据源中的相关数据实体(表)包含了可导航的实例成员
如何使用 LINQ 轻松地构建 XML	LINQ to XML 包含非常强大的函数构造方式，用于从任何 LINQ 查询中生成 XML 文档
如何从数据库中创建 XML	在一个查询中组合 LINQ to Entities、LINQ to Objects 和 LINQ to XML，可以从数据库中构建 XML
如何创建 XML 文件和片段	使用 LINQ to XML 包含的方法，可以轻松地加载文件中的 XML，把 XML 保存到文件中，以及操作 XML 文档的各个部分

第Ⅴ部分

其 他 技 术

- 第 25 章　Windows Presentation Foundation
- 第 26 章　Windows Communication Foundation
- 第 27 章　Windows Workflow Foundation

第25章

Windows Presentation Foundation

本章内容：

- WPF 的含义
- 基本 WPF 应用程序的组成
- WPF 基础
- 用 WPF 编程

本书介绍了应用程序的两种主要类型：用户直接运行的桌面应用程序，以及用户通过浏览器访问的 Web 应用程序。我们在.NET Framework 的两个不同区域创建它们：Windows 窗体和 ASP.NET 页面。这些应用程序类型有各自的优缺点。桌面应用程序有较大的灵活性和较强的响应能力，而 Web 应用程序可以被许多用户同时在远程访问。

在目前的计算环境下，应用程序之间的界限越来越模糊了。有了 Web 服务和 Windows Communication Foundation(WCF，详见第 26 章)后，桌面应用程序和 Web 应用程序都可以用更分布的方式执行，在局域网和广域网上交换数据。另外，Web 客户应用程序(即 IE 或 Firefox 等浏览器)不再被看作所谓的“瘦”客户程序，因为它们不再仅仅显示信息。最新的浏览器和运行它们的计算机的功能已经远不止此。

近年来，用户体验大有渐趋融合之势。Web 应用程序现在一般使用 JavaScript、Flash、Java 应用程序和其他技术，越来越像桌面应用程序了。例如，Google Docs 的功能就说明了这一点。而桌面应用程序相互连接的趋势越来越明显，其功能从简单(自动更新、联机帮助等)到高级(例如，联机数据源和对等联网)，应有尽有。这一点可以通过图 25-1 加以说明。

Windows Presentation Foundation(WPF)是一种统一的技术，利用它编写的应用程序可以在桌面和 Internet 之间搭起桥梁。本章介绍的 WPF 应用程序可以作为桌面应用程序运行，或在浏览器上作为 Web 应用程序运行。WPF 有一个删节版本 Silverlight，可用于给 Web 应用程序添加动态内容。

本章将学习 WPF，理解如何使用它创建下一代应用程序。

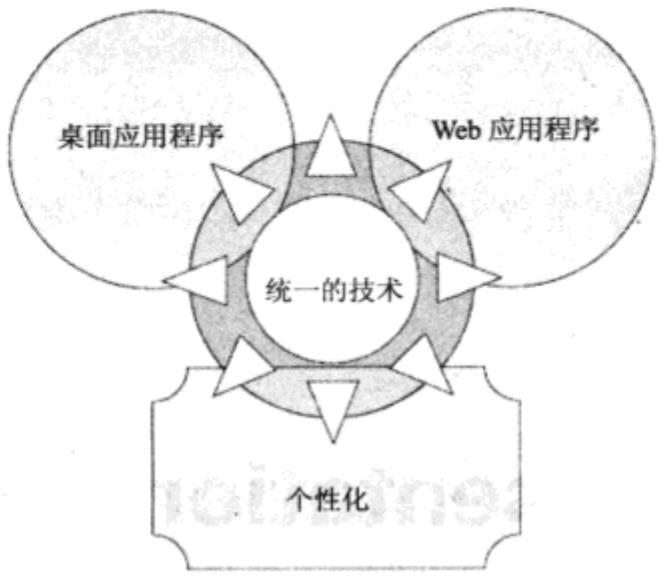


图 25-1

25.1 WPF 的概念

利用 WPF(以前称为 Avalon)技术可以编写出独立于平台的应用程序，它清晰地定义了设计和功能之间的界限。WPF 借用并扩展了以前许多技术的概念和类，包括 Windows 窗体、ASP.NET、XML、数据绑定技术和 GDI+等。如果读者以前使用.NET Framework 创建过 Web 应用程序，第一次看到 WPF 应用程序的代码时，会立即注意到许多类似之处。尤其是，WPF 应用程序使用标记和代码隐藏模型，这与 ASP.NET 类似。但在其内部，它们的不同点和相似点一样多，因此 WPF 对于开发人员和用户而言都是一种全新的技术。

WPF 开发的一个关键概念是设计与功能几乎完全分开。这样，设计人员和 C#开发人员就可以一起开发项目，而在之前，要达到这样的灵活程度，需要使用高级设计概念或第三方工具。这个功能受到所有人的欢迎，包括小型团队、爱好开发的人员、合作开发大项目的大型开发团队和设计人员。

下面几节介绍 WPF 给设计人员和开发人员带来了什么好处，使他们能携手工作。

25.1.1 WPF 给设计人员带来的好处

在 WPF 中，用户界面设计使用的语言是 Extensible Application Markup Language(XAML，读作 zammel)。它类似于 ASP.NET 中使用的标记语言，因为它也使用 XML 语法，允许以声明性的、带层次结构的方式把控件添加到用户界面上。也就是说，可以用 XML 元素的形式添加控件，用 XML 特性指定控件的属性。控件也可以包含其他控件，这对布局和功能都非常重要。

但是，XAML 比 ASP.NET 强大许多，在显示给用户时绝不仅限于 HTML 的功能。XAML 在设计时考虑了目前强大的图形卡，并允许使用这些图形卡通过 DirectX 7 或更高版本提供的全部高级功能。下面列出了这些功能：

- 浮点数坐标和矢量图提供的布局可以缩放、旋转和变换，且没有质量损失。
- 2D 和 3D 高级渲染功能
- 字体的高级处理和渲染

- UI 对象的纯色、渐变色和纹理填充，且可以设置透明度
- 动画故事板功能，可以用于所有情形，包括用户触发的事件，如鼠标单击按钮事件
- 可重用的资源，以动态设置控件的样式

许多功能都专门面向 Microsoft Vista 及以后版本的操作系统(包括 Windows 7、Windows Server 2008 和 Windows Server 2008 R2)，这些操作系统可以通过 Aero 接口访问高级图形功能。但是，WPF 应用程序也可以运行在其他操作系统上，例如 Windows XP。如果图形卡因某种原因无法工作，内置于.NET Framework 4 运行库中的回退渲染系统可以渲染 XAML(但有性能损失)。

VS 和 VCE 包含创建 XAML 代码并设置其样式的功能，但设计人员选择的工具是 Microsoft Expression Blend(EB)。这是一个设计和布局软件包，可用于创建 XAML 文件，接着开发人员就可以使用 XAML 文件创建应用程序。实际上，EB 使用与 VS 和 VCE 相同的解决方案和项目文件，所以可以在这些环境中的一个创建项目，在另一个环境中编辑它。在 EB 中，在编辑代码隐藏文件时，只需在 Files 窗口(等价于 VS 和 VCE 中的 Solution Explorer)中双击它。图 25-2 显示了 EB 界面。

图　25-2

在 http://microsoft.com/expression/products/Blend_Overview.aspx 上可以找到有关 EB 试用版本的许多信息，并可以下载。但是，编写 WPF 应用程序或编辑 XAML 并不需要 EB。图 25-3 显示了图 25-2 中的项目，但该项目在 VCE 中加载。

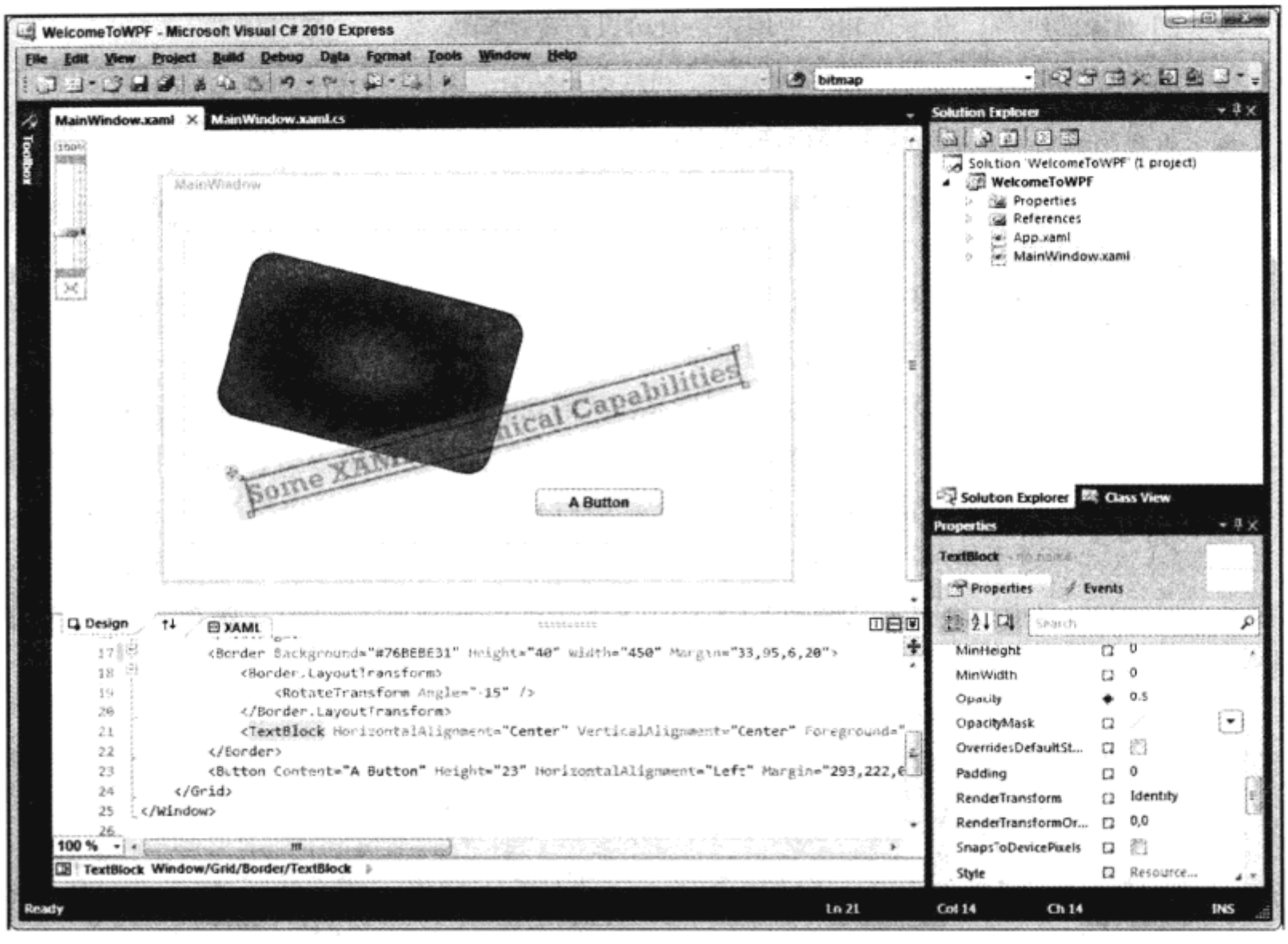


图 25-3

在 VCE 中，默认屏幕会在两个窗格上显示 XAML(目前不必考虑显示在 XAML 视图中的代码)和这些 XAML 的预览。属性编辑器不太直观。

应用程序在两个环境中启动的效果相同，如图 25-4 所示。

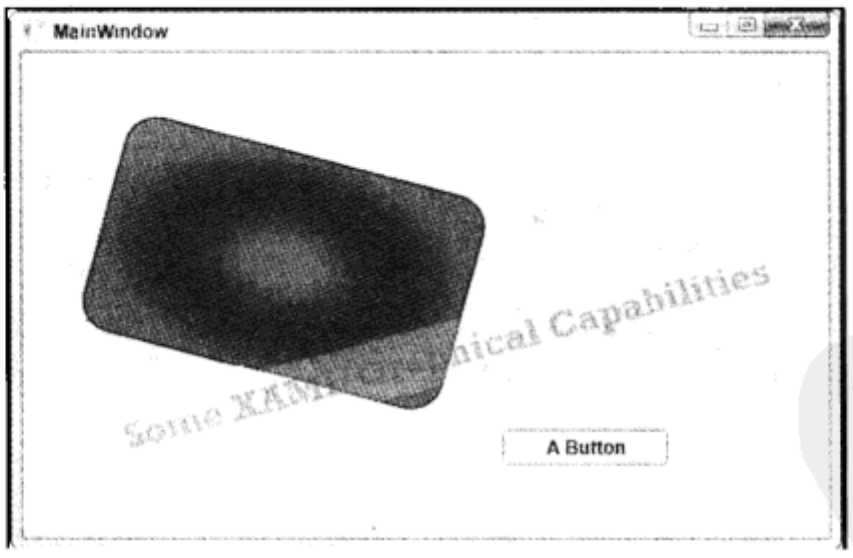


图 25-4

25.1.2 WPF 给 C#开发人员带来的好处

如上一节所述，开发人员可以创建能在 VS 或 VCE 中工作的项目和解决方案，而设计人员可以

在 Expression Blend 中编辑这些项目和解决方案。但与设计人员不同，开发人员主要在 VS 或 VCE 中工作。

如本节的引言所述，WPF 使用与 ASP.NET 非常类似的代码隐藏模式。例如，把 Click 特性添加到表示按钮控件的 XML 元素中，就可以给按钮控件添加一个事件处理程序。这个特性在 XAML 页面的代码隐藏文件中指定了事件处理程序的名称，事件处理程序可以用 C#编写。

注意，还可以在 WPF 应用程序中操作控件，其方式类似于 Windows 窗体应用程序使用编程技术布置用户界面。可以使用代码隐藏实例化控件，设置其属性，添加事件处理程序，给窗口添加控件，这完全绕过了 XAML。其代码一般比对应的 XAML 声明代码长得多，且无法分开设计和功能。编程方式在某些情形下是必须的，但一般应使用 XAML 来布置用户界面上的控件。

本章主要从 C#开发人员的角度来阐述。WPF 是一个需要整本书来介绍的主题，所以本章仅简要探讨它。

25.2　基本 WPF 应用程序的组成

WPF 使用起来很直观，学习它的最佳方式是开始使用它。如果读者已经阅读了本书的其他章节，就会很快熟悉许多技术。

下面的示例要创建一个简单的 WPF 应用程序，在该示例的说明中，会详细探讨代码和结果，了解如何把代码组合起来。

试一试：创建一个基本的 WPF 应用程序

(1) 创建一个新 WPF 应用程序 Ch25Ex01，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(2) 修改 MainWindow.xaml 中的代码，如下所示：

可从 wrox.com 下载源代码

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex01.MainWindow"
  Title="Color Spinner" Height="370" Width="270">
  <Window.Resources>
    <Storyboard x:Key="Spin">
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse1"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse2"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="-360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
```

```
        Storyboard.TargetName="ellipse3"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse4"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="-360"/>
      </DoubleAnimationUsingKeyFrames>
    </Storyboard>
  </Window.Resources>
  <Window.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
      <BeginStoryboard Storyboard="{StaticResource Spin}"
        x:Name="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="goButton">
      <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="stopButton">
      <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
  </Window.Triggers>
  <Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFFFFFFF" Offset="0"/>
      <GradientStop Color="#FFFFC45A" Offset="1"/>
    </LinearGradientBrush>
  </Window.Background>
  <Grid>
    <Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
      HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
      <Ellipse.Effect>
        <BlurEffect Radius="10"/>
      </Ellipse.Effect>
      <Ellipse.Fill>
        <RadialGradientBrush>
          <GradientStop Color="#FF000000" Offset="1"/>
          <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
        </RadialGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
      Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="120" Fill="Red" Opacity="0.5"
      RenderTransformOrigin="0.92,0.5" >
      <Ellipse.Effect>
        <BlurEffect/>
      </Ellipse.Effect>
      <Ellipse.RenderTransform>
        <TransformGroup>
```

```
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="85,15,0,0" Name="ellipse2" Stroke="{x:Null}"
  Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="80" Fill="Blue" Opacity="0.5"
  RenderTransformOrigin="0.5,0.92" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="115,85,0,0" Name="ellipse3" Stroke="{x:Null}"
  Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="120" Opacity="0.5" Fill="Yellow"
  RenderTransformOrigin="0.08,0.5" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="85,115,0,0" Name="ellipse4" Stroke="{x:Null}"
  Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="80" Opacity="0.5" Fill="Green"
  RenderTransformOrigin="0.5,0.08" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
<Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
  Name="goButton" VerticalAlignment="Bottom" Width="75" Content="Go"/>
<Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
  Name="stopButton" VerticalAlignment="Bottom" Width="75"
  Content="Stop"/>
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle"/>
</Grid>
</Window>
```

代码段 Ch25Ex01\MainWindow.xaml

(3) 双击设计视图中的 Toggle 按钮。该按钮在图 25-5 中高亮显示，它会隐藏 XAML 视图。

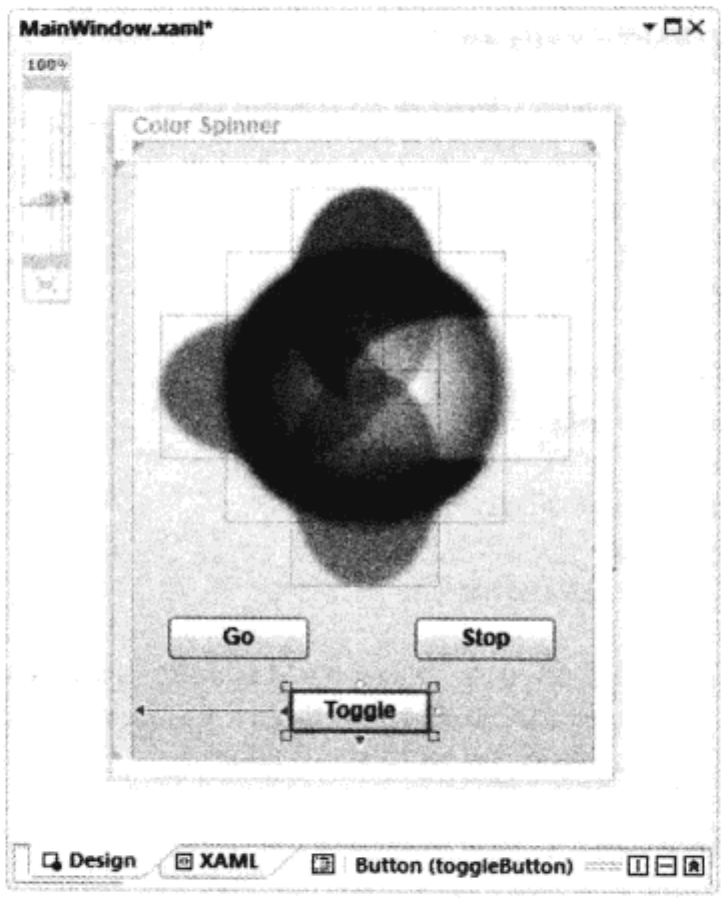


图 25-5

(4) 修改 MainWindow.xaml.cs 中的代码，如下所示(双击按钮时，会在 toggleButton_Click()事件处理程序中添加 using 语句和新代码)：

可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Media.Animation;

namespace Ch25Ex01
{
   /// <summary>
   /// Interaction logic for MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window
   {
      public MainWindow()
      {
           InitializeComponent();
       }
```

```
        private void toggleButton_Click(object sender, RoutedEventArgs e)
        {
            Storyboard spinStoryboard = Resources["Spin"] as Storyboard;
            if (spinStoryboard != null)
            {
                if (spinStoryboard.GetIsPaused(this))
                {
                    spinStoryboard.Resume(this);
                }
                else
                {
                    spinStoryboard.Pause(this);
                }
            }
        }
    }
}
```

代码段 Ch25Ex01\MainWindow.xaml.cs

(5) 执行应用程序，开始、停止、切换动画，示例如图 25-6 所示。

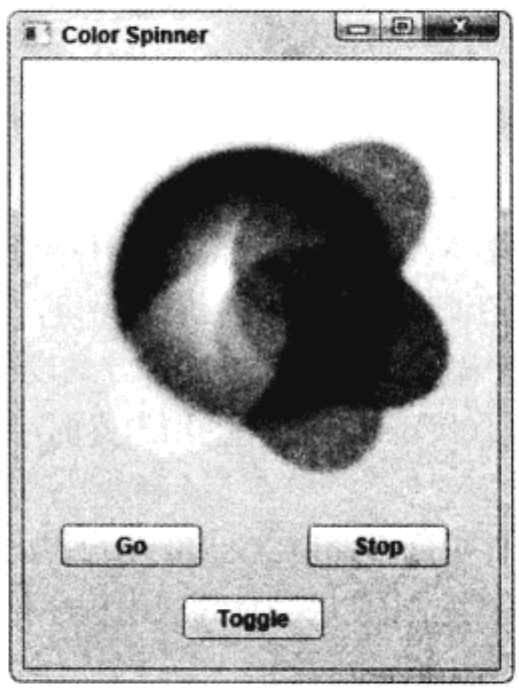


图 25-6

(6) 创建一个新的 WPF 浏览器应用程序 Ch25Ex01Web，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(7) 在 page1.xaml 中把<Page>元素的 Title 特性的值改为 Color Spinner Web。

(8) 打开 Ch25Ex01 应用程序中的 MainWindow.xaml 文件，把该文件中<Window>元素的所有代码复制到 page1.xaml 的<Page>元素中。

(9) 把<Window.Resources>、<Window.Triggers>和<Window.Background>元素分别改为<Page.Resources>、<Page.Triggers>和<Page.Background>元素(注意，要修改这些元素的开始和结束标记)。

(10) 从 page1.xaml 中删除 5 个<Ellipse.Effect>元素及其内容。

(11) 把 toggleButton_Click()事件处理程序和 System.Windows.Media.Animation 名称空间的 using

语句从 MainWindow.xaml.cs 复制到 page1.xaml.cs 中。

(12) 执行 Ch25Ex01Web 应用程序，结果如图 25-7 所示。

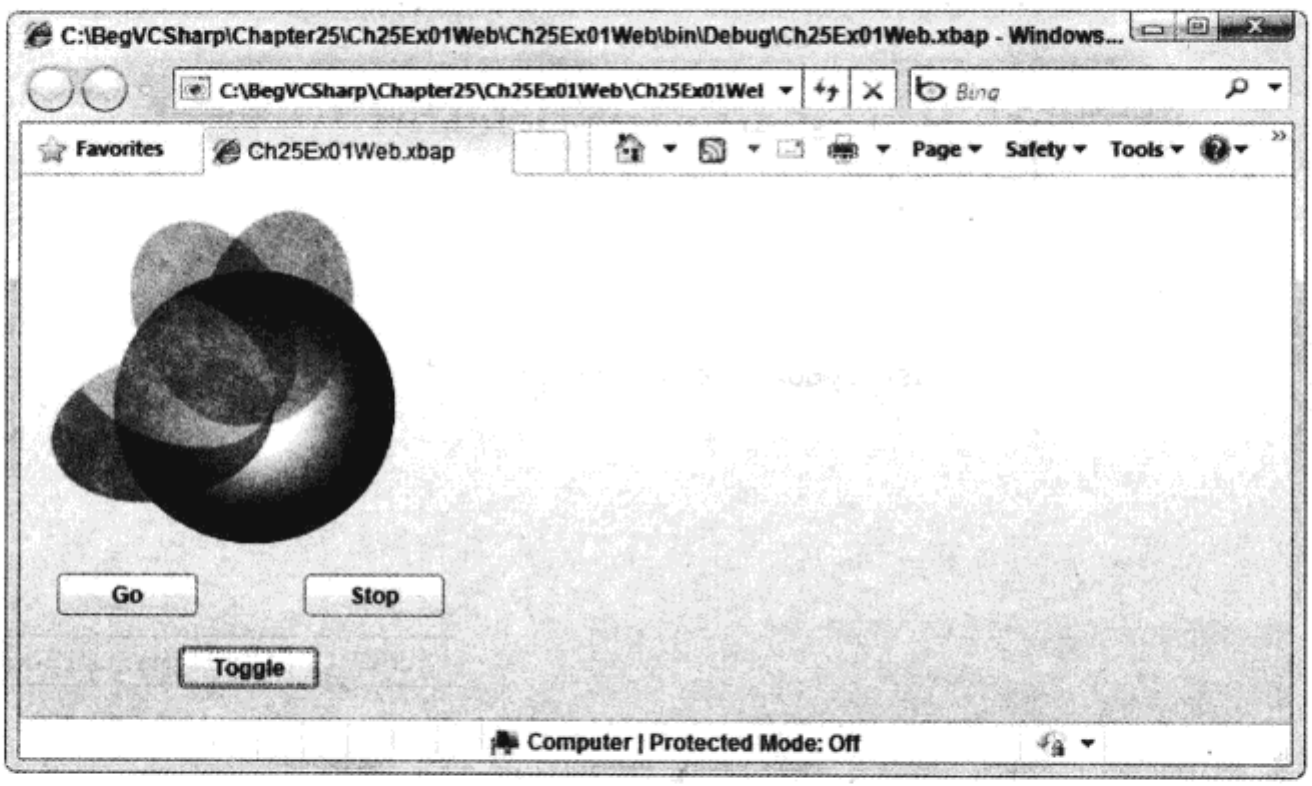


图 25-7

示例的说明

这个示例创建了一个简单应用程序，得到可以开始或停止的彩色旋转椭圆。但黑白屏幕图不能完全传达这种效果，只有运行代码，才会发现应用程序很漂亮。

为了达到这种效果，添加了许多 XAML 代码，但如果仔细查看代码，就会发现许多代码都是重复的——这是必要的，因为要使 4 个椭圆产生动画效果。另外，在代码隐藏文件中几乎没有添加什么 C#代码，仅为三个按钮中的一个添加了代码。以这种方式设计代码说明了两个要点：

- 设计人员可以仅用 XAML 代码创建出漂亮的用户界面，其中涉及高级图形功能、动画和用户交互操作。
- 需要时可以在代码隐藏文件中完全控制 XAML 用户界面。

我们还介绍了如何在 Web 应用程序中使用与桌面应用程序相同的代码。这需要进行几处修改，如后面所述，但基本功能在两个环境中是相同的。

这个应用程序中的代码演示了 WPF 的许多特性，介绍了一些关键技术。首先创建桌面应用程序，再看看 Web 应用程序需要的修改。下面列出了桌面应用程序 MainWindow.xaml 的 XAML，以及代码的顶级元素：

```
< Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="Ch25Ex01.MainWindow"
   Title="Color Spinner" Height="370" Width="270">


   ...


< /Window >
```

<Window>元素用于定义窗口。一个应用程序可以包含几个窗口，每个窗口都包含在一个单独的 XAML 文件中。但这并不是说 XAML 文件总是定义一个窗口。XAML 文件可以包含用户控件、笔刷和其他资源以及 Web 页面等。Ch25Ex01 项目中的 XAML 文件甚至定义了应用程序 App.xaml。本章后面会介绍应用程序和 App.xaml 文件。

在 MainWindow.xaml 中，注意<Window>元素包含一些通过名称即可了解其含义的特性。其中有两个名称空间声明，一个用于全局名称空间(用于 XML)，另一个用于 x 名称空间。这两个名称空间对于 WPF 功能来说非常重要，定义了 XAML 语法的词汇。接着是一个 Class 特性，它取自 x 名称空间。这个特性把 XAML <Window>元素链接到代码隐藏文件中的一个部分类定义上，在这个例子中是 Ch25Ex01.Window。这类似于 ASP.NET，一个类用于一个页面，并允许代码隐藏文件与 XAML 文件共享相同的编码模型，包括 XAML 元素定义的控件等。注意，x:Class 特性只能用于 XAML 文件的根元素。

其他 3 个特性 Title、Height 和 Width 指定了在窗口标题栏中显示的文本以及用于该窗口的尺寸(单位是像素)。这些特性映射到 System.Windows.Window 类的属性上，Ch25Ex01.Window 派生于 System.Windows.Window 类。

System.Windows.Window 类有其他几个属性可以定义额外的功能。许多属性都比<Window>元素上使用的 3 个属性复杂，它们不只是字符串或数字。XAML 语法允许使用嵌套的元素，为这些属性指定值。

“XAML 语法”一节详细讨论了定义对象、属性和内容的 XAML 语法。

例如，下面的代码用一个嵌套的<Window.Background>元素定义了 Background 属性：

```
<Window.Background>
 <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FFFFFFFF" Offset="0"/>
    <GradientStop Color="#FFFFC45A" Offset="1"/>
 </LinearGradientBrush>
</Window.Background>
```

这段代码把 Background 属性设置为 LinearGradientBruch 类的一个实例。在这个示例中，笔刷定义了一个自上而下从白色过渡到桃色的渐变色。

这段代码在嵌套的元素中还定义了另外两个复杂的属性：定义动画的<Window.Resources>和定义触发器的<Window.Triggers>(它控制动画)。这两个属性的功能都远不止此，本章后面会详细介绍它们。

在查看这些属性的实现代码之前，需要先看看<Grid>元素。<Grid>元素定义了 System.Windows.Controls.Grid 控件的一个实例。这是可以在 WPF 应用程序中用于布局的几个控件之一。它允许用相对于矩形 4 条边的坐标来定位嵌套的控件。其他控件允许用不同的方式定位控件。所有的布局控件都在本章后面的“控件布局”一节中介绍。

<Grid>元素包含 5 个<Ellipse>元素(System.Windows.Shapes.Ellipse 控件)和 3 个<Button>元素(System.Windows.Controls.Button 控件)。这些元素定义的椭圆用于在应用程序中显示旋转图形，按钮用于控制应用程序。

第一个<Ellipse>元素如下所示：

```
<Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
```

```
  HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
  <Ellipse.Effect>
    <BlurEffect Radius="10"/>
  </Ellipse.Effect>
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF000000" Offset="1"/>
      <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

这个元素定义了 System.Windows.Shapes.Ellipse 类的一个实例，用于显示椭圆图形，还设置了这个实例的几个属性，如下所示：

- **Name**：用于控件的标识符。
- **Margin**：通过指定图形周围的边距，来指定网格中 Ellipse 控件定义的图形的位置。这些数据在代码中以像素为单位。因此，这个属性根据 HorizontalAlignment 和 VerticalAlignment 属性，映射为图形的实际位置。
- **HorizontalAlignment 和 VerticalAlignment**：指定使用 Grid 定义的矩形的哪一条边布置图形。例如，Left 和 Bottom 值指定图形相对于网格的左下角定位。
- **Height 和 Width**：图形的尺寸。
- **Stroke**：Ellipse 控件定义的勾画图形轮廓的笔刷。
- **Fill**：Ellipse 控件定义的填充图形内部区域的笔刷。
- **Effect**：显示 Ellipse 控件时使用的特效。

Fill 属性使用的笔刷是 RadialGradientBrush。在这个例子中，笔刷指定了一个从白色(椭圆的中心)过渡到黑色(椭圆的边界)的渐变色。

Effect 属性设置为使用 BlurEffect。这是可以应用于 WPF 中图形的几个特效之一。这个特效会用 BlurEffect.Radius 属性定义的数值模糊图形，它不能应用于 Web 应用程序，因此在第(10)步删除了它。这是桌面应用程序和 Web 应用程序的几个区别之一。

可以定义应用于 XAML 项的定制特效，但这是一个高级主题，本章没有介绍。

代码中的另外 4 个<Ellipse>元素非常类似。每个元素都定义了四个具有动画效果的彩色椭圆中的一个，第一个元素如下：

```
<Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
  Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="120" Fill="Red" Opacity="0.5"
  RenderTransformOrigin="0.92,0.5" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>
```

```
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
```

这段代码非常类似于上一个椭圆的代码，但有如下区别：

- Stroke 属性设置为{x:null}。在 XAML 中，放在花括号中的值称为标记扩展，用于为不能在 XAML 语法中简化为简单字符串的属性提供值。在这个例子中，{x:null}为属性指定 null 值，表示 Stroke 没有使用笔刷。
- Opacity 属性指定为 0.5，这表示椭圆是半透明的。
- Effect 属性使用 BlurEffect，没有指定 Radius 特性，这里给 Radius 使用默认值 5。
- 指定了 RenderTransform 属性。这个属性设置为带一个 RotateTransform 变换的 TransformGroup 对象。这个变换在椭圆连续变化时使用。它指定了一个属性 Angle。椭圆根据这个角度(单位为度)旋转，它最初设置为 0°。
- RenderTransformOrigin 用于设置中心点，椭圆在进行 RotateTransform 变换时，围绕该中心点旋转。

最后两个属性与 XAML 中定义的动画有关，动画是用 System.Windows.Media.Animation. Storyboard 对象定义的。这个对象在<Window.Resources>元素中定义，表示可以通过窗口的 Resources 集合使用 Storyboard 对象。这段代码还定义了一个 x:Key 特性，它允许使用键通过 Resources 引用 Storyboard 对象：

```
<Window.Resources>
  <Storyboard x:Key="Spin">
    ...
  </Storyboard>
</Window.Resources>
```

Storyboard 对象包含 4 个DoubleAnimationUsingKeyFrames 对象。使用这些对象，可以指定包含 double 值的属性应随时间变化，还可以进一步定义这个操作。这段代码中的每个元素都定义了一个彩色椭圆使用的动画。例如，ellipse1 椭圆的动画如下所示：

```
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
      Storyboard.TargetName="ellipse1"
      Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0]
.(RotateTransform.Angle)"
      RepeatBehavior="Forever">
      <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
    </DoubleAnimationUsingKeyFrames>
```

这里不深入探讨这个元素，但它指定了 RotateTransform 变换的 Angle 属性应在 10 秒内从其初始值变成 360°，而且在变换结束后应重复进行。本章的“动画”一节将详细介绍动画。

定义了椭圆后，有 3 个<Button>元素定义了按钮(注意 Click 特性没有出现在示例的代码中，它是在第(3)步双击按钮时由 IDE 添加的)：

```
<Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
  Name="goButton" VerticalAlignment="Bottom" Width="75"
  Content="Go"/>
```

```
<Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
  Name="stopButton" VerticalAlignment="Bottom" Width="75"
  Content="Stop"/>
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggl" Click="toggleButton_Click"/>
```

这些元素都指定了 Button 对象的名称、位置和尺寸，所使用的属性与前面的<Ellipse>元素的相同。它们还用 Content 属性确定了在按钮上显示什么内容——在这个例子中，就是按钮上显示的字符串文本。按钮不仅可以用这种方式显示简单的字符串，还可以使用嵌套的形状或其他图形内容，详见本章的“控件的样式”一节。

toggleButton 按钮的 Click 特性为 Click 事件定义了一个事件处理方法。这个方法即为 toggleButton_Click()，它实际上是一个路由的事件(routed event)处理程序。路由的事件详见本章后面的“路由事件”一节。现在只需知道，单击按钮就会触发这个事件，并调用事件处理程序。

在事件处理程序的代码中，首先获得 Storyboard 对象的一个引用，该对象定义了动画。如前所述，这个对象包含在 Window 对象的 Resources 属性中，它使用了键 Spin。之后应是检索 Storyboard 对象的代码：

```
private void toggleButton_Click(object sender, RoutedEventArgs e)
{
   Storyboard spinStoryboard = Resources["Spin"] as Storyboard;
   if (spinStoryboard != null)
   {
      if (spinStoryboard.GetIsPaused(this))
      {
         spinStoryboard.Resume(this);
      }
      else
      {
         spinStoryboard.Pause(this);
      }
   }
}
```

注意，所有这些方法都需要对包含故事板(storyboards)的对象的引用。这是因为故事板本身没有跟踪时间。包含故事板的窗口有自己的时钟，故事板就使用这个时钟。给故事板传送对窗口的引用(通过 this)，故事板就可以访问这个时钟。

另外两个按钮 goButton 和 stopButton 没有链接到后台代码的任何事件处理方法上。它们的功能是由触发器决定的。在这个例子中，定义了 3 个触发器，如下所示：

```
<Window.Triggers>
  <EventTrigger RoutedEvent="FrameworkElement.Loaded">
    <BeginStoryboard Storyboard="{StaticResource Spin}"
      x:Name="Spin_BeginStoryboard"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="ButtonBase.Click"
```

```
      SourceName="goButton">
      <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click"
      SourceName="stopButton">
      <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
  </Window.Triggers>
```

第一个触发器将 FrameworkElement.Loaded 事件(在加载应用程序时启动)和 BeginStoryboard 动作链接起来。这个动作启动 Spin 动画。注意，Spin 动画是通过代码(StaticResource Spin)使用标记扩展语法引用的。这个语法用于引用包含窗口中的资源，它在 WPF 应用程序中很常见。BeginStoryboard 动作的名称是 Spin_BeginStoryboard，在另外两个触发器中引用，这两个触发器分别链接了 goButton 和 stopButton 的 Click 事件。这些触发器使用了 ResumeStoryboard 和 PauseStoryboard 动作，恢复和暂停动画的执行。

这段代码在作为桌面应用程序时工作正常，但把它们转换为 Web 应用程序时，需要进行几处修改。实际上，这个示例创建了一个新的 Web 浏览器应用程序，隐藏了几个细节。例如，在浏览器上运行代码有一些安全方面的限制，所以 Web 浏览器应用程序使用了一个临时的键，它用于给应用程序签名。如果希望让应用程序执行浏览器应用程序禁止执行的某些动作，例如，访问本地文件系统，使用临时键就是必须的。

还要注意，桌面应用程序的根元素<Window>在 Web 应用程序中被<Page>元素替换。这是因为浏览器提供的功能略微不同于运行桌面应用程序的主机程序。因此，WPF 使用不同的类表示这些不同的主机。但是如代码所示，许多对象都可以在两个环境中使用相同的代码，详见本章后面的内容。

这就完成了这个示例应用程序的分析。这里复习了许多基础知识，还简要学习了一些新概念，所以现在最好休息一下。本章剩余的内容将深入阐述这里提到的技术，规范需要的语法，再介绍一些新知识。

25.3 WPF 基础

希望本章第一部分的示例能激起读者对 WPF 编程的兴趣。虽然还有许多新概念需要理解，但我们已经介绍了如何组合 XAML 与.NET 代码快速创建动态的应用程序。还学习了许多功能，包括设计人员不需要 C#的任何知识，就可以设计应用程序的 UI。最后说明了如何用几乎相同的代码创建桌面应用程序和 Web 应用程序。

但是，在开始创建 WPF 应用程序之前，应花点时间学习基础知识。本节就介绍 WPF 应用程序的几个基础主题，学习实现它们所需的语法。还要探讨许多可以在应用程序中进一步研究的其他方法。

本节的内容如下：

- XAML 语法
- 桌面应用程序和 Web 应用程序
- Application 对象
- 控件基础知识，包括依赖属性、关联属性、路由事件和关联事件

- 控件的布局和样式
- 触发器
- 动画
- 静态和动态资源

25.3.1 XAML 语法

本章第一部分的示例虽然介绍了许多 XAML 语法，但没有正式描述它们。这个示例忽略了许多规则和可能性，以便只考虑基本结构和功能。本节将详细介绍 XAML，理解 XAML 文件的组成部分。

1. 对象元素语法

XAML 文件的基本结构使用对象元素语法来描述对象的一个层次结构，这个层次结构有一个根对象，它包含了其他所有对象。顾名思义，对象元素语法描述了用 XML 元素表示的对象(或结构)。例如，在前面的示例中，<Button>元素用于表示 System.Windows.Controls.Button 对象。

XAML 文件的根元素总是使用对象元素语法，但在前面的示例中，用于根对象的类不是用元素名(<Window>或<Page>)定义的，而是用 x:Class 特性定义的。这个语法只能用于根元素。对于桌面应用程序，根元素必须继承于 System.Windows.Window，而对于 Web 应用程序，根元素必须继承于 System.Windows.Controls.Page。

用对象元素语法定义的许多对象实际上都是控件，如前面示例中使用的 Button 控件。

2. 特性语法

在许多情况下，元素用于表示对象(使用对象元素语法)时，就用特性指定属性和事件。例如，前面的<Button>元素使用了如下特性：

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle" Click="toggleButton_Click"/>
```

这里的每个特性都设置了 toggleButton 对象的一个属性值，但 Click 和 Name 除外，Click 特性把一个路由的事件处理程序赋予 toggleButton 的 Click 事件。这些都是特性语法的例子。

这里使用的 Name 特性是一个特例：它定义了控件的标识符，以便在代码隐藏文件和其他 XAML 代码中引用它。

特性可以用它们引用的基类和一个句点来限定。例如，Button 控件从 ButtonBase 中继承了 Click 事件，所以可以把上面的代码重写为：

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle" ButtonBase.Click="toggleButton_Click"/ >
```

注意，这个语法也称为关联属性，参见本章的“控件基础”一节。

3. 属性元素语法

在许多情况下，需要使用比简单字符串略微复杂的方式来初始化属性值。在示例应用程序中，

Fill 属性就是这样，它设置为各种笔刷对象：

```
<Ellipse ...>
  ...
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF000000" Offset="1"/>
      <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

这里属性通过子元素设置，该子元素的名称根据下面的约定来指定：

```
[Parent Element Name].[Property Name]
```

这称为属性元素语法。

4. 内容语法

许多控件其实都是内容表示器(content presenter)。这表示可以给控件提供内容，这些内容根据控件模板显示出来。例如，为 Button 控件提供的内容显示在按钮表面。这些内容可以是文本，如示例所示，也可以是图形。

使用内容语法可以为控件指定内容，为此，只需把内容添加为表示控件的元素的内容：

```
<Button ...>Go</Button>
```

这行代码用于 Button 控件，以显示文本 Go。这个示例使用了一种比较简单但不太灵活的方式：特性 Content。这里使用的完整内容语法对于较复杂的内容而言是必须的，例如，本节引言中提到的图形内容。

给控件使用复杂的内容时，XAML 代码会有复杂的嵌套层次。因此，XAML 允许控件用其他更微妙的方式设置样式。有关内容表示器和样式设置的知识详见“控件的样式”一节。

5. 合并属性元素语法和内容语法

XAML 页面上用对象元素语法格式化的控件可能包括属性元素语法和内容语法。此时，必须遵循下述语法规则：

- 用于属性元素语法的元素不必是连续的，即一个使用属性元素语法的元素后面可以跟一个使用内容语法的元素，其后的第 3 个元素再使用属性元素语法。
- 用于内容语法的元素(和文本内容)必须是连续的——使用内容语法的文本或元素后不能跟一个使用属性元素语法的元素，其后的第 3 个元素再使用文本或内容语法。

因此下面的代码是正确的：

```
<Button ...>
  <Button.Effect>
    <BlurEffect Radius="10"/>
  </Button.Effect>
  Go
  <Button.RenderTransform>
    <RotateTransform Angle="20"/>
```

```
  </Button.RenderTransform>
</Button>
```

但是，下面的代码不能工作，因为有两个地方使用了内容语法，它们用一个使用属性元素语法的元素隔开了：

```
<Button ...>
  Don't
  <Button.Effect>
    <BlurEffect Radius="10"/>
  </Button.Effect>
  Go
</Button>
```

6. 标记扩展

前面的示例说明了标记扩展也可以用于属性值——例如，值{x:null}。只要使用了花括号{}，就是在使用标记扩展。这些标记扩展都可以在特性语法和属性元素语法代码中使用。

本章的所有标记扩展都专门用于 WPF。WPF 特有的扩展包括用于引用资源和数据绑定的扩展。

25.3.2 桌面和 Web 应用程序

本章前面的示例演示了 WPF 应用程序如何作为独立的桌面应用程序和 Web 应用程序运行。前面把 WPF Application 和 WPF Browser Application 项目模板作为起点，添加了 XAML 和 C#代码，以完成该应用程序。WPF Application 模板把项目编译为.exe 文件，WPF Browser Application 模板把项目编译为.xbap 文件。

XBAP(读作 ex-bap)是 XAML Browser Application 的首字母的缩写，用 WPF 创建的 Web 应用程序常称为 XBAP 应用程序。

这些应用程序类型之间的大多数区别都是项目文件(.csproj)之间的区别。Web 浏览器应用程序定义了一些额外的设置，包括两个应用程序的签名和应用程序的清单(如前所述，为了安全起见)，以及允许在浏览器中调试的设置。还有一个测试证书，可用于这个签名。在生产环境下，需要用证书颁发机构颁发的证书替代这个证书。

在桌面 WPF 和 Web WPF 应用程序之间转换是一个难度很高的过程，因为必须改变许多设置，还要修改一些 XAML 代码，如示例所示。这些改动包括把<Window>元素改为<Page>元素，删除位图效果等功能。最好的方法通常是创建独立的项目，如前面的示例所示。

25.3.3 Application 对象

在 WPF 中，大多数应用程序(包括所有的 XBAP 应用程序和使用 WPF Application 模板的桌面应用程序)都包含一个派生自 System.Windows.Application 的类实例。在前面的示例应用程序中，这个对象由 App.xaml 和 App.xaml.cs 文件定义。Ch25Ex01 的 App.xaml 如下所示：

可从 wrox.com 下载源代码

```
<Application x:Class="Ch25Ex01.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 >
```

代码段 Ch25Ex01\App.xaml

<Application>元素的语法类似于前面讨论的<Window>元素，它以相同的方式使用 x:Class 特性，把代码链接到代码隐藏文件中的部分类定义上。

这段代码定义的对象是 WPF 应用程序的入口。这个对象只能有一个实例，使用静态属性 Application.Current 可以通过代码访问它。应用程序的 Application 对象非常有用，原因如下：

- 它提供了在应用程序的生命周期的特定时刻引发的许多事件，包括前面介绍的 LoadCompleted 和 DispatcherUnhandledException，LoadCompleted 在应用程序加载并显示时引发，DispatcherUnhandledException 在抛出一个未处理的异常时引发。
- 它包含的方法可以用于设置或加载 cookie，定位和加载资源等。
- 它的几个属性可用于访问应用程序范围内的资源(参见“静态和动态资源”一节)和应用程序中的窗口。

Application 对象引发的事件是这个列表中最有用的特性，也是最常用的特性。

25.3.4　控件基础

WPF 提供了许多可用于创建应用程序的控件。本章简要介绍 WPF 及其功能，所以不详细探讨每个 WPF 控件，而是通过使用它们来学习。

下面列出了 WPF 提供的控件：

Border	Image	Slider
BulletDecorator	Label	StatusBar
Button	ListBox	TabControl
Calendar	ListView	TextBlock
CheckBox	ListView	TextBox
ComboBox	Menu	ToolBar
ContextMenu	MediaElement	ToolTip
DataGrid	PasswordBox	TreeView
DatePicker	Popup	Viewbox
DocumentViewer	ProgressBar	
Expander	RadioButton	
FlowDocumentPageViewer	RepeatButton	
FlowDocumentReader	RichTextBox	
FlowDocumentScrollViewer	ScrollBar	
Frame	ScrollViewer	
GroupBox	Separator	

这个列表不包含用于布局的 WPF 控件，这些控件参见本章后面的内容。

这里列出的一些控件的名称您或许非常熟悉，实际上，它们的功能与 Windows 窗体和 ASP.NET 应用程序中的对应控件非常类似。例如，Button 控件可用于显示按钮。其他一些控件您可能不太熟悉，所以需要试用，以了解它们的功能。

这些控件最初都只有非常基本的外观。为了美化它们，必须给它们设置样式，这会使 WPF 的强大功能变得非常明显，如本章后面所述。除了设置样式之外，WPF 控件还使用了其他几个特性。本节将介绍如下内容：

- 依赖属性
- 关联属性
- 路由事件

与其他桌面应用程序和 Web 应用程序开发一样，也可以创建自己的控件，而且您几乎肯定会创建这种控件。在创建控件时，可以使用这里介绍的所有特性。下面几节会举几个实现例子。

1. 依赖属性

依赖属性(dependency property)是在整个 WPF 中使用的一种属性，尤其是在控件上使用，它提供了扩展一般.NET 属性的功能。为了说明这一点，考虑某个一般的.NET 属性。在.NET 中创建类时，一般使用非常简单的代码来实现属性：

```
private string aStringProperty;

public string AStringProperty
{
    get
    {
        return aStringProperty;
    }
    set
    {
        aStringProperty = value;
    }
}
```

这里定义了一个公共属性，它以一个私有字段为基础。这些简单的实现代码绝对适用于大多数目的，但除了基本的状态访问之外，没有包含太多的功能。例如，如果要给控件 ControlA 添加一个 AStringProperty，让另一个控件 ControlB 响应对该属性的改动，就必须执行如下步骤：

(1) 使用前面的代码给控件 ControlA 添加 AStringProperty。

(2) 给 ControlA 添加一个事件。

(3) 给 ControlA 添加一个方法，来引发事件。

(4) 在 ControlA 中为 AStringProperty 的 set 访问器添加代码，以调用事件，引发方法。

(5) 给 ControlB 添加代码，以订阅 ControlA 中的事件。

这里不需要列出添加和响应简单属性的变化的代码，因为本书已经多次列出了这些代码。

这种方法存在的问题是要达到这个效果，没有可以遵循的明确标准。不同开发人员可能采用不同方式添加代码，来达到相同的效果。而且，这需要在开发控件时确定所有可能需要通知他人的属性。

这个问题的WPF 解决方案是用依赖属性替代前面代码中使用的简单属性定义，然后使用格式化的、结构化的技术提供属性改变通知。依赖属性用WPF 属性系统注册，允许使用扩展的功能。这个扩展功能包括(但不仅限于)属性改动的自动通知。确切地讲，依赖属性有如下特性：

- 可以使用样式改变依赖属性的值。
- 可以使用资源或通过数据绑定设置依赖属性的值。
- 可以改变动画中依赖属性的值。
- 可以在 XAML 中按层次设置依赖属性——在父元素上设置的依赖属性值可以用于设置其子元素的对应依赖属性的默认值。
- 使用定义好的编码模式可以配置属性值改动的通知。
- 可以配置一系列相关属性，在改变其中一个属性值时，它们就会全部更新。这称为强制转换。改变的属性会强制转换其他属性的值。
- 可以把元数据应用于依赖属性，指定其他行为特征。例如，可以指定如果给定属性改变了，就需要重新布置用户界面。

实际上，由于依赖属性的实现方式，我们可能最初注意不到它们与一般属性有什么区别。但是，在创建自己的控件时，很快会发现如果使用一般的.NET 属性，有许多功能都突然消失了。

依赖属性在 WPF 中使用得非常普遍，所以后面将介绍如何实现它们。

2. 关联属性

关联属性是一种属性，定义这种属性的类的实例的每个子对象都可以访问它。例如，假定有一个类 Recipe，它可以包含表示成分的子对象。在 Recipe 类定义中可以定义一个关联属性 Quantity，它可以由每个子对象使用。注意，子对象不必为关联属性指定值。

这么做的主要原因是用于设置关联属性值的 XAML 代码很容易理解：

```
<Recipe Name="Simple Vegetable Chili">
  <TinOfKidneyBeans Recipe.Quantity="2" Mashed="true" />
  <TinOfChoppedTomatoes Recipe.Quantity="2" />
  <FreshChili Recipe.Quantity="5" Notes="Chopped fine, vary to taste." />
  <Onion Recipe.Quantity="1" Notes="Chopped and fried in olive oil." />
  <LBVPort Notes="Just a dash." />
</Recipe>
```

这里使用的语法是前面介绍的特性语法的一种形式。其中关联属性使用父元素名、句点和关联属性名来表示。

在 WPF 中，关联属性有许多用途。稍后在“控件的布局”一节中介绍如何定位控件时，会介绍许多关联属性。我们将学习容器控件如何定义关联属性，让每个子控件都可以确定(例如)要停靠在容器的哪条边上。

3. 路由事件

WPF 应用程序的本质是层次结构，所以其中的控件常常包含其他控件，这些控件又包含更多的控件，依此类推。路由事件(routed event)是一个机制，借助这个机制，影响层次结构中一个控件的事件可以影响层次结构中的其他事件，且不需要复杂的代码。

一个很好的例子是允许用户用鼠标与应用程序交互操作，当然这非常常见。用户单击应用程序中的按钮时，一般要响应单击事件。Windows 窗体和 ASP.NET 开发中的一种常见方式是为按钮的事件提供事件处理程序，来响应鼠标的单击。

这种技术存在很大的局限性，且在一些 Windows 窗体应用程序中会导致代码比较混乱，尽管初看起来不是这样。其原因是需要某种机制引发某个按钮的单击事件，标识该控件应响应鼠标单击，而这并不是非常明显。在这里举出的简单例子中，是应引发按钮的单击事件，还是应引发包含按钮的窗口的单击事件？如果按钮和窗口都有事件处理程序，且只引发了其中一个事件，一般希望引发按钮的事件。但如果希望引发两个事件，那么引发事件的顺序如何？对于 Windows 窗体应用程序，这需要编写相当复杂的定制代码。

WPF 控件(包括 Button 和 Window)的鼠标单击事件实现为路由事件，解决了这个问题。路由事件由层次结构中的所有对象按指定顺序引发，可以完全控制响应它们的方式。

例如，假定 Window 包含一个 Grid，Grid 又包含一个 Rectangle。单击 Rectangle 时，事件的引发顺序如下：

(1) 引发 Window 上的鼠标按下事件

(2) 引发 Grid 上的鼠标按下事件

(3) 引发 Rectangle 上的鼠标按下事件

(4) 引发 Rectangle 上的另一个鼠标按下事件

(5) 引发 Grid 上的另一个鼠标按下事件

(6) 引发 Window 上的另一个鼠标按下事件。

添加适当的事件处理方法，就可以响应上述序列中的任一个事件。还可以在事件处理方法的任一点中断该序列，但事件处理程序默认不会中断该序列。这说明，可以在一个事件(这里是一个鼠标按下事件)中触发多个事件处理方法。

在描述前面的事件序列时，WPF 引入了一些有用的术语。事件在控件的层次结构中向下移动时，称为通道(tunneling)；向上移动时，称为冒泡(bubbling)。

另外，只要在 WPF 中使用了路由事件，事件名就可以指出该事件是通道事件还是冒泡事件。所有通道事件都以前缀 Preview 开头。例如，Window 控件有 PreviewMouseDown 和 MouseDown 事件。可以给它们中的一个或两个添加处理程序，或者不添加任何处理程序。

图 25-8 列出了上述事件的引发顺序，并说明了事件在控件层次结构中如何通过通道和冒泡。

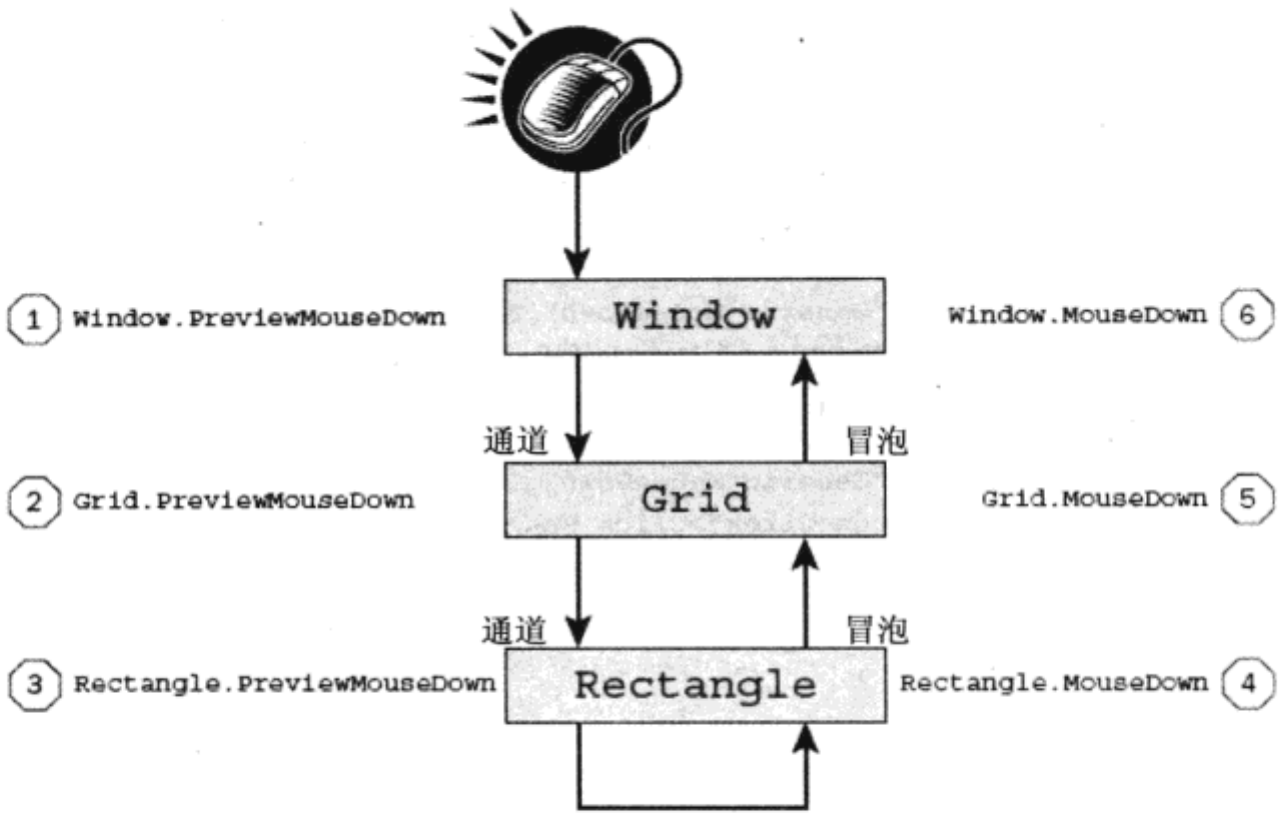


图　25-8

路由事件处理程序有两个参数：事件源和 RoutedEventArgs 实例或派生于 RoutedEventArgs 的类。

实现路由事件的事件处理程序时，可以根据需要将 RoutedEventArgs 对象的 Handled 属性设置为 true。这样，就不会执行进一步的处理了，即该事件不会引发更多的事件处理程序。

RoutedEventArgs 还有一个属性 Source，它允许指定哪个控件最先引发了事件。这是 WPF 最初检测到事件的控件，所以在图 25-8 中，该控件应是 Rectangle。这是非常有用的，因为父控件可以确定单击了哪个子控件。注意，这个“单击测试”是相当复杂的，例如，WPF 可以忽略控件透明区域上的单击，我们不需要做任何工作，就可以启用这个功能。另外，还可以创建可响应鼠标单击的透明控件，所以非常灵活。

WPF 在遇到单击测试时，会区分控件的“透明”区域和“空”区域。只有“透明”区域会响应单击测试，“空”区域会被忽略。

路由事件覆盖了多个鼠标单击，它们可以用于各种目的，包括键盘交互操作、数据绑定和计时器等。稍后介绍的关联事件会使路由事件更有用。

下面的示例演示了本节描述的情形，还介绍了路由事件的其他信息。

试一试：处理路由事件

(1) 创建一个新 WPF 应用程序 Ch25Ex02，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(2) 修改 MainWindow.xaml 的代码，如下所示：

可从 wrox.com 下载源代码

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex02.MainWindow"
  Title="Routed Events" Height="400" Width="800"
```

```
    MouseDown="Generic_MouseDown" PreviewMouseDown="Generic_MouseDown"
    MouseUp="Window_MouseUp">
    <Grid Name="contentGrid" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown" Background="Azure">
      <Rectangle Name="clickMeRectangle" Margin="10,10,0,0"
        Height="23" HorizontalAlignment="Left" VerticalAlignment="Top"
        Width="70" Stroke="Black" MouseDown="Generic_MouseDown"
        PreviewMouseDown="Generic_MouseDown" Fill="CadetBlue" />
      <Button Name="clickMeButton" Margin="0,10,10,0" Height="23"
        HorizontalAlignment="Right" VerticalAlignment="Top" Width="70"
        MouseDown="Generic_MouseDown"
        PreviewMouseDown="Generic_MouseDown"
        Click="clickMeButton_Click">Click Me</Button>
      <TextBlock Name="outputText" Margin="10,40,10,10"
        Background="Cornsilk" />
    </Grid>
</Window>
```

代码段Ch25Ex02\MainWindow.xaml

(3) 修改MainWindow.xaml.cs中的代码，如下所示(注意，根据所使用的IDE和输入XAML代码的方式，可能会自动添加空事件处理方法)：

可从wrox.com下载源代码

```
public partial class MainWindow : Window
{
    ...

    private void Generic_MouseDown(object sender,
        MouseButtonEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}Event {1} raised by control {2}. e.Source={3}\n",
            outputText.Text,
            e.RoutedEvent.Name,
            sender.ToString(),
            ((FrameworkElement)e.Source).Name);
    }

    private void Window_MouseUp(object sender, MouseButtonEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}==========\n\n",
            outputText.Text);
    }

    private void clickMeButton_Click(object sender, RoutedEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}Button clicked!\n==========\n\n",
            outputText.Text);
    }
}
```

代码段Ch25Ex02\MainWindow.xaml.cs

(4) 运行应用程序。当程序运行时，依次单击左上角的矩形、矩形和按钮之间的浅蓝色区域，以及按钮，结果如图 25-9 所示。

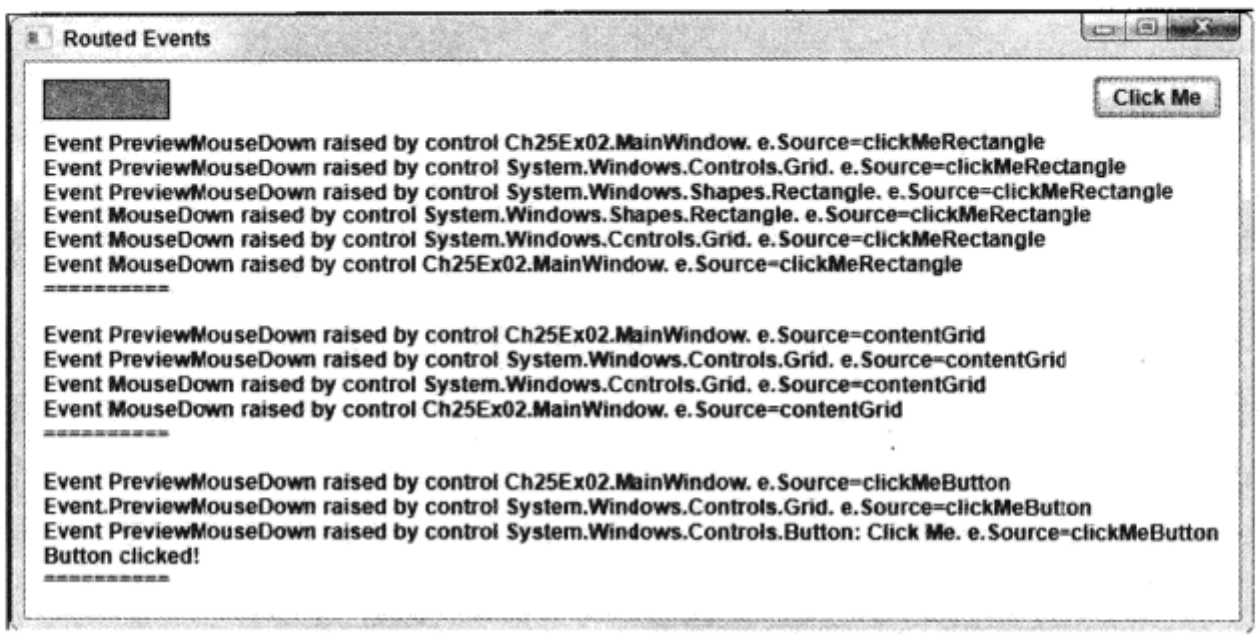


图　25-9

示例的说明

这个示例突出显示了所有 WPF 控件都有的 PreviewMouseDown 和 MouseDown 事件，说明了如何处理路由事件。还介绍了在事件链中包含按钮时会发生什么。所使用的 XAML 代码非常简单，其基本部分(在这个示例中)如下：

```
<Window
  x:Class="Ch25Ex02.MainWindow" MouseDown="Generic_MouseDown"
  PreviewMouseDown="Generic_MouseDown" MouseUp="Window_MouseUp">
  <Grid Name="contentGrid" MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown">
    <Rectangle Name="clickMeRectangle" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown" />
    <Button Name="clickMeButton" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown"
      Click="clickMeButton_Click" />
    <TextBlock Name="outputText" />
  </Grid>
</Window>
```

这里所有对性能没有影响的属性都被删除了，以便主要考虑与路由事件处理相关的代码。本例使用了 3 个事件处理程序，其配置如表 25-1 中所示。

表　25-1

事件处理程序	处理的事件
Generic_MouseDown()	Window.PreviewMouseDown
	Window.MouseDown
	Grid.PreviewMouseDown
	Grid.MouseDown
	Rectangle.PreviewMouseDown
	Rectangle.MouseDown

(续表)

事件处理程序	处理的事件
Window_MouseUp()	Window.MouseUp
clickMeButton_Click()	Button.Click

事件处理方法只是把信息输出到 TextBlock 控件中，说明发生了什么。文本输出包括事件名、引发事件的控件和事件的源控件(通过 RoutedEventArgs.Source 获得)。

运行应用程序时，第一次单击 Rectangle 控件会发生示例前面描述的事件序列。Generic_MouseDown()事件处理程序会调用 6 次：其中 3 次为 PreviewMouseDown 通道事件调用，另外 3 次为 MouseDown 冒泡事件调用。所有这些事件的事件源都是满足单击测试的控件，即矩形 clickMeRectangle。在这些处理程序之后还调用了 Window_MouseUp()事件处理程序，添加了一些文本，把这个测试与下一次测试分开。

此后单击控件之间的区域，实际上这会单击 Grid 控件的背景。这次调用了 4 次 Generic_ MouseDown()事件处理程序：其中 2 次为 PreviewMouseDown 通道事件调用，另外 2 次为 MouseDown 冒泡事件调用。在所有情况下，事件源是 Grid 控件 contentGrid。在调用 Generic_MouseDown()后，又调用了 Window_MouseUp()事件处理程序。

最后单击 Button 控件。这次 Generic_MouseDown()仅调用了 3 次，接着触发 Button.Click 事件，调用 clickMeButton_Click()。然后调用 Window_MouseUp()事件处理程序。

在最后这个事件链中，MouseDown 事件由按钮处理，用于触发其 Click 事件。按钮的 MouseDown 事件处理程序的底层实现代码把 RoutedEventArgs 事件参数的 Handled 属性设置为 true。这会中断事件流，所以 MouseDown 事件不会按层次结构向上传送。

为了弄明白按钮控件中断事件路由的方式，我们在介绍示例前单击了 Rectangle 控件。

4. 关联事件

前面的示例在页面的 Button 控件上添加了 Button.Click 事件。但没有给 Grid 或 Window 的 Click 事件添加处理程序，因为这两个控件没有 Click 事件。但有时我们希望它们有 Click 事件。

例如，假定一个窗口包含 1000 个按钮，而您希望处理每个按钮的 Click 事件。这需要 1000 个事件处理程序，或者简化编码，使用一个共享的事件处理程序。但即使只有一个事件处理程序，也必须把它关联到每个 Button.Click 事件上。

WPF 为处理这种情况提供了另一种更好的方式：关联事件。使用关联事件系统，可以在没有提供事件的控件上处理这些事件，所以上面的示例可以处理包含按钮的 Grid 上的 Button.Click 事件——即使 Grid 控件没有 Click 事件。实际上，我们处理的是 ButtonBase.Click 事件，因为 ButtonBase 是定义 Click 事件的类，而 Button 控件继承了这个类。

关联事件的语法与关联属性的特性语法相同：

```
<Grid Name="contentGrid" ButtonBase.Click="contentGrid_Click"...>
 <Button Name="button1" ...>Button1</Button>
 <Button Name="button2" ...>Button2</Button>
 ...
 <Button Name="button1000" ...>Button1000</Button>
```

```
</Grid>
```

在事件处理程序中，从 sender 参数中获得 Grid 控件的一个引用，然后使用 RoutedEventArgs.Source 属性确定单击了哪个按钮，并做出相应的响应。这个事件仅在单击按钮时触发，而在单击 Grid 控件的背景时不会触发，因为 Grid 控件没有 Click 事件。

25.3.5　控件的布局

本章前面一直使用 Grid 元素布置控件，这主要是因为在创建新的 WPF 应用程序时，这是默认提供的控件。但是我们还没有充分利用这个类的全部功能，也没有学习能得到其他布局效果的其他布局容器。本节将详细介绍控件布局，因为这是 WPF 中需要掌握的一个基本概念。

所有的内容布局控件都派生于抽象类 Panel。这个类仅定义了一个容器，该容器可以包含派生于 UIElement 的对象集合。所有的 WPF 控件都派生于 UIElement。不能直接使用 Panel 类控制布局，但可以派生于它。另外，还可以使用下面派生于 Panel 的布局控件：

- **Canvas**：这个控件可以按任意方式定位子控件。它对子控件的定位没有任何限制，但也没有给子控件的定位提供任何帮助。
- **DockPanel**：这个控件可以把子控件停靠在它的 4 条边上。最后一个子控件会占用剩余的空间。
- **Grid**：前面已经介绍了这个控件如何灵活地定位子控件。但还没有介绍如何把这个控件的布局分成行和列，使控件在栅格布局中对齐。
- **StackPanel**：这个控件以水平或垂直布局来布置其子控件。
- **WrapPanel**：这个控件与 StackPanel 一样，也以水平或垂直布局来布置其子控件，但它不是仅在单行或单列上布置控件，而是根据可用的空间，允许把子控件放在多行或多列上。

下面详细介绍这些控件的用法。但首先需要理解几个基本概念：

- 控件如何以堆栈顺序显示
- 如何使用对齐、页边距和填充来定位控件及其内容
- 如何使用 Border 控件

1. 堆栈顺序

容器控件包含多个子控件时，它们会以指定的堆栈顺序显示。读者应在绘图软件包中熟悉这个概念。理解堆栈顺序的最佳方式是假定每个控件放在一个玻璃盘上，而容器控件包含一叠这样的玻璃盘。因此，如果通过这些透明的玻璃从上向下看，就可以看到容器的内容。包含在容器中的控件重叠放置，因此看到的内容取决于玻璃盘的顺序。如果某个控件在该叠玻璃盘偏上的位置，就可以在重叠的区域中看到它。而位于该叠玻璃盘偏下位置的控件就会被其上的控件部分或全部覆盖掉。

用鼠标单击窗口时，这也会影响单击测试。考虑到控件的重叠，目标控件应总是位于堆栈最上面的那个控件。控件的堆栈顺序由它们显示在容器的子控件列表中的顺序确定。容器中的第一个子控件放在堆栈最下面的一层，最后一个控件放在堆栈最上面的一层。在第一个和最后一个子控件中间的其他子控件放在中间层上。在可以在 WPF 中使用的一些布局控件中，控件的堆栈顺序还有其他含义，如后面所述。

2. 对齐、页边距、填充和尺寸

前面的示例说明了 Margin、HorizontalAlignment 和 VerticalAlignment 如何在 Grid 容器中定位控件。还说明了如何使用 Height 和 Width 指定尺寸。这些属性和尚未探讨的 Padding 属性对所有的(或大多数)布局控件都很有用，但它们的使用方式不同。不同布局控件也可以为这些属性设置默认值。后面的各小节会介绍一些示例，但在此之前先介绍基本概念。

两个对齐属性确定了控件的对齐方式，但我们并没有列出这些属性的所有值。例如，HorizontalAlignment 可以设置为 Left、Right、Center 或 Stretch。Left 和 Right 会把控件定位在容器的左或右边界上，Center 把控件定位在中心，Stretch 会改变控件宽度，使其边界延伸到容器的边界上。VerticalAlignment 与此类似，其值有 Top、Bottom、Center 或 Stretch。

Margin 和 Padding 分别指定在控件的边界周围和控件边界的内部预留多少空间。前面的示例使用 Margin 指定控件相对于 Grid 左上角等地点的位置。这是有效的，因为把 HorizontalAlignment 设置为 Left，VerticalAlignment 设置为 Top，控件就会位于容器的左上角，而 Margin 在控件的边界周围插入了一个空隙。Padding 的用法类似，但在控件的内容与其边界之间空出了一些地方。这对 Border 控件尤其有用，如下一节所述。Margin 和 Padding 都可以指定为 4 个部分值(其形式是 leftAmount、topAmount、rightAmount、bottomAmount)或者指定为一个值(Thickness 值)。

Height 和 Width 常由其他属性控制。例如，把 HorizontalAlignment 设置为 Stretch，控件的 Width 属性就随着其容器宽度的变化而变化。

3. Border 控件

Border 控件是一个简单实用的容器控件，它包含一个子控件，而不像稍后介绍的更复杂的控件那样包含多个控件。这个子控件会完全填满 Border 控件。这似乎没有什么用，但可以使用 Margin 和 Padding 属性把 Border 定位在其容器中，把 Border 的内容定位在 Border 的边界中。也可以设置 Border 的 Background 属性，使之可见。稍后介绍这个控件。

4. Canvas 控件

如前所述，Canvas 控件对控件的定位提供了完全的自由。另外，用于某个子元素的 HorizontalAlignment 和 VerticalAlignment 属性不会影响其他元素的定位。

使用 Margin 可以在 Canvas 中定位元素，如前面的示例所示。但更好的方式是使用 Canvas 类的 Canvas.Left、Canvas.Top、Canvas.Right 和 Canvas.Bottom 关联属性：

```
<Canvas ...>
   <Button Canvas.Top="10" Canvas.Left="10" ...>Button1</Button>
</Canvas>
```

上面的代码定位了一个按钮，使其顶边距离 Canvas 的顶边 10 个像素，其左边界距离 Canvas 的左边界 10 个像素。注意 Top 和 Left 属性的优先级高于 Bottom 和 Right。例如，如果指定了 Top 和 Bottom 属性，就忽略 Bottom 属性。

图 25-10 显示了 Canvas 控件中定位的两个 Rectangle 控件，但窗口有两个不同尺寸。

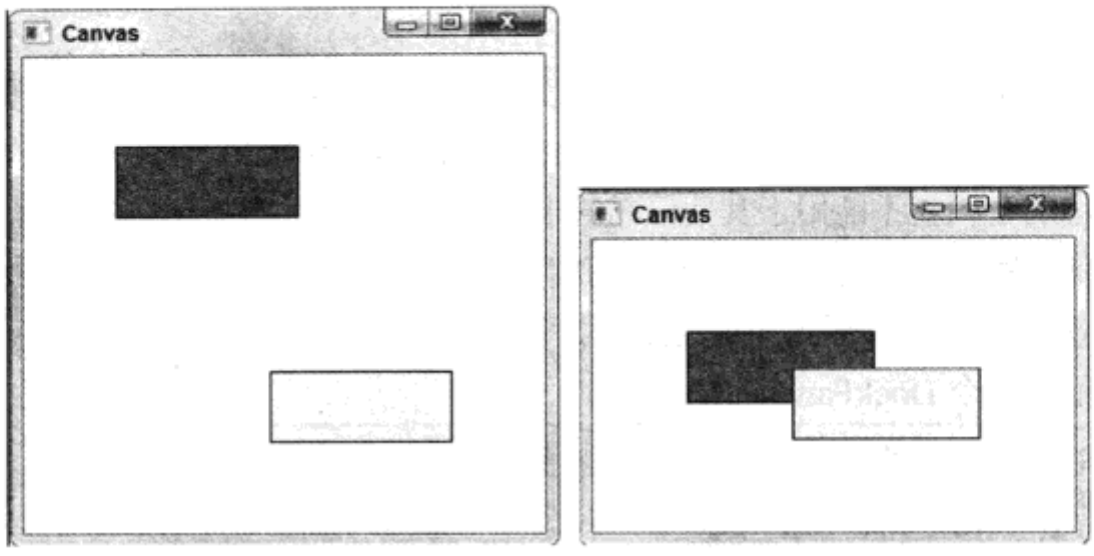


图　25-10

本节的所有示例布局都在本章下载代码的 LayoutExamples 项目中。

一个 Rectangle 相对于 Canvas 控件的左上角定位，另一个 Rectangle 相对于 Canvas 控件的右下角定位。在重新设置窗口的大小时，这些相对位置保持不变。还可以看出 Rectangle 控件的堆栈顺序的重要性。右下角的 Rectangle 在堆栈顺序中比较高，所以两个 Rectangle 重叠时，就可以看到右下角的 Rectangle。

这个示例的代码如下：

可从 wrox.com 下载源代码

```
<Canvas Background="AliceBlue">
  <Rectangle Canvas.Left="50" Canvas.Top="50" Height="40" Width="100"
    Stroke="Black" Fill="Chocolate" />
  <Rectangle Canvas.Right="50" Canvas.Bottom="50" Height="40" Width="100"
    Stroke="Black" Fill="Bisque" />
</Canvas>
```

代码段 LayoutExamples\CanvasWindow.xaml

5. DockPanel 控件

顾名思义，DockPanel 控件允许把控件停靠在它的一个边界上。读者即使以前从来没有注意过，也应很熟悉这类布局。Word 中的 Ribbon 控件就总是位于 Word 窗口的顶部，VS 和 VCE 中的各个窗口也是用这种方式定位的。在 VS 和 VCE 中，拖动窗口就会改变窗口的停靠方式。

DockPanel 有一个关联属性 DockPanel.Dock，子控件可以使用该属性指定它停靠在哪条边上。这个属性可以设置为 Left、Top、Right 和 Bottom。

DockPanel 中控件的堆栈顺序非常重要，因为每次把控件停靠在一条边上时，都会相应地减小后续子控件的可用空间。例如，把一个工具条停靠在 DockPanel 的顶部，再把第二个工具条停靠在 DockPanel 的左边。第一个控件会延伸到 DockPanel 显示区域的整个顶部，但第二个控件只能从第一个工具条的底部开始沿着 DockPanel 的左边界延伸到 DockPanel 的底部。

在前面的子控件都定位好后，最后一个指定的子控件通常会占据剩余的空间(可以控制这个

行为)。

在 DockPanel 中定位控件时，该控件占用的空间可能比 DockPanel 给该控件预留的区域小。例如，如果把 Width 为 100、Height 为 50、HorizontalAlingment 为 Left 的按钮停靠在 DockPanel 的顶部，则该按钮右边的空间就不能用于其他停靠的子控件。另外，如果按钮控件的Margin 为 20，就要在 DockPanel 顶部预留总共90 个像素(控件的高度加上边距和下边距)。在用 DockPanel 进行布局时，要考虑到这种情况；否则就得不到期望的结果。

图 25-11 显示了一个 DockPanel 布局示例。

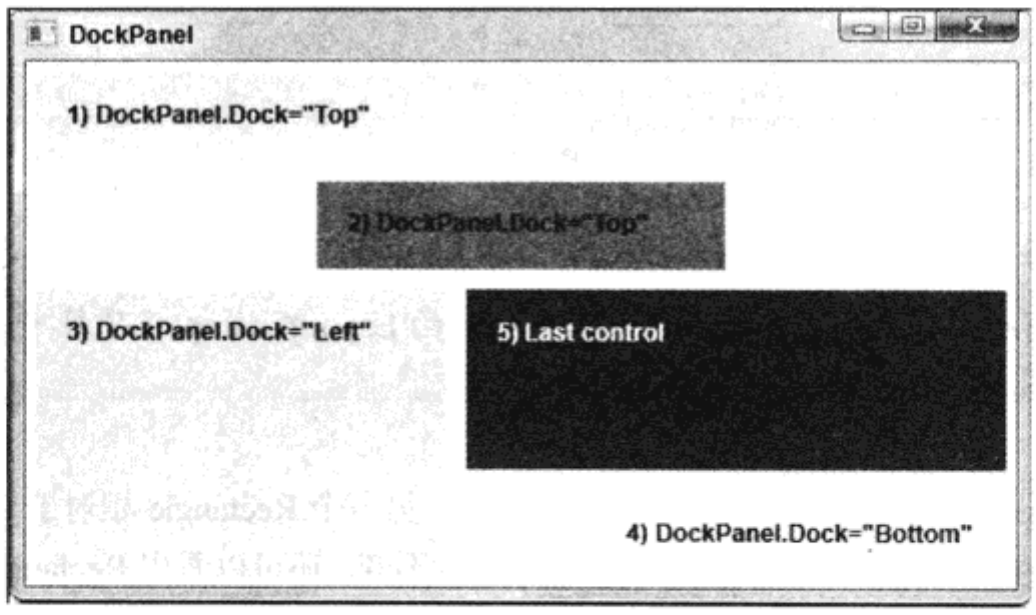


图 25-11

这个布局的代码如下所示：

可从 wrox.com 下载源代码

```
<DockPanel Background="AliceBlue">
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
   Background="Aquamarine" Height="45">
   <Label>1) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
   Background="PaleVioletRed" Height="45" Width="200">
   <Label>2) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Left" Padding="10" Margin="5"
   Background="Bisque" Width="200">
   <Label>3) DockPanel.Dock="Left"</Label>
  </Border>
  <Border DockPanel.Dock="Bottom" Padding="10" Margin="5"
   Background="Ivory" Width="200" HorizontalAlignment="Right">
   <Label>4) DockPanel.Dock="Bottom"</Label>
  </Border>
  <Border Padding="10" Margin="5" Background="BlueViolet">
   <Label Foreground="White">5) Last control</Label>
  </Border>
</DockPanel>
```

代码段 LayoutExamples\DockPanelWindow.xaml

这段代码在示例布局中使用前面介绍的Border 控件清楚地划分了停靠控件的区域，再使用 Label 控件输出简单的信息文本。为了理解该布局，必须自上而下读取代码，依次查看每个控件：

(1) 第一个 Border 控件停靠在 DockPanel 的顶部。DockPanel 中被占据的总区域是顶部的 55 个像素(Height+2×Margin)。注意，Padding 属性对这个布局没有影响，因为它在 Border 边界的内部，但这个属性控制着其内嵌 Lable 控件的定位。如果没有 Height 或 Width 属性的限制，Border 控件就会在 DockPanel 中延伸，填满 DockPanel 顶部的所有可用空间。

(2) 第二个 Border 控件也停靠在 DockPanel 的顶部，占据了显示区域顶部的另外 55 个像素。这个 Border 控件还包含 Width 属性，使 Border 仅占据 DockPanel 的部分宽度。它位于中心，因为 DockPanel 中的 HorizontalAlignment 的默认值是 Center。

(3) 第三个 Border 控件停靠在 DockPanel 的左边，占据显示区域左边的 210 个像素。

(4) 第四个 Border 控件停靠在 DockPanel 的底部，占据 30 个像素加上它包含的 Label 控件的高度。这个高度由 Margin、Padding 和 Border 控件的内容确定，因为它没有明确指定。Border 控件锁定在 DockPanel 的右下角，因为它的 HorizontalAlignment 为 Right。

(5) 第五个也是最后一个 Border 控件填满了剩余的空间。

运行这个示例，试着改变其内容的尺寸。注意，控件的堆栈顺序越大，获得其空间的优先级就越高。缩小窗口，第五个 Border 控件会很快被堆栈顺序较大的控件覆盖，使用 DockPanel 控制布局时要小心避免这种情况，例如，可以设置窗口的最小尺寸。

6. Grid 控件

Grid 控件可以包含多行和多列，用于布置子控件。本章前面已经使用过几次 Grid 控件，但所有示例都使用单行单列的 Grid 控件。要添加更多行和列，必须使用 RowDefinitions 和 ColumnDefinitions 属性，它们分别是 RowDefinition 和 ColumnDefinition 对象的集合，用属性元素语法指定：

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  ...
</Grid>
```

这段代码定义了一个包含 3 行 2 列的 Grid 控件。注意，这里不需要额外的信息。在这段代码中，Grid 控件重新设置大小时，都会自动重置每一行和每一列的大小。每一行的高度都是 Grid 控件高度的 1/3，每一列的宽度都是 Grid 控件宽度的一半。把 Grid.ShowGridlines 属性设置为 true，就可以在 Grid 的单元格之间显示线条。

Width、Height、MinWidth、MaxWidth、MinHeight 和 MaxHeight 可用于控制尺寸的重置。例如，设置列的 Width 属性可以确保该列的宽度保持不变。也可以把列的 Width 属性设置为*，表示“计算完其他列的宽度后填满剩余的空间”。这实际上是默认值。多个列的宽度都是*时，剩余的空间就由它们平分。*值也可以用于行的 Height 属性。Height 和 Width 的另一个值是 Auto，表示根据控件的内容设置行或列的大小。也可以使用 GridSplitter 控件，让用户通过单击和拖动操作来定制行和列的尺寸。

Grid 控件的子控件可以使用关联属性 Grid.Column 和 Grid.Row 指定它们包含在什么单元格中。这些属性的默认值都是 0，所以如果忽略这两个属性，子控件就位于左上单元格中。子控件也可以使用 Grid.ColumnSpan 和 Grid.RowSpan 放在表的多个单元格中，此时左上单元格由 Grid.Column 和 Grid.Row 指定。

图 25-12 中的 Grid 控件包含多个椭圆和一个 GridSplitter，且窗口有两个不同尺寸。

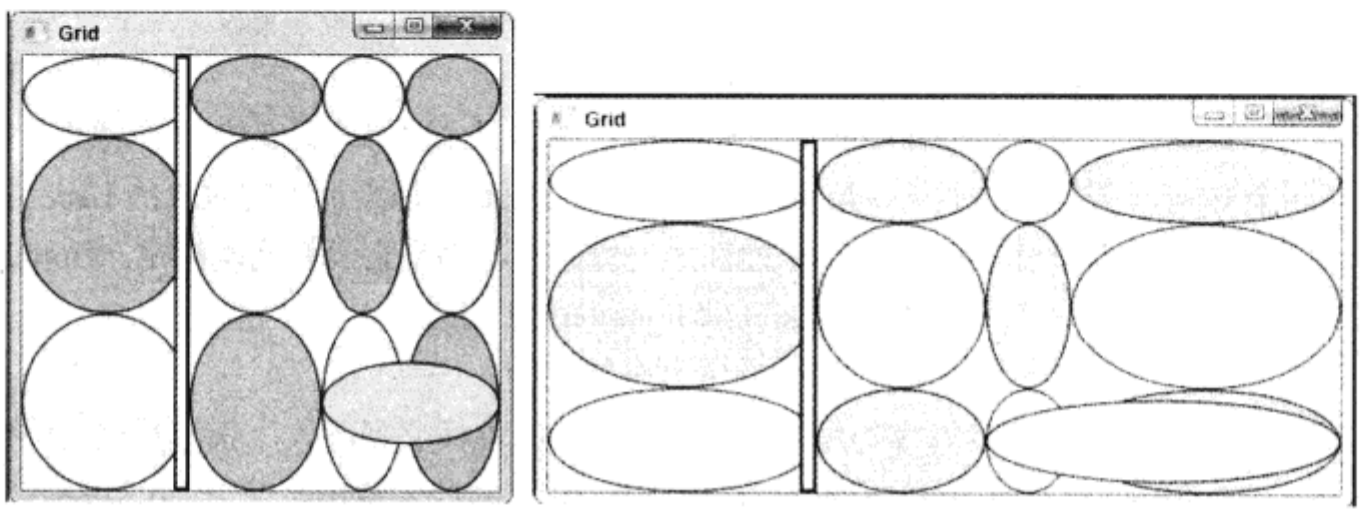


图 25-12

所用的代码如下：

可从 wrox.com 下载源代码

```
<Grid Background="AliceBlue">
  <Grid.ColumnDefinitions>
    <ColumnDefinition MinWidth="100" MaxWidth="200" />
    <ColumnDefinition MaxWidth="100" />
    <ColumnDefinition Width="50" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition MinHeight="100" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Ellipse Grid.Row="0" Grid.Column="0" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="1" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="2" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="3" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="0" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="1" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="2" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="3" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="0" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="1" Fill="BurlyWood"
```

```
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="2" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="3" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="2" Grid.ColumnSpan="2" Fill="Gold"
    Stroke="Black" Height="50"/>
  <GridSplitter Grid.RowSpan="3" Width="10" BorderThickness="2">
    <GridSplitter.BorderBrush>
      <SolidColorBrush Color="Black" />
    </GridSplitter.BorderBrush>
  </GridSplitter>
</Grid>
```

代码段 LayoutExamples\GridWindow.xaml

这段代码使用行和列定义上的各种属性，在重置显示区域的大小时，获得了有趣的效果，所以最好测试一下。

首先考虑行。顶行的高度是固定的 50 像素，第二行的最小高度设置为 100 像素，第三行填满了剩余的空间。这表示如果 Grid 的高度小于 150 像素，第三行就是不可见的。Grid 的高度在 150～250 像素之间时，只有第三行的尺寸会在 0～100 像素之间变化。这是因为剩余的空间应计算为总高度减去高度固定的行的高度之和。剩余的空间位于第二行和第三行之间，但因为第二行的最小高度是 100 像素，所以它的高度不会变化，除非 Grid 的总高度达到 250 像素。最后，Grid 的总高度超过 250 像素时，第二行和第三行会分享剩余的空间，所以它们的高度都等于或大于 100 像素。

接着分析列。只有第三列的尺寸是固定的 50 像素。第一列和第二列的总宽度是 300 像素。因此，在 Grid 控件的总宽度超过 550 像素时，只有第四列的尺寸会增大。为了弄明白这一点，应考虑列可用的像素值是多大，它们是如何分配的。首先，把 50 个像素分配给第三列，剩余的 500 像素分配给其余的列。第三列的最大宽度是 100 像素，第一列和第四列的宽度就剩下 400 像素。第一列的最大宽度是 200 像素，所以即使宽度超过了这个值，第一列也不会占用更多的空间。而第四列的尺寸会增加。

注意这个示例的另外两点。首先，最后一个定义的椭圆占据了第三列和第四列，以演示 Grid.ColumnSpan 的用法。其次，提供了一个 GridSplitter，允许重置第一列和第二列的大小。但是，一旦 Grid 控件的总宽度超过 550 像素，这个 GridSplitter 就不能设置这些列的大小，因为第一列和第二列都不能增大尺寸了。

GridSplitter 控件很有用，但其外观很呆板。这是一个需要设置样式的控件，至少要把其 Background 属性设置为 Transparent，使之不可见。

如果窗口中有多个 Grid 控件，还可以在行或列定义上使用 ShareSizeGroup 属性为行或列定义共享的尺寸组。ShareSizeGroup 属性可以设置为字符串标识符。例如，如果在一个 Grid 控件中，共享尺寸组中的一列改变了，则该组中另一个 Grid 控件的列也会改变，以匹配前一个 Grid 控件。通过 Grid.IsShareSizeScope 属性可以启用或禁用这个功能。

7. StackPanel 控件

了解到 Grid 控件的复杂性之后，读者会发现 StackPanel 控件是一个较简单的布局控件。可以将 StackPanel 控件看作 DockPanel 的一个删节版本，因为子控件停靠的边是固定的。这两个控件的另

一个区别是 StackPanel 的最后一个子控件不填满剩余的空间。但是，控件会默认延伸到 StackPanel 控件的边界上。

控件堆叠的方向由 3 个属性确定。Orientation 可以设置为 Horizontal 或 Vertical，HorizontalAlignment 和 VerticalAlignment 可以确定控件是沿着 StackPanel 的顶边、底边、左边还是右边定位。给所使用的对齐属性设置 Center 值，甚至可以使控件在 StackPanel 的中心处堆叠。

图 25-13 显示了两个 StackPanel 控件，它们都包含 3 个按钮。StackPanel 控件使用包含两行一列的 Grid 控件定位。

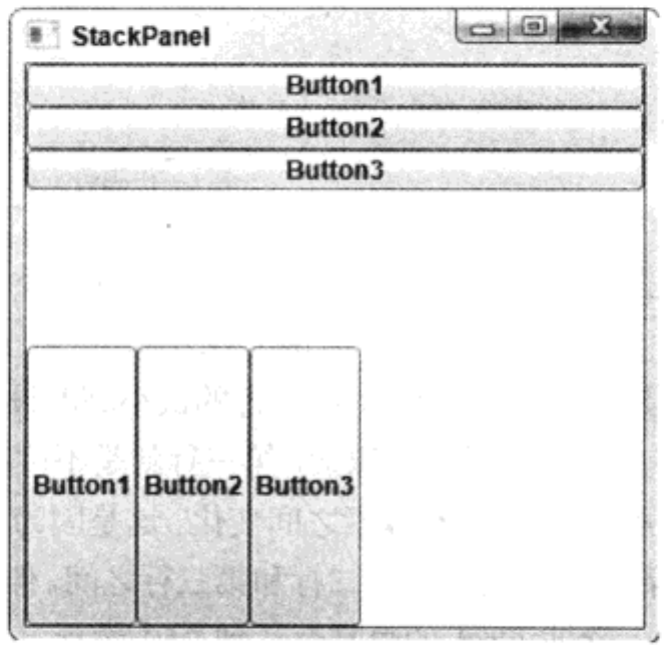


图 25-13

这里使用的代码如下：

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <StackPanel Grid.Row="0">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
  <StackPanel Grid.Row="1" Orientation="Horizontal">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Grid>
```

代码段 LayoutExamples\StackPanelWindow.xaml

使用 StackPanel 来布局时，经常需要添加滚动条，以便查看包含在 StackPanel 中的所有控件。WPF 为我们做了许多工作。使用 ScrollViewer 控件就可以添加滚动条——只需在这个控件中包含 StackPanel：

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
```

```
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <ScrollViewer>
    <StackPanel Grid.Row="0">
      <Button>Button1</Button>
      <Button>Button2</Button>
      <Button>Button3</Button>
    </StackPanel>
  </ScrollViewer>
  <StackPanel Grid.Row="1" Orientation="Horizontal">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Grid>
```

可以使用更复杂的技术以不同的方式滚动，或者用编程方式滚动，但上述代码常常就是需要编写的代码。

8. WrapPanel 控件

WrapPanel 实际上是 StackPanel 的一个扩展版本，其中“不合适”的控件会移动到附加的行或列上。图 25-14 显示的 WrapPanel 控件包含多个形状，而窗口设置为两个不同的尺寸。

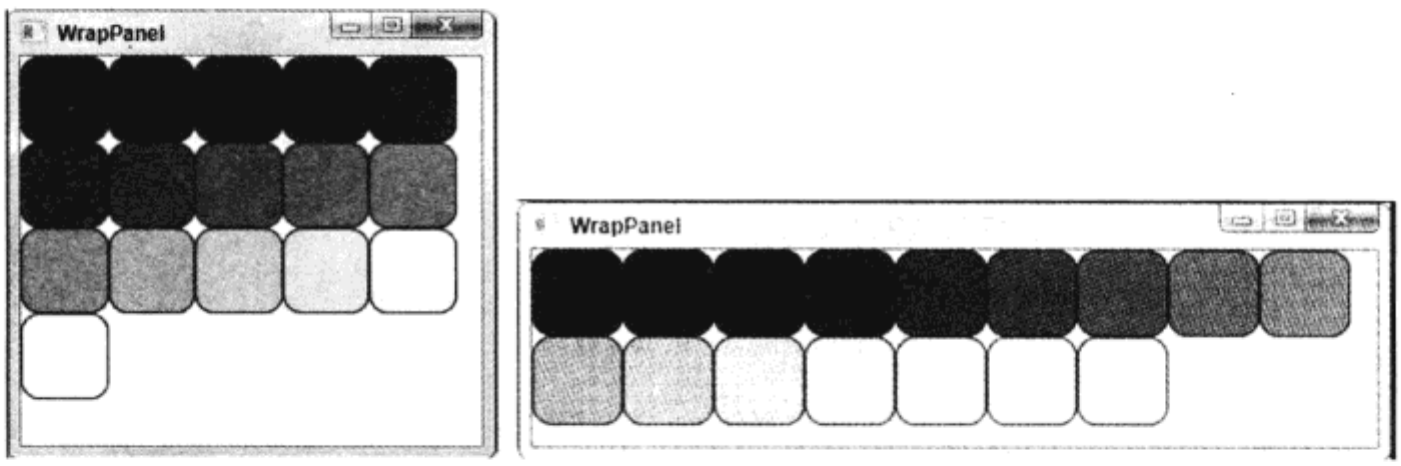


图　25-14

代码的缩减版本如下：

可从 wrox.com 下载源代码

```
<WrapPanel Background="AliceBlue">
  <Rectangle Fill="#FF000000" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF111111" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF222222" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  ...
  <Rectangle Fill="#FFFFFFFF" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
</WrapPanel>
```

代码段 LayoutExamples\WrapPanelWindow.xaml

WrapPanel 控件是创建动态布局的一种好方法，允许用户精确控制内容的显示方式。

25.3.6 控件的样式

WPF 的一个杰出特性是它允许设计人员全面控制用户界面的外观和操作方式。其核心是可以给控件设置任意二维或三维样式。前面一直使用的是.NET 3.5 为控件提供的基本样式，但控件可以使用的样式是无穷多的。

本节介绍两种基本技术：

- **样式**：成批应用于控件的属性组
- **模板**：用于建立控件外观的控件

这两种技术有些重叠，因为样式可以包含模板。

1. 样式

WPF 控件有一个 Style 属性(继承自 FrameworkElement)，它可以设置为 Style 类的实例。Style 类相当复杂，具有高级样式化功能，但其核心是一组 Setter 对象。每个 Setter 对象都根据 Property 属性(要设置的属性名)及 Value 属性(属性要设置的值)设置属性值。可以把在 Property 中使用的名称完全限定为控件类型(如Button.Foreground)，也可以设置Style 对象的TargetType 属性(如 Button)，使之可以解析属性名。

下面的代码说明了如何使用 Style 对象设置 Button 控件的 Foreground 属性：

```
<Button>
 Click me!
 <Button.Style>
  <Style TargetType="Button">
   <Setter Property="Foreground">
    <Setter.Value>
     <SolidColorBrush Color="Purple" />
    </Setter.Value>
   </Setter>
  </Style>
 </Button.Style>
</Button>
```

显然，在这段代码中，以通常的方式设置按钮的 Foreground 属性要简单得多。把样式转换为资源后，样式会非常有用，因为资源是可以重用的。本章后面将介绍其转换过程。

2. 模板

控件用模板构建，而模板是可以定制的。模板包含一个控件层次结构，用于建立控件的显示外观，该外观可能包含控件的内容显示器，例如，显示内容的按钮。

控件的模板常常存储在其 Template 属性中，Template 属性是 ControlTemplate 类的一个实例。ControlTemplate 类包含 TargetType 属性，它可以设置为用于定义模板的控件类型，这个属性可以包含单个控件，也可以是一个容器，例如 Grid，所以它不会限制我们的操作。

一般使用样式来设置类的模板。这只需用如下方式为要使用的控件提供 Template 属性：

```
<Button>
```

```
Click me!
<Button.Style>
  <Style TargetType="Button">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          ...
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
 </Button.Style>
</Button>
```

一些控件需要多个模板。例如，CheckBox 控件使用一个模板表示其复选框(CheckBox. Template)，使用另一个模板输出复选框旁边的文本(CheckBox. ContentTemplate)。

需要内容显示器的模板可以在输出内容的位置上包含 ContentPresenter 控件。一些控件，尤其是输出数据项集合的控件，使用另一种技术，本章不讨论该技术。

在与资源合并使用时，替代模板是很有用的。但是，设置控件的样式是一种非常常见的技术，需要用一个示例来说明。

试一试：使用样式和模板

(1) 创建一个新 WPF 应用程序 Ch25Ex03，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(2) 修改 MainWindow.xaml 的代码，如下所示：

可从
wrox.com
下载源代码

```
<Window x:Class="Ch25Ex03.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="Nasty Button" Height="150" Width="550">
 <Grid Background="Black">
  <Button Margin="20" Click="Button_Click">
   Would anyone use a button like this?
   <Button.Style>
    <Style TargetType="Button">
     <Setter Property="FontSize" Value="18" />
     <Setter Property="FontFamily" Value="arial" />
     <Setter Property="FontWeight" Value="bold" />
     <Setter Property="Foreground">
      <Setter.Value>
       <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <LinearGradientBrush.GradientStops >
         <GradientStop Offset="0.0" Color="Purple" />
         <GradientStop Offset="0.5" Color=" Azure" />
         <GradientStop Offset="1.0" Color=" Purple" />
        </LinearGradientBrush.GradientStops>
       </LinearGradientBrush>
      </Setter.Value>
     </Setter>
     <Setter Property="Template">
      <Setter.Value>
       <ControlTemplate TargetType="Button">
```

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="50" />
    <ColumnDefinition />
    <ColumnDefinition Width="50" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition MinHeight="50" />
  </Grid.RowDefinitions>
  <Ellipse Grid.Column="0" Height="50">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
          <GradientStop Offset="0.0" Color="Yellow" />
          <GradientStop Offset="1.0" Color="Red" />
        </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
 <Grid Grid.Column="1">
  <Rectangle RadiusX="10" RadiusY="10">
    <Rectangle.Fill>
      <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
          <GradientStop Offset="0.0" Color="Yellow" />
          <GradientStop Offset="1.0" Color="Red" />
        </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
                    <ContentPresenter Margin="20,0,20,0"
                        HorizontalAlignment="Center"
                        VerticalAlignment="Center" />
                  </Grid>
                  <Ellipse Grid.Column="2" Height="50">
                    <Ellipse.Fill>
                      <RadialGradientBrush>
                        <RadialGradientBrush.GradientStops>
                          <GradientStop Offset="0.0" Color="Yellow" />
                          <GradientStop Offset="1.0" Color="Red" />
                        </RadialGradientBrush.GradientStops>
                      </RadialGradientBrush>
                    </Ellipse.Fill>
                  </Ellipse>
                </Grid>
              </ControlTemplate>
            </Setter.Value>
          </Setter>
        </Style>
      </Button.Style>
    </Button>
  </Grid>
</Window>
```

代码段Ch25Ex03\MainWindow.xaml

(3) 修改 MainWindow.xaml.cs 中的代码，如下所示：

可从
wrox.com
下载源代码

```
public partial class MainWindow : Window
{
    ...

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Button clicked.");
    }
}
```

代码段 Ch25Ex03\MainWindow.xaml.cs

(4) 运行应用程序，单击一次按钮，结果如图 25-15 所示。

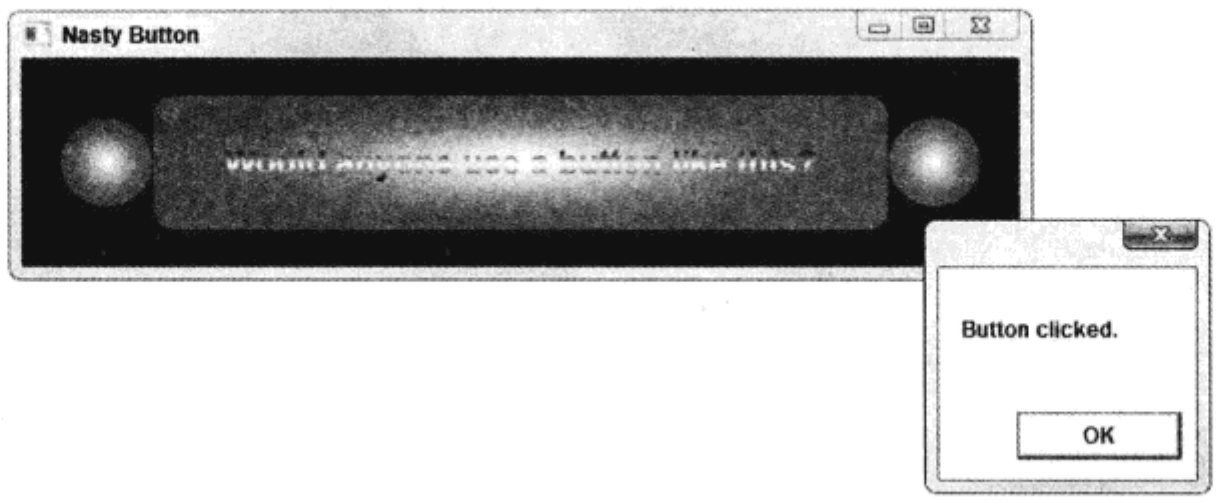


图　25-15

示例的说明

这个示例中的按钮非常难看。但是，抛去外观方面的因素，这个示例其实说明了可以在 WPF 中毫不费力地全面控制按钮的外观。但改变按钮的模板时，注意按钮的功能没有变化。即可以单击按钮，在事件处理程序中响应该单击。

注意在这里使用的模板上没有实现与 Windows 按钮关联的一些操作。例如，把鼠标停放在按钮上或单击按钮时，没有可视化的反馈。这个按钮无论是否获得焦点，其外观都是相同的。为了得到这些效果，需要学习触发器，这是下一节的内容。

但在此之前，先详细分析一下示例代码，主要关注样式和模板，看看模板是如何创建的。

刚开始的代码很普通，用于显示按钮控件：

```
<Button Margin="20" Click="Button_Click">
  Would anyone use a button like this?
```

这给按钮提供了基本属性和内容。接着，Style 属性设置为 Style 对象，开始为按钮控件设置 3 个简单的字体属性：

```
<Button.Style>
  <Style TargetType="Button">
    <Setter Property="FontSize" Value="18" />
    <Setter Property="FontFamily" Value="arial" />
    <Setter Property="FontWeight" Value="bold" />
```

接着，使用属性元素语法设置 Button.Foreground 属性，因为使用了一个笔刷：

```
<Setter Property="Foreground">
 <Setter.Value>
  <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
   <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.0" Color=" Purple" />
    <GradientStop Offset="0.5" Color="Azure" />
    <GradientStop Offset="1.0" Color="Purple" />
   </LinearGradientBrush.GradientStops>
  </LinearGradientBrush>
 </Setter.Value>
</Setter>
```

Style 对象的剩余代码将 Button.Template 属性设置为 ControlTemplate 对象：

```
      <Setter Property="Template">
       <Setter.Value>
        <ControlTemplate TargetType="Button">
            ...
        </ControlTemplate>
       </Setter.Value>
      </Setter>
     </Style>
    </Button.Style>
   </Button>
```

模板代码可以汇总为一个 Grid 控件，该控件在一行中有 3 个单元格。这些单元格包含一个 Ellipse、一个 Rectangle、模板的 ContentPresenter 和另一个 Ellipse：

```
<Grid>
 <Ellipse Grid.Column="0" Height="50">
  ...
 </Ellipse>
 <Grid Grid.Column="1">
  <Rectangle RadiusX="10" RadiusY="10">
   ...
  </Rectangle>
  <ContentPresenter Margin="20,0,20,0"
     HorizontalAlignment="Center"
     VerticalAlignment="Center" />
  </Grid>
  <Ellipse Grid.Column="2" Height="50">
   ...
  </Ellipse>
</Grid>
```

这些代码都较为简单，下面要进一步分析它们。

25.3.7 触发器

在本章的第一个示例中，学习了触发器可以把事件关联到动作上。WPF 中的事件可以包含所有的操作，包括按钮单击、应用程序启动和停止等。在 WPF 中有几类触发器，它们都继承自基类 TriggerBase。示例中使用的触发器类型是 EventTrigger。EventTrigger 类包含一个动作集合，每个动

作都是派生于基类 TriggerAction 的一个对象。这些动作都在激活触发器时执行。

在 WPF 中，并不是许多类都继承自 TriggerAction，当然，我们可以定义自己的类。可以使用 EventTrigger 通过 BeginStoryboard 动作触发动画，通过 ControllableStoryboardAction 处理故事板，通过 SoundPlayerAction 触发声音效果。最后一个触发器在动画中最常用，所以下一节介绍它。

每个控件都有 Triggers 属性，用于直接在该控件上定义触发器。也可以在该层次结构中进一步定义触发器，例如，在前面所示的 Window 对象上定义触发器。在给控件设置样式时，最常用的触发器类型是 Trigger(也可以使用 EventTrigger 引发控件动画)。Trigger 类用于设置属性，以响应其他属性的变化，在 Style 对象中使用该类时特别有用。

Trigger 对象的配置如下：

- 要定义 Trigger 对象监控的属性，使用 Trigger.Property 属性。
- 要定义 Trigger 对象的激活时间，设置 Trigger.Value 属性。
- 要定义 Trigger 对象执行的动作，把 Trigger.Setters 属性设置为 Setters 对象的集合。

这里的 Setter 对象就是前面“样式”一节中的 Setter 对象。

例如，下面的触发器将检查属性 MyBooleanValue 的值，在该属性为 true 时，把 Opacity 属性的值设置为 0.5：

```
<Trigger Property="MyBooleanValue" Value="true">
 <Setter Property="Opacity" Value="0.5" />
</Trigger>
```

这段代码并没有给出许多信息，因为它未与任何控件或样式关联。下面的代码较易理解，因为它说明了 Trigger 在 Style 对象中的用法：

```
<Style TargetType="Button">
 <Style.Triggers>
  <Trigger Property="IsMouseOver" Value="true">
   <Setter Property="Foreground" Value="Yellow" />
  </Trigger>
 </Style.Triggers>
</Style>
```

这段代码在 Button.IsMouseOver 属性为 true 时，把按钮控件的 Foreground 属性改为 Yellow。IsMouseOver 是几个非常有用的属性之一，可以用作查找控件信息和控件状态的快捷方式。顾名思义，如果鼠标停放在控件上，这个属性就是 true。此时可以给鼠标的停放操作编码。类似于这个属性的其他属性有 IsFocused，它确定控件是否获得焦点；IsHitTestVisible，它指定是否能单击某个控件(即该控件没有被堆栈顺序更高的控件遮住)；以及 IsPressed，它指定按钮是否被按下。最后一个属性仅应用于继承自 ButtonBase 的按钮，而其他属性可以应用于所有控件。

除了 Style.Triggers 属性之外，还可以使用 ControlTemplate.Triggers 属性获得许多信息。这个属性可以为包含触发器的控件创建模板。所以默认的 Button 模板可以响应鼠标的停放、单击和焦点的变化。我们必须修改这个模板，自己实现该功能。

25.3.8　动画

动画是用故事板创建的。毫无疑问，定义动画的最佳方式是使用 Expression Blend 等设计器。也可以直接编辑 XAML 代码，根据代码隐藏中的含义来定义动画(因为 XAML 只是建立 WPF 对象

模型的一种方式)。

用 Storyboard 对象定义故事板，Storyboard 对象包含一个或多个时间线。定义时间线可以使用关键帧，也可以使用几个更简单的对象来封装整个动画。复杂的故事板甚至可以包含嵌套的故事板。

如示例所示，Storyboard 包含在一个资源目录中，所以必须用 x:Key 属性标识它。

在故事板的时间线中，可以连续改变应用程序中的任意元素的动画属性，属性类型可以是 double、Point 或 Color。这涵盖了需要连续改变的所有元素属性，所以非常灵活。有一些操作不能执行，例如，用一种笔刷完全替代另一种笔刷，但只要有了这 3 种类型，就总有办法达到需要的任何效果。

这 3 种类型都有两个关联的时间线控件，可用作 Storyboard 的子控件。这 6 个控件是 DoubleAnimation 、 DoubleAnimationUsingKeyFrames 、 PointAnimation 、 PointAnimationUsingKeyFrames、ColorAnimation 和 ColorAnimationUsingKeyFrames。每个时间线控件都可以使用关联属性 Storyboard.TargetName 和 Storyboard.TargetProperty 关联到特定控件的特定属性上。例如，如果要连续改变一个 Rectangle 控件(其 Name 属性是 MyRectangle)的 Width 属性，就可以把 Storyboard.TargetName 和 Storyboard.TargetProperty 属性设置为 MyRectangle 和 Width。也可以使用 DoubleAnimation 或 DoubleAnimationUsingKeyFrames 连续改变这个属性。

Storyboard.TargetProperty 属性可以解释相当高级的语法，以便定位动画中感兴趣的属性。在本章开头的示例中，为两个关联属性使用了下面的值：

```
Storyboard.TargetName="ellipse1"
Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0]
  .(RotateTransform.Angle)"
```

控件 ellipse1 的类型是 Ellipse，TargetProperty 指定椭圆在变换时旋转的角度。这个角度通过 Ellipse 的 RenderTransform 属性(继承自 UIElement)和 TransformGroup 对象的第一个子对象(就是这个属性的值)来定位。第一个子对象是 RotateTransform，角度是这个对象的 Angle 属性。

这个语法比较长，但用起来很简单。最困难的地方是确定给定的属性继承自哪个基类，但对象浏览器可以提供这方面的帮助。

下面看看无关键帧的简单动画时间线，然后再介绍使用关键帧的时间线。

1. 没有关键帧的时间线

没有关键帧的时间线是 DoubleAnimation、PointAnimation 和 ColorAnimation。这些时间线有相同的属性名，但这些属性的类型随时间线的类型而异(注意，所有的持续时间属性在 XAML 代码中都以格式[days.]hours:minutes:seconds 指定)，如表 25-2 所示。

表 25-2

属 性	用 法
Name	时间线的名称，用于在其他地方引用它
BeginTime	在触发故事板之后、时间线启动之前的时间
Duration	时间线的持续时间
AutoReverse	时间线完成时是否倒回，是否把属性返回为初始值。这个属性是一个布尔值

(续表)

属　性	用　法
RepeatBehavior	把这个属性设置为指定的持续时间，会使时间线根据指定的值重复执行——一个整数后跟 x(例如 5x)表示时间线重复指定的次数；也可以使用 Forever，让时间线重复执行到暂停或停止故事板为止
FillBehavior	如果时间线完成时，故事板仍旧在执行，时间线该如何操作。使用 HoldEnd 可以在时间线完成时不改变属性值(默认)，使用 Stop 可以把属性值返回为初始值
SpeedRatio	控制动画相对于其他属性的指定值的执行速度。默认为 1，但可以在其他代码中修改它，加快或减慢动画的执行速度
From	在动画开始时属性的初始值。可以忽略这个值，使用属性的当前值
To	在动画结束时属性的最终值。可以忽略这个值，使用属性的当前值
By	使用这个值使属性从当前值连续变化为当前值与指定值之和。这个属性可以单独使用，也可以与 From 属性结合使用

例如，下面的时间线使 Rectangle 控件(其 Name 属性是 MyRectangle)的 Width 属性在 5 秒内在 100~200 之间连续变化：

```
<Storyboard x:Key="RectangleExpander">
<DoubleAnimation Storyboard.TargetName="MyRectangle"
   Storyboard.TargetProperty="Width" Duration="00:00:05"
   From="100" To="200" />
</Storyboard>
```

2. 有关键帧的时间线

有关键帧的时间线是 DoubleAnimationUsingKeyFrames、PointAnimationUsingKeyFrames 和 ColorAnimationUsingKeyFrames。这些时间线类使用的属性与上一节的时间线类的相同，但没有 From、To 和 By 属性，而是有一个 KeyFrames 属性(它是关键帧对象的集合)。

这些时间线可以包含任意多个关键帧，每个关键帧都可以用不同的方式改变属性值。每类时间线都有 3 种关键帧：

- **Discrete**：离散的关键帧会使连续变化的值跳到指定的值上，没有任何切换。
- **Linear**：线性的关键帧会使连续变化的值以线性变换方式连续改变为指定的值。
- **Spline**：样条曲线的关键帧会使连续变化的值按照三次贝塞尔曲线函数定义的非线性变换方式，连续改变为指定的值。

因此关键帧对象有 9 种：DiscreteDoubleKeyFrame、LinearDoubleKeyFrame、SplineDoubleKeyFrame、DiscreteColorKeyFrame、LinearColorKeyFrame、SplineColorKeyFrame、DiscretePointKeyFrame、LinearPointKeyFrame 和 SplinePointKeyFrame。

关键帧类的 3 个属性与上一节介绍的时间线类的相同，但样条曲线的关键帧还有一个额外的属性，如表 25-3 中所示。

表 25-3

属 性	用 法
Name	关键帧的名称，用于在其他位置引用关键帧
KeyTime	关键帧的位置，表示为时间线启动后经历的时间
Value	到达关键帧时的属性值，或到达关键帧时应设置的属性值
KeySpline	两组 cp1x,cp1y cp2x,cp2y 形式的数字，它们定义了用于连续改变属性的三次贝塞尔函数(仅用于样条曲线的关键帧)

例如，连续改变一个方块中 Ellipse 的 Center 属性(其类型是 Point)，就可以建立该 Ellipse 的位置动画：

```
<Storyboard x:Key="EllipseMover">
 <PointAnimationUsingKeyFrames Storyboard.TargetName="MyEllipse"
  Storyboard.TargetProperty="Center" RepeatBehavior="Forever">
  <LinearPointKeyFrame KeyTime="00:00:00" Value="50,50" />
  <LinearPointKeyFrame KeyTime="00:00:01" Value="100,50" />
  <LinearPointKeyFrame KeyTime="00:00:02" Value="100,100" />
  <LinearPointKeyFrame KeyTime="00:00:03" Value="50,100" />
  <LinearPointKeyFrame KeyTime="00:00:04" Value="50,50" />
 </PointAnimationUsingKeyFrames>
</Storyboard>
```

Point 值在 XAML 代码中以 x,y 的形式指定。

25.3.9 静态和动态资源

WPF 的另一个优秀特性是可以定义资源，如控件样式和模板，它们可以在应用程序中重用。如果在正确的地方定义资源，还可以在多个应用程序中使用它们。

资源定义为 ResourceDictionary 对象中的项。顾名思义，这是指定了键的对象集合。因此在本章前面的示例代码中定义资源时使用 x:Key 特性：指定与资源关联的键。可以在多个不同位置访问 ResourceDictionary 对象。可以把资源包含在本地控件中、本地窗口中、本地应用程序中或者放在外部程序集中。

引用资源有两种方式：静态和动态。注意这个区别并不意味着资源本身有任何区别，这说明不能把资源定义为静态的或动态的。区别在于如何使用它。

1. 静态资源

在设计期间知道使用什么资源，而且知道该引用不会在应用程序的生存期中变化时，就使用静态资源。例如，如果定义了一个按钮样式，将其用于应用程序中的按钮，在应用程序运行时就不太可能改变它。此时，就应静态引用资源。另外，使用静态资源时，资源类型在编译期间解析，所以性能很好。

要引用静态资源，可以使用下面的标记扩展语法：

```
{StaticResource resourceName}
```

例如，如果为按钮控件定义了一个样式，将其 x:Key 特性设置为 MyStyle，就可以在控件中引用它，如下所示：

```
<Button Style="{StaticResource MyStyle}" ...>...</Button>
```

2. 动态资源

用动态资源定义的属性可以在运行期间改变为另一个动态资源。这在许多情况下都是很有用的。有时希望用户能控制应用程序的一般主题，此时就可以动态分配资源。另外，有时在运行期间并不关心资源需要的键，例如，动态关联了资源程序集。

因此动态资源比静态资源更灵活。但是，动态资源有一个缺点：使用动态资源的系统开销略大，所以如果要优化应用程序的性能，在使用动态资源时应该极其谨慎。

动态引用资源的语法非常类似于静态引用资源的语法：

```
{DynamicResource resourceName}
```

例如，如果为按钮控件定义了一个样式，将其 x:Key 特性设置为 MyDynamicStyle，就可以在控件中引用它，如下所示：

```
<Button Style="{DynamicResource MyDynamicStyle}" ...>...</Button>
```

3. 引用样式资源

前面介绍了如何动态和静态引用按钮控件中的样式资源。这里使用的样式资源位于本地 Window 控件的 Resources 属性中，例如：

```
<Window ...>
  <Window.Resources>
    <Style x:Key="MyStyle" TargetType="Button">
      ...
    </Style>
  </Window.Resources>
    ...
</Window>
```

每个要使用这个控件样式的按钮控件都必须在其 Style 属性中引用它(以静态或动态方式)。另外，还可以定义一个样式资源，使之对给定的控件类型是全局可用的。即 Style 对象应用于应用程序中给定类型的每个控件。为此，只需忽略 x:Key 特性：

```
<Window ...>
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
    ...
</Window>
```

这是给应用程序添加主题的好方法。我们可以为所使用的各种控件类型定义一组全局样式，这样在任意地方都可以使用它们。

前几节讨论了许多基础知识，下面在一个示例中综合使用它们。这个示例修改了上一个示例中的按钮控件，使用触发器和动画，并把样式定义为全局可重用的资源。

试一试：触发器、动画和资源

(1) 创建一个新的 WPF 应用程序 Ch25Ex04，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(2) 将 Ch25Ex03 的 MainWindow.xaml 代码复制到 Ch25Ex04 的 MainWindow.xaml 中，但修改 Window 元素上的名称空间引用，如下所示：

```
<Window x:Class="Ch25Ex04.MainWindow"
```

代码段 Ch25Ex04\MainWindow.xaml

(3) 将Ch25Ex03 的 MainWindow.xaml.cs 中的 Button_Click()事件处理程序复制到 Ch25Ex04 的 MainWindow.xaml.cs 中。

(4) 在<Window>元素上添加一个<Window.Resources>子元素，把<Style>定义从<Button.Style>元素移动到<Window.Resources>元素中。删除空的<Button.Style>元素。结果如下所示(删节)：

```
<Window x:Class="Ch25Ex04.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Nasty Button" Height="150" Width="550">
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  <Grid Background="Black">
    <Button Margin="20" Click="Button_Click">
      Would anyone use a button like this?
    </Button>
  </Grid>
</Window>
```

(5) 运行应用程序，验证结果与上一个示例相同。

(6) 在模板的主 Grid 控件和包含 ContentPresenter 元素的 Rectangle 中添加 Name 属性：

```
<Setter Property="Template">
  <Setter.Value >
    <ControlTemplate TargetType="Button">
      <Grid Name="LayoutGrid">
        <Grid.ColumnDefinitions>
          ...
        <Grid Grid.Column="1">
          <Rectangle RadiusX="10" RadiusY="10" Name="BackgroundRectangle">
          <Rectangle.Fill>
            ...
          </Rectangle.Fill>
        </Rectangle>
          ...
      </Grid>
```

```
      ...
    </Grid>
  </ControlTemplate>
 </Setter.Value>
</Setter>
```

(7) 在<ControlTemplate>元素中，在</ControlTemplate>标记的前面添加如下代码：

```
  </Grid>
  <ControlTemplate.Resources>
    <Storyboard x:Key="PulseButton">
      <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
        RepeatBehavior="Forever"
        Storyboard.TargetName="BackgroundRectangle"
        Storyboard.TargetProperty=
"(Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
        <LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
        <LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
        <LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
      </ColorAnimationUsingKeyFrames>
    </Storyboard>
  </ControlTemplate.Resources>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="LayoutGrid" Property="Effect">
        <Setter.Value>
          <DropShadowEffect ShadowDepth="0" Color="Red"
            BlurRadius="40" />
        </Setter.Value>
      </Setter>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter TargetName="LayoutGrid" Property="Effect">
        <Setter.Value>
          <DropShadowEffect ShadowDepth="0" Color="Yellow"
          BlurRadius="80" />
        </Setter.Value>
      </Setter>
    </Trigger>
    <EventTrigger RoutedEvent="UIElement.MouseEnter">
      <BeginStoryboard Storyboard="{StaticResource PulseButton}"
        x:Name="PulseButton_BeginStoryboard" />
    </EventTrigger>
    <EventTrigger RoutedEvent="UIElement.MouseLeave">
      <StopStoryboard
        BeginStoryboardName="PulseButton_BeginStoryboard" />
    </EventTrigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

(8) 运行应用程序，把鼠标停放在按钮上，如图 25-16 所示，按钮会闪动并发光。

(9) 单击按钮，如图 25-17 所示，发光效果改变了。

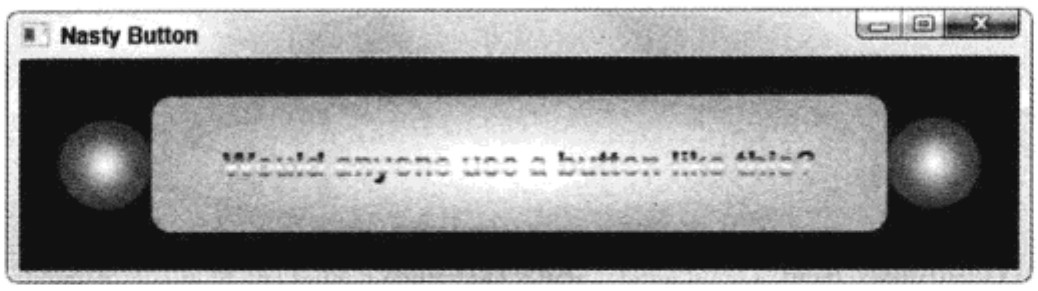


图 25-16

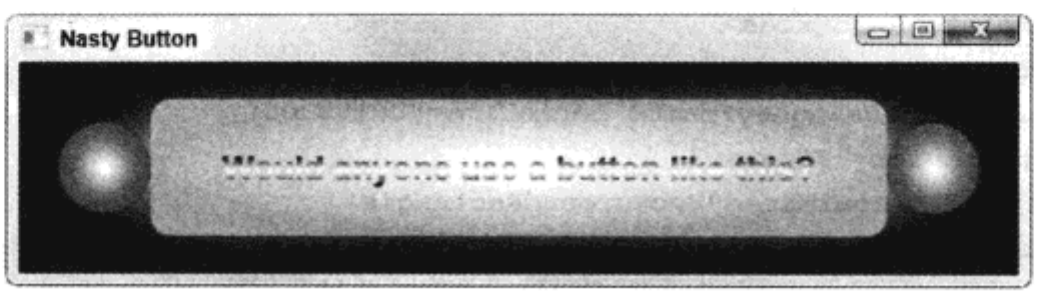


图 25-17

示例的说明

这个示例完成了两个工作。第一，定义了一个全局资源，用于格式化应用程序中的所有按钮(目前只有一个按钮，但这不重要)。第二，给上一个示例创建的样式添加了一些特性，使该样式能基本被接受。该样式会闪动并发光，以响应鼠标停放在按钮上和单击按钮操作。

把样式设置为全局资源，只需把<Style>样式移动到 Window 的资源部分。可以添加 x:Key 特性，但因为这里不需要设置页面上按钮控件的 Style 属性，所以该样式立即可以全局使用。

把样式设置为资源后，还要修改它。首先，给样式中的两个控件添加 Name 特性。这样才能在其他代码中引用它们，本例要为样式的控件模板在动画和触发器中引用它们。

接着，添加一个动画，作为样式中指定的控件模板的一个本地资源。动画 Storyboard 对象用 PulseButton 的 x:Key 值标识：

```
<ControlTemplate.Resources>
  <Storyboard x:Key="PulseButton">
```

故事板包含一个 ColorAnimationUsingKeyFrames 元素，它可以连续改变控件模板中使用的颜色。连续改变的属性是红色，它是在 BackgroundRectangle 控件中使用的放射性填充的外部颜色。在控件中定位这个属性需要为 Storyboard.TargetProperty 关联属性使用相当复杂的语法：

```
<ColorAnimationUsingKeyFrames BeginTime="00:00:00"
  RepeatBehavior="Forever"
  Storyboard.TargetName="BackgroundRectangle"
  Storyboard.TargetProperty=
"(Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
```

动画的时间线包含 3 个关键帧，在 2 秒内把颜色从 Red 连续改变为 Orange，再从 Orange 连续改变为 Red：

```
    <LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
    <LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
    <LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
  </ColorAnimationUsingKeyFrames>
 </Storyboard>
```

```
        </ControlTemplate.Resources>
```

把动画添加为资源，就不会执行动画了。要执行动画，需要添加两个 EventTrigger 触发器：

```
<EventTrigger RoutedEvent="UIElement.MouseEnter">
  <BeginStoryboard Storyboard="{StaticResource PulseButton}"
    x:Name="PulseButton_BeginStoryboard" />
</EventTrigger>
<EventTrigger RoutedEvent="UIElement.MouseLeave">
  <StopStoryboard
    BeginStoryboardName="PulseButton_BeginStoryboard" />
</EventTrigger>
```

这段代码使用 Button 控件的基类 UIElement 中的 MouseEnter 和 MouseLeave 事件，来控制动画的执行。MouseEnter 通过 BeginStoryboard 元素启动动画，MouseLeave 通过 StopStoryboard 元素停止动画。

注意，使用静态资源引用来定位故事板资源。这是很合理的，因为故事板定义为控件的本地资源，不需要在运行期间改变它。

还定义了另外两个触发器，使用 DropShadowEffect 效果产生鼠标停放在按钮上和单击按钮时的发光效果。这里使用本章前面介绍的 IsMouseOver 和 IsPressed 属性来达到这个效果：

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Red"
        BlurRadius="40" />
    </Setter.Value>
  </Setter>
</Trigger>
<Trigger Property="IsPressed" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Yellow"
        BlurRadius="80" />
    </Setter.Value>
  </Setter>
</Trigger>
```

鼠标停放在按钮上时，这里定义的发光效果比较小，显示为红色；单击按钮时，该发光效果比较大，显示为黄色。

25.4 用 WPF 编程

学习了 WPF 编程的所有基本技术后，就可以开始创建自己的应用程序了。但这里没有足够的篇幅介绍 WPF 的其他优秀特性，如数据绑定、格式化列表显示的一些好方法。但是，熟悉了 WPF 编程后，不应裹足不前，因此在完成本章之前，我们还要介绍两个主题，因为它们不太复杂，而且是 WPF 应用程序中经常执行的任务。

- 如何创建和使用自己的控件

- 如何在控件上实现依赖属性

最后还要举一个例子来说明本章介绍的许多技术，并简要讨论 WPF 数据绑定。

25.4.1 WPF 用户控件

WPF 提供了一组在许多情况下都很有用的控件。但是与所有的.NET 开发框架一样，WPF 也允许扩展这个功能。确切地讲，我们可以从 WPF 类层次结构的类中派生自己的类，从而创建自己的控件。

可以派生的一个最有用的控件是 UserControl。这个类提供了 WPF 控件需要的所有基本功能，允许控件无缝地捕捉到已有的 WPF 控件上。希望通过 WPF 控件获得的所有功能，如动画、样式设置、模板化等，都可以通过用户控件获得。

使用 Project | Add User Control 菜单项可以在项目中添加用户控件。这个菜单项会提供一个空白的画布(实际上是一个空白的 Grid)。用户控件在 XAML 中使用顶级的 UserControl 元素定义，代码隐藏类派生于 System.Windows.Controls.UserControl 类。

在项目中添加了用户控件后，就可以添加布局控件和配置控件的代码隐藏内容了。之后，就可以在应用程序中使用用户控件了，甚至可以在其他应用程序中重用它。

创建用户控件时，必须知道的一个重要事项是如何实现依赖属性。如本章前面所述，依赖属性是 WPF 编程的一个重要组成部分。在创建自己的控件时，一般不希望错过这些属性提供的功能。

25.4.2 实现依赖属性

可以在继承自 System.Windows.DependencyObject 类的任何类中添加依赖属性。这个类在 WPF 的许多类的继承层次结构中，包括所有控件和 UserControl。

为了给类实现依赖属性，需要在类型 System.Windows.DependencyProperty 的类定义中添加一个公共静态成员。这个成员的名称由开发人员确定，但最好遵循命名约定<PropertyName> Property：

```
public static DependencyProperty MyStringProperty;
```

把这个属性定义为静态似乎很古怪，因为可以为类的每个实例唯一地定义该属性。WPF 属性框架会自动进行跟踪，所以无需担心它。

必须用静态方法 DependencyProperty.Register()配置刚才添加的成员：

```
public static DependencyProperty MyStringProperty =
    DependencyProperty.Register(...);
```

这个方法带有 3～5 个参数，如表 25-4 所示(按照顺序，前 3 个参数是必选参数)。

表 25-4

参 数	用 法
string name	属性名
Type propertyType	属性的类型
Type ownerType	包含属性的类的类型
PropertyMetadata typeMetadata	其他属性设置：属性的默认值，用于属性改变通知和强制转换的回调方法
ValidateValueCallback validateValueCallback	用于验证属性值的回调方法

可以使用其他方法注册依赖属性，如 RegisterAttached()，它可以用于实现关联属性。本章不介绍其他方法，但读者应知道它们。

例如，可以使用如下 3 个参数注册 MyStringProperty 依赖属性：

```
public class MyClass : DependencyObject
{
  public static DependencyProperty MyStringProperty = DependencyProperty.Register(
       "MyString",
       typeof(string),
       typeof(MyClass));
}
```

也可以包含能直接访问依赖属性的.NET 属性(但这不是强制的，如后面所示)。但是，因为依赖属性定义为静态成员，所以不能对它们使用与普通属性相同的语法。要访问依赖属性的值，必须使用从 DependencyObject 继承的方法，如下所示：

```
public class MyClass : DependencyObject
{
   public static DependencyProperty MyStringProperty = DependencyProperty.Register(
     "MyString",
     typeof(string),
     typeof(MyClass));

     public string MyString
     {
        get { return (string)GetValue(MyStringProperty); }
        set { SetValue(MyStringProperty, value); }
     }
}
```

其中 GetValue()和 SetValue()方法分别获取和设置当前实例的 MyStringProperty 依赖属性值。这两个方法都是公共的，所以客户代码可以直接使用它们处理依赖属性的值。因此添加一个.NET 属性来访问依赖属性并非是强制的。

如果要为属性设置元数据，就必须使用派生于 PropertyMetadata 的对象，如 FrameworkPropertyMetadata，把这个实例作为第四个参数传送给 Register()。FrameworkPropertyMetadata 构造函数有 11 个重载版本，它们带有一个或多个参数，如表 25-5 所示。

表　25-5

属 性 类 型	用　　法
object defaultValue	属性的默认值
FrameworkProperty-MetadataOptionsflags	标记的组合(来自于 FrameworkPropertyMetadataOptions 枚举)，用于指定属性的其他元数据。例如，使用 AffectArrange 可以声明对属性的改变会影响控件的布局。这会使窗口的布局引擎在属性改变时重新计算控件的布局。MSDN 文档中列出了这个选项的完整列表

(续表)

属 性 类 型	用 法
propertyChangedCallback propertyChangedCallback	属性值改变时使用的回调方法
CoerceValueCallback coerceValueCallback	属性值强制转换时使用的回调方法
bool isAnimationProhibited	指定这个属性是否被动画改变
UpdateSourceTrigger defaultUpdateSourceTrigger	对属性值进行数据绑定操作时，这个属性根据枚举 UpdateSourceTrigger 中的值确定何时更新数据源。默认值是 PropertyChanged，表示只要属性改变，就更新绑定源。这未必合适，例如，TextBox.Text 属性给这个属性使用 LostFocus 值，确保绑定源并不过早更新。还可以使用 Explicit 值指定绑定源仅在请求时更新(调用派生于 DependancyObject 的类的 UpdateSource()方法)

下面是使用 FrameworkPropertyMetadata 的一个简单示例，它只设置了属性的默认值：

```
public class MyClass : DependencyObject
{
   public static DependencyProperty MyStringProperty =
   DependencyProperty.Register(
      "MyString",
      typeof(string),
      typeof(MyClass),
      new FrameworkPropertyMetadata("Default value"));
}
```

前面学习了可以指定的 3 个回调方法，分别用于属性改变通知、属性强制转换和属性值的验证。这些回调方法与依赖属性一样，必须实现为公共静态方法。每个回调方法都有一个指定的返回类型和参数列表。

下面的示例要创建并使用一个用户控件，它有两个依赖属性。我们将学习如何在用户控件代码中为这些属性实现回调方法。

试一试：用户控件

(1) 创建一个新的 WPF 应用程序 Ch25Ex05，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(2) 在应用程序中添加一个新的用户控件 Card，修改 Card.xaml 中的代码，如下所示：

```
<UserControl x:Class="Ch25Ex05.Card"
  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="150" d:DesignWidth="100"
  Height="150" Width="100" x:Name="UserControl"
  FontSize="16" FontWeight="Bold">
  <UserControl.Resources>
```

```
  <DataTemplate x:Key="SuitTemplate">
    <TextBlock Text="{Binding}"/>
  </DataTemplate>
</UserControl.Resources>
<Grid>
  <Rectangle Stroke="{x:Null}" RadiusX="12.5" RadiusY="12.5">
    <Rectangle.Fill>
      <LinearGradientBrush EndPoint="0.47,-0.167" StartPoint="0.86,0.92">
        <GradientStop Color="#FFD1C78F" Offset="0"/>
        <GradientStop Color="#FFFFFFFF" Offset="1"/>
      </LinearGradientBrush>
    </Rectangle.Fill>
    <Rectangle.Effect>
      <DropShadowEffect/>
    </Rectangle.Effect>
  </Rectangle>
  <Label x:Name="SuitLabel"
    Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Margin="8,51,8,60" />
  <Label x:Name="RankLabel"
    Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="8,8,0,0" />
  <Label x:Name="RankLabelInverted"
    Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Right" VerticalAlignment="Bottom"
    Margin="0,0,8,8" RenderTransformOrigin="0.5,0.5">
    <Label.RenderTransform>
      <RotateTransform Angle="180"/>
    </Label.RenderTransform>
  </Label>
  <Path Fill="#FFFFFFFF" Stretch="Fill" Stroke="{x:Null}"
    Margin="0,0,35.218,-0.077" Data="F1 M110.5,51 L123.16457,51 C116.5986,
      76.731148 115.63518,132.69684 121.63533,149.34013 133.45299,
      182.12018 152.15821,195.69803 161.79765,200.07669 L110.5,200 C103.59644,
      200 98,194.40356 98,187.5 L98,63.5 C98,56.596439 103.59644,51 110.5,51 z">
    <Path.OpacityMask>
      <LinearGradientBrush EndPoint="0.957,1.127" StartPoint="0,-0.06">
        <GradientStop Color="#FF000000" Offset="0"/>
        <GradientStop Color="#00FFFFFF" Offset="1"/>
      </LinearGradientBrush>
    </Path.OpacityMask>
  </Path>
</Grid>
</UserControl>
```

代码段 Ch25Ex05\Card.xaml

(3) 修改 Card.xaml.cs 中的代码，如下所示：

```
public partial class Card : UserControl
{
    public static string[] Suits = { "Club", "Diamond", "Heart", "Spade" };

    public static DependencyProperty SuitProperty = DependencyProperty.Register(
        "Suit",
        typeof(string),
        typeof(Card),
        new PropertyMetadata("Club", new PropertyChangedCallback(OnSuitChanged)),
        new ValidateValueCallback(ValidateSuit));

    public static DependencyProperty RankProperty = DependencyProperty.Register(
        "Rank",
        typeof(int),
        typeof(Card),
        new PropertyMetadata(1),
        new ValidateValueCallback(ValidateRank));

    public Card()
    {
        InitializeComponent();
    }

    public string Suit
    {
        get { return (string)GetValue(SuitProperty); }
        set { SetValue(SuitProperty, value); }
    }

    public int Rank
    {
        get { return (int)GetValue(RankProperty); }
        set { SetValue(RankProperty, value); }
    }

    public static bool ValidateSuit(object suitValue)
    {
        string suitValueString = (string)suitValue;
        if (suitValueString != "Club" && suitValueString != "Diamond"
            && suitValueString != "Heart" && suitValueString != "Spade")
        {
            return false;
        }
        return true;
    }

    public static bool ValidateRank(object rankValue)
    {
        int rankValueInt = (int)rankValue;
        if (rankValueInt < 1 || rankValueInt > 12)
        {
            return false;
        }
```

```
        return true;
    }

    private void SetTextColor()
    {
        if (Suit == "Club" || Suit == "Spade")
        {
            RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
            SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
            RankLabelInverted.Foreground =
                new SolidColorBrush(Color.FromRgb(0, 0, 0));
        }
        else
        {
            RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
            SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
            RankLabelInverted.Foreground =
                new SolidColorBrush(Color.FromRgb(255, 0, 0));
        }
    }

    public static void OnSuitChanged(DependencyObject source,
        DependencyPropertyChangedEventArgs args)
    {
        ((Card)source).SetTextColor();
    }
}
```

代码段 Ch25Ex05\Card.xaml.cs

(4) 修改 MainWindow.xaml 中的代码，如下所示：

```
<Window x:Class="Ch25Ex05.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Card Dealer" Height="600" Width="800">

    <Grid Name="contentGrid" MouseLeftButtonDown="Grid_MouseLeftButtonDown"
      MouseLeftButtonUp="Grid_MouseLeftButtonUp" MouseMove="Grid_MouseMove">
      <Grid.Background>
        <LinearGradientBrush EndPoint="0.364,0.128" StartPoint="0.598,1.042">
          <GradientStop Color="#FF0D4F1A" Offset="0"/>
          <GradientStop Color="#FF448251" Offset="1"/>
        </LinearGradientBrush>
      </Grid.Background>
    </Grid>
</Window>
```

代码段 Ch25Ex05\MainWindow.xaml

(5) 修改 MainWindow.xaml.cs 中的代码，如下所示：

```
public partial class MainWindow : Window
{
    private Card currentCard;
    private Point offset;
```

```
private Random random = new Random();

public MainWindow()
{
    InitializeComponent();
}

private void Grid_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (e.Source is Card)
    {
        currentCard = (Card)e.Source;
        offset = Mouse.GetPosition(currentCard);
        contentGrid.Children.Remove(currentCard);
    }
    else
    {
        currentCard = new Card
        {
            Suit = Card.Suits[random.Next(0, 4)],
            Rank = random.Next(1, 13)
        };
        currentCard.HorizontalAlignment = HorizontalAlignment.Left;
        currentCard.VerticalAlignment = VerticalAlignment.Top;
        offset = new Point(50, 75);
    }
    contentGrid.Children.Add(currentCard);
    PositionCard();
}

private void Grid_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    currentCard = null;
}

private void Grid_MouseMove(object sender, MouseEventArgs e)
{
    if (currentCard != null)
    {
        PositionCard();
    }
}

private void PositionCard()
{
    Point mousePos = Mouse.GetPosition(this);
    currentCard.Margin =
        new Thickness(mousePos.X - offset.X, mousePos.Y - offset.Y, 0, 0);
}
}
```

代码段Ch25Ex05\MainWindow.xaml.cs

(6) 运行应用程序。单击窗口表面，会添加随机扑克牌，单击并拖动可以重新定位扑克牌。单

击已有的扑克牌时，它会跳到堆栈顺序的顶部。结果如图 25-18 所示。

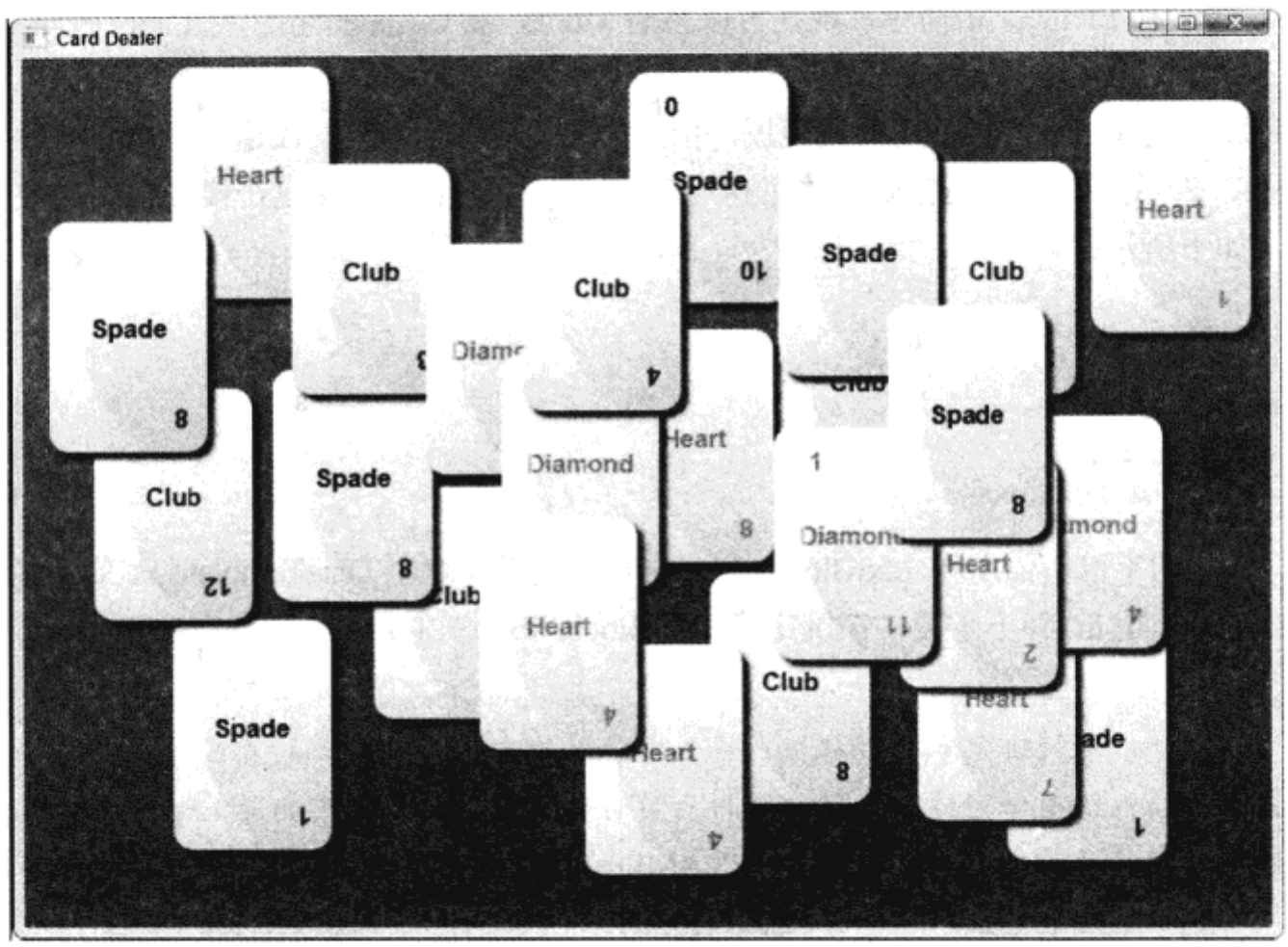


图　25-18

示例的说明

这个示例创建了一个用户控件，它带有两个依赖属性。该示例包含的客户代码使用这个控件。这个示例包含许多基础知识，首先要在代码中查看的是 Card 控件。

Card 控件包含的大多数代码都在本章前面使用过了。布局代码没有使用新特性，但结果比前面两个例子的不很美观按钮好多了。有一个全新的特性：这个控件使用了一小部分数据绑定功能。在使用数据绑定操作时，控件的属性绑定到数据源上，其中包含许多技术。WPF 很容易把属性绑定到所有的数据源上，如数据库中的数据、XML 数据和依赖属性值(本例就使用了依赖属性值)。

确切地讲，Card 中的代码为客户代码提供了两个依赖属性：Suit 和 Rank，并把这两个属性绑定到控件布局中的可视化元素上。结果是把 Suit 设置为 Club 时，单词 Club 就显示在扑克牌的中心。同样，Rank 的值显示在扑克牌的两个角上。

稍后讨论 Suit 和 Rank 的实现代码。现在只要了解这些属性分别是 string 和 int 值就足够了。可以给这两个属性使用枚举值，但这需要编写不少新代码，所以这个示例使用基本属性，以使代码尽可能简单。

要把一个值绑定到属性上，应使用绑定语法，即标记扩展。这个语法表示，把属性值指定为{Binding...}。以这种方式配置绑定有许多可能性。在本示例中，SuitLabel 标签的绑定配置为：

```
<Label x:Name="SuitLabel"
 Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
 ContentTemplate="{DynamicResource SuitTemplate}" HorizontalAlignment="Center"
 VerticalAlignment="Center"Margin="8,51,8,60"/>
```

这里为绑定指定了 3 个属性：Path(属性名)、ElementName(与属性关联的元素)和 Mode(执行绑定的方式)。Path 和 Element 很简单，现在可以忽略 Mode，要点是这个指定把 Label.Content 属性绑定到 Card.Suit 属性上。

在绑定属性值时，还必须使用数据模板，指定如何显示绑定内容。在这个示例中，数据模板是 SuitTemplate，表示为一个动态资源(但这里也可以使用静态资源绑定)。这个模板在用户控件资源部分中定义，如下所示：

```
<UserControl.Resources>
  <DataTemplate x:Key="SuitTemplate">
    <TextBlock Text="{Binding}"/>
  </DataTemplate>
</UserControl.Resources>
```

因此，Suit 的字符串值用作 TextBlock 控件的 Text 属性。这个 DataTemplate 定义也重用于两个 Rank 标签。Rank 是 int 并不重要，在绑定到 TextBlock.Text 属性上时，它会转换为 string。

显然，数据绑定和数据模板有许多内容，但本书没有足够的篇幅介绍其细节。本章的小结仅给出学习这个主题的资料。最后要注意，如果使用的是 Expression Blend，就可以高效地绑定数据，无需考虑 XAML 语法，因为 Expression Blend 会处理语法。

为了使这个数据绑定起作用，必须使用上一节介绍的技术定义两个依赖属性。它们在用户控件的代码隐藏内容中定义，如下所示(它们都是简单的.NET 属性包装器，所以不需要显示其代码)：

```
public static DependencyPropertySuitProperty =
    DependencyProperty.Register(
        "Suit",
        typeof(string),
        typeof(Card),
         new PropertyMetadata(
           "Club", new PropertyChangedCallback(OnSuitChanged)),
        new ValidateValueCallback(ValidateSuit));

public static DependencyProperty RankProperty =
    DependencyProperty.Register(
        "Rank",
        typeof(int),
        typeof(Card),
        new PropertyMetadata(1),
        new ValidateValueCallback(ValidateRank));
```

两个依赖属性都使用回调方法验证其值，Suit 属性还有一个在其值改变时执行的回调方法。验证回调方法的返回类型是 bool，它只有一个 object 类型的参数，它是客户代码试图设置的属性值。如果该值是 OK，就应返回 true，否则就返回 false。在示例代码中，只允许 Suit 属性使用以下 4 个字符串中的一个：

```
public static bool ValidateSuit(object suitValue)
{
    string suitValueString = (string)suitValue;
```

```
    if (suitValueString != "Club"&& suitValueString != "Diamond"
        && suitValueString != "Heart"&& suitValueString != "Spade")
    {
        return false;
    }
    return true;
}
```

这是相当麻烦的，显然这里使用枚举比较好，但没有这么做，原因如前所述。同样，Rank 属性也被限制为 1(ace)～12(king)之间：

```
public static bool ValidateRank(object rankValue)
{
    int rankValueInt = (int)rankValue;
    if (rankValueInt < 1 || rankValueInt > 12)
    {
        return false;
    }
    return true;
}
```

Suit 的值改变时，就调用 OnSuitChanged()回调方法。这个方法负责把文本颜色设置为红色(表示红心和方块)或黑色(表示黑桃和梅花)。为此，在方法调用的源上调用一个实用方法。这是必须的，因为回调方法实现为静态方法，但要把引发事件的用户控件实例传送为一个参数，才能与用户控件交互操作。所调用的方法是 SetTextColor()：

```
public static void OnSuitChanged(DependencyObject source,
    DependencyPropertyChangedEventArgs args)
{
    ((Card)source).SetTextColor();
}
```

SetTextColor()方法是私有方法，但显然仍可在 OnSuitChanged()中访问，因为它们都是同一个类的成员，分别是实例和静态方法。SetTextColor()仅根据 Suit 值把控件的各个标签的 Foreground 属性设置为黑色或红色的纯色笔刷：

```
private void SetTextColor()
{
    if (Suit == "Club" || Suit == "Spade")
    {
        RankLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
        SuitLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
        RankLabelInverted.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
    }
    else
    {
        RankLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(255, 0, 0));
        SuitLabel.Foreground =
```

```
            new SolidColorBrush(Color.FromRgb(255, 0, 0));
        RankLabelInverted.Foreground =
            new SolidColorBrush(Color.FromRgb(255, 0, 0));
    }
}
```

这就是 Card 控件中需要查看的代码。MainWindow.xaml 和 MainWindow.xaml.cs 中的客户代码相当简单，它使用基本的样式提供渐变的绿色背景，还提供了许多事件处理代码(使用路由事件和关联路由事件)进行用户交互操作。其中有几个难点——例如，页边距如何用于定位扑克牌，如何使用偏移值，把已有的扑克牌从单击的位置拖动开——阅读这些代码，就可以攻克这些难点。

25.5 小结

本章学习了用 WPF 编程所需的所有基础知识，还简要讨论了一些高级技术，了解了高级 WPF 编程提供的功能。WPF 主题的范围极广，不可能用一章的篇幅详细论述，如果读者对 WPF 感兴趣，可以参阅这个主题的其他资料，例如，如果希望更多地了解 Web 环境下的 XAML，开始时可以参考 *WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4* (Wrox, 2010)或者 *Silverlight 3 Programmer's Reference* (Wrox, 2009)。Silverlight 是一个尤其需要关注的领域，它的功能变得越来越好了。由于 Silverlight 是 Web 开发环境下的 WPF 的一个子集，所以注意 WFP 和 Silverlight 技巧很多都是通用的。

当然，也可以使用可用的工具(尤其是 Expression Blend)看看能得到什么效果。MSDN 文档也有一定的帮助，只是 WPF 是一种新技术，这方面的资料还比较欠缺，必须到其他地方查找相关信息。

有一些很好的网站提供了额外的信息。例如 WPF 的社区站点 http://windowsclient.net/，还可以访问 Scott Guthrie 的博客 http://weblogs.asp.net/scottgu。

本章介绍的内容如下：

- WPF 的含义，它对桌面和 Web 软件开发的潜在影响
- WPF 如何让使用 Expression Blend 的设计人员和使用 VS 或 VCE 的开发人员协同工作，完成项目
- XAML 的含义，它的基本语法和一些术语
- 使用 Application 对象
- WPF 中的控件的工作原理，包括依赖属性、关联属性、路由事件和关联事件的概念
- WPF 中的布局系统，使用各种布局容器定位控件
- 用样式和模板定制控件的外观和操作方式
- 使用触发器和动画改进用户体验
- 在内部和外部资源字典中定义资源，静态和动态访问资源
- 创建带依赖属性的用户控件

第 26 章将介绍.NET 3.0 最近引入的另一个技术：Windows Communication Foundation。

25.6　练习

(1) 有人说，可以给 WPF 桌面应用程序和 WPF 浏览器应用程序使用相同的 XAML 代码，这种说法正确吗？

(2) 为了允许子控件给父控件上定义的属性设置值，需要使用什么技术？应在 XAML 中使用什么语法？给出一个 XAML 示例，其中有两个子控件 Branch 为父控件 Tree 上定义的属性 LeafCount 设置不同的值。

(3) 下面哪个关于依赖属性的论述是正确的？

a. 必须通过关联的.NET 属性来访问依赖属性。

b. 依赖属性定义为公共静态成员。

c. 每个类定义只能有一个依赖属性。

d. 必须用命名约定[PropertyName]Property 来命名依赖属性。

e. 可以用回调方法验证赋给依赖属性的值。

(4) 可以使用哪个布局控件在单行或单列中显示控件？

(5) WPF 中的通道事件以特殊的方式命名，以便识别它们。这个命名约定是什么？

(6) 哪些属性类型可以连续改变？

(7) 何时使用动态资源引用，而不是静态资源引用？

附录 A 给出了练习答案。

25.7　本章要点

主　题	重 要 概 念
WPF 的含义	WPF 是 Microsoft 创建 Windows 和 Web 应用程序的最新方式，它使用标记和代码隐藏模型，能清晰地分开设计和功能，当设计人员和开发人员必须同时处理一个项目时，这种技术非常有效。WPF 中的标记称为 XAML。设计人员常使用 Expression Blend 处理 XAML
XAML	XAML 的语法允许在标记中表示对象。XAML 用 XML 编写，使用标记扩展来提供 WPF 功能
控件	WPF 使用控件建立用户界面，非常类似于 Windows 窗体。WPF 控件使用依赖属性提供与框架各个方面的集成，例如更改通知。依赖属性可以关联到未定义它们的对象上，并在需要时提供上下文信息。WPF 中的控件事件常常是路由事件，它们会沿着 WPF 应用程序的控件层次结构向上或向下路由
布局	WPF 提供了几个布局控件，它们可以包含其他控件。根据所需布局的类型，可以使用 Canvas、DockPanel、Grid、StackPanel 或 WrapPanel 布局控件
样式	可以使用 Style 对象给控件指定样式，Style 对象主要包含应用于控件属性的 Setter 对象。通过修改其模板，可以完全控制控件的外观和操作方式，还可以使用触发器响应用户的交互操作和数据变化
动画	在某个时间段内，属性可以在一个取值范围内连续变化。使用关键帧可以定义动画中的重要点，通过编程方式或触发器启动和停止动画
资源	可以为任意作用域定义资源(尤其是样式和模板)，例如在控件、窗口或应用程序的范围内定义。可以使用 StaticResource 或 DynamicResource 标记扩展访问资源，这取决于引用是否需要随时间变化

第26章

Windows Communication Foundation

本章内容：

- WCF 的含义
- WCF 概念
- WCF 编程

第 19 章学习了 Web 服务，以及如何使用它们在应用程序之间提供简单的通信。该章探讨了如何使用 HTTP GET 和 POST 技术与 Web 服务交换数据，以及如何使用 SOAP。自从 Web 服务可供.NET 开发人员使用以来，尽管 Web 服务很强大，但这种技术的扩展显然是有一定限制的。为此，微软公司发布了 Web Service Enhancements(WSE)插件来解决这个问题。WSE 允许 Web 服务开发人员通过消息的安全保护、路由技术和各种其他策略来提高 Web 服务，但仍有进一步提升的空间。

另一种.NET 技术——远程技术可以在一个进程中创建对象实例，在另一个进程中使用它们。远程技术甚至允许在一台计算机上使用位于另一台计算机上的对象。但这种技术仍有它自己的问题。远程技术是有限制的，刚入门的程序员要掌握它也不容易。

Windows Communication Foundation(WCF)是 Web 服务和远程技术的替代品，它从 Web 服务中提取了服务、独立于平台的 SOAP 消息传输等概念，把它们与远程技术中的宿主服务器应用程序和高级绑定功能结合在一起，所以可以将这种技术看作一个超集，包含了Web 服务和远程技术，但比 Web 服务强大，比远程技术更易于掌握。使用 WCF 可以从简单的应用程序转向使用面向服务的体系结构(SOA)的应用程序。SOA 表示可以分散处理，并在需要时连接跨本地网络和 Internet 的服务和数据，使用分布式处理。

本章将学习 WCF 的原理，以及如何在应用程序代码中创建和使用 WCF 服务。

不能在 VCE(Visual C# 2010 Express)中创建 WCF 服务，但可以在 VS 的完全版本中创建。还可以在 Visual Web Developer 2010 Express 中创建 IIS 承载的 WCF 服务，但本章使用 VS 来介绍所有的选项。

26.1 WCF 的含义

WCF 技术允许创建服务，可以跨进程、计算机和网络从其他应用程序访问这些服务。利用这些服务，可以共享跨多个应用程序的功能，提供数据源，或者抽象复杂的进程。

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

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

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

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

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

WCF 架构允许定制本节介绍的几乎所有方面。但这是一个高级主题，本章仅使用.NET 4 默认提供的技术。

了解了 WCF 服务的基础知识后，下面将详细介绍这些概念。

26.2 WCF 概念

本节描述 WCF 的如下方面：

- WCF 通信协议
- 地址、端点和绑定
- 合同

- 消息模式
- 行为
- 驻留

26.2.1　WCF 通信协议

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

- **HTTP**：它允许与任何地方(包括跨 Internet)的 WCF 服务通信。可以使用 HTTP 通信技术创建 WCF Web 服务。
- **TCP**：如果正确配置了防火墙，它允许与本地网络或跨 Internet 的 WCF 服务通信。TCP 比 HTTP 高效，功能也比较多，但配置起来更加复杂。
- **命名管道**：它允许与 WCF 服务通信，该 WCF 服务与调用代码位于同一台计算机的不同进程上。
- **MSMQ**：这是一种排队技术，允许应用程序发送的消息通过队列路由到目的地。MSMQ 是一种可靠的消息传输技术，可以确保发送给队列的消息一定达到该队列。MSMQ 还是一种异步技术，所以只有排在前面的消息都处理完了，服务仍有效时，才能处理当前的消息。

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

图 26-1 列出了这些传输协议如何把应用程序与不同位置上的 WCF 服务连接起来。本章将介绍所有这些协议，但 MSMQ 除外(这个主题需要较深入的讨论)。

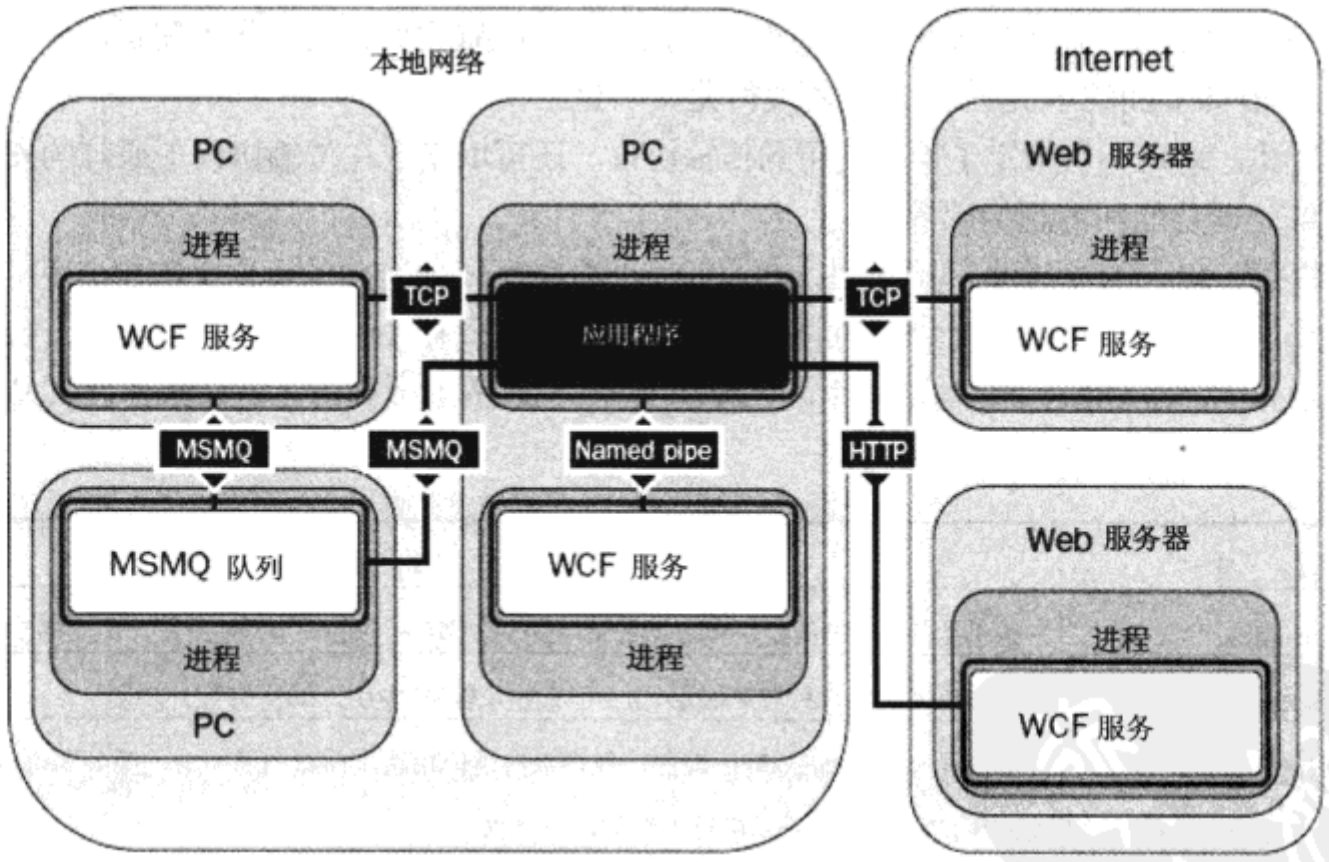


图　26-1

为了连接 WCF 服务，必须知道它在什么地方。这表示必须知道端点的地址。

26.2.2　地址、端点和绑定

用于服务的地址类型取决于所使用的协议。本章前面介绍的 3 个协议(不包括 MSMQ)都需要格

式化的服务地址：

- HTTP：HTTP 协议的地址是 URL，其格式很常见：http://<server>:<port>/<service>。对于 SSL 连接，也可以使用 https://<server>:<port>/<service>。如果在 IIS 中承载服务，<service>就是扩展名为.svc 的文件(.svc 文件类似于 Web 服务中使用的.asmx 文件)。IIS 地址可能包含比这个示例更多的子目录，即.svc 文件之前有更多使用/字符分隔的部分。
- TCP：TCP 的地址采用 net.tcp:// <server>:<port>/<service>形式。
- **命名管道：命名**管道连接的地址与上述类似，但没有端口号。其形式是 net.pipe:// <server>/<service>。

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

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

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

接着就可以给操作使用下面的地址：

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

在.NET 4 中，可以给操作使用默认端点，而无需明确配置它们。这简化了配置，如果需要使用标准端点地址(如上面的示例所示)，这表现得尤其明显。

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

绑定提供了极大的灵活性，所以.NET Framework 提供了一些可用的预定义绑定。还可以把这些绑定用作起点，修改它们，得到需要的绑定类型。预定义绑定有一些必须遵循的原则。每种绑定类型都用 System.ServiceModel 名称空间中的一个类表示。表 26-1 中列出了这些绑定及其基本信息。

表 26-1

绑 定	说 明
BasicHttpBinding	最简单的 HTTP 绑定，Web 服务使用的默认绑定，它的安全功能有限，不支持事务处理
WSHttpBinding	HTTP 绑定的一种较高级形式，可以使用 WSE 中引入的所有额外功能
WSDualHttpBinding	扩展了 WSHttpBinding 功能，包含双向通信功能。在双向通信中，服务器可以启动与客户机的通信，还可以进行一般的消息交换
WSFederationHttpBinding	扩展了 WSHttpBinding 功能，包含联合功能。联合功能允许第三方实现单一登录(single sign-on)和其他专用安全措施。这是一个高级主题，本章不予讨论
NetTcpBinding	用于 TCP 通信，允许配置安全性、事务处理等
NetNamedPipeBinding	用于指定管道的通信，允许配置安全性、事务处理等
NetPeerTcpBinding	允许与多个客户机进行广播通信，是一个高级类，本章不予讨论

(续表)

绑　定	说　明
NetMsmqBinding 和 MsmqIntegrationBinding	这些绑定用于 MSMQ，本章不予讨论
NetPeerTcpBinding	用于对等绑定，本章不予讨论
WebHttpBinding	用于使用 HTTP 请求(而不是 SOAP 消息)的 Web 服务
NetTcpContextBinding	类似于 NetTcpBinding，但允许通过 SOAP 标题交换环境信息
BasicHttpContextBinding 和 WSHttpContextBinding	类似于 BasicHttpBinding 和 WSHttpBinding，但允许分别通过 HTTP Cookie 或 SOAP 标题交换环境信息

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

在.NET 4 中，端点的默认绑定因所用协议而异。这些默认绑定如表 26-2 所示。

表 26-2

协　议	默 认 绑 定
HTTP	BasicHttpBinding
TCP	NetTcpBinding
命名管道	NetNamedPipeBinding
MSMQ	NetMsmqBinding

26.2.3　合同

合同定义了 WCF 服务的用法。可以定义如下几种合同：

- **服务合同**：包含服务的一般信息和服务提供的操作的一般信息。例如，该合同可以包含服务使用的名称空间。在为 SOAP 消息定义模式时，服务使用唯一的名称空间，以免与其他服务冲突。
- **操作合同**：定义操作的用法，这包括操作方法的参数和返回类型，以及其他信息，例如，方法是否返回响应消息。
- **消息合同**：允许定制 SOAP 消息内部的信息格式化方式。例如，数据应包含在 SOAP 标题中还是 SOAP 消息体中。在创建必须与以前的系统集成的 WCF 服务时，就可以使用消息合同。
- **错误合同**：定义操作可能返回的错误。使用.NET 客户程序时，错误会导致可以捕获的异常，并以通常方式处理。
- **数据合同**：如果使用复杂类型，如用户定义的结构和对象(作为操作的参数或返回类型)，就必须为这些类型定义数据合同。数据合同根据通过属性显示的数据来定义类型。

一般使用特性把合同添加到服务类和方法中，如本章后面所述。

26.2.4　消息模式

上一节提到，操作合同可以定义操作是否返回一个值，WSDualHttpBinding 允许进行双向通信。

这些都是消息模式，消息模式有 3 种类型：

- **请求/响应消息传输**：交换消息的“一般”方式，每个发送给服务的消息都会从客户机中得到一个返回的响应。这并不意味着客户机等待响应，因为可以用一般方式异步调用操作。
- **单向消息传输**：消息从客户机传输给 WCF 操作，但不发送响应。不需要响应时，就可以使用这种消息模式。例如，创建一个 WCF 操作，它会重启 WCF 宿主服务器，此时不需要等待响应。
- **双向消息传输**：一种较高级的模式，客户机可以用作服务器，服务器也可以用作客户机。启动后，双向消息传输允许客户机和服务器彼此发送消息，这些消息可能有响应，也可能没有。这类似于创建一个对象，并订阅该对象公开的事件。

本章的后面将使用这些消息模式。

26.2.5 行为

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

在.NET 4 中，可以在不同的级别上指定默认行为，而无需给每个服务和操作指定每个行为的各个方面。还可以在需要时提供默认设置，重写设置，减少所需的配置量。

本章仅介绍 WCF 服务的基本知识，讨论行为的最基本功能。

26.2.6 驻留

本章的引言曾提到，WCF 服务可以存储在几个不同进程中，包括：

- **Web 服务器**：驻留在 IIS 的 WCF 服务是 WCF 提供的最接近 Web 服务的服务。还可以使用 WCF 服务中的高级功能和安全特性，这些功能和特性很难在 Web 服务中实现。也可以集成 IIS 特性，如 IIS 安全特性。
- **可执行文件**：可以把 WCF 服务驻留在.NET 中创建的任意应用程序类型中，如控制台应用程序、Windows 窗体应用程序和 WPF 应用程序。
- **Windows 服务**：可以把 WCF 服务驻留在 Windows 服务中，这表示可以使用 Windows 服务提供的有用特性，包括自动启动和错误恢复。
- **Windows Activation Service(WAS)**：专门用于驻留 WCF 服务，基本上是 IIS 的一个简化版本，可以在任何没有 IIS 的地方使用。

上述列表中的两个选项 IIS 和 WAS 为 WCF 服务提供了有用的特性，例如激活、处理循环和对象池。如果使用另外两个驻留选项，WCF 服务就是自驻留的。这未必是件坏事，因为我们可能不需要驻留环境提供的其他功能。但是自驻留的服务需要编写更多的代码。

26.3 WCF 编程

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

- 定义 WCF 服务合同
- 自驻留的 WCF 服务

试一试：一个简单的 WCF 服务和客户程序

(1) 在 C：BegVCSharp\Chapter26 目录中创建一个新的 WCF 服务应用程序项目 Ch26Ex01。

(2) 在解决方案中添加一个控制台应用程序 Ch26Ex01Client。

(3) 在 Build 菜单上单击 Build Solution 选项。

(4) 在 Solution Explorer 中右击 Ch26Ex01Client 项目，选择 Add Service Reference 选项。

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

(6) 启动开发 Web 服务器，加载 WCF 服务的信息后，展开该引用，查看其细节，如图 26-2 所示(读者的端口号可能与本图中的不同)。

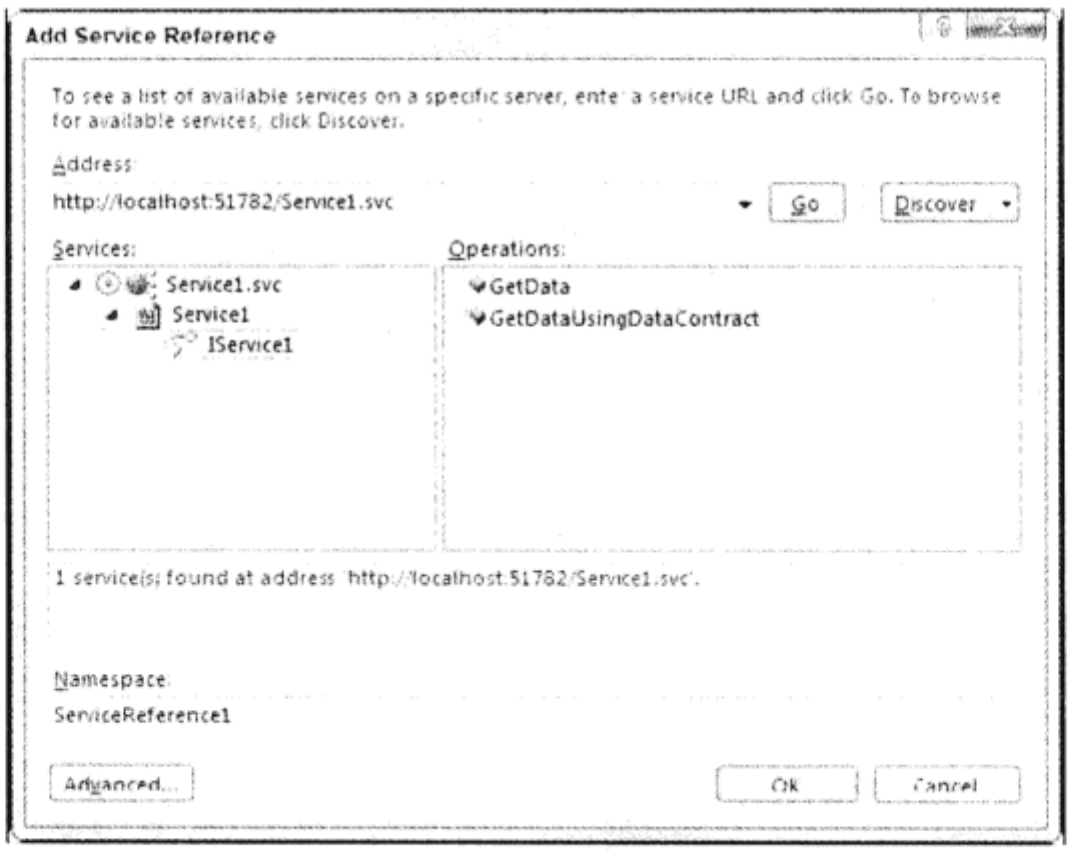


图　26-2

(7) 单击 OK 按钮，添加服务引用。

(8) 在 Ch26Ex01Client 应用程序中修改 Program.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex01Client.ServiceReference1;

namespace Ch26Ex01Client
{
   class Program
   {
      static void Main(string[] args)
      {
         string numericInput = null;
         int intParam;
         do
```

```
            {
                Console.WriteLine(
                  "Enter an integer and press enter to call the WCF service.");
                numericInput = Console.ReadLine();
            }
            while (!int.TryParse(numericInput, out intParam));
            Service1Client client = new Service1Client();
            Console.WriteLine(client.GetData(intParam));
            Console.WriteLine("Press an key to exit.");
            Console.ReadKey();
        }
    }
}
```

代码段 Ch26Ex01Client\Program.cs

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

(10) 运行应用程序。出现提示后，单击 OK 按钮启用 Web.config 中的调试功能。在控制台应用程序窗口中输入一个数字，按下回车键。结果如图 26-3 所示。

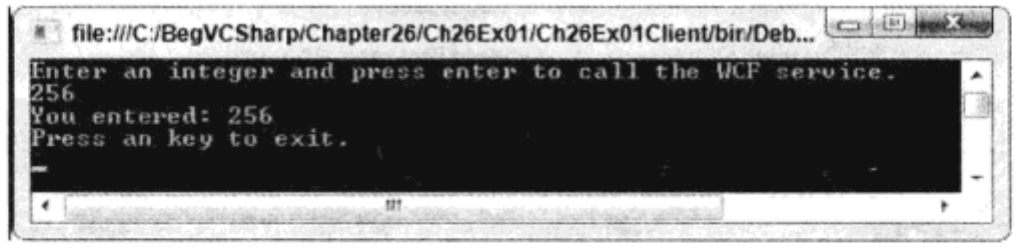


图 26-3

(11) 退出应用程序，在 Solution Explorer 中右击 Ch26Ex01 项目中的 Service1.svc 文件，单击 View in Browser。

(12) 查看窗口中的信息，如图 26-4 所示。

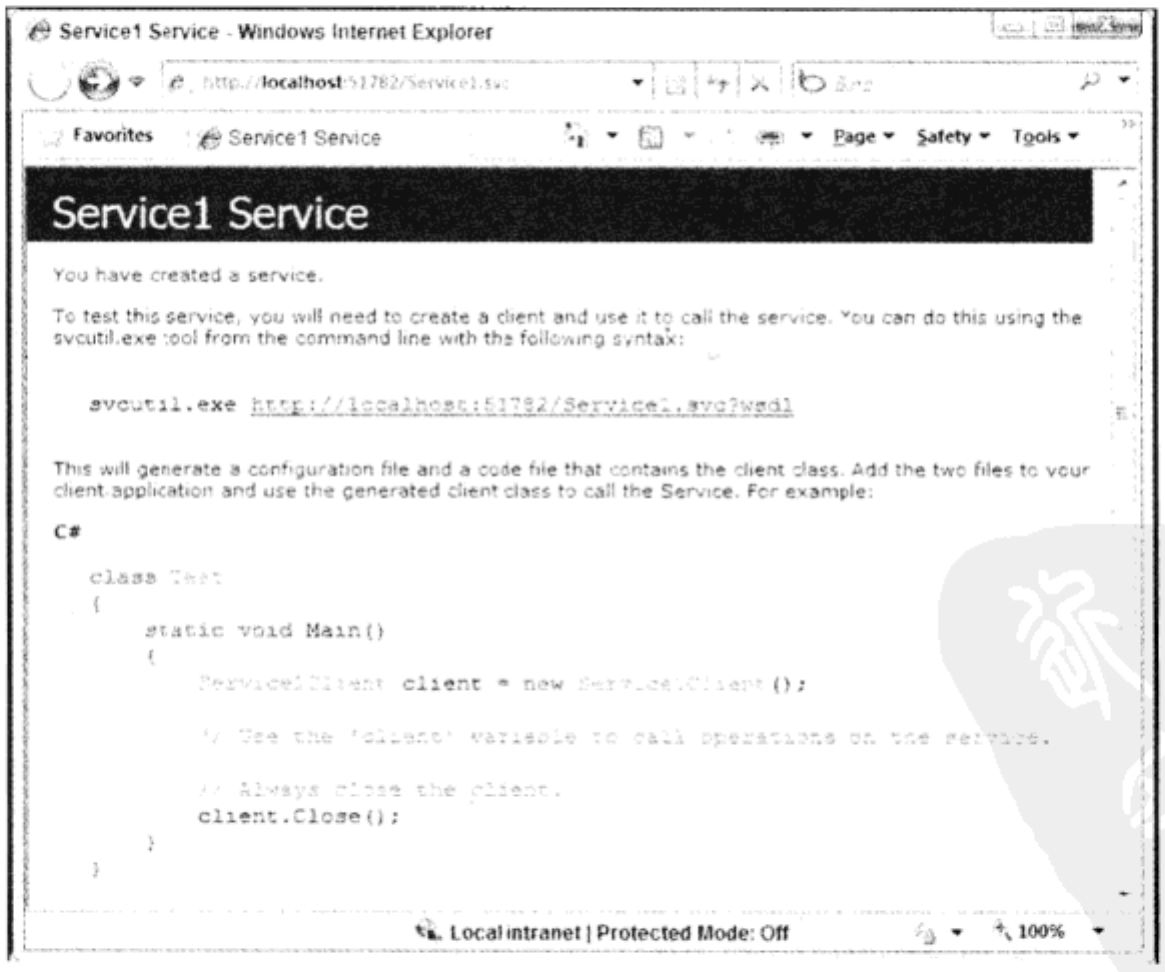


图 26-4

(13) 单击 Web 页面顶部的链接，查看服务的 WSDL。现在还不需要了解 WSDL 文件中的所有内容的含义。

示例的说明

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

首先看看服务器项目 Ch26Ex01，它包含：

- Service1.svc 文件，它定义了服务的宿主。
- 类定义 CompositeType，它定义了服务使用的数据合同(位于 IService1.cs 代码文件中)。
- 接口定义 IService1，它定义了服务合同和两个操作合同。
- 类定义 Service1，它实现 IService1 接口，定义了服务的功能(位于 Service1.svc.cs 代码文件)。
- 配置段<system.serviceModel>(在 Web.config 中)，它配置了服务。

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

可从 wrox.com 下载源代码

```
< %@ ServiceHost Language="C#" Debug="true" Service="Ch26Ex01.Service1"
  CodeBehind="Service1.svc.cs"%>
```

代码段 Ch26Ex01\Service1.svc

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

显然，没有驻留在 Web 服务器上的 WCF 服务不需要这个文件。本章后面将学习自驻留的 WCF 服务。

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

可从 wrox.com 下载源代码

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

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

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

```
    }
}
```

代码段 Ch26Ex01\IService1.cs

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

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

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

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

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

可从 wrox.com 下载源代码

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

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

代码段 Ch26Ex01\Service1.svc.cs

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

把服务的实现代码(类)和服务合同(接口)分开的效果极佳。客户程序不需要了解类的任何信息，类包含的功能可能远远超过了服务实现的功能。一个类甚至可以实现多个服务合同。

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

WCF 配置代码包含在 Web.config 或 app.config 文件的配置段<system.serviceModel>中。这个示例没有进行很多服务配置，因为使用了默认值。在 Web.config 文件中，配置段包含一个子段，它为服务行为<behaviors>重写了默认值。Web.config 中<system.serviceModel>配置段的代码如下(为了简洁起见，删除了注释)：

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

代码段 Ch26Ex01\Web.config

这个配置段可以在<behavior>子段中定义一个或多个行为，这些行为可以在多个其他元素上重用。可以给<behavior>段指定一个名称，以便进行重用(这样就可以在其他地方引用它)，也可以不指定名称来使用(如本例所示)，以指定重写默认的行为设置。

如果使用了非默认的配置，在<system.serviceModel>中就会包含一个<services>段，其中包含一个或多个<service>子段，<service>段又可以包含<endpoint>子段，每个<endpoint>子段都定义了服务的一个端点。实际上，所定义的端点是服务的基端点。可以从中推断出操作的端点。

在 Web.config 中，重写的一个默认行为如下：

```
<serviceDebug includeExceptionDetailInFaults="false"/>
```

这个设置可以是 true，在传输给客户程序的任意错误中提供异常详情，通常只允许在开发过程中传输这些异常信息。

在 Web.config 中，另一个默认的重写行为与元数据相关。元数据允许客户程序获得 WCF 服务的描述。默认配置给服务定义了两个默认端点，一个端点由客户程序用于访问服务，另一个端点用于获得服务的元数据。在 Web.config 文件中，可以禁用这个功能，如下所示：

```
<serviceMetadata httpGetEnabled="false"/>
```

另外，还可以完全删除这行配置代码，因为默认行为不允许交换元数据。

如果在本例中尝试禁用这个功能，并不能阻止客户程序访问服务，因为客户程序已经在添加服务引用时获得了需要的元数据。但禁用元数据会禁止其他客户程序使用 Add Service Reference 工具访问这个服务。一般情况下，产品环境中的 Web 服务不需要提供元数据，所以应在开发阶段完成后禁用这个功能。

如果没有元数据，访问 Web 服务的另一种常见方式是在一个独立的程序集中为 WCF 服务定义合同，由宿主项目和客户项目引用。接着客户程序就可以直接使用这些合同生成一个代理，而不是通过元数据来访问服务。

前面介绍了 WCF 服务的代码，现在看看客户程序，尤其是使用 Add Service Reference 工具做了什么。注意在 Solution Explorer 中，客户程序包含一个文件夹 Service References，如果展开该文件夹，会看到一项 ServiceReference1，它是添加引用时选择使用的名称。

Add Service Reference 工具创建了访问服务所需的所有类。这包括服务的代理类，服务的代理类包含服务的所有操作方法(Service1Client)，以及从数据合同中生成的客户端类(CompositeType)。

也可以浏览 Add Service Reference 工具生成的代码(显示项目中的所有文件，包括隐藏的文件)，但目前最好不要浏览代码，因为有许多容易引起混淆的代码。

该工具还为项目添加了一个配置文件 app.config，这个配置定义了两个内容：

- 服务端点的绑定信息
- 端点的地址和合同

从服务描述中提取绑定信息，在客户程序中，每个可配置的选项都复制到配置文件中：

可从 wrox.com 下载源代码

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IService1"
          closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard"
          maxBufferSize="65536" maxBufferPoolSize="524288"
          maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8"
          transferMode="Buffered" useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None"
              proxyCredentialType="None" realm="" />
            <message clientCredentialType="UserName"
              algorithmSuite="Default" />
```

```
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
```

代码段 Ch26Ex01Client\app.config

这个绑定、服务的基地址(这是 Web 服务器存储的服务的.svc 文件地址)和合同的客户端版本 IService1 在端点配置中使用：

```
  <client>
    <endpoint address="http://localhost:51782/Service1.svc"
      binding="basicHttpBinding"
      bindingConfiguration="BasicHttpBinding_IService1"
      contract="ServiceReference1.IService1"
      name="BasicHttpBinding_IService1" />
    </client>
  </system.serviceModel>
</configuration>
```

Add Service Reference 工具是非常全面的。实际上，大多数信息都不是必要的，可以用下面的代码替代这个配置文件：

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://localhost:51782/Service1.svc"
        binding="basicHttpBinding"
        contract="ServiceReference1.IService1"
        name="BasicHttpBinding_IService1" />
    </client>
  </system.serviceModel>
</configuration>
```

这段代码删除了整个<bindings>段和<endpoint>元素的 bindingConfiguration 属性，这表示客户程序将使用默认的绑定配置。

但是为了学习 WCF 服务，掌握工具的全面性是非常重要的。它会显示包含在 BasicHttpBinding 默认绑定中的所有设置。本章不深入探讨 WCF 服务配置，但介绍了其中的一些配置，如超时设置，这些配置的命名很浅显，很容易理解。

这个示例介绍了许多基础知识，下面总结一下前面的内容：

- WCF 定义
  - 服务由服务合同接口定义，其中包括操作合同成员
  - 服务在实现了服务合同接口的类中实现
  - 数据合同只是使用数据合同特性的类型定义
- WCF 服务配置
  - 可以使用配置文件(Web.config 或 app.config)来配置 WCF 服务

- WCF Web 服务器驻留：
  - Web 服务器驻留把.svc 文件用作服务基地址
- WCF 客户机配置：
  - 可以使用配置文件(web.config 或 app.config)来配置 WCF 服务客户机

下一节详细介绍合同。

26.3.1 WCF 测试客户程序

上面的示例创建了服务和客户程序，说明了 WCF 基本体系结构的工作原理，以及 WCF 服务的配置方式。但实际要使用的客户应用程序会比较复杂，也难以正确地测试服务。

为了便于开发 WCF 服务，VS 提供了一个测试工具，可用于确保 WCF 操作正常工作。这个工具会自动配置为处理 WCF 服务项目，所以如果运行项目，该工具就会显示出来。只需确保要测试的服务(即.svc 文件)设置为 WCF 服务项目的启动页面即可。另外，也可以把测试客户程序运行为独立的应用程序运行。在 64 位操作系统上，测试客户程序位于 C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\WcfTestClient.exe。

如果使用 32 位操作系统，该路径是相同的，只是根文件夹是 Program Files。

可以使用该工具调用服务操作，还可以用其他方式检查服务。如下面的示例所示。

试一试：使用 WCF 测试客户程序

(1) 打开上一个示例中的 WCF Service Application 项目 Ch26Ex01。

(2) 在 Solution Explorer 中右击 Service1.svc，选择 Set As Start Page。

(3) 在 Solution Explorer 中右击 Ch26Ex01 项目，选择 Set As StartUp Project。

(4) 在 Web.config 中，确保启用元数据。

可从 wrox.com 下载源代码

```
<serviceMetadata httpGetEnabled="true"/>
```

代码段 Ch26Ex01\Web.config

(5) 运行应用程序。WCF 测试客户程序就会显示出来，如图 26-5 所示(需要一定的时间添加服务)。

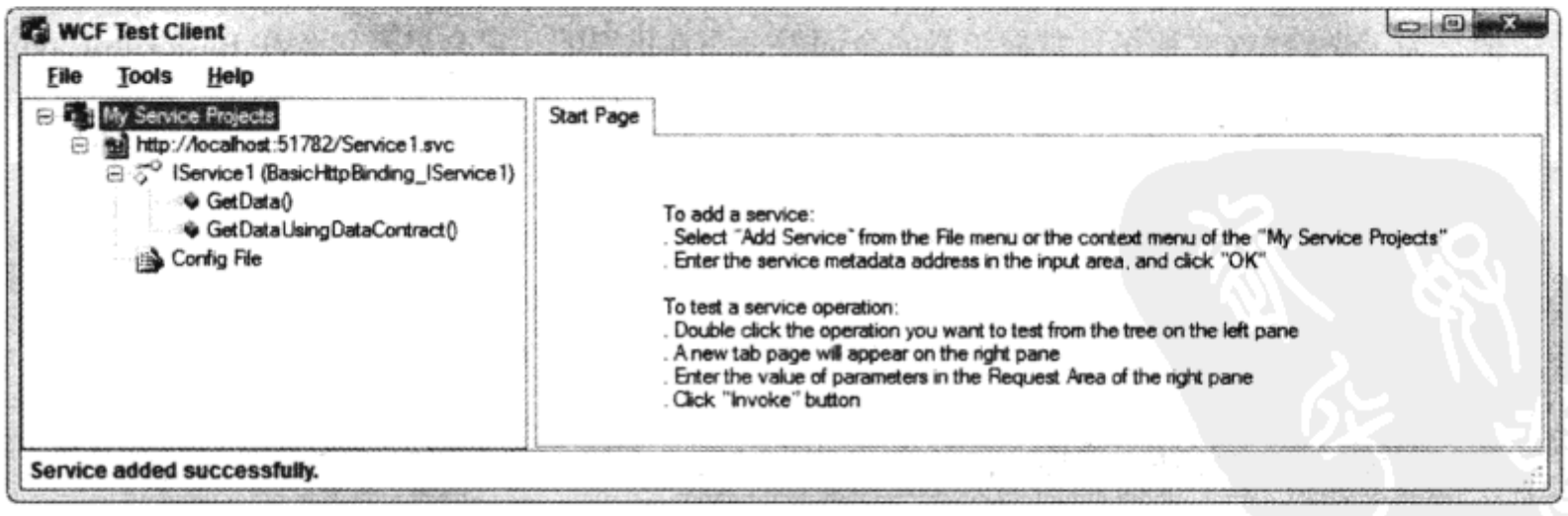


图 26-5

(6) 在测试客户程序的左面板上，双击 Config File。用于访问服务的配置文件就会显示在右面板上，如图 26-6 所示。

Client.dll.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IService1" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
          maxBufferSize="65536" maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
          useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
              realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:51782/Service1.svc" binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IService1" contract="IService1"
        name="BasicHttpBinding_IService1" />
    </client>
  </system.serviceModel>
</configuration>
```

图　26-6

(7) 在左面板上，双击 GetDataUsingDataContract()操作。

(8) 在右面板上，把 BoolValue 的值改为 True，StringValue 改为 Test String，再单击 Invoke。

(9) 如果显示了安全提示对话框，单击 OK 确认要把信息发送给服务。

(10) 显示操作的结果，如图 26-7 所示。

GetDataUsingDataContract

Request

Name	Value	Type
◢ composite	Ch26Ex01.CompositeType	Ch26Ex01.CompositeType
BoolValue	True	System.Boolean
StringValue	Test String	System.String

☐ Start a new proxy　　Invoke

Response

Name	Value	Type
◢ (return)		Ch26Ex01.CompositeType
BoolValue	True	System.Boolean
StringValue	"Test StringSuffix"	System.String

Formatted　XML

图　26-7

(11) 单击 XML 标签页，查看请求和响应的 XML，如图 26-8 所示。

GetDataUsingDataContract

Request

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <Action s:mustUnderstand="1" xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">http://tempuri.org/IService1/GetDataUsingDataContract</Action>
  </s:Header>
  <s:Body>
    <GetDataUsingDataContract xmlns="http://tempuri.org/">
      <composite xmlns:d4p1="http://schemas.datacontract.org/2004/07/Ch26Ex01" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <d4p1:BoolValue>true</d4p1:BoolValue>
        <d4p1:StringValue>Test String</d4p1:StringValue>
      </composite>
    </GetDataUsingDataContract>
  </s:Body>
</s:Envelope>
```

Response

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <GetDataUsingDataContractResponse xmlns="http://tempuri.org/">
      <GetDataUsingDataContractResult xmlns:a="http://schemas.datacontract.org/2004/07/Ch26Ex01" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:BoolValue>true</a:BoolValue>
        <a:StringValue>Test StringSuffix</a:StringValue>
      </GetDataUsingDataContractResult>
    </GetDataUsingDataContractResponse>
  </s:Body>
</s:Envelope>
```

Formatted XML

图 26-8

示例的说明

这个示例使用 WCF 测试客户程序在上一个示例创建的服务上检查和调用操作。首先注意，服务的加载有一点儿延迟，这是因为测试客户程序必须检查服务，以确定其功能。这个检查过程使用与 Add Service Reference 工具相同的元数据，所以必须确保元数据是可用的(在上一个示例中可能禁用了元数据)。检查完后，在工具的左面板上就会显示服务及其操作。

接着，查看用于访问服务的配置。与上一个示例中的客户应用程序一样，这些配置也是从服务的元数据中自动生成的，且包含在与服务相同的代码中。如有必要，可以通过该工具编辑这个配置文件，方法是右击 Config File 项，单击 Edit with SvcConfigeditor。

接着调用了一个操作。测试客户程序允许输入要使用的参数，并调用方法，然后显示结果，所有这些都不需要编写任何客户代码。我们还查看了为获得结果而发送和接收的 XML，这些信息的技术性很强，但在调试比较复杂的服务时，这些信息是绝对必需的。

26.3.2 定义 WCF 服务合同

从前面的示例可以了解到，通过 WCF 体系结构，可以结合使用类、接口和特性来方便地为 WCF 服务定义合同。本节将深入介绍这种技术。

1. 数据合同

要给服务定义数据合同，需要把 DataContractAttribute 特性应用于类定义。这个特性在名称空间 System.Runtime.Serialization 中。可以使用表 26-3 中所示的属性配置它。

表 26-3

属 性	说 明
Name	用不同于类定义的名称来命名数据合同。这个名称在 SOAP 消息和服务元数据定义的客户端数据对象上使用
Namespace	指定数据合同在 SOAP 消息中使用的名称空间

当需要与已有的 SOAP 消息格式交互操作时(类似于其他合同的指定属性)，需要使用这两个属性，否则就不需要它们。

数据合同中的每个类成员都必须使用 DataMemberAttribute 特性，它在名称空间 System.Runtime.Serialization 中。这个特性具有表 26-4 中所示的属性。

表 26-4

属 性	说 明
Name	指定序列化时数据成员的名称(默认为成员名称)
IsRequired	指定成员是否必须显示在 SOAP 消息中
Order	int 值，指定序列化或反序列化成员的顺序，如果一个成员必须在另一个成员之前出现，这个顺序就是必须的。先处理 Order 较低的成员
EmitDefaultValue	把它设置为 false，如果成员的值是默认值，就禁止该成员包含在 SOAP 消息中

2. 服务合同

把 System.ServiceModel.ServiceContractAttribute 特性应用于接口定义，就定义了服务合同。表 26-5 中所示的属性可用于定制服务合同。

表 26-5

属 性	说 明
Name	按照 WSDL 中<portType>元素中的定义，指定服务合同的名称
Namespace	定义 WSDL 中<portType>元素使用的服务合同的名称空间
ConfigurationName	在配置文件中使用的服务合同名称
HasProtectionLevel	指定服务使用的消息是否有明确定义的保护级别。保护级别允许签名消息，或者签名和加密消息
ProtectionLevel	保护级别，用于保护消息
SessionMode	确定是否为消息启用会话。如果使用会话，就可以确保关联上发送给服务的不同端点的消息，即它们使用同一个服务实例，因此可以共享状态
CallbackContract	对于双向消息传输，客户机提供了合同和服务。这是因为，如前所述，双向通信中的客户机也用作服务器。这个属性允许指定客户机使用的合同

3. 操作合同

在定义服务合同的接口中，应用 System.ServiceModel.OperationContractAttribute 特性，就可以把成员定义为操作。这个特性具有表 26-6 中所示的属性。

表 26-6

属 性	说 明
Name	指定服务操作的名称。默认为成员名称
IsOneWay	指定操作是否返回一个响应。如果把它设置为 true，则客户机不等待操作完成，就会继续执行
AsyncPattern	设置为 true，操作就会实现为两个方法：Begin<methodName>和 End<methodName>，这两个方法可用于异步调用操作
HasProtectionLevel	参见表 26-5
ProtectionLevel	参见表 26-5
IsInitiating	如果使用会话，这个属性就确定调用这个操作是否可以启动新会话
IsTerminating	如果使用会话，这个属性就确定调用这个操作是否会中断当前会话
Action	如果使用寻址功能(WCF 服务的一个高级功能)，操作就有一个关联的动作名称，通过这个属性可以指定该名称
ReplyAction	同上，但为操作的响应指定动作名称

4. 消息合同

前面的示例中没有使用消息合同规范。如果使用消息合同，就应定义一个表示消息的类，再给类应用 MessageContractAttribute 特性。接着给这个类的成员应用 MessageBodyMemberAttribute、MessageHeaderAttribute 或 MessageHeaderArrayAttribute 特性。所有这些特性都在 System.ServiceModel 名称空间中。如果要高度控制 WCF 服务使用的 SOAP 消息，就不要使用消息合同，所以这里不详细讨论它。

5. 误合同

如果客户应用程序可以使用特定的异常类型，如定制异常，就可以给可能生成该异常的操作应用 System.ServiceModel.FaultContractAttribute 特性。在使用 WCF 时一般不希望这么做。

试一试：WCF 合同

(1) 创建一个新 WCF 服务应用程序项目 Ch26Ex02，将其保存在 C: \BegVCSharp \Chapter26 目录中。

(2) 给解决方案添加一个类库项目 Ch26Ex02Contracts，删除 Class1.cs 文件。

(3) 在 Ch26Ex02Contracts 项目中添加对 System.Runtime.Serialization 和 System.Service Model.dll 程序集的引用。

(4) 在 Ch26Ex02Contracts 项目中添加 Person 类，修改 Person.cs 中的代码，如下所示：

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

namespace Ch26Ex02Contracts
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public int Mark { get; set; }
    }
}
```

代码段 Ch26Ex02Contracts\Person.cs

(5) 在 Ch26Ex02Contracts 项目中添加 IAwardService 类，修改 IAwardService.cs 中的代码，如下所示：

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

namespace Ch26Ex02Contracts
{
    [ServiceContract(SessionMode=SessionMode.Required)]
    public interface IAwardService
    {
        [OperationContract(IsOneWay=true,IsInitiating=true)]
        void SetPassMark(int passMark);

        [OperationContract]
        Person[] GetAwardedPeople(Person[] peopleToTest);
    }
}
```

代码段 Ch26Ex02Contracts\IAwardService.cs

(6) 对于 Ch26Ex02 项目，添加对 Ch26Ex02Contracts 项目的引用。

(7) 删除 Ch26Ex02 项目中的 IService1.cs 和 Service1.svc。

(8) 在 Ch26Ex02 中添加一个新的 WCF 服务 AwardService。

(9) 删除 Ch26Ex02 项目中的 IAwardService.cs 文件。

(10) 修改 AwardService.svc.cs 文件中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using Ch26Ex02Contracts;

namespace Ch26Ex02
{
   public class AwardService : IAwardService
   {
      private int passMark;

      public void SetPassMark(int passMark)
      {
         this.passMark = passMark;
      }

      public Person[] GetAwardedPeople(Person[] peopleToTest)
      {
         List < Person > result = new List < Person > ();
         foreach (Person person in peopleToTest)
         {
            if (person.Mark > passMark)
            {
               result.Add(person);
            }
         }
         return result.ToArray();
      }
   }
}
```

代码段 Ch26Ex02\AwardService.svc.cs

(11) 修改 Web.config 中的服务配置段，如下所示：

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="http" binding="wsHttpBinding"/>
  </protocolMapping>
  ...
</system.serviceModel>
```

(12) 打开 Ch26Ex02 的项目属性。在 Web 部分，选择 Specific port，输入端口号 51425，如图 26-9 所示。

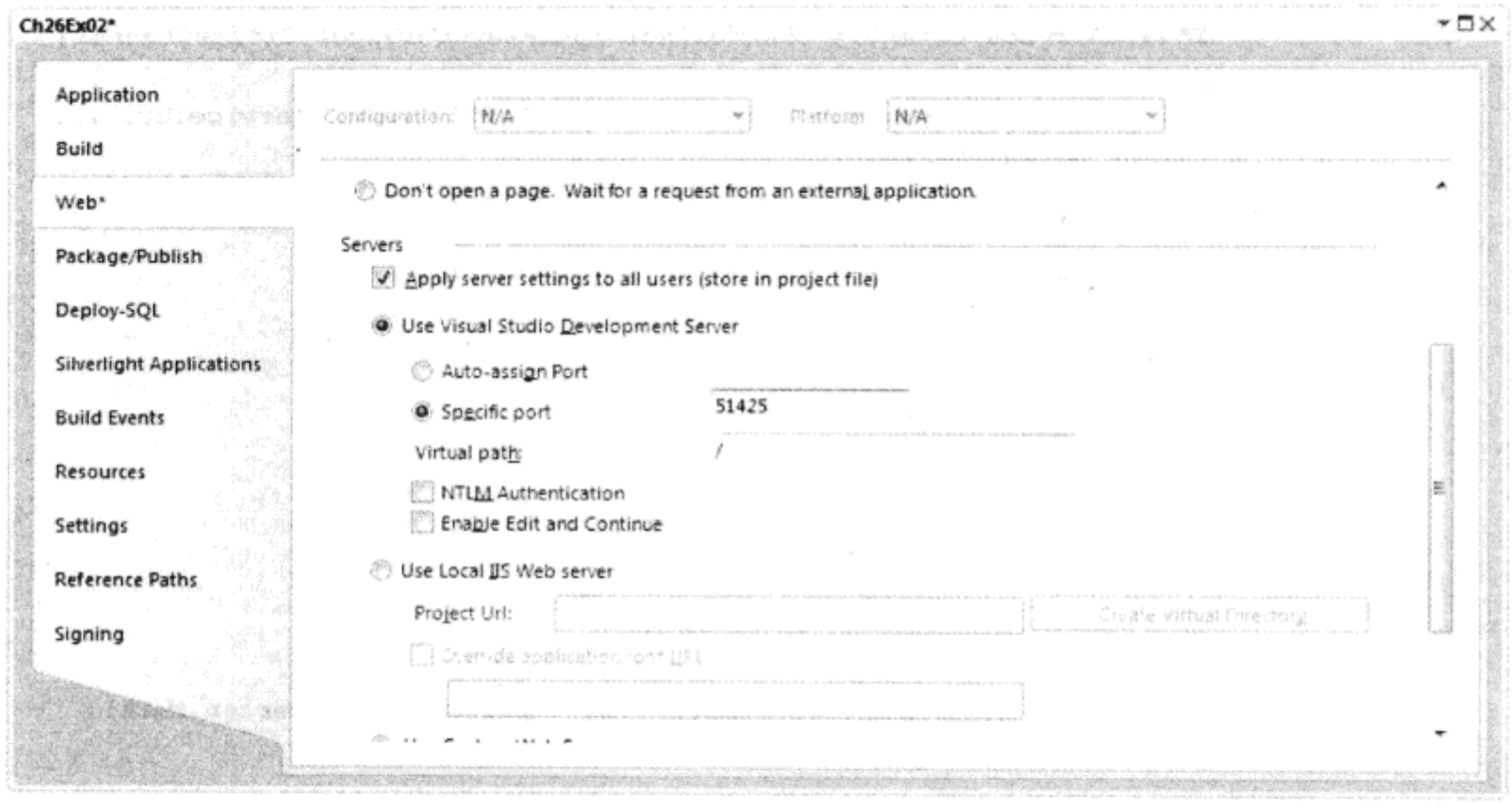


图　26-9

(13) 在解决方案中添加一个新的控制台项目 Ch26Ex02Client，把它设置为启动项目。

(14) 在 Ch26Ex02Client 项目中添加对 System.ServiceModel.dll 程序集和 Ch26Ex02Contracts 项目的引用。

(15) 在 Ch26Ex02Client 项目中修改 Program.cs 中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using Ch26Ex02Contracts;

namespace Ch26E02Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Person[] people = new Person[]
            {
                new Person { Mark = 46, Name="Jim" },
                new Person { Mark = 73, Name="Mike" },
                new Person { Mark = 92, Name="Stefan" },
                new Person { Mark = 84, Name="George" },
                new Person { Mark = 24, Name="Arthur" },
                new Person { Mark = 58, Name="Nigel" }
            };

            Console.WriteLine("People:");
            OutputPeople(people);
```

```
            IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
                new WSHttpBinding(),
                new EndpointAddress("http://localhost:51425/AwardService.svc"));
            client.SetPassMark(70);
            Person[] awardedPeople = client.GetAwardedPeople(people);

            Console.WriteLine();
            Console.WriteLine("Awarded people:");
            OutputPeople(awardedPeople);

            Console.ReadKey();
        }

        static void OutputPeople(Person[] people)
        {
            foreach (Person person in people)
            {
                Console.WriteLine("{0}, mark: {1}", person.Name, person.Mark);
            }
        }
    }
}
```

代码段 Ch26Ex02Client\Program.cs

(16) 运行应用程序，结果如图 26-10 所示。

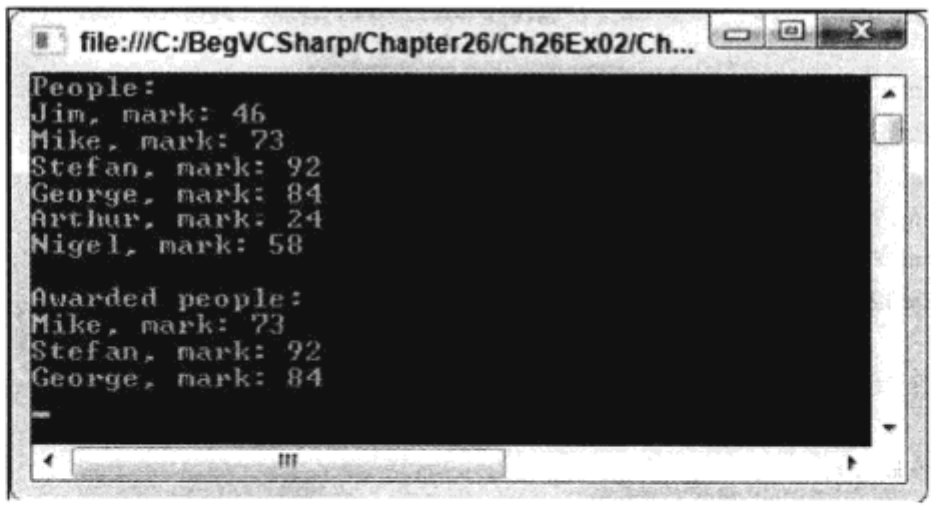


图 26-10

示例的说明

这个示例在类库项目中创建了一系列合同，在 WCF 服务和客户程序中使用了这个类库。与前面的示例一样，这个服务也驻留在 Web 服务器上。这个服务的配置也被减少到最低程度。

在这个示例中，主要区别是客户程序不需要元数据，因为客户程序可以访问合同程序集。客户程序不是从元数据中生成一个代理类，而是通过另一种方法获得服务合同接口的引用。这个示例中另一个值得注意的地方是使用会话维护服务中的状态，这需要 WSHttpBinding 绑定，而不是 BasicHttpBinding 绑定。

这个示例使用的数据合同是一个简单的类 Person，它有一个 string 属性 Name 和一个 int 属性 Mark。使用的 DataContractAttribute 特性和 DataMemberAttribute 特性没有进行定制，也不需要给这

个合同重复迭代代码。

定义服务合同时，给 IAwardService 接口应用了 ServiceContractAttribute 特性。这个特性的 SessionMode 属性设置为 SessionMode.Required，因为这个服务需要状态：

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IAwardService
{
```

第一个操作合同SetPassMark()设置状态，因此其OperationContractAttribute 的 IsInitiating 属性设置为 true。这个操作不返回任何值，所以将 IsOneWay 设置为 true，把操作定义为单向操作：

```
[OperationContract(IsOneWay=true,IsInitiating=true)]
void SetPassMark(int passMark);
```

另一个操作合同 GetAwardedPeople()不需要进行任何定制，使用前面定义的数据合同：

```
   [OperationContract]
   Person[] GetAwardedPeople(Person[] peopleToTest);
}
```

这两个类型 Person 和 IAwardService 都可以用于服务和客户程序。服务在 AwardService 类型中实现了 IAwardService 合同，它不包含任何可标记的代码。这个类与前面的服务类的唯一区别是，这个类是有状态的。这是允许的，因为定义了一个会话，来关联来自客户程序的消息。

为了确保服务使用 WSHttpBinding 绑定，给服务添加了如下 Web.config：

```
<protocolMapping>
  <add scheme="http" binding="wsHttpBinding"/>
</protocolMapping>
```

这重写了 HTTP 绑定的默认映射。另外，也可以手工配置服务，保留已有的默认配置，但这个重写的配置要简单得多。注意这类重写配置会应用于项目中的所有服务。如果项目中有多个服务，就必须确保每个服务都能接受这个绑定。

客户程序比较有趣，主要是因为下面这行代码：

```
IAwardService client = ChannelFactory < IAwardService > .CreateChannel(
   new WSHttpBinding(),
   new EndpointAddress("http://localhost:51425/AwardService.svc"));
```

客户程序没有用 app.config 文件来配置与服务的通信，也没有从元数据中定义代理类，来与服务通信。而是通过 ChannelFactory<T>.CreateChannel()方法创建代理类。这个方法创建了一个实现 IAwardService 客户程序的代理类，但在后台生成的类与服务通信，就像前面通过元数据生成的代理一样。

如果通过 ChannelFactory<T>.CreateChannel()方法创建代理类，通信信道就默认为在 1 分钟后超时，导致通信错误。使连接一直处于激活状态有许多方式，但这些都超出了本章的讨论范围。

以这种方式创建代理类是一种非常有用的技术，可以快速生成客户应用程序。

26.3.3 自驻留的 WCF 服务

本章前面介绍了驻留在 Web 服务器上的 WCF 服务。它们可以在 Internet 上通信，但对于本地网络通信而言，这并不是最高效的方式。一方面，需要用计算机上的 Web 服务器驻留服务，另一方面，在应用程序的体系结构上出现一个独立的 WCF 服务可能并不合适。

因此应使用自驻留的 WCF 服务。自驻留的 WCF 服务存在于创建它的进程中，而不存在于特别建立的主机应用程序(如 Web 服务器)的进程中。这样，就可以使用控制台应用程序或 Windows 应用程序驻留服务了。

要建立自驻留的 WCF 服务，需要使用 System.ServiceModel.ServieceHost 类。用要驻留的服务类型或服务类的一个实例来实例化这个类。通过属性或方法可以配置服务宿主，也可以通过配置文件来配置。实际上，宿主进程(如 Web 服务器)使用 ServiceHost 实例完成该驻留任务。自驻留时，区别是直接与这个类交互操作。但是，在宿主应用程序的 app.config 文件中，<system.serviceModel>段中的配置使用的语法与本章前面的配置段中的相同。

可以通过任意协议提供自驻留的 WCF 服务，但是一般在这种类型的应用程序中使用 TCP 或指定管道绑定。通过 HTTP 访问的服务常常位于 Web 服务器进程中，因为可以获得 Web 服务器提供的额外功能，如安全性等。

如果要驻留 MyService 服务，可以使用下面的代码创建 ServiceHost 的一个实例：

```
ServiceHost host = new ServiceHost(typeof(MyService));
```

如果要存储 MyService 的实例 MyServiceObject，可以编写如下代码，创建 ServiceHost 的一个实例：

```
MyService myServiceObject = new MyService();
ServiceHost host = new ServiceHost(myServiceObject);
```

警告：只有配置了服务，使调用总是可以路由到同一个对象实例上，才能在 ServiceHost 中存储服务实例。为此，必须给服务类应用 ServiceBehaviorAttribute 特性，将这个特性的 InstanceContextMode 属性设置为 InstanceContextMode.Single。

创建了 ServiceHost 实例后，就可以通过属性配置服务、其端点和绑定。另外，如果把配置放在.config 文件中，就会自动配置 ServiceHost 实例。

有了配置好的ServiceHost 实例后，为了开始驻留服务，应使用 ServiceHost.Open()方法。同样，通过 ServiceHost.Close()方法可以停止驻留服务。第一次驻留 TCP 绑定的服务时，如果启用它，可能会收到 Windows 防火墙服务发出的一个警告，因为它阻塞了默认的 TCP 端口。只有给这个服务打开 TCP 端口，才能开始监听该端口。

下面的示例使用自驻留技术通过 WCF 服务提供 WPF 应用程序的一些功能。

试一试：自存储的 WCF 服务

(1) 创建一个新的 WPF 应用程序 Ch26Ex03，将其保存在 C:\BegVCSharp\Chapter26 目录中。

(2) 使用 Add New Item 向导给项目添加一个新的 WCF 服务 AppControlService。

(3) 修改 MainWindow.xaml 中的代码，如下所示：

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch26Ex03.MainWindow"
  Title="Solar Evolution" Height="450" Width="430"
  Loaded="Window_Loaded" Closing="Window_Closing">
  <Grid Height="400" Width="400" HorizontalAlignment="Center"
   VerticalAlignment="Center">
   <Rectangle Fill="Black" RadiusX="20" RadiusY="20"
    StrokeThickness="10">
    <Rectangle.Stroke>
     <LinearGradientBrush EndPoint="0.358,0.02"
      StartPoint="0.642,0.98">
      <GradientStop Color="#FF121A5D" Offset="0"/>
      <GradientStop Color="#FFB1B9FF" Offset="1"/>
     </LinearGradientBrush>
    </Rectangle.Stroke>
   </Rectangle>
   <Ellipse Name="AnimatableEllipse" Stroke="{x:Null}" Height="0"
    Width="0" HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Ellipse.Fill>
     <RadialGradientBrush>
      <GradientStop Color="#FFFFFFFF" Offset="0"/>
      <GradientStop Color="#FFFFFFFF" Offset="1"/>
     </RadialGradientBrush>
    </Ellipse.Fill>
    <Ellipse.Effect>
     <DropShadowEffect ShadowDepth="0" Color="#FFFFFFFF"
      BlurRadius="50"/>
    </Ellipse.Effect>
   </Ellipse>
  </Grid>
</Window>
```

代码段 Ch26Ex03\MainWindow.xaml

(4) 修改 MainWindow.xaml.cs 中的代码，如下所示：

```
using System.Windows.Shapes;
using System.ServiceModel;
using System.Windows.Media.Animation;

namespace Ch26Ex03
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private AppControlService service;
        private ServiceHost host;
        public MainWindow()
        {
```

```
        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        service = new AppControlService(this);
        host = new ServiceHost(service);
        host.Open();
    }

    private void Window_Closing(object sender,
        System.ComponentModel.CancelEventArgs e)
    {
        host.Close();
    }

    internal void SetRadius(double radius, string foreTo,
        TimeSpan duration)
    {
        if (radius > 200)
        {
            radius = 200;
        }
        Color foreToColor = Colors.Red;
        try
        {
            foreToColor =
                (Color)ColorConverter.ConvertFromString(foreTo);
        }
        catch
        {
            // Ignore color conversion failure.
        }
        Duration animationLength = new Duration(duration);

        DoubleAnimation radiusAnimation = new DoubleAnimation(
            radius * 2, animationLength);
        ColorAnimation colorAnimation = new ColorAnimation(
            foreToColor, animationLength);
        AnimatableEllipse.BeginAnimation(Ellipse.HeightProperty,
            radiusAnimation);
        AnimatableEllipse.BeginAnimation(Ellipse.WidthProperty,
            radiusAnimation);
        ((RadialGradientBrush)AnimatableEllipse.Fill).GradientStops[1]
            .BeginAnimation(GradientStop.ColorProperty, colorAnimation);
    }
  }
}
```

代码段 Ch26Ex03\MainWindow.xaml.cs

(5) 修改 IAppControlService.cs 中的代码，如下所示：

可从
wrox.com
下载源代码

```
[ServiceContract]
public interface IAppControlService
{
    [OperationContract]
    void SetRadius(int radius, string foreTo, int seconds);
}
```

代码段 Ch26Ex03\IAppControlService.cs

(6) 修改 AppControlService.cs 中的代码，如下所示：

可从
wrox.com
下载源代码

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
    private MainWindow hostApp;

    public AppControlService(MainWindow hostApp)
    {
        this.hostApp = hostApp;
    }

    public void SetRadius(int radius, string foreTo, int seconds)
    {
        hostApp.SetRadius(radius, foreTo, new TimeSpan(0, 0, seconds));
    }
}
```

代码段 Ch26Ex03\AppControlService.cs

(7) 修改 app.config 中的代码，如下所示：

可从
wrox.com
下载源代码

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Ch26Ex03.AppControlService">
        <endpoint address="net.tcp://localhost:8081/AppControlService"
          binding="netTcpBinding"
          contract="Ch26Ex03.IAppControlService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

代码段 Ch26Ex03\Web.config

(8) 在项目中添加一个新的控制台应用程序 Ch26Ex03Client。

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

(10) 配置解决方案，使之有多个启动项目，让多个项目同时启动，如图 26-11 所示。

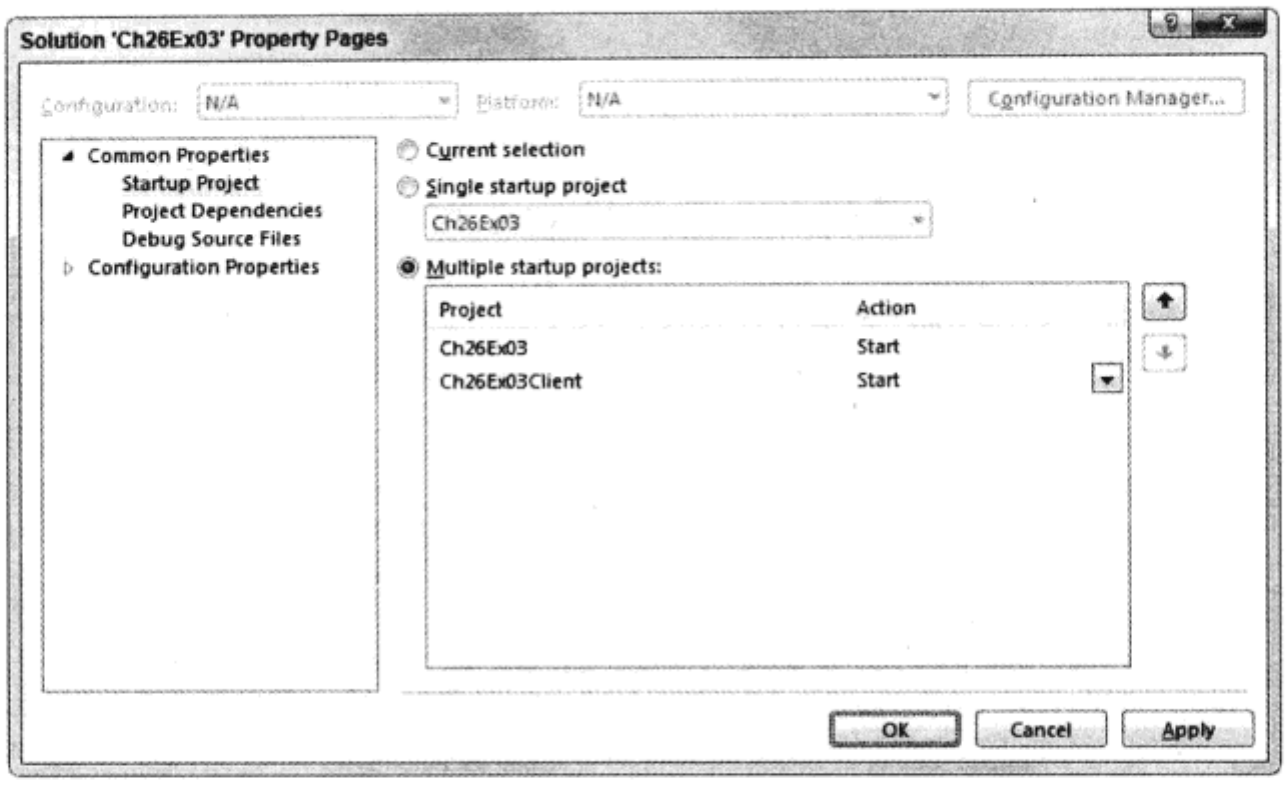


图 26-11

(11) 在 Ch26Ex03Client 项目中添加对 System.ServiceModel.dll 和 Ch26Ex03 的引用。

(12) 修改 Program.cs 中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex03;
using System.ServiceModel;

namespace Ch26Ex03Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press enter to begin.");
            Console.ReadLine();
            Console.WriteLine("Opening channel.");
            IAppControlService client =
                ChannelFactory<IAppControlService>.CreateChannel(
                    new NetTcpBinding(),
                    new EndpointAddress(
                        "net.tcp://localhost:8081/AppControlService"));
            Console.WriteLine("Creating sun.");
            client.SetRadius(100, "yellow", 3);
            Console.WriteLine("Press enter to continue.");
            Console.ReadLine();
            Console.WriteLine("Growing sun to red giant.");
            client.SetRadius(200, "Red", 5);
            Console.WriteLine("Press enter to continue.");
            Console.ReadLine();
            Console.WriteLine("Collapsing sun to neutron star.");
            client.SetRadius(50, "AliceBlue", 2);
```

```
            Console.WriteLine("Finished. Press enter to exit.");
            Console.ReadLine();
        }
    }
}
```

代码段 Ch26Ex03Client\Program.cs

(13) 运行解决方案。出现提示时，打开 Windows 防火墙 TCP 端口，使 WCF 可以监听连接。

(14) 显示 Solar Evolution 窗口和控制台应用程序窗口时，在控制台窗口中按下回车键。结果如图 26-12 所示。

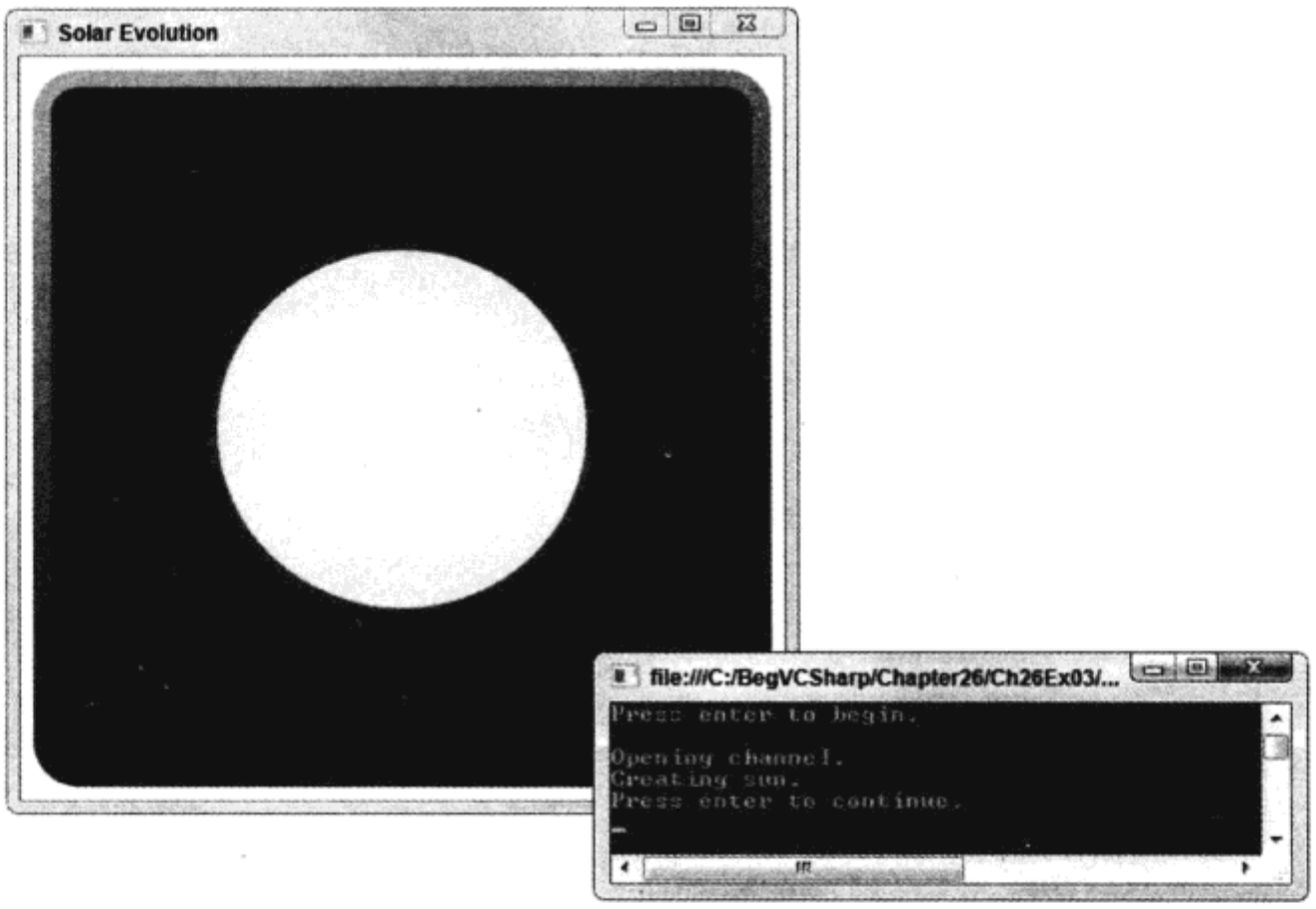


图　26-12

(15) 在控制台窗口中继续按下回车键，继续太阳演化循环。

示例的说明

这个示例在 WPF 应用程序中添加了一个 WCF 服务，用它控制 Ellipse 控件的动画。我们创建了一个简单的客户应用程序来测试服务。如果不熟悉WPF，不必过多地考虑示例中的 XAML 代码，我们只对 WCF 感兴趣。

WCF 服务 AppControlService 有一个操作 SetRadius()，客户程序调用这个操作来控制动画。这个方法与和它同名的方法通信，同名方法在 WPF 应用程序的 Window1 类中定义。为此，服务必须引用应用程序，所以必须寄宿该服务的一个对象实例。如前所述，这表示服务必须使用行为特性：

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
   ...
}
```

在 Window1.xaml.cs 中，在 Windows_Loaded()事件处理程序中创建服务实例。这个方法也为服务创建了一个 ServiceHost 对象，并调用了其 Open()方法，以便开始寄宿：

```
public partial class MainWindow : Window
{
   private AppControlService service;
   private ServiceHost host;
   ...

   private void Window_Loaded(object sender, RoutedEventArgs e)
   {
      service = new AppControlService(this);
      host = new ServiceHost(service);
      host.Open();
   }
```

在 Window_Closing()事件处理程序中，应用程序关闭时，寄宿过程中断。

配置文件非常简单，它定义了 WCF 服务的一个端点，监听端口 8081 的 net.tcp 地址，使用默认的 NetTcpBinding 绑定：

```
<service name="Ch26Ex03.AppControlService">
 <endpoint address="net.tcp://localhost:8081/AppControlService"
   binding="netTcpBinding"
   contract="Ch26Ex03.IAppControlService"/>
</service>
```

这与客户应用程序中的代码相匹配：

```
IAppControlService client =
   ChannelFactory < IAppControlService > .CreateChannel(
      new NetTcpBinding(),
      new EndpointAddress(
         "net.tcp://localhost:8081/AppControlService"));
```

客户程序创建客户代理类时，可以调用 SetRadius()方法，给它传递半径、颜色和动画持续时间等参数，这些都会通过服务转发给 WPF 应用程序。接着，WPF 应用程序中的简单代码定义并使用动画，来改变椭圆的大小和颜色。

如果使用的计算机名不是 localhost，网络允许在指定的端口上通信，这段代码就可以在网络上工作。另外，还可以把客户程序和宿主应用程序进一步分离，并通过 Internet 连接起来。无论采用什么方式，WCF 服务都提供了很好的通信方式，建立它不需要付出过多努力。

26.4 小结

本章介绍了使用 WCF 服务在应用程序、进程和计算机之间通信的基本技术。首先学习了 WCF 服务的概念，它与 Web 服务或远程实现的区别，使用 WCF 服务需要了解的概念。接着讨论了如何编写 WCF 服务，如何在客户程序中使用 WCF 服务，如何以各种方式存储 WCF 服务。

这是在应用程序中使用 WCF 服务所必需的最少知识。本章仅触及 WCF 服务的皮毛，尤其是.config 文件配置和行为。WCF 架构允许集成高级的安全体系，以任意方式定制通信。

如果希望学习 WCF 服务的更多知识，可以阅读 *Professional* WCF *4*(Wrox，2010 年 6 月)。第 27 章将介绍.NET 3.5 中的最后一种重要的新技术：Workflow Foundation(在.NET 4 中做了很大的改进)。

26.5 练习

(1) 下述哪些应用程序可以寄宿 WCF 服务？

a. Web 应用程序

b. Windows 窗体应用程序

c. Windows 服务

d. COM+应用程序

e. 控制台应用程序

(2) 如果要与 WCF 服务交换 MyClass 类型的参数，应实施什么类型的合同？需要什么特性？

(3) 如果把 WCF 服务寄宿在 Web 应用程序中，应对服务使用的基端点进行什么扩展？

(4) 在自寄宿 WCF 服务时，必须设置 ServiceHost 类的属性，调用它的方法，来配置服务。对吗？

(5) 提供服务合同 IMusicPlayer 的代码，它定义了 Play()、Stop()和 GetTrackInformation()操作。在合适的地方使用单向方法。还要为这个服务定义什么其他合同？

附录 A 给出了练习答案。

26.6 本章要点

主　题	重 要 概 念
WCF 基础	WCF 提供了创建远程服务并与之通信的框架，它合并了 Web 服务和远程体系结构的元素，并使用新技术来达到该目标
通信协议	可以通过几个协议与 WCF 服务通信，包括 HTTP 和 TCP。这表示可以使用客户应用程序的本地服务，也可以使用其他计算机或网络上的服务。为此，应通过对应于该协议的绑定及需要的特性，在特定的端点上访问该服务。可以通过行为控制这些特性，例如使用会话状态或提供元数据。.NET 4 包含许多默认的设置，非常便于定义简单的服务
通信负载	对来自 WCF 服务的响应的调用一般编码为 SOAP 消息。也可以使用普通的 HTTP 消息，如果需要，还可以从头开始定义自己的负载类型
寄宿	WCF 服务可以寄宿在 IIS 或 Windows 服务中，也可以是自寄宿的。使用 IIS 等宿主来存储 WCF 服务，就可以利用该宿主内置的功能，包括安全性和应用程序池。自寄宿比较灵活，但可能需要更多的配置和编码
合同	通过合同定义 WCF 服务和客户代码之间的接口。服务及其提供的操作是通过服务和操作合同定义的。数据类型用数据合同定义。可使用消息合同和误合同来进一步定制通信
客户应用程序	客户应用程序与 WCF 服务通过代理类来通信。代理类为服务实现了服务合同接口，对这个接口的操作方法的所有调用都重定向到服务上。可以使用 Add Service Reference 工具生成代理，也可以通过信道工厂方法以编程方式创建代理。为了成功通信，必须配置客户程序，以匹配服务的配置

第27章

Windows Workflow Foundation

本章内容：

- 工作流的含义和执行工作流的方法
- 活动的含义
- 如何创建定制活动
- 如何从活动中发送电子邮件

Windows Workflow Foundation (WF)最初出现在.NET 3.0 中，在.NET 3.5 中做了修订，另添加了一些功能，使之更容易与 Windows Communication Foundation (WCF)集成在一起。.NET 4 完全重写了 Workflow，虽然其核心概念没有变化，但实现方式却完全不同。本章将介绍 Windows Workflow Foundation 4 (WF4)。

工作流的一个简化定义是“一个活动集合”，但这不是一个令人完全满意的定义。使用一个类比方法可能更加有效。

编写程序时，我们会使用语句(例如 if/else)，调用函数(Console.WriteLine)，也会执行循环中的一些代码。但终端用户可能并不理解编程，所以他们会向我们反映，他们希望系统做什么工作，而我们编写代码，以便满足他们的要求。

现在假定可以为终端用户提供一个做了极大简化的编程环境，在这个环境中，预先建立了语句和流程控制逻辑，终端用户只需把这些部分插入他们希望的位置即可。此时就可以使用 Workflow 4。语句和控制逻辑都称为活动，这些活动可以插入到工作流中。

27.1 Hello World

每本编程图书都需要一个 Hello World 示例，本书也不例外。但这个示例不使用传统的编程语言，而使用 Workflow 4。下面的示例将创建一个 Workflow 项目，添加一个活动，并执行该工作流。

试一试：简单的 Workflow 4 应用程序

(1) 在 Visual Studio 2010 中，创建一个新的 Workflow Console Application 项目，确保在屏幕中上部的下拉列表中选择.NET Framework 4，如图 27-1 所示。

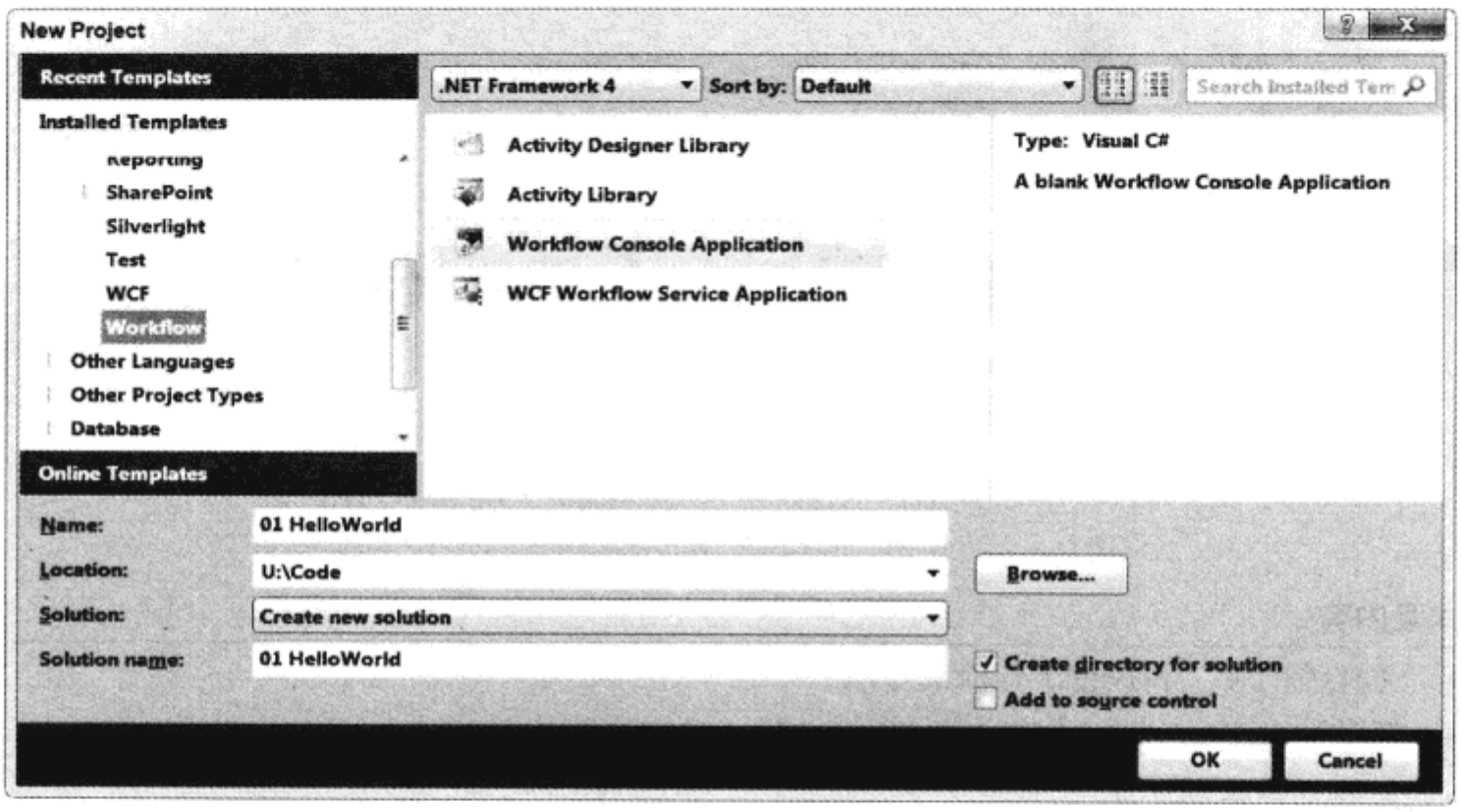


图 27-1

(2) 从工具箱中把一个 WriteLine 活动拖放到主设计区中。

(3) 在文本框中键入 Hello Workflow World，如图 27-2 所示。

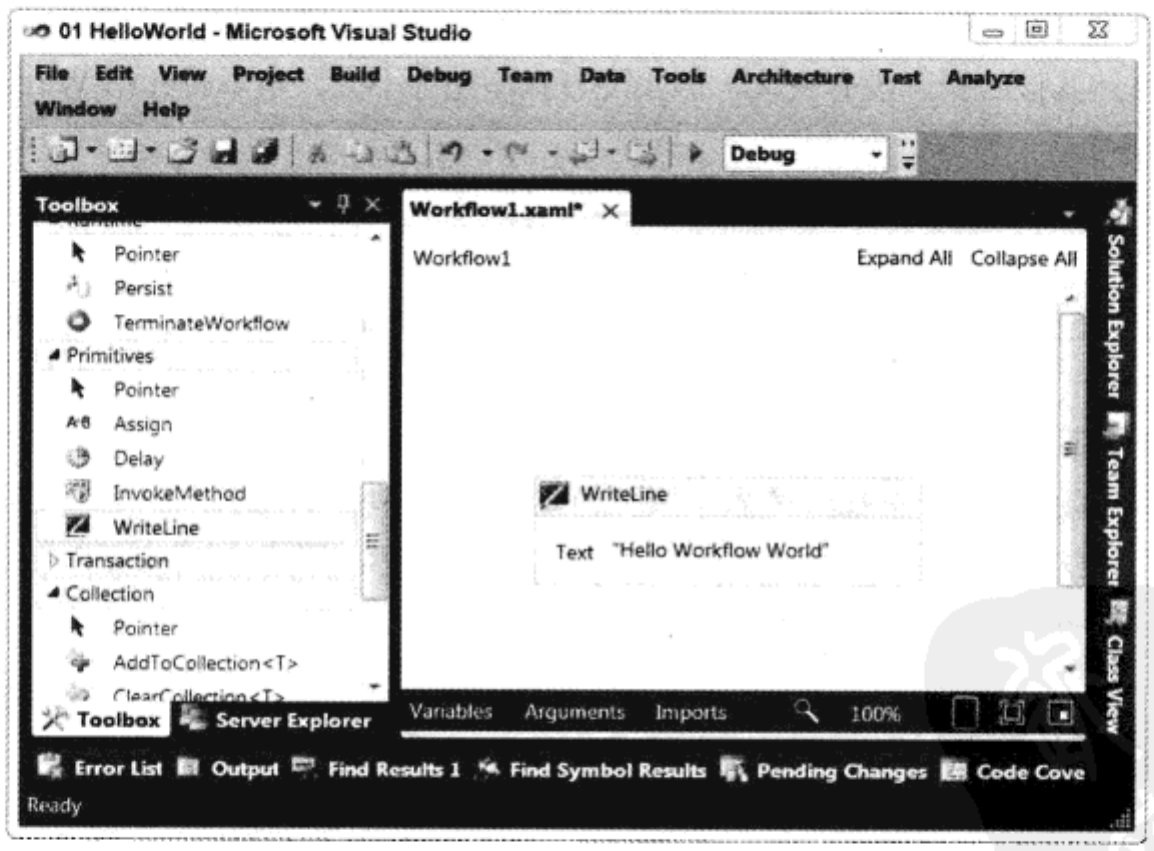


图 27-2

(4) 运行应用程序，查看输出的文本。

示例的说明

运行应用程序时，会执行工作流中的活动。在这个示例中只有一个活动，它输出一些文本。活动执行完毕后，工作流就完成了，应用程序随之退出。当然，可以在工作流中提供更多活动，执行更有效的任务，而不是仅仅在控制台上输出一条消息而已。

27.2　工作流和活动

上一节列举了一个简短的工作流例子，该工作流使用了一个简单的活动。工作流是一个活动集合，它定义了这些活动的执行顺序。该示例使用了顺序工作流，这种工作流包含了多个按顺序执行的活动。

活动是一个工作单元，分为两种类型。第一种是上例中的简单活动，例如 WriteLine 活动。此类活动只执行一个任务。另一种是复合活动，这种活动有几个熟悉的例子，例如 While 活动，它包含其他子活动。

因此，工作流类似于程序——工作流中的简单活动类似于常规的编程语句，流程控制活动类似于流程控制语句，执行方式也类似于程序。

如果工作流类似于程序，那么，我们可以像编程那样创建自己的函数吗？也许我们需要一个发送电子邮件的函数，或者把数据写入审核踪迹的函数。此时可以使用定制活动——可以编写这些低级的功能，用户只需把它们拖放到工作流中即可。

Windows Workflow Foundation 4 提供了许多活动，下一节将讨论其中的一些活动，说明如何在工作流中使用它们。

27.2.1　If 活动

这个活动的工作原理类似于 C#中的 if/else 语句。执行 If 活动时，将计算一个条件，接着根据该条件的计算结果确定工作流的执行路径。

使用 If 活动时，它会显示在工作流中，如图 27-3 所示。

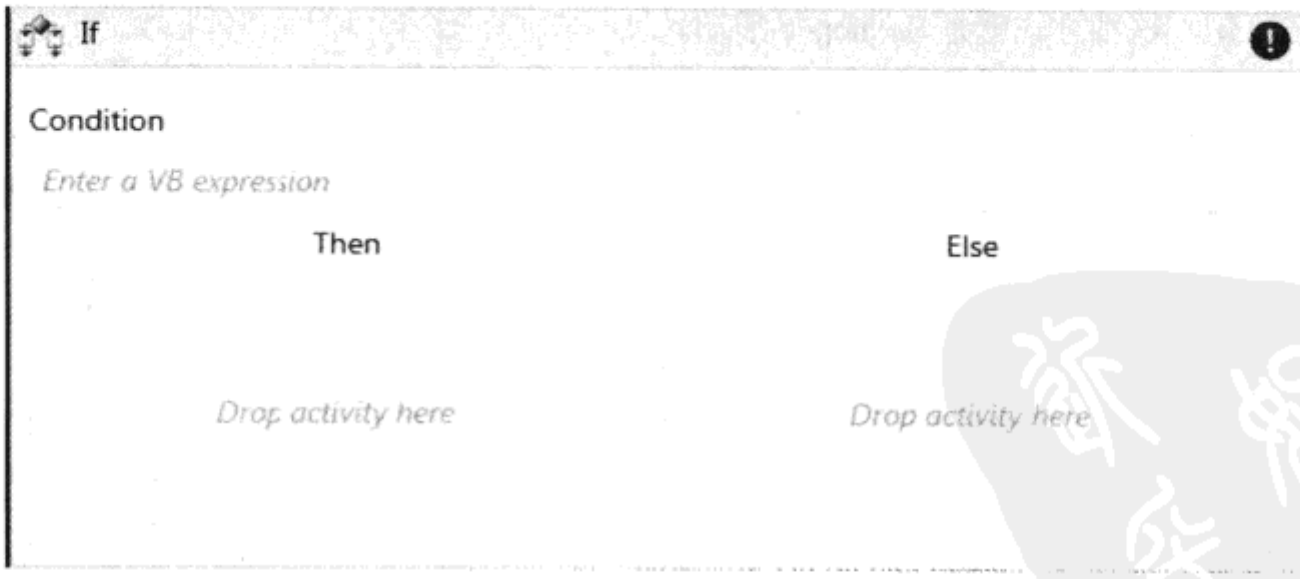


图　27-3

If 活动包含一个在执行活动时计算的条件表达式，以及用于 Then 和 Else 活动的占位符。Condition 属性是一个表达式，计算结果是一个布尔值，所以可以在这里包含任意有效的表达式。

Workflow 4 包含一个使用 Visual Basic 语法的表达式引擎，这对于 C#程序员而言是一个陌生的领域，因为 Visual Basic 与 C#大相径庭。但目前的情况就是这样，所以要使用内置的活动，必须学习 Visual Basic。只要记住 Visual Basic 的语法非常罗嗦，并且删除分号即可。

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，可以定义传送到工作流中的变元，然后就可以在 If 活动中由表达式计算该变元。本章后面会讨论变元和变量。

27.2.2 While 活动

任何程序员都很熟悉 While 活动，它会计算一个条件，若计算结果为 true，就执行活动体，如图 27-4 所示。

图 27-4

While 体中只能包含一个活动，但大多数程序都要求循环中有多个语句，所以必须通过某种方式添加更多语句。这种方式就是 Sequence 活动。

27.2.3 Sequence 活动

这个活动允许构建一系列其他活动，在执行时，会从第一个子活动开始，然后依次执行每个子活动，如图 27-5 所示。

图 27-5 显示了一个 Sequence 活动，它包含 3 个子活动：一个 WriteLine、一个 While 和另一个 WriteLine。如果执行这个工作流，会把第一条消息写到控制台中，接着执行 while 循环，最后把另一条消息写到控制台中。图 27-5 还显示了工作流设计器的另一个有用特性：折叠(和展开)活动。本例子中折叠了 While 活动，只显示该活动的名字——活动右边带两个 V 形图标的按钮允许切换活动内容的可见性。

在设计大型工作流时，折叠和展开活动的功能非常有用，因为可以折叠当前暂不处理的工作流部分，展开正在处理的部分。

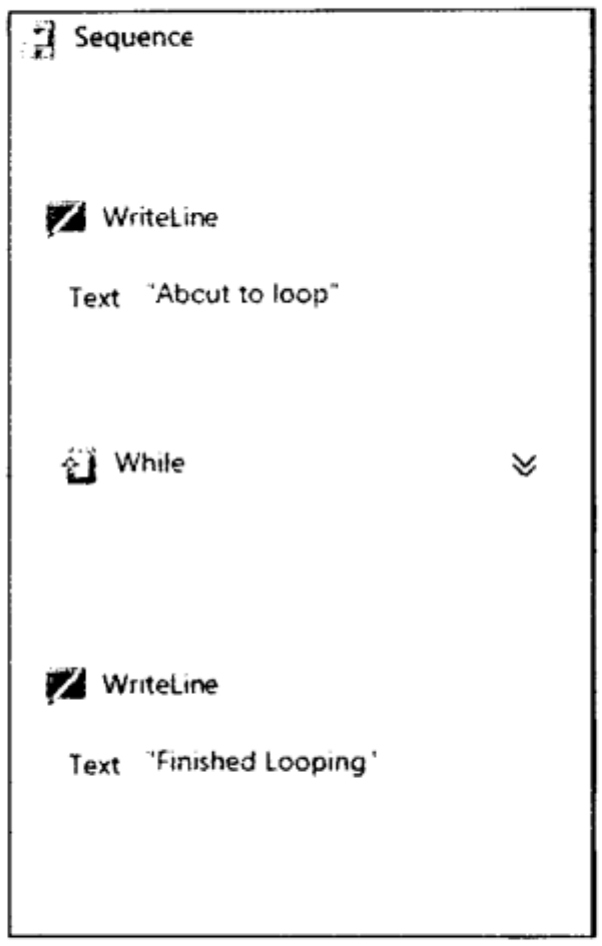


图　27-5

工作流设计器的其他特性使工作流变得更容易处理。例如，设计器右下角包含控件组，如图 27-6 所示。

这些控件(从左到右)可以把当前工作流放大到 100%，放大到某个定制的级别，使当前工作流的大小刚好能放在屏幕上，显示或隐藏概览窗口(显示工作流的缩略图)。另外，还可以双击某个复合活动，例如 While 或 If，以便隐藏所有更高级的活动，仅显示被单击的活动及其子活动。屏幕左上角的导览图显示了当前位置，如图 27-7 所示。

图　27-6

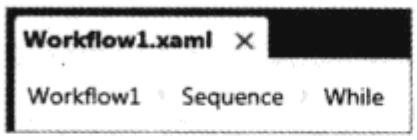


图　27-7

可以单击序列中的任意项，导航到工作流的更高层次，直至到达顶层为止。

27.3　变元和变量

在任何常规的编程语言中，都可以使用变元给函数传送值(以及检索响应)，可以在函数中定义变量，并把变量用作临时存储器。Workflow 也是一种编程语言，所以也可以使用这些结构。

但在进一步解释变元和变量之前，先考虑一下工作流的实际组成部分。如果在 Visual Studio 2010 中打开 Solution Explorer，查看本章第一部分创建的解决方案，会看到图 27-8 中突出显示的文件。

图　27-8

双击 Workflow1.xaml，会在设计器中显示工作流。但这只是一个普通的 XML 文件，所以不要双击它，而是右击，再选择 Open With。在打开的对话框中选择 XML Editor，就会打开相应文件，显示组成工作流的 XML：

```
<Activity mc:Ignorable="sap"
 x:Class="_01_HelloWorld.Workflow1"
 mva:VisualBasic.Settings="Assembly references and imported
         namespaces serialized as XML namespaces"
 xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
 xmlns:mva="clr-namespace:Microsoft.
  VisualBasic.Activities;assembly=System.Activities"
 xmlns:s="clr-namespace:System;assembly=mscorlib"
 xmlns:s1="clr-namespace:System;assembly=System"
 xmlns:s2="clr-namespace:System;assembly=System.Xml"
 xmlns:s3="clr-namespace:System;assembly=System.Core"
 xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
 xmlns:sap="http://schemas.microsoft.com/netfx/2009/xaml/activities/presentation"
 xmlns:scg="clr-namespace:System.Collections.Generic;assembly=System"
 xmlns:scg1="clr-namespace:System.
  Collections.Generic;assembly=System.ServiceModel"
 xmlns:scg2="clr-namespace:System.Collections.Generic;assembly=System.Core"
 xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=mscorlib"
 xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
 xmlns:sd1="clr-namespace:System.Data;assembly=System.Data.DataSetExtensions"
 xmlns:sl="clr-namespace:System.Linq;assembly=System.Core"
 xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine sad:XamlDebuggerXmlReader.FileName=
   "U:\Code\01 HelloWorld\01 HelloWorld\Workflow1.xaml"
   sap:VirtualizedContainerService.HintSize="209.6,200"
   Text="Hello Workflow World" />
</Activity>
```

代码中引用了许多 XML 名称空间，但主要部分仅显示了一个带有 Text 属性的 WriteLine 活动。

要创建传送给工作流的变元，可以使用工作流设计器中的 Arguments 设计器。这个选项显示在设计器界面的左下角，如图 27-9 所示。

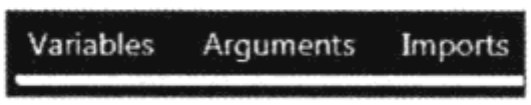


图 27-9

下面的示例将创建一个输入变元和一个变量，并在一个简单的工作流中使用它们。

试一试：使用变元和变量

(1) 在 Visual Studio 2010 中创建一个新的 Workflow Console Application 项目。

(2) 显示工作流时，单击 Arguments 按钮，创建一个字符串变元，如图 27-10 所示。把变元的名字设置为 Name，方向设置为 In，数据类型设置为 String。

(3) 在工作流中添加一个 Sequence 活动，再给 Sequence 添加一个 WriteLine 活动，在表达式文

本框中键入 Name。注意不要包括引号，因为这是变元的名字，而不是字面量字符串。

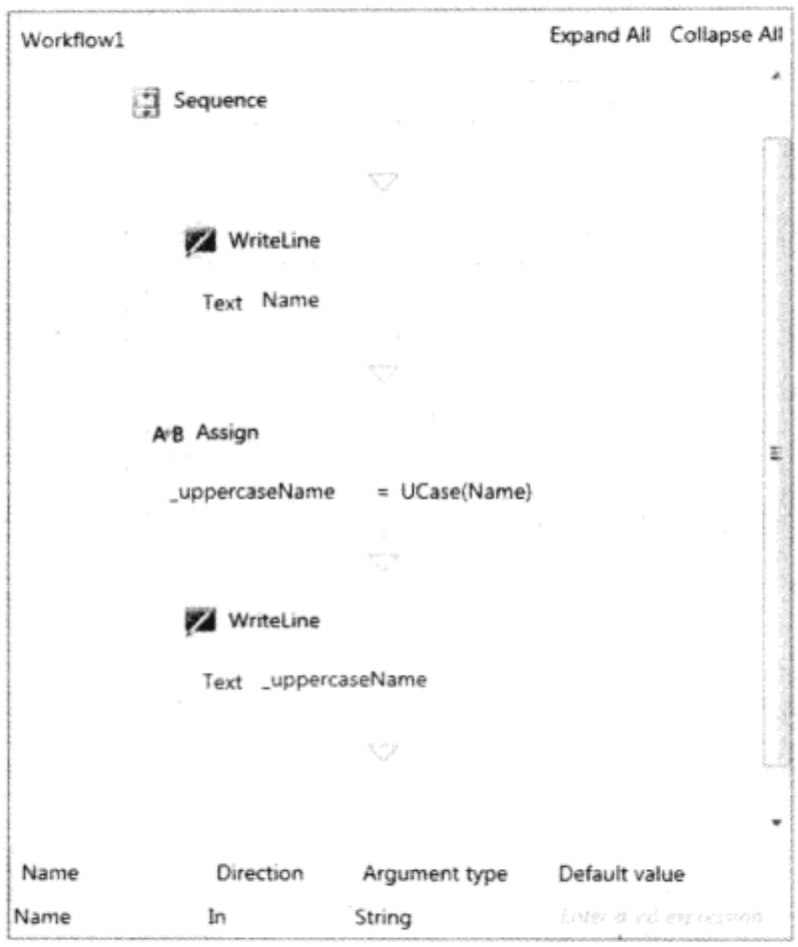


图 27-10

(4) 现在单击 Variables 按钮，定义一个 String 类型的_uppercaseName 变量。这个变量将用于存储 Name 变元的大写值。

(5) 把一个 Assign 活动拖放到设计器上。把该活动左边的文本框设置为_uppercaseName，右边的文本框设置为 UCase(Name)。这个活动给变量赋值，在本例中，该值是 VB 中的一个函数调用，该函数把其参数转换为大写字符串。

(6) 把一个 WriteLine 活动拖放到设计器上，将其文本框设置为_uppercaseName。

(7) 现在切换到 MainProgram.cs 文件，在其中给 Name 变元创建一个值，并传送给工作流。在该文件中包含如下代码：

```
class Program
{
    static void Main(string[] args)
    {
        WorkflowInvoker.Invoke(new Workflow1());
    }
}
```

(8) 需要修改上面的代码，以传送 Name 变元的值，进行如下修改：

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, object> parms = new Dictionary<string, object>();
        parms.Add("Name", "Morgan");
        WorkflowInvoker.Invoke(new Workflow1(), parms);
```

```
    }
}
```

如果愿意，还可以替换为自己的名字。

(9) 建立并运行项目，结果应是两行，其中一行是大写。

示例的说明

执行这个应用程序时，会把一个字符串参数传送到工作流中。输入参数被数据绑定到 WriteLine 活动和 Assign 活动上，在 Assign 活动中用一个表达式把输入参数转换为大写形式，然后保存到一个本地变量中。这个变量供第二个 WriteLine 活动使用。

如果现在使用 XAML 编辑器打开示例中的 Workflow1.xaml 文件，就可以在该文件的顶部看到变元的定义：

```
<x:Members>
    <x:Property Name="Name" Type="InArgument(x:String)" />
</x:Members>
```

另外还可以看到序列中变量的定义：

```
<Sequence.Variables>
    <Variable x:TypeArguments="x:String" Name="_uppercaseName" />
</Sequence.Variables>
```

与常规程序中的变量一样，工作流中的变量也有作用域，定义了变量在何时是可用的。一旦定义变量的活动结束了，就会释放该变量。

在上面的代码示例中，创建了一个名称/值对字典，接着使用键 Name 把一个值添加到字典中。这里使用的键值必须准确匹配变元的定义，否则就不会正确设置变元值，工作流就可能使用不正确的数据来执行。

在一般的编程语言中，函数的另一个行为是可以有返回值。类似地，不仅可以把变元传送给工作流，也可以从工作流中传出变元。前面创建了一个字典，以便把值传送给工作流，也可以用相同的方式在工作流结束时，从字典中返回输出变元。

变元包含方向的概念，方向可以是如下 3 个值中的一个：

- In：变元传送到工作流中。
- Out：从工作流返回变元。
- In/Out：变元传送到工作流中，在工作流结束时从工作流中返回。

只有定义为 Out 或 In/Out 的变元才能在工作流结束时返回。为了读取从工作流返回的值，可以使用以下代码：

```
IDictionary<string,object> returnValues = WorkflowInvoker.Invoke(new Workflow1(),
    parms);
```

这里给 returnValues 变量赋予一个名称/值对字典，该字典包含工作流上定义的所有 Out 和 In/Out 变元。

下面的示例说明了如何将变元传给工作流以及如何从工作流传出变元。

试一试：返回变元

(1) 在 Visual Studio 2010 中创建一个新 Workflow Console Application 项目。

(2) 在工作流中创建一个字符串变元，把它定义为 In/Out 变元 Person。

(3) 把一个 Sequence 活动拖放到工作流中。

(4) 把一个 WriteLine 活动拖放到工作流中，把其文本表达式设置为：

```
String.Format ( "Person is called : {0}", Person )
```

这是 VB，并非 C#；这是一个表达式，所以不需要分号。

(5) 把一个 Assign 活动拖放到工作流中，把其左边的文本框设置为 Person，右边的文本框设置为如下表达式：

```
String.Format("You entered the name : {0}", Person)
```

此时的工作流应如图 27-11 所示。

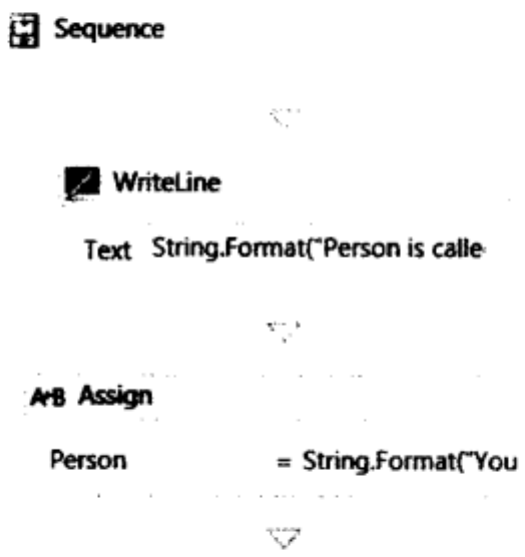


图 27-11

(6) 修改主 program.cs 文件，使之把一个变元传送给工作流，并输出所有 Out 或 In/Out 变元的值，如下面的代码段所示：

```
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("Person", "Morgan");

foreach (KeyValuePair<string, object> kvp in
   WorkflowInvoker.Invoke(new Workflow1(), parms))
{
    Console.WriteLine("{0} = {1}", kvp.Key, kvp.Value);
}
```

执行时，工作流应输出 Person 变元的值，再从上述代码中输出一个修改过的值。这说明，一旦

工作流完成，就把工作流中修改过的变元传送给调用者。

示例的说明

执行这个工作流时，给它传送一个输入变元。在工作流执行时，这个变元是可用的，在这个例子中，工作流结束时还会返回该变元，因为该变元定义为 In/Out。

27.4 定制活动

本章前面的示例仅使用了内置活动，而工作流还允许编写定制活动，接着就可以像内置活动那样使用定制活动。

本章的前面提到，有两大类活动：单一活动和复合活动。本节将创建这两类活动。

活动由拥有它的工作流(或父活动)安排执行。接着要执行什么操作主要取决于活动的编写者。例如对于 WriteLine 活动，在该活动的代码中一般会调用 Console.WriteLine。

编写活动时，一般会重写 Execute 方法，以便提供定制代码。这个方法随活动的基类而变化。这些基类及其执行方法如表 27-1 所示。

表 27-1

基 类	执 行 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)

最简单的基类是 CodeActivity，CodeActivity 还有一个泛型版本，该版本接受一个类型变元——用作执行该活动时的返回值。工作流可以返回变元，同样，活动在执行后也可以返回一个值，这个数据可以在工作流中绑定，这样，一个活动的输出就可以变成下一个活动的输入。

假定要在工作流中使用当前时间，就可以创建一个返回 DateTime 值的活动，该活动执行时通过调用 DateTime.Now 获得这个时间信息。只要创建一个定制活动即可，而无需把一个字符串输出到控制台上。下面的示例说明了创建定制活动的过程。

试一试：编写定制活动

(1) 在 Visual Studio 2010 中创建一个新的 Workflow Console Application 项目。

(2) 在解决方案中添加第二个项目，但这个项目使用 Activity Library 项目模板。这会创建一个

默认活动(Activity1.xaml)，可以从项目中删除这个默认活动，因为目前不使用它。

(3) 给类库添加一个新类 Timestamp，需要的代码如下：

```
using System;
using System.Activities;
namespace CustomActivities
{
    public class Timestamp : CodeActivity<DateTime>
    {
        protected override DateTime Execute(CodeActivityContext context)
        {
            return DateTime.Now;
        }
    }
}
```

这段代码定义了定制活动，并提供了对应 Execute 方法的实现代码(返回当前的日期/时间值)。

(4) 编译解决方案，再把对定制活动项目的引用添加到工作流项目中。这样就可以在工作流项目中使用定制活动，并把定制活动添加到工具箱中。

(5) 编辑主工作流，拖放一个 Sequence 活动。在 Sequence 活动中定义一个 DateTime 类型的变量，把这个变量命名为 currentDateTime。

(6) 拖放一个 Timestamp 活动，并显示其属性。需要把 Result 属性改为 currentDateTime，将活动的结果值赋予这个变量。图 27-12 显示了属性值。

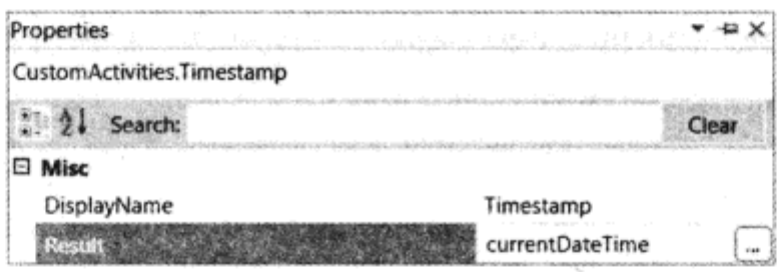


图 27-12

(7) 拖放一个 WriteLine 活动，设置其表达式，如下所示：

```
String.Format ( "The time read from the Timestamp activity is '{0}'",
    currentDateTime)
```

(8) 运行工作流。结果应显示了当前的日期和时间。

示例的说明

执行工作流时，会依次运行每个活动。如果活动派生自 CodeActivity，在安排运行该活动时就会调用 Execute 方法。这里的活动有一个返回值，该返回值在 Execute 方法中设置为当前的日期和时间。在这个示例中，Timestamp 活动的值存储在一个工作流变量中，接着使用 WriteLine 活动输出到控制台上。

Timestamp 活动非常简单。一般情况下，我们创建的活动要在其 Execute 方法中执行更多的操作。活动一般是自包含的单元，类似于传统编程语言中的函数。函数一般有一个或多个变元，通常这些变元作为参数传送给函数。但有时函数还通过当前的应用程序环境接收数据。

ASP.NET 中有一个很好的例子。静态属性 HttpContext.Current 允许访问各种属性，例如当前应用程序的状态、HttpRequest 和 HttpResponse。这个对象由 ASP.NET 处理管道定义，接着就可用于在该管道内部调用的任何对象。

在工作流中，使用“扩展”概念也可以实现类似的功能。

27.4.1 工作流扩展

扩展只是一个要从活动的 Execute 方法中访问的对象。一般需要为扩展定义一个接口，活动则针对这个接口进行编码。这样就可以替换该扩展的实现代码，而无需重新编写整个活动的代码。

例如，考虑一个发送电子邮件的活动。可以在活动内部硬编码电子邮件提供程序，但该活动只能使用这个提供程序。在这个例子中，最好定义一个活动使用的接口，再提供该接口的几个实现方式——其中一个使用 Outlook 发送邮件，另一个使用 Microsoft Exchange，等等。简言之，可以扩展电子邮件提供程序的列表，而无需修改活动。

为了使用电子邮件活动，还需要能定义传送给该活动的变元。我们肯定不希望硬编码电子邮件的接收人、主题或正文。与 WriteLine 活动一样，我们希望能定义可以在活动中设置的属性。为此，需要使用派生自 Argument 的类，例如 InArgument 和 OutArgument。

使用这些变元类替代活动中的属性。使用变元类型的原因与工作流在执行时存储其状态的方式相关，参见本章后面的内容。下面创建接口和定制活动。

试一试：定义 ISendEmail 接口和活动

(1) 在 Visual Studio 2010 中创建一个新 Class Library 项目，命名为 SharedInterfaces。

(2) 在库中添加一个接口 ISendEmail，如下所示：

```
/// <summary>
/// Interface used by the SendEmail activity to send an email
/// </summary>
public interface ISendEmail
{
    /// <summary>
    /// Sends an email
    /// </summary>
    /// <param name="sender">The person sending the email</param>
    /// <param name="recipient">The recipient of the email</param>
    /// <param name="subject">The subject</param>
    /// <param name="body">The body</param>
    void SendEmail(string sender, string recipient, string subject, string body);
}
```

(3) 现在给解决方案添加第二个项目。这次选择 Activity Library，命名为 CustomActivities。删除自动创建的 Activity1.xaml 文件。

(4) 给项目添加一个新类 SendEmail，其定义如下所示。需要基于活动库引用 SharedInterfaces 项目，以便包含 ISendEmail 接口的定义：

```
public class SendEmail : NativeActivity
{
    public InArgument<string> Sender { get; set; }
    public InArgument<string> Recipient { get; set; }
```

```
        public InArgument<string> Subject { get; set; }
        public InArgument<string> Body { get; set; }
        protected override void Execute(NativeActivityContext context)
        {
            context.GetExtension<ISendEmail>().SendEmail
                (Sender.Get(context),Recipient.Get(context),
                Subject.Get(context), Body.Get(context));
        }
    }
```

该活动定义了 4 个输入变元，并在 Execute 方法中使用它们。

示例的说明

执行这个活动时，只是查找 ISendEmail 扩展，并调用其 SendEmail 方法。发送电子邮件还需要更多的功能，这是下面两个示例的主题。

定义变元时，应使用泛型类 InArgument< >、OutArgument< >或 InOutArgument< >。在 Execute 方法中，使用有点古怪的语法 Argument.Get(context)，从当前执行环境中检索这些变元的值。这是由数据在工作流中的存储方式决定的。

在使用常规.NET 属性的传统类中，该类的数据存储在对象实例中。这说明，数据对于外部的调用者而言是不透明的；而在工作流中包含活动，意味着要在磁盘上存储工作流实例，只有完全序列化每个活动，才能把工作流存储在磁盘上。

这是 Workflow 3.x 的工作方式，它导致一些工作流需要使用磁盘上的大量空间。Workflow 4 使用了新的模型，只需要存储发生了变化的数据，因为传送给活动的环境对象可以跟踪这些变元的实际值。在 SendEmail 活动中，使用 Get()方法从环境中读取数据，所以在这个活动执行时，工作流的状态会保存下来。例如，如果调用 Set()方法，改变变元的值，工作流执行逻辑就会设置一个标记，表示有数据改变了，以便仅把这些变化的数据存储到磁盘上。这极大地提高了 Workflow 4 的性能，磁盘的使用量也显著减少了。

除了维护所有变元的值之外，环境对象还包含一个扩展集合。在上面的代码中，可以调用 GetExtension< >泛型方法，从环境对象中检索 ISendEmail 接口。这里传送了我们请求的接口类型，这个方法中的查找逻辑给代码返回扩展实例，以便在该扩展上调用 SendEmail 方法。

如下面的示例所示，下一步是把扩展添加到工作流中，为此，需要利用工作流程序集中的另一个类 WorkflowApplication。

试一试：使用 WorkflowApplication 类

(1) 给解决方案添加第三个项目。这应是一个 Workflow Console Application 项目。之后，把它设置为启动项目。

(2) 添加对 SharedInterfaces 和 CustomActivities 程序集的引用。

(3) 添加 ISendEmail 接口的实现代码，如下所示(这些代码不发送电子邮件，但至少把数据输出到控制台上)：

```
public class ConsoleSendEmail : ISendEmail
{
    public void SendEmail(string sender, string recipient, string subject,
```

```
        string body)
        {
            Console.WriteLine("Email to:      {0}", recipient);
            Console.WriteLine("      from:    {0}", sender);
            Console.WriteLine("      subject: {0}", subject);
            Console.WriteLine("      body:    {0}", body);
        }
    }
```

(4) 在工作流中添加一个 SendEmail 活动，设置所有的属性。按 F4 键会显示所选活动的属性表。

(5) 修改 Program.cs 文件，使用 WorkflowApplication 类执行工作流。这个类允许添加扩展，而 WorkflowInvoker 则不然。

```
class Program
{
    static void Main(string[] args)
    {
        WorkflowApplication app = new WorkflowApplication(new Workflow1());
        app.Extensions.Add(new ConsoleSendEmail());
        ManualResetEvent finished = new ManualResetEvent(false);
        app.Completed = (completedArgs) => { finished.Set(); };
        app.Run();
        finished.WaitOne();
    }
}
```

这里创建了 WorkflowApplication 实例，接着给它添加 ConsoleSendEmail 扩展。然后创建 ManualResetEvent，给它关联 Completed 事件。工作流结束时会引发该事件。之后调用 Run 方法以执行工作流，在事件完成后，工作流就结束了。如果生成并运行程序，则控制台上显示的输出应匹配在 SendEmail 活动上设置的属性值。

示例的说明

执行工作流应用程序时，添加到应用程序上的扩展会存储在 WorkflowApplication 对象的集合中。该对象安排每个活动的执行，在执行一个活动时，会创建一个环境对象，并将其传送给 Execute 方法。

环境对象的类型随活动使用的基类而变化。由于 SendEmail 活动派生自 NativeActivity，所以可以调用 GetExtension 方法，以检索添加到工作流应用程序中的任何扩展。

如果活动派生自 CodeActivity，传送给其 Execute 方法的环境对象就不能访问任何扩展，实际上，根据设计，CodeActivity 不能访问任何环境信息，

在工作流应用程序上调用 Run 时，会使用线程池中的线程来执行工作流，因此在执行工作流时，允许代码在后台继续执行。上面的代码使用 ManualResetEvent，并在处理程序中给 Completed 事件设置了 ManualResetEvent，使主应用程序与工作流的完成同步。

测试了前面的代码后，就可以确认 SendEmail 活动在工作，所以现在可以使用 System.Net.Mail 名称空间中的类创建 ISendEmail 的实际实现代码。理解这个名称空间的一个很好的资源是 www.systemnetmail.com。

另一种方法是用 Outlook 发送电子邮件，下一个例子就完成这个功能。

试一试：用 Outlook 发送电子邮件

(1) 给解决方案添加第三个项目。这应是一个 Workflow Console Application 项目。之后，把它设置为启动项目。

(2) 添加对 Outlook 对象模型的引用。为此，打开 Add Reference 对话框，选择 COM 选项卡。再向下滚动，找到 Microsoft Outlook Object Library。用于演示这个示例的计算机安装了 Microsoft Office 2007，其内部的版本号是 12.0，所以它会显示在对话框中，如图 27-13 所示。

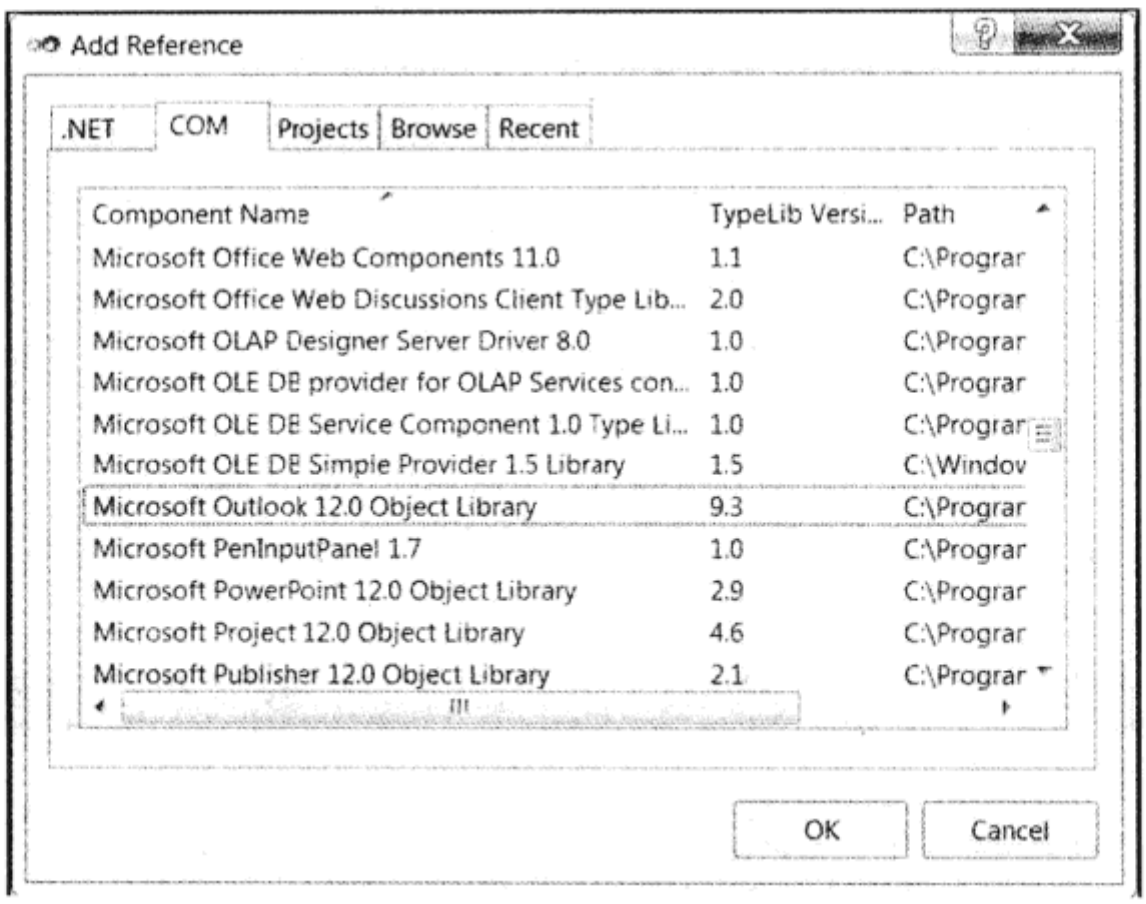


图 27-13

(3) 创建一个新类 OutlookSendEmail，键入如下代码：

```
public class OutlookSendEmail : ISendEmail
{
    public void SendEmail(string sender, string recipient, string subject,
    string body)
    {
        Application app = new Application();
        var mapi = app.GetNamespace("MAPI");
        mapi.Logon(ShowDialog: false, NewSession: false);
        var outbox = mapi.GetDefaultFolder(OlDefaultFolders.olFolderOutbox);

        MailItem email = app.CreateItem(OlItemType.olMailItem);
        email.To = recipient;
        email.Subject = subject;
        email.Body = body;
        email.Send();
    }
}
```

注意在这个示例中，没有指定发件人，因为电子邮件使用当前登录的用户的配置信息来发送。

(4) 修改 Program.cs 文件，把这个类用作电子邮件扩展：

```
app.Extensions.Add(new OutlookSendEmail());
```

应删除 ConsoleSendEmail 扩展，或者将其注释掉。

(5) 运行程序(确保 Outlook 也在运行)。如果查看 Sent Items 文件夹，应看到自动生成的电子邮件。这说明我们刚才发送了第一封自动生成的电子邮件。

示例的说明

OutlookSendEmail 类使用 Outlook 对象模型，通过 MailItem 类新建一封电子邮件。为了用 Outlook 发送电子邮件，需要构建 Application 对象的一个实例，再获取 MAPI 名称空间的引用。

如果尚未运行 Outlook，就需要在对 Login 的调用中指定用户名和密码。如果已在运行 Outlook，就可以忽略这些参数，此时 Outlook 将使用当前登录的用户的配置信息。

创建了 MailItem 后，就可以设置 To 属性，以指定收件人。要发送给多个收件人，可以在电子邮件地址之间添加分号。然后需要指定电子邮件的 Body 和 Subject，最后调用 Send( )。如果希望发送格式化的电子邮件，也可以使用 HTMLBody 属性。

如果在发件箱中看不到电子邮件，或者从代码中接收到异常，就需要跟踪这个异常。为此，可以在 Program.cs 文件中添加一些额外的代码，如下面的示例所示。

试一试：处理工作流的错误

在这个例子中，将学习如何捕获和处理工作流中的错误。

(1) 使用上一个例子的项目，修改 Program.cs 文件，如下所示：

```
static void Main(string[] args)
{
    WorkflowApplication app = new WorkflowApplication(new Workflow1());
    app.OnUnhandledException = (e) =>
        {
            return UnhandledExceptionAction.Abort;
        };
    app.Extensions.Add(new OutlookSendEmail());
    ManualResetEvent finished = new ManualResetEvent(false);
    app.Completed = (completedArgs) => { finished.Set(); };
    app.Aborted = (abortedEventArgs) =>
        {
            Console.WriteLine("Workflow Aborted.\r\n{0}",abortedEventArgs.Reason);
            finished.Set();
        };
    app.Run();
    finished.WaitOne();
}
```

已经添加了突出显示的条目。

(2) 运行应用程序。如果出现了异常，就显示一个消息 Workflow Aborted，然后把异常输出到控制台上。

示例的说明

工作流中出现未处理的异常时，首先调用的是 OnUnhandledException 委托。这里可以选择执行的动作有 Abort、Cancel 或 Terminate。这个委托传送给异常的一个实例，以便根据所抛出异常的类型确定执行什么操作。

如果选择 Abort(中止)工作流，就接着调用 Aborted 委托。默认是 Terminate(终止)工作流。

27.4.2　活动的有效性验证

如果没有定义变元，许多活动都不能执行，目前我们无法把给定的变元标记为必选变元。使用一些标准活动时，可能会在工作流设计器上显示一个错误消息，因为这些活动包含一些必选变元。

为了把属性标记为必选属性，可以在定义变元时使用[RequiredArgument]特性。在给变元添加这个特性时，在活动的右边会显示一个红色的感叹号图标，如图 27-14 所示。

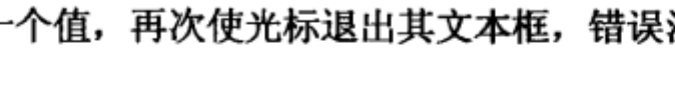


图　27-14

这表示一个或多个属性存在错误，如果把光标停放在该图标上，会显示一个描述该错误的工具提示。下面的示例将更新 SendMail 活动，把除 Sender 之外的所有变元标记为必选变元。

试一试：把变元标记为必选变元

(1) 打开 SendEmail.cs 文件，进行如下修改：

```
public class SendEmail : NativeActivity
{
    public InArgument<string> Sender { get; set; }
    [RequiredArgument]
    public InArgument<string> Recipient { get; set; }
    [RequiredArgument]
    public InArgument<string> Subject { get; set; }
    [RequiredArgument]
    public InArgument<string> Body { get; set; }
    protected override void Execute(NativeActivityContext context)
    {
        context.GetExtension<ISendEmail>().SendEmail(Sender.Get(context),
          Recipient.
    Get(context), Subject.Get(context), Body.Get(context));
    }
}
```

(2) 编译应用程序。

(3) 打开 Workflow1.xaml 文件，显示 SendEmail 活动的属性。修改一个加了[RequiredArgument]特性的属性，使光标退出其文本框。此时应看到错误图标，把鼠标停放在该图标上，查看错误的描述。

(4) 在必选变元中输入一个值，再次使光标退出其文本框，错误消息应消失了。

示例的说明

工作流可以检查活动，查找 RequiredArgument 特性。如果找到了带这个特性的属性，但该属性

没有定义其值，设计器类就会给活动加上错误图标。

定制活动就快完成了。最后一个任务是创建一个定制设计器，用于提供活动在设计期间可视化表示。

27.4.3 活动设计器

把活动拖放到设计界面上时，设计器会提供其可视化表示。一般这是一个 Windows 窗体类，而在 Visual Studio 2010 中，可以使用 XAML 定义活动的设计器。

XAML 详见第 22 章，这里不再重复。本节仅讨论对定制活动比较重要的部分。

一般在独立的程序集中创建活动的设计器类，它仅需要在设计期间使用，在执行活动时不需要它。Visual Studio 2010 包含 Activity Designer Library 项目类型，提供了足以开始创建设计器的功能，下面的例子就使用这个项目类型。

除了提供活动的可视化表示之外，设计器还可以提供数据输入字段。没有设计器，活动的所有属性都必须在属性表中设置；而对于定制设计器，可以选择在设计界面上包含一些属性。这可以为活动的用户提供非常棒的设计期间的体验。

下面的示例将再次更新 SendEmail 活动，添加一个定制设计器。

试一试：添加活动设计器

(1) 打开前面的解决方案，再添加一个新的 Activity Designer Library 项目，命名为 CustomActivities.Design。

(2) 这会创建一个空白的设计器 ActivityDesigner1。可以重命名这个设计器，或者添加一个新的设计器 SendEmailDesigner，无论采用哪种方式，都应有一个名称与活动名相似的设计器。

(3) 默认创建的 XAML 提供了一个空白的设计界面，在其中需要添加一些文本字段和标签。在设计器的 XAML 文件中添加如下 XAML——这定义了一组列和行，在这些列和行中将放置设计元素：

```
<sap:ActivityDesigner x:Class="CustomActivities.Design.SendEmailDesigner"
    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.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
    </Grid>
</sap:ActivityDesigner>
```

(4) 在屏幕上添加接收用户输入的元素：

```
        </Grid.RowDefinitions>
        <TextBlock Text="Recipient"/>
        <TextBox Text="{Binding ModelItem.Recipient}" Grid.Column="1"/>
        <TextBlock Text="Subject" Grid.Row="1"/>
        <TextBox Text="{Binding ModelItem.Subject}" Grid.Row="1" Grid.Column="1"/>
        <TextBlock Text="Body" Grid.Row="2"/>
        <TextBox Text="{Binding ModelItem.Body}" Grid.Row="2" Grid.Column="1"/>
    </Grid>
```

这些元素定义了一系列标签和文本框，它们数据绑定到底层的活动上——前缀 ModelItem 是实际活动的同义词。

(5) 需要把设计器与活动关联起来。最简单的方法是使用 Designer 特性。在 SendEmail 活动的顶部添加如下代码：

```
using System.ComponentModel;
namespace CustomActivities
{
    [Designer("CustomActivities.Design.SendEmailDesigner,
    CustomActivities.Design")]
    public class SendEmail : NativeActivity
    {
```

Designer 特性由 Visual Studio 读取，用于确定把哪个设计器关联到活动上，以及哪个程序集包含设计器。上面使用的字符串是设计器的 TypeName，它一般输入为字符串，以免在设计器程序集和活动程序集之间进行程序集引用。

(6) 在主工作流程序集中添加对 PresentationCore 程序集的引用。如果接着编译解决方案，打开包含 SendEmail 活动的工作流，将看到如图 27-15 所示的结果。

图 27-15

(7) 这个设计器功能完备，但不是很吸引人，为此可以在字段周围添加一些空格。修改 XAML，如下所示，就会得到比较好的设计结果。当然也可以添加色彩和图形，使设计界面更加鲜活。

```
<Grid>
    <Grid.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Margin" Value="0,2,4,2"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="Margin" Value="0,2,0,2"/>
        </Style>
    </Grid.Resources>
    <Grid.ColumnDefinitions>
```

这些资源定义了与文本块和文本框关联的样式。这里的这些样式仅应用了统一的页边距和对齐方式，使活动在屏幕上看起来更加美观。

示例的说明

Visual Studio 使用 Designer 特性查找与活动关联的类。如果找到了，就在屏幕上显示活动时使用该类。

在 XAML 中常常使用数据绑定把可视化类和后台类链接起来——在这个例子中，可视化类是设计器，后台类是活动。

27.5 小结

本章学习了 Windows Workflow Foundation 4，主要内容如下：

- 工作流的含义和执行方式
- 如何使用一些内置活动
- 如何创建自己的活动

27.6 练习

(1) 如何创建复合活动？

(2) 可以通过 WCF 展示工作流吗？如果能，如何展示？

(3) 如何确保工作流从停止的地方重新启动？

附录 A 给出了练习答案。

27.7 本章要点

主 题	重 要 概 念
工作流基础	工作流由活动组成，活动类似于传统编程语言中的语句。可以编写自己的活动。工作流一般包含一些内置的活动和一些定制的活动
If 活动	可以在工作流中使用 If 活动计算一个表达式，选择两条路径中的一条。表达式可以很简单，也可以很复杂，可以根据需要引用变量和变元
While 活动	这个活动允许在工作流中定义一个循环。循环的条件是一个表达式，该活动包含一个子活动，该子活动一般是一个序列，这样就可以在循环的每次迭代中添加其他多个活动
Sequence 活动	Sequence 活动允许自上而下地执行许多子活动
变元和变量	可以在工作流中传入传出变元，在工作流中还可以定义全局或本地变量。变元通过数据类型来定义，例如 String 或 Int32，还需要定义变元的方向。变量遵守的规则与传统的编程语言相同
工作流扩展	扩展可以用于改变工作流在运行期间的行为，而无需改变工作流。扩展一般编写为接口和该接口的实现代码
活动的有效性验证	可以把活动的一些属性定义为必选属性，这样终端用户就可以看出哪些属性必须定义其值，而没有定义其值的必选属性会在用户界面上显示错误图标
活动设计器	设计器可以用于美化活动的用户界面，使终端用户更容易使用活动。设计器是 XAML，可以创建任何标记，来显示定制活动的用户界面

附 A 录

习 题 答 案

第 1、2 章没有习题。

第 3 章

习题 1

```
super.smashing.great
```

习题 2

b)，因为它以数字开头；e)，因为它包含一个句点。

习题 3

不，理论上没有限制包含在 string 变量中的字符串的长度。

习题 4

这里，*和/运算符的优先级最高，其次是+, <<，最后是+=。本习题中的优先级可以用括号来演示，如下所示：

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

或者：

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

二者的结果相同。

习题 5

```
static void Main(string[] args)
{
    int firstNumber, secondNumber, thirdNumber, fourthNumber;
```

```
        Console.WriteLine("Give me a number:");
        firstNumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        secondNumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        thirdNumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        fourthNumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("The product of {0}, {1}, {2}, and {3} is {4}.",
                          firstNumber, secondNumber, thirdNumber, fourthNumber,
                          firstNumber * secondNumber * thirdNumber * fourthNumber);
    }
```

注意这里使用了 Convert.ToInt32()，本章没有介绍它们。

第 4 章

习题 1

```
(var1 > 10) ^ (var2 > 10)
```

习题 2

```
static void Main(string[] args)
{
   bool numbersOK = false;
   double var1, var2;
   var1 = 0;
   var2 = 0;
   while (!numbersOK)
   {
       Console.WriteLine("Give me a number:");
       var1 = Convert.ToDouble(Console.ReadLine());
       Console.WriteLine("Give me another number:");
       var2 = Convert.ToDouble(Console.ReadLine());
       if ((var1 > 10) ^ (var2 > 10))
       {
          numbersOK = true;
       }
       else
       {
          if ((var1 <= 10) && (var2 <= 10))
          {
             numbersOK = true;
          }
          else
          {
             Console.WriteLine("Only one number may be greater than 10.");
          }
       }
   }
   Console.WriteLine("You entered {0} and {1}.", var1, var2);
}
```

注意使用另一个逻辑时，执行效果更佳，如下所示：

```
static void Main(string[] args)
{
    bool numbersOK = false;
    double var1, var2;
    var1 = 0;
    var2 = 0;
    while (!numbersOK)
    {
        Console.WriteLine("Give me a number:");
        var1 = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        var2 = Convert.ToDouble(Console.ReadLine());
        if ((var1 > 10) && (var2 > 10))
        {
            Console.WriteLine("Only one number may be greater than 10.");
        }
        else
        {
            numbersOK = true;
        }
    }
    Console.WriteLine("You entered {0} and {1}.", var1, var2);
}
```

习题 3

代码如下：

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

使用赋值运算符=而不是布尔运算符==，是一个十分常见的错误。

习题 4

```
static void Main(string[] args)
{
    double realCoord, imagCoord;
    double realMax = 1.77;
    double realMin = -0.6;
    double imagMax = -1.2;
    double imagMin = 1.2;
    double realStep;
    double imagStep;
    double realTemp, imagTemp, realTemp2, arg;
    int iterations;
    while (true)
    {
```

```
realStep = (realMax - realMin) / 79;
imagStep = (imagMax - imagMin) / 48;
for (imagCoord = imagMin; imagCoord >= imagMax;
     imagCoord += imagStep)
{
    for (realCoord = realMin; realCoord <= realMax;
         realCoord += realStep)
    {
        iterations = 0;
        realTemp = realCoord;
        imagTemp = imagCoord;
        arg = (realCoord * realCoord) + (imagCoord * imagCoord);
        while ((arg < 4) && (iterations < 40))
        {
            realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                - realCoord;
            imagTemp = (2 * realTemp * imagTemp) - imagCoord;
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
        }
        switch (iterations % 4)
        {
            case 0:
                Console.Write(".");
                break;
            case 1:
                Console.Write("o");
                break;
            case 2:
                Console.Write("O");
                break;
            case 3:
                Console.Write("@");
                break;
        }
        Console.Write("\n");
    }
    Console.WriteLine("Current limits:");
    Console.WriteLine("realCoord: from {0} to {1}", realMin, realMax);
    Console.WriteLine("imagCoord: from {0} to {1}", imagMin, imagMax);

    Console.WriteLine("Enter new limits:");
    Console.WriteLine("realCoord: from:");
    realMin = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("realCoord: to:");
    realMax = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("imagCoord: from:");
    imagMin = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("imagCoord: to:");
    imagMax = Convert.ToDouble(Console.ReadLine());
  }
}
```

第 5 章

习题 1

a)和 c)的转换不能隐式进行。

习题 2

```
enum color : short
{
    Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black, White
}
```

可以，byte 类型可以包含 0～255 之间的数字。如果枚举项使用不同值，基于 byte 的枚举可以包含 256 项，如果给枚举项使用重复的值，就可以包含更多的项。

习题 3

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

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

习题 4

不，原因如下：

- 遗漏了语句末尾的分号。
- 第二行尝试访问 blab 中不存在的第 6 个元素。
- 第二行尝试指定未包含在双引号中的字符串。

习题 5

```
static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    string reversedString = "";
    for (int index = myString.Length - 1; index >= 0; index--)
    {
       reversedString += myString[index];
    }
    Console.WriteLine("Reversed: {0}", reversedString);
}
```

习题 6

```
static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    myString = myString.Replace("no", "yes");
    Console.WriteLine("Replaced \"no\" with \"yes\": {0}", myString);
}
```

习题 7

```
static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    myString = "\"" + myString.Replace(" ", "\" \"") + "\"";
    Console.WriteLine("Added double quotes around words: {0}", myString);
}
```

或者使用 String.Split():

```
static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
```

```
        string[] myWords = myString.Split(' ');
        Console.WriteLine("Adding double quotes around words:");
        foreach (string myWord in myWords)
        {
            Console.Write("\"{0}\" ", myWord);
        }
    }
```

第 6 章

习题 1

第一个函数的返回类型是 bool，但不返回一个 bool 值。

第二个函数有一个 params 变元，但这个变元不在变元列表的末尾。

习题 2

```
static void Main(string[] args)
{
    if (args.Length != 2)
    {
        Console.WriteLine("Two arguments required.");
        return;
    }
    string param1 = args[0];
    int param2 = Convert.ToInt32(args[1]);
    Console.WriteLine("String parameter: {0}", param1);
    Console.WriteLine("Integer parameter: {0}", param2);
}
```

注意这个答案包含的代码检查是否提供了两个变元，本题没有这个要求，但在本题中似乎应进行检查。

习题 3

```
class Program
{
    delegate string ReadLineDelegate();

    static void Main(string[] args)
    {
        ReadLineDelegate readLine = new ReadLineDelegate(Console.ReadLine);
        Console.WriteLine("Type a string:");
        string userInput = readLine();
        Console.WriteLine("You typed: {0}", userInput);
    }
}
```

习题 4

```
struct order
{
    public string itemName;
```

```
    public int unitCount;
    public double unitCost;

    public double TotalCost()
    {
        return unitCount * unitCost;
    }
}
```

习题 5

```
struct order
{
    public string itemName;
    public int unitCount;
    public double unitCost;

    public double TotalCost()
    {
        return unitCount * unitCost;
    }

    public string Info()
    {
        return "Order information: " + unitCount.ToString() + " " + itemName +
               " items at $" + unitCost.ToString() + " each, total cost $" +
               TotalCost().ToString();
    }
}
```

第 7 章

习题 1

这个语句仅对要用于所有版本的信息有效。而且，我们常常希望仅在使用调试版本时输出调试信息。此时，应首选 Debug.WriteLine()版本。

使用 Debug.WriteLine()版本还有一个优点，该版本不会编译到发布版本中，从而使最终代码文件变得更小。

习题 2

```
static void Main(string[] args)
{
    for (int i = 1; i < 10000; i++)
    {
        Console.WriteLine("Loop cycle {0}", i);
        if (i == 5000)
        {
            Console.WriteLine(args[999]);
        }
    }
}
```

在 VS 中，可以把断点放在下面的代码行上：

```
Console.WriteLine("Loop cycle {0}", i);
```

应修改断点的属性，把执行次数的条件设置为“执行次数等于 5000 时中断”。

在 VCE 中，可以把断点放在出错的代码行上，因为在 VCE 中无法采用上述方式修改断点的属性。

习题 3

错误，finally 块总是执行，它可能在处理 catch 块之后执行。

习题 4

```
static void Main(string[] args)
{
    Orientation myDirection;
    for (byte myByte = 2; myByte < 10; myByte++)
    {
        try
        {
            myDirection = checked((Orientation)myByte);
            if ((myDirection < Orientation.North) ||
                (myDirection > Orientation.West))
            {
                throw new ArgumentOutOfRangeException("myByte", myByte,
                    "Value must be between 1 and 4");
            }
        }
        catch (ArgumentOutOfRangeException e)
        {
            // If this section is reached then myByte < 1 or myByte > 4.
            Console.WriteLine(e.Message);
            Console.WriteLine("Assigning default value, Orientation.North.");
            myDirection = Orientation.North;
        }

        Console.WriteLine("myDirection = {0}", myDirection);
    }
}
```

注意这是一个小问题。因为枚举基于 byte 类型，所以可以给它赋予任何 byte 值，即使该值不是枚举中指定的名称，也是如此。在上面的代码中，如有必要，我们会生成自己的异常。

第 8 章

习题 1

Public、private 和 protected 是实际的可访问级别。

习题 2

错误，永远都不应手工调用对象的析构函数，.NET 运行库环境会在垃圾回收过程中自动完成该任务。

习题 3

不，可以在没有任何类实例的情况下调用静态方法。

习题 4

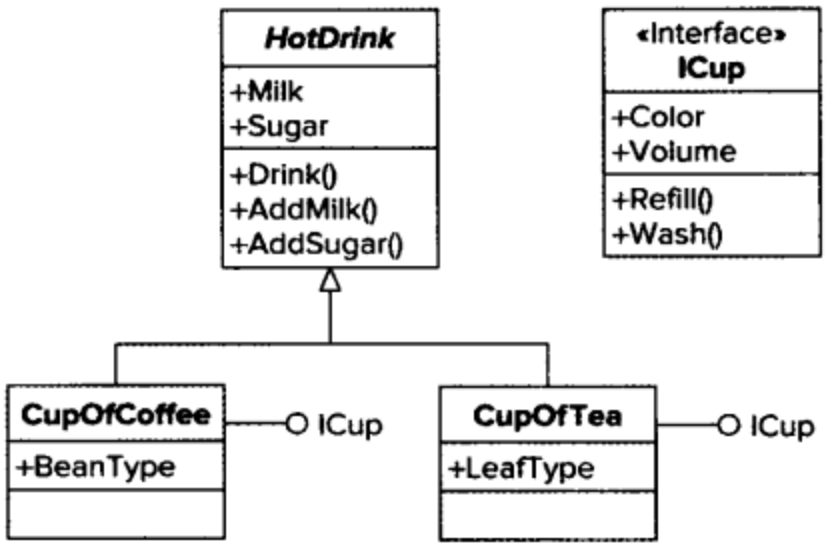


图 A-1

习题 5

```
static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    ICup cupInterface = (ICup)drink;
    cupInterface.Wash();
}
```

注意显式转换为 ICup 的代码行。这是必须的，因为 HotDrink 不支持 ICup 接口，但我们知道传送给这个函数的两个 cup 对象支持 ICup 接口。这很危险，因为也可以给这个函数传送其他类，但这些类也可能派生于 HotDrink，而 HotDrink 却不支持 ICup 接口。为了更正这个问题，应检查该接口是否得到支持：

```
static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    if (drink is ICup)
    {
        ICup cupInterface = drink as ICup;
        cupInterface.Wash();
    }
}
```

这里使用的 is 和 as 操作符在第 11 章介绍。

第 9 章

习题 1

myDerivedClass 派生于 MyClass，但是 MyClass 密封的，不能从 MyClass 中派生其他类。

习题 2

把它定义为静态类，或者将其所有构造函数定义为私有。

习题 3

不能创建的类可以通过它们拥有的静态成员来使用。实际上，甚至可以通过这些成员获取这些类的实例，如下所示：

```
class CreateMe
{
   private CreateMe()
   {
   }

   static public CreateMe GetCreateMe()
   {
      return new CreateMe();
   }
}
```

这里，公共构造函数可以访问私有构造函数，因为它在同一个类的定义中。

习题 4

为简单起见，下面的类定义显示为一个代码文件的一部分，而没有给每个类定义列出单独的代码文件：

```
namespace Vehicles
{
   public abstract class Vehicle
   {
   }
   public abstract class Car : Vehicle
   {
   }
   public abstract class Train : Vehicle
   {
   }
   public interface IPassengerCarrier
   {
   }
   public interface IHeavyLoadCarrier
   {
```

```
    }
    public class SUV : Car, IPassengerCarrier
    {
    }
    public class Pickup : Car, IPassengerCarrier, IHeavyLoadCarrier
    {
    }
    public class Compact : Car, IPassengerCarrier
    {
    }
    public class PassengerTrain : Train, IPassengerCarrier
    {
    }
    public class FreightTrain : Train, IHeavyLoadCarrier
    {
    }
    public class T424DoubleBogey : Train, IHeavyLoadCarrier
    {
    }
}
```

习题 5

```
using System;
using Vehicles;

namespace Traffic
{
  class Program
  {
    static void Main(string[] args)
    {
      AddPassenger(new Compact());
      AddPassenger(new SUV());
      AddPassenger(new Pickup());
      AddPassenger(new PassengerTrain());
    }

    static void AddPassenger(IPassengerCarrier Vehicle)
    {
       Console.WriteLine(Vehicle.ToString());
    }
  }
}
```

第 10 章

习题 1

```
class MyClass
{
   protected string myString;
```

```
    public string ContainedString
    {
        set
        {
            myString = value;
        }
    }

    public virtual string GetString()
    {
        return myString;
    }
}
```

习题 2

```
class MyDerivedClass : MyClass
{
    public override string GetString()
    {
        return base.GetString() + " (output from derived class)";
    }
}
```

习题 3

如果方法有返回类型，就可以将其用作表达式的一部分：

```
x = Manipulate(y, z);
```

如果没有给部分方法提供实现代码，编译器就会在使用该部分方法的所有地方删除该方法。在上面的代码中，这会使 x 的结果变得模糊，因为 Manipulate()方法没有替代方法。如果没有这个方法，可能只需忽略整行代码，但编译器无法确定我们是否的确希望忽略它。

没有返回类型的方法不能作为表达式的一部分来调用，所以编译器可以安全地删除对部分方法调用的所有引用。

同样，也禁止使用 out 参数，因为在方法调用之前，用作 out 参数的变量必须是未定义的，而应在方法调用之后定义。删除方法定义会违反这个规则。

习题 4

```
class MyCopyableClass
{
    protected int myInt;

    public int ContainedInt
    {
        get
        {
            return myInt;
        }
        set
        {
            myInt = value;
```

```
        }
    }

    public MyCopyableClass GetCopy()
    {
        return (MyCopyableClass)MemberwiseClone();
    }
}
```

客户端代码：

```
class Program
{
    static void Main(string[] args)
    {
        MyCopyableClass obj1 = new MyCopyableClass();
        obj1.ContainedInt = 5;
        MyCopyableClass obj2 = obj1.GetCopy();
        obj1.ContainedInt = 9;
        Console.WriteLine(obj2.ContainedInt);
    }
}
```

这些代码显示 5，说明所复制对象有自己专用的 myInt 字段。

习题 5

```
using System;
using Ch10CardLib;

namespace Exercise_Answers
{
    class Class1
    {
        static void Main(string[] args)
        {
            while(true)
            {
                Deck playDeck = new Deck();
                playDeck.Shuffle();
                bool isFlush = false;
                int flushHandIndex = 0;
                for (int hand = 0; hand < 10; hand++)
                {
                    isFlush = true;
                    Suit flushSuit = playDeck.GetCard(hand * 5).suit;
                    for (int card = 1; card < 5; card++)
                    {
                        if (playDeck.GetCard(hand * 5 + card).suit != flushSuit)
                        {
                            isFlush = false;
                        }
                    }
                    if (isFlush)
                    {
```

```
                    flushHandIndex = hand * 5;
                    break;
                }
            }
            if (isFlush)
            {
                Console.WriteLine("Flush!");
                for (int card = 0; card < 5; card++)
                {
                    Console.WriteLine(playDeck.GetCard(flushHandIndex + card));
                }
            }
            else
            {
                Console.WriteLine("No flush.");
            }
            Console.ReadLine();
        }
    }
}
```

这些代码会循环下去，因为同花色是不常见的。可能需要按几次回车键，才能在洗好的扑克牌中找到一个同花色。为了验证一切像期望的那样执行，可以尝试将洗牌的代码行注释掉。

第 11 章

习题 1

```
using System;
using System.Collections;

namespace Exercise_Answers
{
    public class People : DictionaryBase
    {
        public void Add(Person newPerson)
        {
            Dictionary.Add(newPerson.Name, newPerson);
        }

        public void Remove(string name)
        {
            Dictionary.Remove(name);
        }

        public Person this[string name]
        {
            get
            {
                return (Person)Dictionary[name];
            }
```

```
            set
            {
                Dictionary[name] = value;
            }
        }
    }
}
```

习题 2

```
public class Person
{
   private string name;
   private int age;

   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }

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

public static bool operator >(Person p1, Person p2)
{
   return p1.Age > p2.Age;
}

public static bool operator <(Person p1, Person p2)
{
   return p1.Age < p2.Age;
}

public static bool operator >=(Person p1, Person p2)
{
   return !(p1 < p2);
}

public static bool operator <=(Person p1, Person p2)
```

```
    {
        return !(p1 > p2);
    }
}
```

习题 3

```
public Person[] GetOldest()
{
    Person oldestPerson = null;
    People oldestPeople = new People();
    Person currentPerson;
    foreach (DictionaryEntry p in Dictionary)
    {
        currentPerson = p.Value as Person;
        if (oldestPerson == null)
        {
            oldestPerson = currentPerson;
            oldestPeople.Add(oldestPerson);
        }
        else
        {
            if (currentPerson > oldestPerson)
            {
                oldestPeople.Clear();
                oldestPeople.Add(currentPerson);
                oldestPerson = currentPerson;
            }
            else
            {
                if (currentPerson >= oldestPerson)
                  {
                  oldestPeople.Add(currentPerson);
                }
            }
        }
    }
    Person[] oldestPeopleArray = new Person[oldestPeople.Count];
    int copyIndex = 0;
    foreach (DictionaryEntry p in oldestPeople)
    {
        oldestPeopleArray[copyIndex] = p.Value as Person;
        copyIndex++;
    }
        return oldestPeopleArray;
}
```

这个函数比较复杂，因为没有为 Person 定义==运算符，但仍可以构建逻辑。另外，返回 People 实例更加简单，因为在处理过程中比较容易操作这个类。作为一个折中方法，在整个函数中都使用了 People 实例，再在最后转换为一个 Person 实例数组。

习题 4

```
public class People : DictionaryBase, ICloneable
```

```
{
    public object Clone()
    {
        People clonedPeople = new People();
        Person currentPerson, newPerson;
        foreach (DictionaryEntry p in Dictionary)
        {
            currentPerson = p.Value as Person;
            newPerson = new Person();
            newPerson.Name = currentPerson.Name;
            newPerson.Age = currentPerson.Age;
            clonedPeople.Add(newPerson);
        }
        return clonedPeople;
    }

    ...
}
```

在 Person 类上实现 ICloneable 接口，可以简化这段代码。

习题 5

```
public IEnumerable Ages
{
    get
    {
        foreach (object person in Dictionary.Values)
            yield return (person as Person).Age;
    }
}
```

第 12 章

习题 1

a、b 和 e 是

c 和 d 否，但它们可以使用由包含它们的类提供的泛型类型参数。

f 否。

习题 2

```
public static double? operator *(Vector op1, Vector op2)
{
    try
    {
        double angleDiff = (double)(op2.ThetaRadians.Value -
            op1.ThetaRadians.Value);
        return op1.R.Value * op2.R.Value * Math.Cos(angleDiff);
    }
    catch
    {
```

```
            return null;
        }
    }
```

习题 3

不在T上强制new()约束，就不能实例化T。在T上强制new()约束可以确保有一个公共的默认构造函数是可用的。

```
public class Instantiator<T>
    where T : new()
{
    public T instance;

    public Instantiator()
    {
        instance = new T();
    }
}
```

习题 4

同一个泛型类型参数T既用于泛型类，又用于泛型方法。需要重命名其中的一个或两个。例如：

```
public class StringGetter<U>
{
    public string GetString<T>(T item)
    {
        return item.ToString();
    }
}
```

习题 5

一种方式如下：

```
public class ShortCollection<T> : IList<T>
{
    protected Collection<T> innerCollection;
    protected int maxSize = 10;

    public ShortCollection() : this(10)
    {
    }

    public ShortCollection(int size)
    {
        maxSize = size;
        innerCollection = new Collection<T>();
    }

    public ShortCollection(List<T> list) : this(10, list)
    {
    }
```

```
public ShortCollection(int size, List<T> list)
{
    maxSize = size;
    if (list.Count <= maxSize)
    {
        innerCollection = new Collection<T>(list);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

protected void ThrowTooManyItemsException()
{
    throw new IndexOutOfRangeException(
        "Unable to add any more items, maximum size is " + maxSize.ToString()
        + " items.");
}
#region IList<T> Members

public int IndexOf(T item)
{
    return (innerCollection as IList<T>).IndexOf(item);
}

public void Insert(int index, T item)
{
    if (Count < maxSize)
    {
        (innerCollection as IList<T>).Insert(index, item);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

public void RemoveAt(int index)
{
    (innerCollection as IList<T>).RemoveAt(index);
}

public T this[int index]
{
    get
    {
        return (innerCollection as IList<T>)[index];
    }

    set
    {
        (innerCollection as IList<T>)[index] = value;
    }
}
```

```
#endregion

#region ICollection<T> Members

public void Add(T item)
{
    if (Count < maxSize)
    {
        (innerCollection as ICollection<T>).Add(item);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

public void Clear()
{
    (innerCollection as ICollection<T>).Clear();
}

public bool Contains(T item)
{
    return (innerCollection as ICollection<T>).Contains(item);
}

public void CopyTo(T[] array, int arrayIndex)
{
    (innerCollection as ICollection<T>).CopyTo(array, arrayIndex);
}

public int Count
{
    get
    {
        return (innerCollection as ICollection<T>).Count;
    }
}

public bool IsReadOnly
{
    get
    {
        return (innerCollection as ICollection<T>).IsReadOnly;
    }
}

public bool Remove(T item)
{
    return (innerCollection as ICollection<T>).Remove(item);
}

#endregion
```

```
        #region IEnumerable<T> Members

        public IEnumerator<T> GetEnumerator()
        {
            return (innerCollection as IEnumerable<T>).GetEnumerator();
        }

        #endregion
    }
```

习题 6

不。类型参数 T 定义为协变。但协变参数类型只能用作方法的返回值，不能用作方法变元。否则就会得到如下编译错误(假定使用名称空间 VarianceDemo)：

```
Invalid variance: The type parameter 'T' must be contravariantly valid on
'VarianceDemo.IMethaneProducer<T>.BelchAt(T)'. 'T' is covariant.
```

第 13 章

习题 1

```
public void ProcessEvent(object source, EventArgs e)
{
    if (e is MessageArrivedEventArgs)
    {
        Console.WriteLine("Connection.MessageArrived event received.");
        Console.WriteLine("Message: {0}",
                          (e as MessageArrivedEventArgs).Message);
    }
    if (e is ElapsedEventArgs)
    {
        Console.WriteLine("Timer.Elapsed event received.");
        Console.WriteLine("SignalTime: {0}",
                          (e as ElapsedEventArgs ).SignalTime);
    }
}

public void ProcessElapsedEvent(object source, ElapsedEventArgs e)
{
    ProcessEvent(source, e);
}
```

注意需要这个额外的 ProcessElapsedEvent()方法，因为 ElapsedEventHandler 委托不能转换为 EventHandler 委托。MessageHandler 委托不需要进行这个转换，因为它的语法与 EventHandler 相同：

```
public delegate void MessageHandler(object source, EventArgs e);
```

习题 2

修改 Player.cs，如下所示(修改了一个方法，添加了两个新方法——代码中的注释说明了这些变化)：

```
public bool HasWon()
{
    // get temporary copy of hand, which may get modified.
    Cards tempHand = (Cards)hand.Clone();

    // find three and four of a kind sets
    bool fourOfAKind = false;
    bool threeOfAKind = false;
    int fourRank = -1;
    int threeRank = -1;

    int cardsOfRank;
    for (int matchRank = 0; matchRank < 13; matchRank++)
    {
        cardsOfRank = 0;
        foreach (Card c in tempHand)
        {
            if (c.rank == (Rank)matchRank)
            {
                cardsOfRank++;
            }
        }
        if (cardsOfRank == 4)
        {
            // mark set of four
            fourRank = matchRank;
            fourOfAKind = true;
        }
        if (cardsOfRank == 3)
        {
            // two threes means no win possible
            // (threeOfAKind will only be true if this code
            // has already executed)
            if (threeOfAKind == true)
            {
                return false;
            }
            // mark set of three
            threeRank = matchRank;
            threeOfAKind = true;
    }
}

// check simple win condition
if (threeOfAKind && fourOfAKind)
{
    return true;
}

// simplify hand if three or four of a kind is found, by removing used cards
if (fourOfAKind || threeOfAKind)
{
    for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
    {
        if ((tempHand[cardIndex].rank == (Rank)fourRank)
```

```
            || (tempHand[cardIndex].rank == (Rank)threeRank))
        {
            tempHand.RemoveAt(cardIndex);
        }
    }
}

// at this point the method may have returned, because:
// - a set of four and a set of three has been found, winning.
// - two sets of three have been found, losing.
// if the method hasn't returned then either:
// - no sets have been found, and tempHand contains 7 cards.
// - a set of three has been found, and tempHand contains 4 cards.
// - a set of four has been found, and tempHand contains 3 cards.

// find run of four sets, start by looking for cards of same suit in the same
// way as before
bool fourOfASuit = false;
bool threeOfASuit = false;
int fourSuit = -1;
int threeSuit = -1;

int cardsOfSuit;
for (int matchSuit = 0; matchSuit < 4; matchSuit++)
{
    cardsOfSuit = 0;
    foreach (Card c in tempHand)
    {
        if (c.suit == (Suit)matchSuit)
        {
            cardsOfSuit++;
        }
    }
    if (cardsOfSuit == 7)
    {
        // if all cards are the same suit then two runs
        // are possible, but not definite.
        threeOfASuit = true;
        threeSuit = matchSuit;
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
    if (cardsOfSuit == 4)
    {
        // mark four card suit.
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
    if (cardsOfSuit == 3)
    {
        // mark three card suit.
        threeOfASuit = true;
        threeSuit = matchSuit;
    }
}
```

```
if (!(threeOfASuit || fourOfASuit))
{
    // need at least one run possibility to continue.
    return false;
}

if (tempHand.Count == 7)
{
    if (!(threeOfASuit && fourOfASuit))
    {
        // need a three and a four card suit.
        return false;
    }

    // create two temporary sets for checking.
    Cards set1 = new Cards();
    Cards set2 = new Cards();

    // if all 7 cards are the same suit...
    if (threeSuit == fourSuit)
    {
        // get min and max cards
        int maxVal, minVal;
        GetLimits(tempHand, out maxVal, out minVal);
        for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
        {
            if (((int)tempHand[cardIndex].rank < (minVal + 3))
                || ((int)tempHand[cardIndex].rank > (maxVal - 3)))
            {
                // remove all cards in a three card set that
                // starts at minVal or ends at maxVal.
                tempHand.RemoveAt(cardIndex);
            }
        }
        if (tempHand.Count != 1)
        {
            // if more then one card is left then there aren't two runs.
            return false;
        }
        if ((tempHand[0].rank == (Rank)(minVal + 3))
            || (tempHand[0].rank == (Rank)(maxVal - 3)))
        {
            // if spare card can make one of the three card sets into a
            // four card set then there are two sets.
            return true;
        }
        else
        {
            // if spare card doesn't fit then there are two sets of three
            // cards but no set of four cards.
            return false;
        }
    }
}
```

```
        // if three card and four card suits are different...
        foreach (Card card in tempHand)
        {
            // split cards into sets.
            if (card.suit == (Suit)threeSuit)
            {
                set1.Add(card);
            }
            else
            {
                set2.Add(card);
            }
        }

        // check if sets are sequential.
        if (isSequential(set1) && isSequential(set2))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // if four cards remain (three of a kind found)
    if (tempHand.Count == 4)
    {
        // if four cards remain then they must be the same suit.
        if (!fourOfASuit)
        {
            return false;
        }
        // won if cards are sequential.
        if (isSequential(tempHand))
        {
            return true;
        }
    }

    // if three cards remain (four of a kind found)
    if (tempHand.Count == 3)
    {
        // if three cards remain then they must be the same suit.
        if (!threeOfASuit)
        {
            return false;
        }
        // won if cards are sequential.
        if (isSequential(tempHand))
        {
            return true;
        }
    }
```

```
    // return false if two valid sets don't exist.
    return false;
}

// utility method to get max and min ranks of cards
// (same suit assumed)
private void GetLimits(Cards cards, out int maxVal, out int minVal)
{
    maxVal = 0;
    minVal = 14;
    foreach (Card card in cards)
    {
        if ((int)card.rank > maxVal)
        {
            maxVal = (int)card.rank;
        }
        if ((int)card.rank < minVal)
        {
            minVal = (int)card.rank;
        }
    }
}

// utility method to see if cards are in a run
// (same suit assumed)
private bool isSequential(Cards cards)
{
    int maxVal, minVal;
    GetLimits(cards, out maxVal, out minVal);
    if ((maxVal - minVal) == (cards.Count - 1))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

第 14 章

习题 1

为了结合使用对象初始化器和类，必须包含一个默认的无参数构造函数。可以给这个类添加一个，或者删除已有的非默认构造函数。此后，就可以使用如下代码，在一步中实例化和初始化这个类：

```
Giraffe myPetGiraffe = new Giraffe
{
    NeckLength = "3.14",
    Name = "Gerald"
};
```

习题 2

错误。使用 var 关键字声明变量时，该变量仍是强类型化的，编译器会确定变量的类型。

习题 3

可以使用已实现的 Equals()方法。注意不能使用==运算符来执行这个操作，因为==运算符会比较变量，确定它们是否引用同一个对象。

习题 4

扩展方法必须是静态的：

```
public static string ToAcronym(this string inputString)
```

习题 5

必须在静态类中包含扩展方法，它可以在包含客户代码的名称空间中访问。为此，可以在同一个名称空间中包含代码，或者导入包含该类的名称空间。

习题 6

一种方式如下：

```
public static string ToAcronym(this string inputString)
{
    return inputString.Trim().Split(' ')
        .Aggregate<string, string>("",
            (a, b) => a + (b.Length > 0 ?
                b.ToUpper()[0].ToString() : ""));
}
```

其中使用了三元运算符以防多个空格引发错误。还要注意需要带两个泛型类型参数的 Aggregate()版本，因为需要一个种子值。

第 15 章

习题 1

Windows 窗体项目中的 Program.cs 文件包含应用程序的 Main()方法。默认情况下，这个方法如下所示：

```
[STAThread]
static void Main()
{
  Application.EnableVisualStyles();
  Application.SetCompatibleTextRenderingDefault(false);
  Application.Run(new Form1());
}
```

代码行

```
Application.EnableVisualStyles();
```

控制 Windows 窗体的可见样式。

请注意这行代码在 Windows 2000 上不起作用。

习题 2

TabControl 包含 SelectedIndexChanged 事件，在用户移动到另一个选项卡页时，该事件可用于执行代码。

(1) 在 Windows 窗体设计器上，选择 TabControl，给它添加两个选项卡。

(2) 把选项卡命名为 Tab Three 和 Tab Four。

(3) 选择 TabControl，添加 SelectedIndexChanged 事件，进入代码窗口。

(4) 输入下述代码：

```
private void tabControl1_SelectedIndexChanged(object sender,
EventArgs e)
{
  string message = "You changed the current tab to `" +
  tabControl1.SelectedTab.Text + "'  from  `" +
  tabControl1.TabPages[mCurrentTabIndex].Text + "' ";
  mCurrentTabIndex = tabControl1.SelectedIndex;
  MessageBox.Show(message);
}
```

(5) 在类的顶部添加私有字段 mCurrentTabIndex，如下所示：

```
partial class Form1 : Form
{
private int mCurrentTabIndex = 0;
```

(6) 运行应用程序。

默认显示 TabControl 中的第一个选项卡页，其索引为 0。把私有字段 mCurrentTabIndex 设置为 0，就可以利用这个默认操作。在 SelectedIndexChanged 方法中，建立要显示的消息。为此，需要使用 SelectedTab 属性，获取刚才选中的选项卡页的 Text 属性，使用 TabPages 集合获取字段 mCurrentTabIndex 指定的选项卡页的属性。建立了消息后，就把 mCurrentTabIndex 字段改为指向新选中的选项卡页。

习题 3

创建一个派生自 ListViewItem 的类，就可以使用它替代“所需要的”ListViewItem 类。这意味着，尽管 ListView 本身不知道类的额外信息，但可以把额外信息直接存储在 ListView 显示的项中。

(1) 创建一个新类 FQListViewItem：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
```

```
namespace ListView
{
  class FQListViewItem : ListViewItem
  {
    private string mFullyQualifiedPath;
    public string FullyQualifiedPath
    {
      get { return mFullyQualifiedPath; }
      set { mFullyQualifiedPath = value; }
    }
  }
}
```

(2) 在 Form.cs 文件中找到 ListViewItem 类型，把它改为 FQListViewItem 类型。

(3) 找到对.Tag 的所有引用，将其改为对.FullyQualifiedPath 的引用。在方法 listViewFilesAndFolders_ItemActivate 中，把第二行中的所选项转换为 FQListViewItem 项，如下所示：

```
string filename =
  ((FQListViewItem)lw.SelectedItems[0]).FullyQualifiedPath;
```

第 16 章

习题 1

为此，应创建一个新属性和两个事件。首先创建属性(private int maxLength = 32767)：

```
public int MaxLength
{
  get { return maxLength; }
  set
  {
    if (value >= 0 && value <= 32767)
    {
      maxLength = value;
      if (MaxLengthChanged != null)
        MaxLengthChanged(this, new EventArgs());
      textBoxText.MaxLength = value;
    }
  }
}
```

接着创建两个新事件：

```
public event System.EventHandler MaxLengthChanged;
public event System.EventHandler MaxLengthReached;
```

在窗体设计器中，选中文本框，给 TextChanged 事件添加事件处理程序，代码如下：

```
private void txtLabelText_TextChanged(object sender, EventArgs e)
{
  if (textBoxText.Text.Length >= maxLength)
  {
    if (MaxLengthReached != null)
```

```
        MaxLengthReached(this, new EventArgs());
      }
    }
```

在普通的文本框中，文本的最大长度是 System.Int32 类型的大小，但默认为 32 767 个字符，这一般足以满足要求。在上述第二步的属性中，检查值是否为负，或者是否超过 32 767，如果是，就忽略更改请求。如果值是可接受的，就设置文本框的 MaxLength 属性，引发 MaxLengthChanged 事件。

txtLabelText_TextChanged 事件处理程序检查文本框中的最大字符数是否等于或超过 maxLength 中指定的数字，如果是，就引发 MaxLengthReached 事件。

习题 2

首先选中状态栏上的 3 个字段，把 Bold 的值改为 false (展开 Font 属性，才能进行修改)。把 3 个字段的 Enabled 属性都改为 True，再双击 Bold 字段，输入如下代码：

```
private void toolStripStatusLabelBold_Click(object sender, EventArgs e)
{
  boldToolStripButton.Checked = !boldToolStripButton.Checked;
}
```

双击 Italic 字段，输入如下文本：

```
private void toolStripStatusLabelItalic_Click(object sender, EventArgs e)
{
  italicToolStripButton.Checked = !italicToolStripButton.Checked;
}
```

双击 Underline 字段，输入如下文本：

```
private void toolStripStatusLabelUnderline_Click(object sender, EventArgs e)
{
  underlineToolStripButton.Checked = !underlineToolStripButton.Checked;
}
```

3 个单击事件处理程序都切换工具栏按钮的 Checked 属性。这会引发 CheckedChanged 事件。这些事件处理程序都负责完成所有的工作，所以它们都需要修改，使状态文本也发生变化：

```
      private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)
  {
    Font oldFont, newFont;

    bool checkState = ((ToolStripButton)sender).Checked;
    oldFont = this.richTextBoxText.SelectionFont;

    if (!checkState)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    richTextBoxText.SelectionFont = newFont;
    richTextBoxText.Focus();
```

```
    boldToolStripMenuItem.CheckedChanged -= new
EventHandler(boldToolStripMenuItem_CheckedChanged);
    boldToolStripMenuItem.Checked = checkState;
    boldToolStripMenuItem.CheckedChanged += new
EventHandler(boldToolStripMenuItem_CheckedChanged);
        //StatusBar
        if (!checkState)
            toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
              toolStripStatusLabelBold.Font.Style & ~FontStyle.Bold);
        else
            toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
              toolStripStatusLabelBold.Font.Style | FontStyle.Bold);
    }

    private void italicToolStripButton_CheckedChanged(object sender, EventArgs e)
    {
      Font oldFont, newFont;

      bool checkState = ((ToolStripButton)sender).Checked;
      oldFont = this.richTextBoxText.SelectionFont;

      if (!checkState)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
      else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

      richTextBoxText.SelectionFont = newFont;
      richTextBoxText.Focus();

      italicToolStripMenuItem.CheckedChanged -= new
EventHandler(italicToolStripMenuItem_CheckedChanged);
      italicToolStripMenuItem.Checked = checkState;
      italicToolStripMenuItem.CheckedChanged += new
EventHandler(italicToolStripMenuItem_CheckedChanged);
//StatusBar
        if (!checkState)
            toolStripStatusLabelItalic.Font = new
                Font (toolStripStatusLabelItalic. Font,
              toolStripStatusLabelItalic.Font.Style & ~FontStyle.Italic);
        else
            toolStripStatusLabelItalic.Font = new
                Font (toolStripStatusLabelItalic. Font,
              toolStripStatusLabelItalic.Font.Style | FontStyle.Italic);

    }

   private void UnderlineToolStripButton_CheckedChanged(object sender, EventArgs e)
   {
     Font oldFont, newFont;

     bool checkState = ((ToolStripButton)sender).Checked;
     oldFont = this.richTextBoxText.SelectionFont;

     if (!checkState)
       newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
```

```
        else
          newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        underlineToolStripMenuItem.CheckedChanged -= new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
        underlineToolStripMenuItem.Checked = checkState;
        underlineToolStripMenuItem.CheckedChanged += new
EventHandler(underlineToolStripMenuItem_CheckedChanged);

//StatusBar
          if (!checkState)
            toolStripStatusLabelUnderline.Font = new
               Font(toolStripStatusLabelUnderline .Font,
            toolStripStatusLabelItalic.Font.Style & ~FontStyle.Underline);
          else
            toolStripStatusLabelUnderline.Font = new
               Font(toolStripStatusLabelUnderline.Font,
            toolStripStatusLabelItalic.Font.
              Style | FontStyle.Underline);
    }
```

当选中工具栏按钮时，事件处理程序现在把 StatusStrip 面板的字体改成 Bold、Italic 或 Underline。当未选中这些按钮时，面板的字体就是正常字体。

第 17 章

习题 1

ClickOnce 部署的优点是安装应用程序的用户不需要管理权限。单击一个超链接，就可以自动安装应用程序。另外，还可以配置应用程序的新版本也自动安装。

习题 2

应用程序清单描述了应用程序和需要的权限，部署清单描述了部署配置(例如更新策略)。

习题 3

如果安装程序需要管理权限，就需要 Windows Installer，而不是 ClickOnce 部署。

习题 4

文件系统编辑器、注册编辑器、文件类型编辑器、用户界面编辑器、定制动作编辑器和启动条件编辑器。

第 18 章

习题 1

LoginView 控件可以添加到母版页上，使这个信息可用于每个内容页面。下面的代码段表示，用户登录时，会在 LoginView 中显示 LoggedInTemplate。LoggedInTemplate 包含一个 Label 和一个 LinkButton。id 为 InfoLabel 的 Label 控件用来显示用户信息。

```
<asp:LoginView ID="LoginView1" runat="server">
    <LoggedInTemplate>
        <asp:Label ID="InfoLabel" runat="server" Text="Hello, User">
        </asp:Label><br />
        <asp:LinkButton ID="LinkButton1" runat="server"
                        OnClick="OnLogout">Logout</asp:LinkButton>
    </LoggedInTemplate>
</asp:LoginView>
```

标签在 Page_Load 事件处理程序的代码隐藏内容中填充。用户名可以通过 Context 属性访问，User.Identity.Name 返回用户名。

```
protected void Page_Load(object sender, EventArgs e)
{
    Control infoLabel = this.LoginView1.FindControl("InfoLabel");
    if (infoLabel != null)
        (infoLabel as Label).Text = "Welcome, " + Context.User.Identity.Name;
}
```

习题 2

DropDownlist 的上一个用法是提供一组定义好的项，供用户选择。现在使用连接到 Events 数据库的 SqlDataSource 来替代：

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$
      ConnectionStrings:BegVCSharpEventsConnectionString %>"
    SelectCommand="SELECT [Id], [Title], [Date] FROM [Events]
    ORDER BY [Date]">
</asp:SqlDataSource>
```

设置了 DropDownList 控件，使 DataSourceId 引用 SqlDataSource 后，使 DataTextField 引用 SQL 选择语句中的 Title，以显示事件的标题：

```
<asp:DropDownList ID="dropDownListEvents" runat="server"
    DataSourceID="SqlDataSource1" DataTextField="Title"
      DataValueField="Id">
</asp:DropDownList>
```

习题 3

通过 File | New Project | ASP.NET Web Application 菜单创建一个项目，就可以创建一个预先建立了许多项的项目。其中有一个母版页 Site.Master，这个母版页使用了 Styles 文件夹中的一个样式表

Site.css。在母版页中，给站点导航使用了一个 Menu 控件。文件 Default.aspx 和 About.aspx 使用了母版页，所以可以导航到这两个文件。

Account 子文件夹中包含几个文件，它们使用身份验证特性，如 Login.aspx、Register.aspx 和 ChangePassword.aspx。

可以将这个项目作为基础，根据需要添加自己的页面和功能。

第 19 章

习题 1

选择 File | New | Project，再选择 ASP.NET Empty Web Application 模板，创建一个新的 Web 服务，命名为 CinemaReservation。选择 Project | Add New Item . . .，再选择 Web Service 模板，添加一个新的 Web 服务，命名为 CinemaReservation.asmx。

习题 2

类的代码如下：

```
public class ReserveSeatRequest
{
    public string Name { get; set; }
    public int Row { get; set; }
    public int Seat { get; set; }
}
```

```
public class ReserveSeatResponse
{
    public string ReservationName { get; set; }
    public int Row { get; set; }
    public int Seat { get; set; }
}
```

习题 3

对于所有座位，应声明一个 reservedSeats 数组，以便记住预订的座位：

```
private const int maxRows = 12;
private const int maxSeats = 16;
private bool[,] reservedSeats = new bool[maxRows, maxSeats];
```

Web 服务方法的实现代码如下。如果所请求的座位是空闲的，就保留这个座位，并从 Web 服务中返回它。如果该座位不是空闲的，就返回下一个空闲的座位。

```
 [WebMethod]
public ReserveSeatResponse ReserveSeat(ReserveSeatRequest req)
{
    ReserveSeatResponse resp = new ReserveSeatResponse();
    resp.ReservationName = req.Name;
    object o = HttpContext.Current.Cache["Cinema"];
```

```
        if (o == null)
        {
            // fill seats with data from the database or a file...
            HttpContext.Current.Cache["Cinema"] = reservedSeats;
        }
        else
        {
            reservedSeats = (bool[,])o;
        }
        if (reservedSeats[req.Row, req.Seat] == false)
        {
            reservedSeats[req.Row, req.Seat] = true;
            resp.Row = req.Row;
            resp.Seat = req.Seat;
        }
        else
        {
            int row;
            int seat;
            GetNextFreeSeat(out row, out seat);
            resp.Row = row;
            resp.Seat = seat;
        }
        return resp;
    }
```

习题 4

创建一个新的 Windows 应用程序，添加一个对 Web 服务的服务引用。对 Web 服务的调用如下：

```
private void OnRequestSeat(object sender, EventArgs e)
{
        CinemaService.ReserveSeatRequest req =
            new CinemaService.ReserveSeatRequest();
        req.Name = textName.Text;
        req.Seat = int.Parse(textSeat.Text);
        req.Row = int.Parse(textRow.Text);

        CinemaService.CinemaReservationSoapClient ws =
            new CinemaService.CinemaReservationSoapClient();
        CinemaService.ReserveSeatResponse resp =
        ws.ReserveSeat(req);
        MessageBox.Show(String.Format("Reserved seat {0} {1}",
                resp.Row, resp.Seat));
}
```

第 20 章

习题 1

复制网站，会复制运行 Web 应用程序所需的所有文件。Visual Studio 2010 有一个对话框可用于双向复制。目标服务器中的较新文件可以在本地复制。如果源代码不应复制到目标 Web 服务器上，

发布时允许创建程序集，再仅把程序集复制到目标 Web 服务器上。

习题 2

复制站点前，需要先在目标服务器上创建虚拟目录。使用安装程序可以在安装过程中在 IIS 内部创建虚拟目录。

习题 3

选项有发布到文件系统上，使用 FrontPage Server Extensions 发布到服务器上，通过 FTP 发布，以及使用 1-Click 发布功能发布。这主要取决于当前使用的主机选项以及提供程序所提供的功能。无论如何，都必须在服务器上创建虚拟目录。发布到文件系统上时，需要能访问文件系统，如果自己运行 IIS，就应是这种情况。用 FrontPage Server Extensions 发布时，必须将这些扩展安装在服务器上。通过 FTP 发布时，必须将 FTP 服务器安装在服务器上。通过 1-Click 发布时，提供程序必须支持这个新的发布选项。

习题 4

首先使用 IIS 管理工具创建一个 Web 应用程序，再使用 Visual Studio 把 Web 服务文件复制到服务器上。

第 21 章

习题 1

System.IO

习题 2

需要随机访问文件时，或者不处理字符串数据时，就使用 FileStream 对象写入文件。

习题 3

- Peek(): 获取文件中下一个字符的值，但不前移到下一个文件位置上
- Read():获取文件中下一个字符的值，并前移到下一个文件位置上
- Read(char[] buffer, int index, int count): 从 buffer[index]开始，把 count 个字符读入 buffer
- ReadLine(): 获取一行文本
- ReadToEnd(): 获取文件中的所有文本

习题 4

DeflateStream

习题 5

确保它不拥有 Serializable 特性。

习题 6

- Changed: 修改文件时发生
- Created: 创建文件时发生
- Deleted: 删除文件时发生
- Renamed: 重命名文件时发生

习题 7

添加一个按钮，切换 FileSystemWatcher.EnableRaisingEvents 属性的值。

第 22 章

习题 1

(1) 双击 Create Node 按钮，让事件处理程序执行操作。

(2) 在创建 XmlComment 后，插入如下 3 行代码：

```
    XmlAttribute newPages = document.CreateAttribute("pages");
    newPages.Value = "1000";
newBook.Attributes.Append(newPages);
```

习题 2

(1) //elements——返回文档中的所有节点。

(2) element——返回文档中的每个元素节点，但不返回元素根节点。

(3) element[@Type='Noble Gas'] ——返回其 Type 特性的值是 Noble Gas 的每个元素。

(4) //mass——返回名为 mass 的所有节点。

(5) //mass/..—— .. 使 XPath 从选中的节点向上移动一个位置，这表示，这个查询返回包含 mass 节点的所有节点。

(6) element/specification[mass='20.1797'] ——选择包含 mass 节点值为 20.1797 的 specification 元素。

(7) element/name[text()='Neon'] ——要选择包含测试内容的节点，可以使用 text()函数，它会选择文本为 Neon 的 name 节点。

习题 3

XML 可以是有效的、格式良好的，也可以是无效的。只要选择 XML 文档的一部分，就会得到整个 XML 文档的一部分。这意味着，所选择的 XML 很可能是无效的。大多数 XML 查看器会拒绝显示格式不正确的 XML，所以不能在标准 XML 查看器中直接显示许多查询的结果。

第 23 章

习题 1

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

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

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

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

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

习题 2

在小于 5 000 000 的集中，查找小于 1000 的数字：

```
static void Main(string[] args)
{
    int[] arraySizes = { 100, 1000, 10000, 100000,
                         1000000, 5000000, 10000000, 50000000 };

    foreach (int i in arraySizes) {
       int[] numbers = generateLotsOfNumbers(i);
       var queryResults = from n in numbers
                          where n < 1000
                          select n;
       Console.WriteLine("number array size = {0}: Count(n < 1000) = {1}",
               numbers.Length, queryResults.Count()
       );
    }

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

习题 3

对于 n < 1000，性能受到的影响并不明显。

```
static void Main(string[] args)
```

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

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

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

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

习题 4

对于非常大的子集，例如 n > 1000，而不是 n < 1000，会非常慢：

```
static void Main(string[] args)
{

    int[] numbers = generateLotsOfNumbers(12345678);

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

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

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

习题 5

会输出所有名字，因为没有查询。

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

            var queryResults = names;
```

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

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

习题 6

```
    var queryResults =
        from c in customers
        where c.Country == "USA"
        select c
        ;
    Console.WriteLine("Customers in USA:");
    foreach (Customer c in queryResults)
    {
        Console.WriteLine(c);
    }
```

习题 7

```
        static void Main(string[] args)
        {
            string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
            // only Min() and Max() are available (if no lambda is used)
            // for a result set like this consisting only of strings
            Console.WriteLine("Min(names) = " + names.Min());
            Console.WriteLine("Max(names) = " + names.Max());
                var queryResults =
                        from n in names
                        where n.StartsWith("S")
                        select n;
            Console.WriteLine("Query result: names starting with S");
              foreach (var item in queryResults)
              {
                        Console.WriteLine(item);
              }

            Console.WriteLine("Min(queryResults) = " + queryResults.Min());
            Console.WriteLine("Max(queryResults) = " + queryResults.Max());

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

第 24 章

习题 1

使用下面的代码：

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

namespace BegVCSharp_24_exercise1
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument xdoc = new XDocument(
                new XElement("employees",
                    new XElement("employee",
                        new XAttribute("ID", "1001"),
                        new XAttribute("FirstName", "Fred"),
                        new XAttribute("LastName", "Lancelot"),
                        new XElement("Skills",
                            new XElement("Language", "C#"),
                            new XElement("Math", "Calculus")
                            )
                        ),
                    new XElement("employee",
                        new XAttribute("ID", "2002"),
                        new XAttribute("FirstName", "Jerry"),
                        new XAttribute("LastName", "Garcia"),
                        new XElement("Skills",
                            new XElement("Language", "French"),
                            new XElement("Math", "Business")
                            )
                        )
                    )
                );
            Console.WriteLine(xdoc);

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

        }
    }
}
```

习题 2

使用如下代码：

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

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);


            Console.WriteLine("Oldest customers: Companies with orders in 1996:");
            var queryResults =
                from c in customers.Descendants("customer")
                where c.Descendants("order").Attributes("orderYear")
                                            .Any(a => a.Value == "1996")
                select c.Attribute("Company");

            foreach (var item in queryResults)
            {
                Console.WriteLine(item);
            }
            Console.Write("Press Enter/Return to continue:");
            Console.ReadLine();
        }
    }
}
```

习题 3

代码如下：

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

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                      @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);

            Console.WriteLine(
```

```
                "Companies with individual orders totaling over $10,000");
            var queryResults =
                from c in customers.Descendants("order")
                where Convert.ToDecimal(c.Attribute("orderTotal").Value) > 10000
                select new { OrderID = c.Attribute("orderID"),
                             Company = c.Parent.Attribute("Company") };

            foreach (var item in queryResults)
            {
                Console.WriteLine(item);
            }
            Console.Write("Program finished, press Enter/Return to continue:");
            Console.ReadLine();
        }
    }
}
```

习题 4

使用如下代码：

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

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);

            Console.WriteLine("Lifetime highest-selling customers: "+
                "Companies with all orders totaling over $100,000");
            var queryResult =
                from c in customers.Descendants("customer")
            where c.Descendants("order").Attributes("orderTotal")
                           .Sum(o => Convert.ToDecimal(o.Value)) > 100000
            select c.Attribute("Company");

        foreach (var item in queryResult)
        {
            Console.WriteLine(item);
        }
        Console.Write("Press Enter/Return to continue:");
        Console.ReadLine();
    }
}
```

习题 5

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

namespace BegVCSharp_24_exercise1
{
    class Program
    {
        static void Main(string[] args)
        {
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            Console.WriteLine("Product Details");
            var queryResults = from p in northWindEntities.Products
                               select new
                               {
                                   ID = p.ProductID,
                                   Name = p.ProductName,
                                   Price = p.UnitPrice,
                                   Discontinued = p.Discontinued
                               };
            foreach (var item in queryResults)
            {
               Console.WriteLine(item);
            }
            Console.WriteLine("Employee Details");
            var queryResults2 = from e in northWindDataContext.Employees
                               select new
                               {
                                   ID = e.EmployeeID,
                                   Name = e.FirstName+" "+e.LastName,
                                   Title = e.Title
                               };
            foreach (var item in queryResults2)
            {
               Console.WriteLine(item);
            }
            Console.WriteLine("Press Enter/Return to continue...");
            Console.ReadLine();

        }
    }
}
```

习题 6

使用如下代码：

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

```
namespace BegVCSharp_24_exercise6
{
    class Program
    {
        static void Main(string[] args)
        {
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            Console.WriteLine("Top-Selling Products (Sales over $50,000)");
            var queryResults =
              from p in northWindEntities.Products
              where p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) > 50000
              orderby p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) descending
              select new
              {
                  ID = p.ProductID,
                  Name = p.ProductName,
                  TotalSales = p.Order_Details.Sum(od => od.Quantity * od.UnitPrice)
              };
              foreach (var item in queryResults)
              {
                Console.WriteLine(item);
              }

              Console.WriteLine("Press Enter/Return to continue...");
              Console.ReadLine();
        }
    }
```

习题 7

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

namespace BegVCSharp_24_exercise7
{
    class Program
    {
        static void Main(string[] args)
        {


    NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            var totalResults = from od in northWindEntities.Order_Details
                               from c in northWindEntities.Customers
                               where c.CustomerID == od.Order.CustomerID
                               select new
                               {
                                   Product = od.Product.ProductName,
                                   Country = c.Country,
```

```
                                    Sales = od.UnitPrice * od.Quantity
                                };

            var groupResults =
                from c in totalResults
                group c by new { Product = c.Product, Country = c.Country } into cg
                select new {
                    Product = cg.Key.Product,
                    Country = cg.Key.Country,
                    TotalSales = cg.Sum(c => c.Sales)
                }
            ;

            var orderedResults =
                from cg in groupResults
                orderby cg.Country, cg.TotalSales descending
                select cg
            ;

            foreach (var item in orderedResults)
            {
                Console.WriteLine("{0,-12}{1,-20}{2,12}",
                  item.Country, item.Product, item.TotalSales.ToString("C2"));
            }
            Console.WriteLine("Press Enter/Return to continue...");
            Console.ReadLine();

        }
    }
```

第 25 章

习题 1

错误。大多数代码都相同，但有一些小区别，例如在 WPF 浏览器应用程序中必须使用 Page 控件，在 WPF 桌面应用程序中必须使用 Window 控件。

习题 2

应使用关联属性。在 XAML 中，使用特性语法和<ParentClassName>. <AttributeName>格式的完全限定的特性名来表示关联属性。下面的代码是一个示例;

```
<Tree>
<Branch Tree.LeafCount="3" />
<Branch Tree.LeafCount="42" />
</Tree>
```

习题 3

语句 b)和 e)是正确的。语句 a)错误，因为.NET 属性是可选的。c)错误，因为一个类可以包含的依赖属性个数是没有限制的。d)错误，因为这是一个最适合的命名约定，而不是强制的。

习题 4

应使用 StackPanel 控件。

习题 5

命名约定指定，通道事件的名称与所关联的冒泡事件使用的名称相同，但加上前缀 Preview。

习题 6

严格地说，这是一个技巧问题，因为可以连续改变任何属性类型。但如果要连续改变的属性类型不是 double、Color 或 Point，就必须创建自己的时间线类，所以一般最好使用这些类型。

习题 7

使用动态资源引用可以在运行期间改变资源引用，或者在运行之前不知道该引用是什么时，使用动态资源引用。

第 26 章

习题 1

上述应用程序都可以。

习题 2

数据合同，需要 DataContractAttribute 和 DataMemberAttribute 特性。

习题 3

使用.svc 扩展。

习题 4

这是一种方式，但把所有 WCF 配置放在一个独立配置文件中通常更简单，例如 web.config 或 app.config。

习题 5

```
[ServiceContract]
public interface IMusicPlayer
{
    [OperationContract(IsOneWay=true)]
    void Play();

    [OperationContract(IsOneWay=true)]
    void Stop();

    [OperationContract]
    TrackInformation GetCurrentTrackInformation();
}
```

还需要一个数据合同来封装跟踪信息，在上面的代码中就是 TrackInformation。

第 27 章

习题 1

复合活动包含两部分——活动本身和设计器(XAML)文件，设计器文件定义了活动在屏幕上的布局。复合活动一般派生于 NativeActivity 类，且包含一个子活动集合。例如，Sequence 活动的 Activities 属性就是一个子活动集合。

需要重写 Execute()方法，以便安排子活动运行——可能会随机选择活动，或者同时运行所有活动。给 Execute()方法传送 NativeActivityContext 类的一个实例，它可用于安排子活动的执行。

最后一步是创建一个设计器，以允许用户把活动拖放到您的活动上。这里应使用 XAML 定义活动的外观，通常最好参考内置的活动，看看它们是如何实现的，以重用一些 XAML 资源。如果下载 Reflector (http://reflector.red-gate.com)，就可以使用 BAML 查看器插件(联机搜索它)，对内置程序集使用的资源进行反编译，以便查看用于定义定制复合活动的 XAML。

习题 2

Workflow 4 有几个活动可以把工作流用作 WCF 服务。最简单的方式是在新建项目对话框的 WCF 项中选择 WCF Workflow Service Application 项目类型，接着就可以添加活动，处理入站的方法调用，根据需要把结果返回给调用者。

习题 3

工作流存在“持久保存”服务的概念，持久保存服务可以保存和重新加载工作流实例。工作流空闲时(即工作流等待某种形式的外部输入或延迟)，就可以进行保存。如果安装了持久保存提供程序，工作流就会保存到该提供程序中。可以使用 SQL Server 提供程序(参见 System.Activities.DurableInstancing 程序集中的 SqlWorkflowInstanceStore 类)。如果在 WorkflowInvoker 下运行工作流，就不能保存工作流——必须使用 WorkflowApplication 或 WorkflowServiceHost 类保存工作流。

[General Information]
书名=C#入门经典　第5版
作者=（美）KarliWatso,（美）ChristianNagel著
页数=887
出版社=北京市：清华大学出版社
出版日期=2010.12
SS号=12715714
DX号=000008019474
URL=http://book2.duxiu.com/bookDetail.jsp?dxNumber=000008019474&d=241CA16544711706638C3B20F373789C0

封面
书名
版权
前言
目录
第Ⅰ部分　C#语言
第1章　C#简介
1.1　.NET Framework的含义
1.1.1　.NET Framework的内容
1.1.2　使用.NET Framework编写应用程序
1.2　C#的含义
1.2.1　用C#能编写什么样的应用程序
1.2.2　本书中的C#
1.3　Visual Studio 2010
1.3.1　Visual Studio 2010 Express产品
1.3.2　解决方案
1.4　小结
1.5　本章要点
第2章　编写C#程序
2.1　开发环境
2.1.1　Visual Studio 2010
2.1.2　Visual C# 2010 Express Edition
2.2　控制台应用程序
2.2.1　Solution Explorer
2.2.2　Properties窗口
2.2.3　Error List窗口
2.3　Windows Forms应用程序
2.4　小结
2.5　本章要点
第3章　变量和表达式
3.1　C#的基本语法
3.2　C#控制台应用程序的基本结构
3.3　变量
3.3.1　简单类型
3.3.2　变量的命名
3.3.3　字面值
3.3.4　变量的声明和赋值
3.4　表达式
3.4.1　数学运算符
3.4.2　赋值运算符
3.4.3　运算符的优先级
3.4.4　名称空间
3.5　小结
3.6　练习
3.7　本章要点
第4章　流程控制
4.1　布尔逻辑
4.1.1　布尔赋值运算符
4.1.2　按位运算符
4.1.3　运算符优先级的更新
4.2　goto语句
4.3　分支
4.3.1　三元运算符
4.3.2　if语句
4.3.3　switch语句
4.4　循环

4．4．1　do循环
4．4．2　while循环
4．4．3　for循环
4．4．4　循环的中断
4．4．5　无限循环
4．5　小结
4．6　练习
4．7　本章要点
第5章　变量的更多内容
5．1　类型转换
5．1．1　隐式转换
5．1．2　显式转换
5．1．3　使用Convert命令进行显式转换
5．2　复杂的变量类型
5．2．1　枚举
5．2．2　结构
5．2．3　数组
5．3　字符串的处理
5．4　小结
5．5　练习
5．6　本章要点
第6章　函数
6．1　定义和使用函数
6．1．1　返回值
6．1．2　参数
6．2　变量的作用域
6．2．1　其他结构中变量的作用域
6．2．2　参数和返回值与全局数据
6．3　Main（）函数
6．4　结构函数
6．5　函数的重载
6．6　委托
6．7　小结
6．8　练习
6．9　本章要点
第7章　调试和错误处理
7．1　VS和VCE中的调试
7．1．1　非中断（正常）模式下的调试
7．1．2　中断模式下的调试
7．2　错误处理
7．2．1　try...catch...finally
7．2．2　列出和配置异常
7．2．3　异常处理的注意事项
7．3　小结
7．4　练习
7．5　本章要点
第8章　面向对象编程简介
8．1　面向对象编程的含义
8．1．1　对象的含义
8．1．2　一切皆对象
8．1．3　对象的生命周期
8．1．4　静态和实例类成员
8．2　OOP技术
8．2．1　接口
8．2．2　继承
8．2．3　多态性

8．2．4　对象之间的关系
8．2．5　运算符重载
8．2．6　事件
8．2．7　引用类型和值类型
8．3　Ｗｉｎｄｏｗｓ应用程序中的ＯＯＰ
8．4　小结
8．5　练习
8．6　本章要点
第9章　定义类
9．1　Ｃ#中的类定义
9．2　Ｓｙｓｔｅｍ．Ｏｂｊｅｃｔ
9．3　构造函数和析构函数
9．4　ＶＳ和ＶＣＥ中的ＯＯＰ工具
9．4．1　Ｃｌａｓｓ　Ｖｉｅｗ窗口
9．4．2　对象浏览器
9．4．3　添加类
9．4．4　类图
9．5　类库项目
9．6　接口和抽象类
9．7　结构类型
9．8　浅度和深度复制
9．9　小结
9．10　练习
9．11　本章要点
第１０章　定义类成员
１０．１　成员定义
１０．１．１　定义字段
１０．１．２　定义方法
１０．１．３　定义属性
１０．１．４　在类图中添加成员
１０．１．５　重构成员
１０．１．６　自动属性
１０．２　类成员的其他议题
１０．２．１　隐藏基类方法
１０．２．２　调用重写或隐藏的基类方法
１０．２．３　嵌套的类型定义
１０．３　接口的实现
１０．４　部分类定义
１０．５　部分方法定义
１０．６　示例应用程序
１０．６．１　规划应用程序
１０．６．２　编写类库
１０．６．３　类库的客户应用程序
１０．７　Ｃａｌｌ　Ｈｉｅｒａｒｃｈｙ窗口
１０．８　小结
１０．９　练习
１０．１０　本章要点
第１１章　集合、比较和转换
１１．１　集合
１１．１．１　使用集合
１１．１．２　定义集合
１１．１．３　索引符
１１．１．４　给ＣａｒｄＬｉｂ添加Ｃａｒｄｓ集合
１１．１．５　关键字值集合和ＩＤｉｃｔｉｏｎａｒｙ
１１．１．６　迭代器
１１．１．７　深复制

11.1.8 给CardLib添加深复制
11.2 比较
11.2.1 类型比较
11.2.2 值比较
11.3 转换
11.3.1 重载转换运算符
11.3.2 as运算符
11.4 小结
11.5 练习
11.6 本章要点
第12章 泛型
12.1 泛型的概念
12.2 使用泛型
12.2.1 可空类型
12.2.2 System.Collections.Generic名称空间
12.3 定义泛型类型
12.3.1 定义泛型类
12.3.2 定义泛型接口
12.3.3 定义泛型方法
12.3.4 定义泛型委托
12.4 变体
12.4.1 协变
12.4.2 抗变
12.5 小结
12.6 练习
12.7 本章要点
第13章 其他OOP技术
13.1 运算符和全局名称空间限定符
13.2 定制异常
13.3 事件
13.3.1 事件的含义
13.3.2 处理事件
13.3.3 定义事件
13.4 扩展和使用CardLib
13.5 小结
13.6 练习
13.7 本章要点
第14章 C#语言的改进
14.1 初始化器
14.1.1 对象初始化器
14.1.2 集合初始化器
14.2 类型推理
14.3 匿名类型
14.4 动态查找
14.4.1 dynamic类型
14.4.2 IdynamicMetaObject-Provider
14.5 高级方法参数
14.5.1 可选参数
14.5.2 命名参数
14.5.3 命名参数和可选参数的规则
14.6 扩展方法
14.7 Lambda表达式
14.7.1 复习匿名方法
14.7.2 把Lambda表达式用于匿名方法
14.7.3 Lambda表达式的参数
14.7.4 Lambda表达式的语句体

14.7.5 Lambda表达式用作委托和表达式树
14.7.6 Lambda表达式和集合
14.8 小结
14.9 练习
14.10 本章要点
第Ⅱ部分 Windows编程
第15章 Windows编程基础
15.1 控件
15.1.1 属性
15.1.2 控件的定位、停靠和对齐
15.1.3 Anchor和Dock属性
15.1.4 事件
15.2 Button控件
15.2.1 Button控件的属性
15.2.2 Button控件的事件
15.2.3 添加事件处理程序
15.3 Label和LinkLabel控件
15.4 TextBox控件
15.4.1 TextBox控件的属性
15.4.2 TextBox控件的事件
15.4.3 添加事件处理程序
15.5 RadioButton和CheckBox控件
15.5.1 RadioButton控件的属性
15.5.2 RadioButton控件的事件
15.5.3 CheckBox控件的属性
15.5.4 CheckBox控件的事件
15.5.5 GroupBox控件
15.6 RichTextBox控件
15.6.1 RichTextBox控件的属性
15.6.2 RichTextBox控件的事件
15.7 ListBox和CheckedListBox控件
15.7.1 ListBox控件的属性
15.7.2 ListBox控件的方法
15.7.3 ListBox控件的事件
15.8 ListView控件
15.8.1 ListView控件的属性
15.8.2 ListView控件的方法
15.8.3 ListView控件的事件
15.8.4 ListViewItem
15.8.5 ColumnHeader
15.8.6 ImageList控件
15.9 TabControl控件
15.9.1 TabControl控件的属性
15.9.2 使用TabControl控件
15.10 小结
15.11 练习
15.12 本章要点
第16章 Windows窗体的高级功能
16.1 菜单和工具栏
16.1.1 两个实质一样的控件
16.1.2 使用MenuStrip控件
16.1.3 手工创建菜单
16.1.4 ToolStripMenuItem控件的其他属性
16.1.5 给菜单添加功能
16.2 工具栏
16.2.1 ToolStrip控件的属性

16.2.2 ToolStrip的项
16.2.3 StatusStrip控件
16.2.4 StatusStripStatusLabel的属性
16.3 SDI和MDI应用程序
16.4 生成MDI应用程序
16.5 创建控件
16.5.1 调试用户控件
16.5.2 扩展LabelTextbox控件
16.6 小结
16.7 练习
16.8 本章要点
第17章 部署Windows应用程序
17.1 部署概述
17.2 ClickOnce部署
17.2.1 创建ClickOnce部署
17.2.2 用ClickOnce安装应用程序
17.2.3 创建和使用应用程序的更新包
17.3 Visual Studio安装和部署项目类型
17.4 Microsoft Windows安装程序结构
17.4.1 Windows安装程序术语
17.4.2 Windows安装程序的优点
17.5 为MDI Editor创建安装软件包
17.5.1 规划安装内容
17.5.2 创建项目
17.5.3 项目属性
17.5.4 安装编辑器
17.5.5 File System编辑器
17.5.6 File Types编辑器
17.5.7 Launch Condition编辑器
17.5.8 User Interface编辑器
17.6 生成项目
17.7 安装
17.7.1 Welcome
17.7.2 Read Me
17.7.3 License Agreement
17.7.4 Optional Files
17.7.5 选择安装文件夹
17.7.6 确认安装
17.7.7 进度
17.7.8 完成安装
17.7.9 运行应用程序
17.7.10 卸载
17.8 小结
17.9 练习
17.10 本章要点
第III部分 Web编程
第18章 ASP.NET Web编程
18.1 Web应用程序概述
18.2 ASP.NET运行库
18.3 创建简单的Web页面
18.4 服务器控件
18.5 ASP.NET同送
18.6 ASP.NET AJAX回送
18.7 输入的有效性验证
18.8 状态管理
18.8.1 客户端的状态管理

18.8.2 服务器端的状态管理
18.9 样式
18.10 母版页
18.11 站点导航
18.12 身份验证和授权
18.12.1 身份验证的配置
18.12.2 使用安全控件
18.13 读写SQL Server数据库
18.14 小结
18.15 练习
18.16 本章要点
第19章 Web服务
19.1 使用Web服务的场合
19.1.1 宾馆旅行社代理应用程序
19.1.2 图书发布应用程序
19.1.3 客户应用程序的类型
19.2 应用程序的体系结构
19.3 Web服务的体系结构
19.3.1 调用方法和WSDL
19.3.2 调用方法
19.3.3 WS-I规范
19.4 Web服务和.NET Framework
19.4.1 创建Web服务
19.4.2 客户程序
19.5 创建简单的ASP.NET Web服务
19.6 测试Web服务
19.7 实现Windows客户程序
19.8 异步调用服务
19.9 实现ASP.NET客户程序
19.10 传送数据
19.11 小结
19.12 练习
19.13 本章要点
第20章 部署Web应用程序
20.1 Internet Information Services
20.2 IIS配置
20.3 复制Web站点
20.4 发布Web站点
20.5 Windows安装程序
20.5.1 创建安装程序
20.5.2 安装Web应用程序
20.6 小结
20.7 练习
20.8 本章要点
第IV部分 数据访问
第21章 文件系统数据
21.1 流
21.2 用于输入和输出的类
21.2.1 File类和Directory类
21.2.2 FileInfo类
21.2.3 DirectoryInfo类
21.2.4 路径名和相对路径
21.2.5 FileStream对象
21.2.6 StreamWriter对象
21.2.7 StreamReader对象
21.2.8 读写压缩文件

２１．３　序列化对象
２１．４　监控文件系统
２１．５　小结
２１．６　练习
２１．７　本章要点
第２２章　XML
２２．１　XML文档
２２．１．１　XML元素
２２．１．２　特性
２２．１．３　XML声明
２２．１．４　XML文档的结构
２２．１．５　XML名称空间
２２．１．６　格式良好并有效的XML
２２．１．７　验证XML文档
２２．２　在应用程序中使用XML
２２．２．１　XML文档对象模型
２２．２．２　选择节点
２２．２．３　XPath
２２．３　小结
２２．４　练习
２２．５　本章要点
第２３章　LINQ简介
２３．１　第一个LINQ查询
２３．１．１　用var关键字声明结果变量
２３．１．２　指定数据源：from子句
２３．１．３　指定条件：where子句
２３．１．４　指定元素：select子句
２３．１．５　完成：使用foreach循环
２３．１．６　延迟执行的查询
２３．２　使用LINQ方法语法
２３．２．１　LINQ扩展方法
２３．２．２　查询语法和方法语法
２３．３　排序查询结果
２３．４　orderby子句
２３．５　用方法语法排序
２３．６　查询大型数据集
２３．７　聚合运算符
２３．８　查询复杂的对象
２３．９　投影：在查询中创建新对象
２３．１０　投影：方法语法
２３．１１　单值选择查询
２３．１２　Any和All
２３．１３　多级排序
２３．１４　多级排序方法语法：ThenBy
２３．１５　组合查询
２３．１６　Take和Skip
２３．１７　First和FirstOrDefault
２３．１８　集运算符
２３．１９　Join查询
２３．２０　小结
２３．２１　练习
２３．２２　本章要点
第２４章　应用LINQ
２４．１　LINQ的变体
２４．２　给数据库使用LINQ
２４．３　安装SQL Server和Northwind示例数据

24.3.1 安装SQL Server Express 2008
24.3.2 安装Northwind示例数据库
24.4 第一个LINQ数据库查询
24.5 浏览数据库关系
24.6 使用LINQ to XML
24.7 LINQ to XML函数构造方法
24.8 保存和加载XML文档
24.8.1 从字符串中加载XML
24.8.2 已保存的XML文档内容
24.9 处理XML片段
24.10 从数据库中生成XML
24.11 查询XML文档的方法
24.12 使用LINQ to XML查询成员
24.12.1 Elements()
24.12.2 Descendants()
24.12.3 Attributes()
24.13 小结
24.14 练习
24.15 本章要点
第V部分 其他技术
第25章 Windows Presentation Foundation
25.1 WPF的概念
25.1.1 WPF给设计人员带来的好处
25.1.2 WPF给C#开发人员带来的好处
25.2 基本WPF应用程序的组成
25.3 WPF基础
25.3.1 XAML语法
25.3.2 桌面和Web应用程序
25.3.3 Application对象
25.3.4 控件基础
25.3.5 控件的布局
25.3.6 控件的样式
25.3.7 触发器
25.3.8 动画
25.3.9 静态和动态资源
25.4 用WPF编程
25.4.1 WPF用户控件
25.4.2 实现依赖属性
25.5 小结
25.6 练习
25.7 本章要点
第26章 Windows Communication Foundation
26.1 WCF的含义
26.2 WCF概念
26.2.1 WCF通信协议
26.2.2 地址、端点和绑定
26.2.3 合同
26.2.4 消息模式
26.2.5 行为
26.2.6 驻留
26.3 WCF编程
26.3.1 WCF测试客户程序
26.3.2 定义WCF服务合同
26.3.3 自驻留的WCF服务
26.4 小结
26.5 练习

26.6 本章要点
第27章 Windows Workflow Foundation
27.1 Hello World
27.2 工作流和活动
27.2.1 If活动
27.2.2 While活动
27.2.3 Sequence活动
27.3 变元和变量
27.4 定制活动
27.4.1 工作流扩展
27.4.2 活动的有效性验证
27.4.3 活动设计器
27.5 小结
27.6 练习
27.7 本章要点
附录A 习题答案