

在一些情况下,需要在网格中隐藏某些属性,此时可以使用 `Browsable` 属性,如下面的代码段所示。标记为 `non-browsable` 的属性不会显示在属性网格中。

```
[Browsable(false)]
public bool IsEmployed
{
    .
}
```

`DataGridView` 使用 `Browsable` 属性确定是显示还是隐藏一个属性。如果没有设置 `Browsable` 属性,就默认为显示属性。如果属性是只读的,网格控件就显示对象的值,但在网格中它是只读的。

在网格视图中的所有修改都会反映到底层对象上。例如,如果在上面的代码中,修改了用户界面中的人名,就会调用该属性的 `Setter` 方法。

39.3.6 DateTimePicker 控件

`DateTimePicker` 允许用户在许多不同的格式中选择一个日期或时间值(或两者)。可以以任何标准的时间和日期格式显示基于 `DateTime` 的值。`Format` 属性接受 `DateTimePickerFormat` 枚举,它可以把格式设置为 `Long`、`Short`、`Time` 或 `Custom`。如果把 `Format` 属性设置为 `DateTimePickerFormat.Custom`,就可以把 `CustomFormat` 属性设置为表示该格式的字符串。

`DateTimePicker` 还包含一个 `Text` 属性和一个 `Value` 属性。`Text` 属性返回 `DateTime` 值的文本表示,而 `Value` 属性返回 `DateTime` 对象。还可以用 `MinDate` 和 `MaxDate` 属性设置日期所允许的最大值和最小值。

在用户单击向下箭头时,会显示一个日历,允许用户选择日历中的一个日期。`DateTimePicker` 还包含一些属性,这些属性可用于设置标题、月份的背景色和前景色,从而改变日期的外观。

`ShowUpDown` 属性确定控件上是否显示 `UpDown` 箭头。单击向上或向下箭头就可以改变当前突出显示的值。

39.3.7 ErrorProvider 组件

`ErrorProvider` 实际上并不是一个控件,而是一个组件。当把该组件拖到设计器中时,它会显示在设计器下方的组件栏中。当存在一个错误条件或验证失败时,`ErrorProvider` 可以在控件的旁边显示一个图标。假定有一个 `TextBox` 控件用于输入年龄,业务规则是年龄值不能大于 65。如果用户试图输入大于 65 的年龄,就必须通知用户该年龄大于所允许的值,需要改变输入的值。有效值的检查在文本框的 `Validated` 事件中进行。如果验证失败,就调用 `SetError()` 方法,传递引起错误的控件和一个字符串,将该错误告知用户。然后,一个感叹号图标开始闪烁,表示出现了一个错误,用户把鼠标悬停在该图标上时,会显示错误文本。图 39-9 显示了文本框中输入无效值时出现的图标。

如果字段的验证失败,就调用 `ErrorProvider` 对象的 `SetIcon()` 方法,并传递验证失败的控件和相应的错误消息。该错误在导致验证失败的控件旁边显示为一个闪烁的图标,错误消息显示为工具提示。可以用一个空的错误字符串再次调用 `SetIcon()` 方法,以删除错误条件。

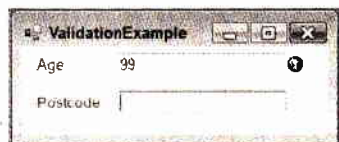


图 39-9

39.3.8 ImageList 组件

顾名思义, ImageList 组件就是一个图像列表。一般情况下, 这个组件用于存储一个图像集合, 这些图像用作工具栏图标或 TreeView 控件中的图标。许多控件都包含 ImageList 属性。这个属性一般和 ImageIndex 属性一起使用。把 ImageList 属性设置为 ImageList 组件的一个实例, 把 ImageIndex 属性设置为 ImageList 中表示应在控件上显示的图像的索引。使用 ImageIndex.Images 属性的 Add() 方法可以把图像添加到 ImageList 组件中。Images 属性返回一个 ImageCollection。

两个最常用的属性是 ImageSize 和 ColorDepth。ImageSize 使用一个 Size 结构作为其值。其默认值是 16×16, 但可以取 1~256 之间的任意值。ColorDepth 使用一个 ColorDepth 枚举作为其值。颜色深度值可以是 4~32 位, 默认是 ColorDepth.Depth8Bit。

39.3.9 Label 控件

Label 控件一般用于给用户提描述性文本。文本可以与其他控件或当前系统状态相关。用户通常看到标签和文本框一起使用。标签为用户提供了在文本框中输入的数据类型的描述。标签控件总是只读的, 用户不能修改 Text 属性的字符串值。但是, 可以在代码中修改 Text 属性。UseMnemonic 属性允许用户启用访问键功能。在 Text 属性中, 在一个字符前面加上与符号(&)时, 标签控件中的该字母就会加上下划线。按 Alt 键和带有下划线的字母就会把焦点移动到按 tab 顺序显示的下一个控件上。如果希望在文本中显示一个“与”符号, 就应用第二个“与”符号转义第一个“与”符号。例如, 如果标签文本应是“Nuts & Bolts”, 就应把属性设置为“Nuts && Bolts”。由于标签控件是只读的, 所以它不能获得焦点。这就是焦点会移动到下一个控件上的原因。因此要记住, 如果启用 mnemonics, 就必须确保正确地设置窗体上的 tab 顺序。

AutoSize 属性是一个布尔值, 指定标签是否根据标签的内容自动设置其大小。在多语言应用程序中, Text 属性的长度会根据当前语言的不同而变化, 此时使用这个属性就很有用。

39.3.10 ListView 控件

ListView 控件允许以 4 种不同的方式显示条目。可以显示文本和可选的大图标、显示文本和可选的小图标、在垂直列表中显示文本和小图标, 以及在详细视图中显示条目文本, 并在列中显示子条目。这听起来应很熟悉, 因为 Windows 资源管理器的右边就用这种方式显示文件夹的内容。ListView 包含一个 ListViewItem 集合。ListViewItem 允许设置一个用于显示的 Text 属性, 它的另一个属性 SubItems 包含在详细视图中显示的文本。

ListView 是 .NET Framework 中最灵活的一个控件, 可以设置许多选项来改变条目的外观。例如, 可以把 CheckBoxes 属性设置为 true, 以模拟带复选框的列表框。还可以给条目添加图像、改变视图, 修改列标题的排序方式, 甚至把条目组合到不同的组中。

39.3.11 PictureBox 控件

PictureBox 控件用于显示图像。图像可以是 BMP、JPEG、GIF、PNG、元文件或图标。SizeMode 属性使用 PictureBoxSizeMode 枚举确定图像在控件中的大小和位置。SizeMode 属性可以是 AutoSize、CenterImage、Normal 和 StretchImage。

设置 ClientSize 属性, 可以改变显示的 PictureBox 的大小。要加载 PictureBox, 首先创建一个基于 Image 的对象。例如, 要把 JPEG 文件加载到 PictureBox 中, 需要编写如下代码:

```
Bitmap myJpeg = Bitmap.FromFile("whatever.png") as Bitmap;
pictureBox1.Image = myJpeg;
```

39.3.12 ProgressBar 控件

ProgressBar 控件是较长操作的状态的可视化表示。它向用户指出正在进行某个操作，用户应等待。ProgressBar 控件有效时要设置 Minimum 和 Maximum 属性。这些属性对应于进度指示器的最左端 (Minimum)和最右端(Maximum)。设置 Step 属性，以确定每次调用 PerformStep()方法时数值的增量。还可以使用 Increment()方法，递增在方法调用中传入的值。Value 属性返回 ProgressBar 的当前值。

39.3.13 TextBox 控件、RichTextBox 控件与 MaskedTextBox 控件

TextBox 控件是工具箱中最常用的控件之一。TextBox、RichTextBox 和 MaskedTextBox 控件都派生自 TextBoxBase。TextBoxBase 提供了 MultiLine 和 Lines 等属性，MultiLine 属性是一个布尔值，它允许 TextBox 控件在多行中显示文本。文本框中的每一行都是字符串数组的一部分。这个数组通过 Lines 属性来提供。Text 属性把整个文本框内容返回为一个字符串。TextLength 是返回的文本字符串的总长度。MaxLength 属性把文本的长度限制为指定的数字。

SelectedText、SelectionLength 和 SelectionStart 都处理文本框中当前选中的文本。选中的文本是控件获得焦点时突出显示的文本。

TextBox 控件添加了几个有趣的属性。AcceptsReturn 属性是一个布尔值，它允许 TextBox 捕获回车键。如果把这个值设置为 false，按回车键就会激活窗体上的默认按钮。把 AcceptsReturn 属性设置为 true 时，按回车键就会在文本框中新建一行。CharactorCasing 确定文本框中文本的大小写。CharactorCasing 枚举包含 3 个值 Lower、Normal 和 Upper。Lower 会使所有输入的文本小写，Upper 则把所有文本转变为大写，Normal 把文本显示为输入时的形式。PasswordChar 属性用一个字符表示用户在文本框中输入文本时要显示给用户的内容，这通常用于输入密码和 PIN。text 属性返回实际输入的文本，只有显示的内容会受这个属性的影响。

RichTextBox 是一个文本编辑控件，它可以处理特殊格式的文本。顾名思义，RichTextBox 控件使用 Rich Text Format(RTF)处理特殊的格式。使用 Selection 属性 SelectionFont、SelectionColor、SelectionBullet 可以修改格式，使用 SelectionIndent、SelectionRightIndent、SelectionHangingIndent 可以修改段落的格式。所有 Selection 属性的工作方式都相同。如果有一段突出显示的文本，对 Selection 属性的修改就会影响选中的文本。如果没有选中文本，这些修改就对当前插入点后面的文本起作用。

控件的文本可以使用 Text 属性或 Rtf 属性检索。Text 属性只返回控件的文本，而 Rtf 属性返回带格式的文本。

LoadFile()方法可以用两种不同的方式从文件中加载文本。它可以使用一个表示文件名和路径的字符串，也可以使用一个流对象。还可以指定 RichTextBoxStreamType。表 39-6 列出了 RichTextBoxStreamType 的值。

表 39-6

值	说 明
PlainText	没有格式信息，包含 OLE 对象，允许使用空格
RichNoOleObjs	Rich 文本格式，但不包含 OLE 对象已经包含的空格

(续表)

值	说 明
RichText	格式化的 RTF, 且包含 OLE 对象
TextTextOleObjs	纯文本, 用文本替换 OLE 对象
UnicodePlainText	与 PlainText 相同, 但编码为 Unicode

SaveFile()方法使用相同的参数, 把控件中的数据保存到指定的文件中。如果同名的文件已经存在, 就覆盖它。在示例中, RTF 文件包含为应用程序中的一个资源, 并在加载窗体时把文本加载到控件中。

MaskedTextBox()可以限制用户在控件中输入的内容, 它还可以自动格式化输入的数据。使用几个属性可以验证或格式化用户的输入。Mask 属性包含掩码字符串, 掩码字符串类似于格式字符串, 使用 Mask 字符串可以设置允许的字符数、允许字符的数据类型和数据的格式。基于 MaskedTextBox 的类也提供了需要的格式化信息和验证信息。MaskedTextBox 只能在传递它的其中构造函数中设置。

有 3 个不同的属性返回 MaskedTextBox 的文本。Text 属性返回控件的当前文本, 它可以根据控件是否获得焦点而不同, 而控件是否获得焦点取决于 HidePromptOnLeave 属性的值。该提示是一个字符串, 告诉用户应输入什么内容。InputText 属性总是只返回用户输入的文本。OutputText 属性根据 IncludeLiterals 和 IncludePrompt 属性返回格式化的文本。例如, 如果对电话号码进行掩码, Mask 字符串就应包含圆括号和几条短横线。这些都是字面量字符, 如果把 IncludeLiteral 属性设置为 true, 圆括号和短横线就应属于文本字符且包含在 OutputText 属性中。

MaskedTextBox 控件还有几个额外的事件。OutputTextChanged 和 InputTextChanged 在 InputText 或 OutputText 改变时触发。

39.3.14 Panel 控件

Panel 控件就是包含其他控件的控件。把控件组合在一起, 并放在一个面板中, 将更容易管理这些控件。例如, 可以禁用面板, 从而禁用该面板中的所有控件。因为 Panel 控件派生自 ScrollableControl, 所以还可以使用 AutoScroll 属性。如果可用区域中有过多的控件要显示, 就可以把它们放在一个面板中, 并把 AutoScroll 属性设置为 true, 现在就可以滚动所有控件了。

面板在默认情况下不显示边框, 但把 BorderStyle 属性设置为除了 none 的其他值, 就可以使用 Panel 通过边框可视化地组合相关控件。这会使用户界面更友好。

Panel 是 FlowLayoutPanel、TableLayoutPanel、TabPage 和 SplitterPanel 的基类。使用这些控件, 可以创建外观非常复杂或专业化的窗体或窗口。FlowLayoutPanel 和 TableLayoutPanel 对创建正确地设置大小的窗体尤其有帮助。

39.3.15 FlowLayoutPanel 和 TableLayoutPanel 控件

FlowLayoutPanel 和 TableLayoutPanel 是 .NET Framework 的新增控件。顾名思义, 面板可以采用与 Web 窗体相同的样式给 Windows 窗体布局。FlowLayoutPanel 是一个容器, 允许以垂直或水平的方式放置包含的控件。除了放置控件之外, 还可以剪辑控件。放置的方向使用 FlowDirection 属性和

FlowDirection 枚举来设置。WrapContents 属性确定在重新设置窗体的大小时, 控件是放在下一行、下一列, 还是剪辑控件。

TableLayoutPanel 使用栅格结构控制控件的布局。任何 Windows 窗体控件都是 TableLayoutPanel 的子控件, 包括另一个 TableLayoutPanel。所以窗口的布局可以非常灵活, 并可以动态设置。把一个控件添加到 TableLayoutPanel 中时, 会给属性页面的 Layout 类别添加 4 个属性。它们分别是 Column、ColumnSpan、Row 和 RowSpan。与 Web 页面上的 html 表一样, 可以给每个控件设置列和行间距。控件默认放置在表的单元格中心处, 但这可以使用 Anchor 和 Dock 属性改变。

行和列的默认样式可以使用 RowStyles 和 ColumnsStyles 集合改变。这两个集合分别包含 RowStyle 和 ColumnsStyle 对象。Style 对象有一个公共属性 SizeType。SizeType 使用 SizeType 枚举来确定列宽或行高。该枚举值包含 AutoSize、Absolute 和 Percent。AutoSize 与其他同等控件共享该空间。Absolute 允许使用一组像素值来设置大小。Percent 要求控件把列或宽度设置为父控件的一个百分比。

行、列和子控件都可以在运行期间添加和删除。GrowStyle 属性使用 TableLayoutPanelGrowStyle 枚举值来设置在已填满的表中添加一个新控件时, 是给表添加列、行, 还是使表保持固定的大小。如果其值是 FixedSized, 则在试图添加另一个控件时, 就抛出一个 ArgumentException 异常。如果表中的单元格为空, 控件就放在空单元格中。这个属性仅在表已满, 但要添加控件时起作用。

示例代码中的窗体有 FlowLayoutPanels 和 TableLayoutPanels, 以说明在面板的控件上重置窗体大小的效果。

39.3.16 SplitContainer 控件

SplitContainer 控件把 3 个控件组合在一起, 其中有两个面板控件, 在它们之间有一个拆分器。用户可以移动拆分器, 重新设置面板的大小。在重新设置面板的大小时, 面板中的控件也可以重新设置大小。SplitContainer 的最佳示例是文件管理器。左侧面板包含文件夹的树型视图, 右侧面板包含文件夹内容的列表视图。用户在拆分器上移动鼠标时, 光标就会改变, 此时可以移动拆分器。SplitContainer 可以包含任意控件, 包括布局面板和其他 SplitContainer。因此, 可以创建非常复杂、高级的窗体。

拆分器的移动和定位可以用 SplitterDistance 和 SplitterIncrement 属性控制。SplitterDistance 属性确定拆分器与控件左边界或顶边的距离, SplitterIncrement 确定在拖动时分隔栏移动的像素值。面板可以使用 Panel1MinSize 和 Panel2MinSize 属性设置其最小尺寸, 这些属性的单位也是像素。

Splitter 控件会引发与移动相关的两个事件 SplitterMoving 和 SplitterMoved。SplitterMoving 事件在移动过程中引发, SplitterMoved 在移动结束后引发。它们都接收一个 SplitterEventArgs。SplitterEventArgs 的 SplitX 和 SplitY 属性表示 Splitter 左上角的 X 和 Y 坐标, X 和 Y 属性表示鼠标指针的 X 和 Y 坐标。

示例代码包含一个穷人的 Windows 资源管理器, 其中使用了 SplitContainer、树型视图和列表视图, 来显示目录和文件。

39.3.17 TabControl 控件和 TabPages 控件

TabControl 允许把相关的组件组合到一系列选项卡页面上。TabControl 管理 TabPages 集合。有几个属性可以控制 TabControl 的外观。Appearance 属性使用 TabAppearance 枚举确定选项卡的外观。

其值是 FlatButtons、Buttons 或 Normal。Multiline 属性对应一个布尔值，确定是否显示多行选项卡。如果把 Multiline 属性设置为 false，而有多个选项卡不能一次显示出来，就提供一组箭头，允许用户滚动查看剩余的选项卡。

TabPage 的 text 属性是在选项卡上显示的内容。Text 属性也在重写的构造函数中用作参数。

一旦创建了 TabPage 控件，它基本上就是一个容器控件，用于放置其他控件。Visual Studio .NET 中的设计器使用集合编辑器，很容易给 TabControl 控件添加 TabPage 控件。在添加每个页面时都可以设置各种属性。接着把其他子控件拖放到每个 TabPage 控件上。

通过查看 SelectedTab 属性可以确定当前的选项卡。每次选择新选项卡时，都会引发 SelectedIndex 事件。通过侦听 SelectedIndex 属性，再用 SelectedTab 属性确认当前选项卡，就可以对每个选项卡进行特定的处理。例如，可以管理为每个选项卡显示的数据。

39.3.18 ToolStrip 控件

ToolStrip 控件是一个用于创建工具栏、菜单结构和状态栏的容器控件。ToolStrip 直接用于工具栏，还可以用作 MenuStrip 和 StatusStrip 控件的基类。

ToolStrip 控件在用作工具栏时，使用一组基于抽象类 ToolStripItem 的控件。ToolStripItem 可以添加公共显示和布局功能，并管理控件使用的大多数事件。ToolStripItem 派生自 System.ComponentModel.Component 类，而不是 Control 类。基于 ToolStripItem 的类必须包含在基于 ToolStrip 的容器中。

Image 和 Text 是要设置的最常见属性。图像可以用 Image 属性设置，也可以使用 ImageList 控件，把它设置为 ToolStrip 控件的 ImageList 属性。然后就可以设置各个控件的 ImageIndex 属性。

ToolStripItem 对应的文本的格式化用 Font、TextAlign 和 TextDirection 属性来处理。TextAlign 设置文本与控件的对齐方式，它可以是 ControlAlignment 枚举中的任意值，默认为 MiddleRight。TextDirection 属性设置文本的方向，其值可以是 ToolStripTextDirection 枚举中的任意值，包括 Horizontal、Inherit、Vertical270 和 Vertical90。Vertical270 把文本旋转 270°，Vertical90 把文本旋转 90°。

DisplayStyle 属性控制在控件上是显示文本、图像、文本和图像，还是什么都不显示。在 AutoSize 设置为 true 时，ToolStripItem 会重新设置其大小，确保只使用最少量的空间。

直接派生自 ToolStripItem 的控件如表 39-7 所示。

表 39-7

Tool Strip Items	说 明
ToolStripButton	表示用户可以选择的按钮
ToolStripLabel	在 ToolStrip 上显示不能选择的文本或图像。ToolStripLabel 还可以显示一个或多个超链接
ToolStripSeparator	用于分解和组合其他 ToolStripItems。根据功能来组合对应项
ToolStripDropDownItem	显示下拉选项。它是 ToolStripDropDownButton、ToolStripMenuItem 和 ToolStripSplitButton 的基类
ToolStripControlHost	在 ToolStrip 上存放其他非 ToolStripItem 的派生控件。它是 ToolStripComboBox、ToolStripProgressBar 和 ToolStripTextBox 的基类

表 39-4 中的后两项 ToolStripDropDownItem 和 ToolStripControlHost 需要详细探讨。ToolStripDropDownItem 是 ToolStripMenuItem 的基类，用于构建菜单结构。把 ToolStripMenuItem 添加到 MenuStrip

控件中。如前所述, `MenuStrips` 派生自 `ToolStrip` 控件。在处理和扩展菜单项时, 这很重要。因为工具栏和菜单派生自同一个类, 所以很容易创建用于管理并执行命令的架构。

`ToolStripControlHost` 可以用于包含其他不派生自 `ToolStripItem` 的控件。可以直接放在 `ToolStrip` 中的控件是派生自 `ToolStripItem` 的控件。

39.3.19 ToolStrip 控件

`MenuStrip` 控件是应用程序菜单结构的容器。如前所述, `MenuStrip` 派生自 `ToolStrip` 类。在构建菜单系统时, 要给 `MenuStrip` 添加 `ToolStripMenu` 对象。这可以在代码中完成, 也可以在 Visual Studio 的设计器中进行。把一个 `MenuStrip` 控件拖放到设计器的一个窗体中, `MenuStrip` 就允许直接在菜单项上输入菜单文本。

`MenuStrip` 控件只有两个额外的属性。`GripStyle` 使用 `ToolStripGripStyle` 枚举把手柄设置为可见或隐藏。`MdiWindowListItem` 属性接受或返回 `ToolStripMenuItem`。这个 `ToolStripMenuItem` 是在 MDI 应用程序中显示所有已打开窗口的菜单。

39.3.20 ContextMenuStrip 控件

要显示上下文菜单, 或在用户右击鼠标时显示一个菜单, 就应使用 `ContextMenuStrip` 类。与 `MenuStrip` 控件一样, `ContextMenuStrip` 也是 `ToolStripMenuItems` 对象的容器, 但它派生自 `ToolStripDropDownMenu`。`ContextMenu` 的创建方法与 `MenuStrip` 相同, 也是添加 `ToolStripMenuItems`, 定义每一项的 `Click` 事件, 执行某个特定的任务。把上下文菜单赋予特定的控件, 为此, 要设置控件的 `ContextMenuStrip` 属性。在用户右击该控件时, 就显示该菜单。

39.3.21 ToolStripMenuItem 控件

`ToolStripMenuItem` 是构建菜单结构的类。每个 `ToolStripMenuItem` 对象都表示菜单系统上的一个菜单选项。每个 `ToolStripMenuItem` 都有一个包含子菜单的 `ToolStripItemCollection`。这个功能继承自 `ToolStripDropDownItem`。

由于 `ToolStripMenuItem` 派生自 `ToolStripItem`, 因此可以使用所有相同的格式化属性。图像在菜单文本的右边显示为小图标。菜单项的旁边可以有复选框标记, 用 `Checked` 和 `CheckState` 属性设置对应标记。

还可以给每个菜单项指定快捷键。快捷键一般包含两个按键, 如 `Ctrl+C`(Copy 的快捷键)。在指定快捷键时, 把 `ShowShortcutKey` 属性设置为 `true`, 还可以在菜单上显示该快捷键。

有用起见, 在用户单击菜单项或使用定义好的快捷键时, 菜单项必须执行某个任务。为此, 最常见的方式是处理 `Click` 事件。如果正在使用 `Checked` 属性, 则还可以使用 `CheckStateChanged` 和 `CheckedChanged` 事件确定选中状态的变化。

39.3.22 ToolStripManager 类

菜单结构和工具栏结构可以很大, 这样就难以管理。`ToolStripManager` 类可以创建较小、易于管理的菜单结构或工具栏结构, 并在需要时合并它们。例如, 如果窗体上有几个不同的控件, 则每个控件都必须显示一个上下文菜单。几个菜单项对于所有控件都要可用, 但每个控件还有几个唯一菜单项。可以在一个 `ContextMenuStrip` 上定义公共菜单选项, 每个唯一菜单项可以预先定义, 或在

运行期间创建。对于需要为其指定上下文菜单的每个控件，应复制公共菜单选项，再使用 `ToolStripManager.Merge()` 方法把唯一菜单选项与公共菜单选项合并起来。把最后得到的菜单赋予控件的 `ContextMenuStrip` 属性。

39.3.23 ToolStripContainer 控件

`ToolStripContainer` 控件用于停靠基于 `ToolStrip` 的控件。当添加一个 `ToolStripContainer` 控件并把 `Docked` 属性设置为 `Fill` 时，就在窗体的两侧边添加了一个 `ToolStripPanel`，在窗体的中间添加了一个 `ToolStripContainerPanel`。在 `ToolStripPanel` 控件中可以添加任意 `ToolStrip` (`ToolStrip`、`MenuStrip` 或 `StatusStrip`)。用户可以选择 `ToolStrips`，把它拖放到窗体的两边或底部。如果把任意 `ToolStripPanel` 控件的 `Visible` 属性设置为 `false`，就不能把 `ToolStrip` 放在面板中了。窗体中心的 `ToolStripContainerPanel` 控件可以用于放置窗体需要的其他控件。

39.4 窗体

本章前面讨论了如何创建简单的 Windows 应用程序。该示例包含一个派生自 `System.Windows.Forms.Form` 的类。根据 .NET Framework 文档，“窗体是应用程序中任何窗口的表示方式。”如果您具有 Visual Basic 背景，就会很熟悉术语“窗体”。如果您是使用 MFC 的 C++ 程序员，就可能习惯把窗体称为窗口、对话框或框架。无论怎样，窗体都是与用户交互的基本方式。我们已经介绍了 `Control` 类中一些较常见的属性、方法和事件，而且因为 `Form` 类派生自 `Control` 类，所以在 `Form` 类中存在所有相同的属性、方法和事件。`Form` 类还在 `Control` 类的基础上添加了大量的功能，本节就介绍这些功能。

39.4.1 Form 类

Windows 客户端应用程序可以包含一个窗体或上百个窗体。它们可以是基于 SDI(单文档界面, Single Document Interface)或 MDI(多文档界面, Multiple Document Interface)的应用程序。但无论怎样, `System.Windows.Forms.Form` 类都是 Windows 端客户的核心。`Form` 类派生于 `ContainerControl`, `ContainerControl` 又派生自 `ScrollableControl`, `ScrollableControl` 又派生自 `Control` 类。因此可以假定, 窗体可以是其他控件的容器, 当所包含的控件在工作区中显示不下时可以滚动显示, 窗体可以拥有与其他控件相同的许多属性、方法和事件。所以, `Form` 类相当复杂。本节就介绍这些功能。

1. 窗体的实例化和析构

理解创建窗体的过程很重要。我们要完成的工作取决于编写初始化代码的位置。对于实例化, 事件以如下顺序发生:

- 构造函数
- Load
- Activated
- Closing
- Closed
- Deactivate

前3个事件在初始化过程中发生。根据初始化的类型,可以确定要关联哪个事件。这个类的构造函数在对象的实例化过程中执行。Load事件在对象实例化后,窗体可见之前发生。它与构造函数的区别是窗体的可行性。在引发Load事件时,窗体已存在,但不可见。在构造函数的执行过程中,窗体还不存在,仍处在实例化过程中。Activated事件在窗体处于可见状态并处于当前状态时发生。

在某种特殊情形下该顺序略有改变。如果在窗体构造函数执行的过程中,把Visible属性设置为true,或调用了Show()方法(它把Visible属性设置为true),就会立即引发Load事件。因为这也会使窗体可见,并处于当前状态,所以还会引发Activate事件。如果在设置Visible属性后还有代码,就执行这些代码。所以启动事件的执行顺序如下所示:

- 构造函数,执行到Visible = true为止
- Load
- Activate
- 构造函数,执行Visible = true之后的代码

这可能会产生一些出乎意料的结果。从最佳实践的角度来看,最好在构造函数中进行尽可能多的初始化。

关闭窗体时会发生什么情况?Closing事件可以取消处理,它把CancelEventArgs作为一个参数,如果把Cancel属性设置为true,就会取消该事件,窗体仍处于打开状态。Closing事件在窗体关闭时发生,而Closed事件在窗体关闭后发生。这两个事件都允许执行必要的清理工作。注意,Deactivate事件在窗体关闭后发生,这是另一个可能产生难以查找的错误的来源。确保不在Deactivate事件中执行防止窗体正常地垃圾回收的操作。例如,设置对另一个对象的引用会使窗体仍处于活动状态。

如果调用Application.Exit()方法,且当前有一个或多个窗体处于打开状态,就不会引发Closing和Closed事件。如果打开了正要清理的文件或数据库连接,这就是一个需要考虑的重要问题,此时应调用Dispose()方法,所以另一种更好的方法是把大多数清理代码放在Dispose()方法中。

与窗体的启动相关的一些属性有StartPosition、ShowInTaskbar和TopMost。StartPosition可以是FormStartPosition枚举中的任意一个值:

- CenterParent——窗体位于父窗体的工作区中心。
- CenterScreen——窗体位于当前屏幕的中心。
- Manual——窗体的位置根据Location属性的值来确定。
- WindowsDefaultBounds——窗体位于默认的Windows位置,使用默认的大小。
- WindowsDefaultLocation——窗体位于默认的Windows位置,但其大小根据Size属性来定。

ShowInTaskbar属性确定窗体是否应在任务栏上可用。只有窗体是一个子窗体,且只希望父窗体显示在任务栏上时,才使用这个属性。TopMost属性指定窗体在应用程序启动时位于层次关系的最上面,即使窗体没有立即获得焦点,也位于最上面。

为了让用户与应用程序交互,用户必须能看到窗体。利用Show()和.ShowDialog()方法就可以实现这一点。Show()方法仅使窗体对用户可见。下面的代码说明了如何创建一个窗体,并把它显示给用户。假定要显示的窗体被命名为MyFormClass:

```
MyFormClass myForm = new MyFormClass();  
myForm.Show();
```

这非常简单。但它的一个缺点是没有给主调代码返回任何通知,说明MyForm已处理完,并退

出(除非关联到窗体上的 Closing 或 Closed 事件)。有时这并不重要, Show()方法工作得很好。如果需要提供某种通知, 使用 ShowDialog()方法是一种比较好的选择。

在调用 Show()方法后, Show()方法后面的代码会立即执行。在调用 ShowDialog()方法后, 主调代码被暂停执行, 等到调用 ShowDialog()方法的窗体关闭后再继续执行。不仅主调代码被暂停执行, 而且窗体也可以返回一个 DialogResult 值。DialogResult 枚举是一个标识符列表, 它们描述了对话框关闭的原因, 包括 OK、Cancel、Yes、No 和其他几个标识符。为了让窗体返回一个 DialogResult 值, 必须设置窗体的 DialogResult 属性, 或者在窗体的一个按钮上设置 DialogResult 属性。

例如, 假定应用程序的一部分要求提供客户的电话号码。窗体包含一个输入电话号码的文本框, 和两个按钮 OK 和 Cancel。如果把 OK 按钮的 DialogResult 属性设置为 DialogResult.OK, 把 Cancel 按钮的 DialogResult 属性设置为 DialogResult.Cancel, 则在选择其中一个按钮时, 窗体不可见, 并给调用它的窗体返回相应的 DialogResult 值。现在注意窗体没有销毁, 只是把 Visible 属性设置为 false。这是因为仍必须从窗体中获取值。在这个示例中, 我们需要电话号码。在窗体上为电话号码创建一个属性, 这样父窗体就可以获取对应值, 并调用窗体上的 Close()方法。下面就是子窗体的代码:

```
namespace FormsSample.DialogSample
{
    partial class Phone: Form
    {
        public Phone()
        {
            InitializeComponent();
            btnOK.DialogResult = DialogResult.OK;
            btnCancel.DialogResult = DialogResult.Cancel;
        }

        public string PhoneNumber
        {
            get { return textBox1.Text; }
            set { textBox1.Text = value; }
        }
    }
}
```

首先要注意, 不包含处理按钮的 Click 事件的代码。因为设置了每个按钮的 DialogResult 属性, 所以在单击 OK 或 Cancel 按钮后, 窗体就消失了。添加的唯一属性是 PhoneNumber。下面的代码显示了父窗体中调用 Phone 对话框的方法:

```
Phone frm = new Phone();
frm.ShowDialog();
if (frm.DialogResult == DialogResult.OK)
{
    label1.Text = "Phone number is " + frm.PhoneNumber;
}
else if (frm.DialogResult == DialogResult.Cancel)
{
    label1.Text = "Form was canceled. ";
}
frm.Close();
```

这看起来非常简单。创建新的 Phone 对象 frm, 在调用 frm.ShowDialog()方法时, 这个方法中的代码会停止执行, 等待 Phone 窗体返回。接着检查 Phone 窗体的 DialogResult 属性。由于窗体还未销毁, 仅使它不可见, 所以仍可以访问公共属性, 其中一个公共属性就是 PhoneNumber。一旦获取了需要的数据, 就可以调用窗体上的 Close()方法。

因为在窗体关闭之前可以在对话框中验证控件, 所以在本例中可以验证电话号码的格式(并相应地更新 OK 按钮的 Enabled 状态)。

2. 外观

用户首先看到的是应用程序的窗体。它应是应用程序中首要的功能。如果应用程序不解决业务问题, 其外观就无关紧要。这并不是说, 窗体和应用程序的整体 GUI 设计不应美观。像颜色组合、字体大小和窗口大小等的设计都可以使应用程序更吸引用户。

有时不希望用户访问系统菜单。在单击窗口左上角的图标时, 这个菜单就会显示出来。一般情况下, 它包含还原、最小化、最大化和关闭等选项。ControlBox 属性允许设置系统菜单的可见性。还可以用 MaximizeBox 和 MinimizeBox 属性设置 Maximize 和 Minimize 按钮的可见性。如果删除所有按钮, 再把 Text 属性设置为空字符串(""), 标题栏就会完全消失。

如果设置了窗体的 Icon 属性, 但没有把 ControlBox 属性设置为 false, 图标就会显示在窗体的左上角。通常应把 Icon 属性设置为 app.ico, 这会使每个窗体的图标都与应用程序的图标相同。

FormBorderStyle 属性用于设置显示在窗体周围的边框类型。它使用 FormBorderStyle 枚举, 其值是:

- Fixed3D
- FixedDialog
- FixedSingle
- FixedToolWindow
- None
- Sizable
- SizableToolWindow

大多数值的意义都一目了然, 只有两个工具窗口边框除外。无论怎样设置 ShowInTaskBar 属性, Tool 窗口都不显示在任务栏中。当用户按 Alt+Tab 组合键时, Tool 窗口也不会显示在窗口列表中。默认设置是 Sizable。

除非明确要求, 否则大多数 GUI 元素的颜色都应设置为系统颜色, 而不是特定的颜色。这样, 如果一些用户喜欢把所有按钮设置为紫字绿底, 应用程序就会采用这种颜色设置。为了把控件设置为使用特定的系统颜色, 必须调用 System.Drawing.Color 类的 FromKnownColor() 方法。FromKnownColor()方法将一个 KnownColor 枚举值作为其参数。在该枚举中定义了许多颜色和 GUI 元素的各种颜色, 如 Control、ActiveBorder 和 Desktop。例如, 如果窗体的背景色应总是匹配 Desktop 颜色, 就应使用下面的代码:

```
myForm.BackColor = Color.FromKnownColor(KnownColor.Desktop);
```

现在如果用户改变桌面的颜色, 窗体的背景色也会随之改变。这将给应用程序增加友好性。用户可以为桌面选择某种奇怪的颜色组合, 但这由他们的偏好决定。

Windows XP 引入了一个命名为可视化样式的特性。当鼠标指针悬停在按钮、文本框、菜单和其他控件上或单击它们时,这些控件的外观会改变,并做出响应,这些相应的改变方式由可视化样式控制。调用 `Application.EnableVisualStyles()` 方法,可以启用应用程序的可视化样式。因为这个方法必须在实例化任何类型的 GUI 之前调用,所以它一般在 `Main()` 方法中调用,如下面的示例所示:

```
[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new Form1());
}
```

这段代码允许支持可视化样式的各种控件采用可视化样式。由于 `EnableVisualStyles()` 方法存在一个问题,所以必须在调用 `EnableVisualStyles()` 方法之后立即添加 `Application.DoEvents()` 方法。这应能解决工具栏上的图标在运行期间开始消失的问题。`EnableVisualStyles()` 方法仅可用于 .NET Framework 1.1 中。

对于控件,还有一个必须完成的任务。大多数控件都提供 `FaltStyle` 属性,它把 `FaltStyle` 枚举作为其值。这个属性可以接受如下 4 个不同值中的一个:

- Flat——控件显示为没有 3D 轮廓的平面图形。
- Popup——类似于 Flat,但当鼠标指针悬停在控件上时,控件显示为 3D 模式。
- Standard——控件显示为 3D 模式。
- System——控件的外观由操作系统控制。

为了启用可视化样式,控件的 `FaltStyle` 属性应设置为 `FaltStyle.System`。应用程序现在采用 XP 的外观并支持 XP 的主题。

39.4.2 多文档界面

当应用程序可以显示同一窗体类型的多个实例,或以某种方式包含不同的窗体时,就应使用 MDI 类型的应用程序。分别例如,可以同时显示多个编辑器窗口的文本编辑器和 Microsoft Access,可以在 Access 中同时打开查询窗口、设计窗口和表窗口。这些窗口都不会超出主 Access 应用程序的边界。

包含本章示例的项目就是一个 MDI 应用程序,项目中的 `MainForm` 窗体就是 MDI 父窗体。把 `IsMdiContainer` 设置为 `true`,会把任何窗体设置为 MDI 父窗体。如果在设计器中创建窗体,注意其背景就会变成暗灰色,这说明它是一个 MDI 父窗体。仍可以给该窗体添加控件,但最好不要这么做。

为了使子窗体成为 MDI 子窗体,子窗体需要一个对其父窗体的链接。为此,应把子窗体的 `MdiParent` 属性设置为父窗体。在这个例子中,所有子窗体都用 `ShowMdiChild()` 方法创建,它的参数是要显示的子窗体的一个引用。把 `MdiParent` 属性设置为 `this`(它引用 `mdiParent` 窗体)后,就显示该窗体。

MDI 应用程序的一个问题是,在任意给定时刻可以打开几个子窗体。对当前活动的子窗体的引用可以使用父窗体的 `ActiveMdiChild` 属性来检索。这通过 Window 菜单上 Current Active 菜单项来说明。单击该菜单项,会显示一个包含窗体的名称和文本值的消息框。

子窗体可以调用 `LayoutMdi()` 方法来安排。`LayoutMdi()` 方法把 `MdiLayout` 枚举值作为参数。其值可以是 `Cascade`、`TileHorizontal` 和 `TileVertical`。

39.4.3 创建自己的用户控件

除了使用内置控件之外，还可以创建自己的自定义控件，再在应用程序中重用它们。自定义控件有两类：派生自 `UserControl` 基类的控件，和派生自 `Control` 基类的控件。`UserControl` 是最容易创建的类型，因为可以从工具箱的其他控件中撰写用户控件。一般仅在内置控件不能用于完成希望的工作时，才编写自定义控件。因为已经有许多图书用整本书的篇幅来讲述自定义控件的创建，所以这里仅讨论用户控件。

本节将构建如图 39-10 所示的用户控件，其中包含一个错误提供程序验证器，并提供其字段，以供重用。这个示例的代码可参见本章代码下载中的 02 `UserControl` 文件夹。

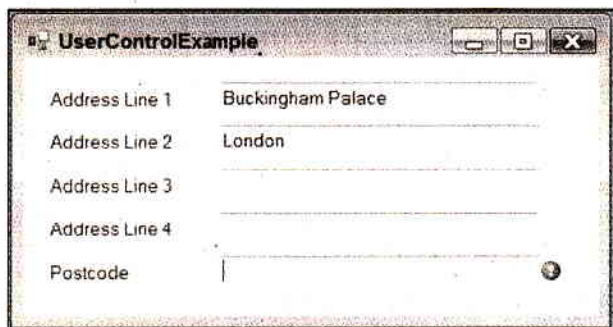


图 39-10

首先需要构建一个新的用户控件，为此可以使用 `Solution Explorer` 或项目菜单。一旦创建它，就有了一个派生自 `UserControl` 的类，接着可以把其他控件拖放到设计界面中。在这个示例中，使用一组标签控件和一组文本框控件来实现图 39-10 所示的功能。

一旦布置好 UI，就可以给控件定义锚定行为。这很重要，因为它允许用户控件在其大小改变时正确地重置大小。在示例控件中，每个文本框都把对齐方式设置为 `Left + Top + Right`，在列表框的右边提供少量空间，以显示任何验证错误。

除了文本框之外，再给用户控件添加一个 `ErrorProvider` 对象，以便在用户在某控件四周来回移动时显示验证错误。因为前两个地址字段和邮政编码指定了验证错误，所以它们是强制验证的。

一旦对用户控件的设计和实现方式满意后，就可以编译它，并在 `Visual Studio` 的工具面板上显示它。接着就可以根据需要在任意应用程序中重用该用户控件。

39.5 小结

本章介绍了建立基于 `Windows` 的客户端应用程序的基础知识。本章解释了许多基本控件，具体方法是讨论了 `Windows.Forms` 名称空间的层次结构，论述了控件的各种属性和方法。

我们还阐述了如何创建基本用户控件。创建自己的控件的作用和灵活性，无论如何强调都不过分。通过创建自己的自定义控件对应的工具箱，基于 `Windows` 的客户端应用程序将更容易开发和测试，因为可以多次重用测试过的同一组件。

第 40 章

核心 ASP.NET

本章内容:

- ASP.NET 简介
- 创建 ASP.NET Web 窗体
- 使用 ADO.NET 绑定数据
- 配置应用程序

如果您是 C# 和 .NET 领域的新手,那么可能会奇怪为什么本书要包含介绍 ASP.NET 的内容。这是一种全新的语言,对吗?但实际上并非如此。事实上,眼见为实,使用 C# 可以创建 ASP.NET 页面。

ASP.NET 是 .NET Framework 的一部分;在通过 HTTP 请求建立文档时,该技术可以在 Web 服务器上动态地创建文档。该文档主要是 HTML 和 XHTML 文档,尽管也可以创建 XML 文档、CSS(Cascading Style Sheet, 级联样式表)文件、图像、PDF 文档,或者支持 MIME 类型的任何其他文档。

在某些方面,ASP.NET 类似于许多其他技术,如 PHP、ASP、ColdFusion 等,但它们有一个重要的区别。顾名思义,ASP.NET 可以与 .NET Framework 完全集成,其中一部分包含了对 C# 的支持。

您可能使用过允许创建动态内容的 ASP(Active Server Page, 动态服务器页面)技术。这种技术使用脚本语言,如 VBScript 或 JavaScript 来编程,结果却不是很好。但对于那些习惯于“正确的”已编译编程语言的人,这种技术很笨拙,肯定会导致性能的损失。

与更高级的编程语言相比,一个主要区别是 ASP.NET 提供了完整的服务器端对象模型,可以在运行时使用。ASP.NET 可以在功能丰富的环境中把页面上的所有控件作为对象来访问。在服务器端,还可以访问其他 .NET 类,与许多有用的服务集成起来。在页面上使用的控件提供了许多功能,实际上可以完成 Windows 窗体类的几乎所有功能,Windows 窗体类非常大的灵活性。因此,生成 HTML 内容的 ASP.NET 页面通常称为 Web 窗体。

本章将详细介绍 ASP.NET,包括 ASP.NET 如何工作,ASP.NET 可以完成什么任务,以及什么地方适合使用 C#。

40.1 ASP.NET 概述

ASP.NET 使用 IIS, Internet 信息服务(Internet Information Service)来传送内容,以响应 HTTP 请求。ASP.NET 页面在 .aspx 文件中,该技术的基本体系结构如图 40-1 所示。

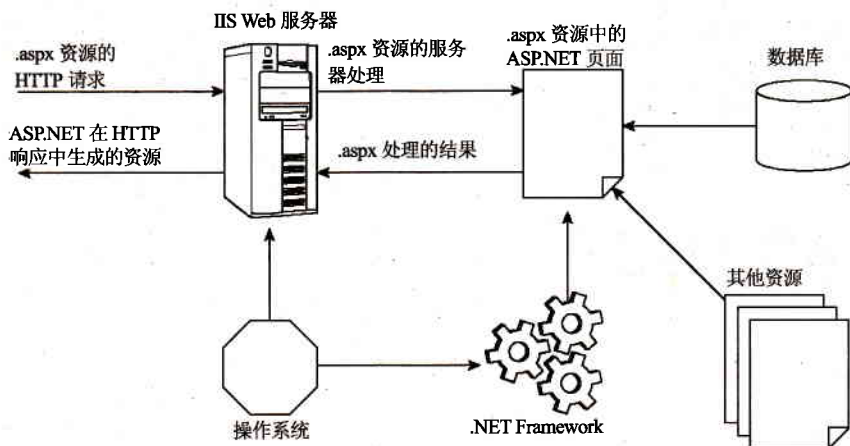


图 40-1

Hello!

40.1.1 ASP.NET 文件的处理方式

在 ASP.NET 处理过程中,可以访问所有 .NET 类、C#或其他语言创建的自定义组件、数据库等。实际上,这与运行 C#应用程序一样;在 ASP.NET 中使用 C#实际上就是在运行 C#程序。

ASP.NET 文件可以包含下述内容:

- 服务器的处理指令
- C#、VB.NET、Jscript.NET 代码或 .NET Framework 支持的其他语言的代码
- 对应已生成资源的窗体内容,如 HTML
- 客户端的脚本代码,如 JavaScript
- 内嵌的 ASP.NET 服务器控件

实际上,ASP.NET 文件也可以很简单,如下所示。

结果很简单,返回一个只包含这个文本的 HTML 页面(因为 ASP.NET 页面的默认输出是 HTML)。本章后面会提到,也可以把代码的某些部分分解为其他文件,这可以提供更合理的结构。

40.1.2 Web 站点和 Web 应用程序

在 ASP.NET 中,可以创建 Web 站点和 Web 应用程序。虽然这两个术语都表示提供 ASP.NET、C#和其他文件的一个集合,但它们的处理方式略有区别。

在 Web 站点上,所提供的任何代码都在需要时动态地编译。这一般意味着,代码在第一次访问站点时编译。Web 站点的.cs 文件存储在 Web 服务器上,这表示可以上传这些文件的新版本,以修改这些文件,这会在下一次访问站点时重新编译代码。

相反,Web 应用程序在部署到 Web 服务器中之前编译,且不包含.cs 文件。而是把一个预编译的程序集和 ASP.NET 页面部署到服务器中。

一般情况下,大多数 ASP.NET 程序员都喜欢使用 Web 应用程序模型,而且有一些技术只能用于 Web 应用程序,如 MVC。Web 站点主要用于开发阶段,在开发阶段,其中需要对代码进行快速

修改, 不适合进行完整的部署。

40.1.3 ASP.NET 中的状态管理

ASP.NET 页面的一个重要属性是它们实际上是无状态的。在默认情况下, 在用户的请求之间, 并没有信息存储在服务器上(尽管有一些方法可以完成存储信息的任务, 详见下面的内容)。这初看起来有点奇怪, 因为状态管理对于用户友好的交互会话非常重要。但是, ASP.NET 提供了一种变通方式来解决这个问题, 从而使会话管理几乎完全透明。

简言之, Web 窗体上控件的状态信息(包括文本框中输入的数据、下拉列表中的选项等)存储在隐藏的 `viewstate` 字段中, 这个字段是服务器生成的页面的一部分, 并传递给用户。后续的操作称为回发(`postback`), 例如, 触发需要服务器端处理的事件, 提交窗体数据, 从而把这些信息发送回服务器。在服务器上, 这些信息用于重新填充页面对象模型, 以便作用于它, 就像在本地进行修改一样。

稍后详细介绍这个主题。

40.2 ASP.NET Web 窗体

如前所述, ASP.NET 中的许多功能使用 Web 窗体实现。稍后我们将创建一个简单的 Web 窗体, 开始深入探讨这种技术。但这里先回顾 Web 窗体设计的一些相关要点。

注意一些 ASP.NET 开发人员仅使用文本编辑器(如 Notepad)创建文件。这里不推荐这么做, 因为 Visual Studio 或 Web Developer Express 等 IDE 提供的优点很重要, 只是使用 Notepad 等文本编辑器是创建文件的一种方法, 所以这里值得提及它。如果使用文本编辑器, 则把 Web 应用程序的哪些部分放在什么地方等方面有非常大的灵活性, 例如, 可以把所有代码都组合到一个文件中。为此, 把代码放在 `<script>` 元素中, 在起始 `<script>` 标记中使用两个属性, 如下所示:

```
<script language="c#" runat="server">
    // Server-side code goes here.
</script>
```

这里的 `runat="server"` 属性很重要, 因为它指示 ASP.NET 引擎在服务器上执行这段代码, 而不是把它发送给客户端, 因此可以访问前面讨论的功能丰富的环境。我们可以在服务器端脚本块中放置函数、事件处理程序等。

如果省略 `runat="server"` 属性, 实际上就是在提供客户端代码, 如果使用本章后面要介绍的服务端端编码风格, 就会失败。但是, 可以使用 `<script>` 元素提供 JavaScript 等语言编写的客户端脚本。例如:

```
<script language="JavaScript" type="text/JavaScript">
    // Client-side code goes here; you can also use "vbscript".
</script>
```



`type` 属性是可选的, 但如果需要兼容 HTML 和 XHTML 标准, 它就是必需的。

在页面中添加 JavaScript 代码的功能也包含在 ASP.NET 中,这好像有点奇怪。但是,JavaScript 允许给 Web 页面添加动态的客户端操作,这非常有用。Ajax 编程就允许添加 JavaScript 代码,详见第 41 章。

可以在 Visual Studio 中创建 ASP.NET 文件,这非常重要,因为我们已经熟悉了在这个环境中进行 C#编程。在这个环境中,Web 应用程序的默认项目设置提供了一种比单个.aspx 文件略微复杂的结构,这并不是问题,但它使程序逻辑性更强(更接近编程,而不像 Web 开发人员的风格)。出于上述原因,本章将使用 Visual Studio 进行 ASP.NET 编程(而不是 Notepad)。

.aspx 文件也可以包含括在“<%”和“%>”标记中的代码块。但是,函数定义和变量声明不能放在这里。然后,可以插入代码,当执行到块时就执行这些代码,当输出简单的 HTML 内容时,这很有效。这种行为类似于旧风格的 ASP 页面的行为,但有一个重要的区别:代码是已经编译好的,不是解释性的。这样,性能会好得多。

下面举一个示例。在 Visual Studio 中,使用 File | New | Web Site 菜单项创建一个新的 Web 站点,或者使用 File | New | Project 菜单项创建一个新的 Web 应用程序。在这两种情况下,都会打开一个对话框。在对话框中选择 Visual C#语言类型和对应的模板。对于 Web 站点,还有另一个选项。Visual Studio 可以在几个不同的位置创建 Web 站点:

- 在本地 IIS Web 服务器上
- 在本地磁盘上,它配置为使用内置的 Visual Web Developer Web 服务器
- 可通过 FTP 访问的任意位置
- 支持 Front Page Server Extensions 的远程 Web 服务器上

不必考虑后两个选项,因为它们使用远程服务器,所以现在应选择前两项。一般情况下,IIS 是安装 ASP.NET Web 站点的最佳位置,因为它最接近部署 Web 站点时需要的配置。另一个选项使用内置的 Web 服务器,适合于测试,但有一些限制:

- 只有本地计算机能访问 Web 站点
- 访问 SMTP 等服务受到限制
- 安全模型与 IIS 不同——应用程序运行在当前用户的账户下,而不是运行在 ASP.NET 的特定账户下

最后一点需要澄清,因为在访问数据库或其他需要验证身份的数据时,安全性非常重要。在默认情况下,运行在 IIS 上的 Web 应用程序会在 IIS5Web 服务器的 ASPNET 账户下运行,或在 IIS6 及其更高版本的 NT AUTHORITY\NETWORK SERVICES 账户下运行。如果使用 IIS,这是可配置的;但如果使用内置的 Web 服务器,就不能配置它。

为了便于说明,但同时因为计算机上可能没有安装 IIS,所以可以使用内置的 Web 服务器。由于在这个阶段不必担心安全性,所以这个阶段很简单。

Web 应用程序不会像 Web 站点那样提示选择位置。Web 站点必须在单独的一步中部署到 Web 服务器中——尽管在测试时,它们会使用 VWD Web 服务器。

在 C:\ProCSharp\Chapter40 目录下使用 File System 选项和 Empty Web Site 模板创建一个新的 ASP.NET Web Site,命名为 PCSWebSite1,如图 40-2 所示。

第 41 章

ASP.NET 的功能

本章内容:

- 用户控件和自定义控件
- 母版页
- 站点导航
- 安全性
- 主题
- Web 部件
- ASP.NET AJAX

本章将介绍 ASP.NET 提供的一些技术,以改进 Web 站点和应用程序。这些技术更便于创建 Web 站点和应用程序,添加高级功能,提升用户的体验。

虽然有时内置的 Web 控件功能强大,但不符合具体项目的需求。本章第一部分将介绍对于控件开发人员可用的选项,并汇编一些简单的用户控件,我们还将介绍构建高级控件的基础知识。

接着介绍母版页,这个技术可以为 Web 站点提供模板。使用母版页可以通过 Web 站点和大量重用代码,在 Web 页面上实现复杂的布局。本章还将说明如何使用 Web 导航服务器控件和母版页,在 Web 站点上提供一致的导航。

可以使站点导航特定于用户,只允许某些用户(通过站点注册的用户,或站点管理员)访问某些部分。本章还将介绍 Web 站点的安全性和登录方式,通过那些登录 Web 登录服务器控件很容易实现该功能。

之后介绍一些高级技巧,即提供和选择 Web 站点的主题,如何使用 Web 部件定位和定制页面上的控件,让用户动态地个性化 Web 页面。

最后,本章用大量篇幅介绍 ASP.NET AJAX。这种技术是提升用户体验的一个强大方式,它允许 Web 站点和应用程序独立地更新页面的一部分,简化添加客户端功能的过程,来加快响应的速度。

本章大部分内容将引用一个大型示例 Web 站点,它包含本章和上一章介绍的所有技术。这个 Web 站点 PCSDemoSite (和本章的其他代码)可以从 www.wrox.com 和本书附赠光盘中找到。相关的代码段会在需要时列出,其他代码(大多数是前面介绍的空白样板内容或简单代码)请读者自学。

41.1 用户控件和自定义控件

有时, 给定控件的工作方式并不像所期望的那样, 也可能是一部分代码本用于在多个页面上重用, 但是多个开发人员实现起来却相当复杂。在这些情况下, 就需要构建用户自己的控件。

.NET Framework 使用简单的编程技术, 为自定义控件的创建提供了一个理想的设置。ASP.NET 服务器控件的每个方面都可以随意定制, 包括模板制作、客户端脚本编写等功能。但是, 也不必为所有这些偶然事件编写代码; 控件越简单, 创建就越容易。

另外, .NET 系统中固有的程序集动态查询功能使 Web 站点和应用程序在新 Web 服务器上的安装如同复制包含代码的目录结构一样简单。要使用自己创建的控件, 只需复制包含这些控件的程序集和其他代码即可。甚至可以把频繁使用的控件放在 Web 服务器上一个位于全局程序集缓存(GAC)的程序集中, 这样服务器上的所有 Web 站点和应用程序就可以访问它们了。

本章将介绍两类不同的控件:

- **用户控件**——如何把现有的 ASP.NET 页面转化为控件
- **自定义控件**——如何即组合几个控件的功能、扩展现有的控件, 以及从头新建控件

我们将创建一个简单的控件, 显示一副扑克牌(黑桃、方块、红桃或梅花), 以便轻松地把它嵌入到其他 ASP.NET 页面中, 以此来阐明用户控件的用法。对于自定义控件, 不打算详细介绍, 只探讨基本规则和查找自定义控件的地址。

41.1.1 用户控件

用户控件是用 ASP.NET 代码创建的控件, 就像在标准的 ASP.NET Web 页面中创建控件一样, 不同之处在于一旦创建了用户控件, 就可以轻松地在多个 ASP.NET 页面中重用它们。

例如, 假定已经创建了一个显示数据库中信息的页面, 信息也许是关于订单的, 就不必创建一个固定的页面去显示信息, 而可以把相关代码放到用户控件中, 然后把该控件插入到任意多个不同的 Web 页面中。

此外, 可以给用户控件定义属性和方法, 例如, 可以指定在 Web 页面上显示数据库表时背景色的属性, 或者指定一个方法, 重新进行数据库查询, 以检查数据库中的变化。

下面创建一个简单的用户控件。

一个简单的用户控件

在 Visual Studio 中, 在 C:\ProCSharp\Chapter41 目录下新建一个空白网站——PCUserCWebApp1。生成 Web 站点后, 就可以选择 Website | Add New Item 菜单选项, 添加名称为 PCUserC1.ascx 的 Web 用户控件。

给项目添加的文件的扩展名为 .ascx 和 .ascx.cs, 它们的工作方式与前面的 .aspx 文件非常相似。 .ascx 文件包含 ASP.NET 代码, 看起来与普通的 .aspx 文件非常相似。 .ascx.cs 文件是代码隐藏文件, 它为用户控件定义了自定义代码, 定义的方式与在 .aspx.cs 文件中扩展窗体的方式相同。

与 .aspx 文件相似, 也可以在设计视图或源视图中查看 .ascx 文件。在源视图中查看文件, 可以发现一个重要的区别: .ascx 文件没有显示 HTML 代码, 特别是没有 <form> 元素。原因在于: 用户控件要插入到其他文件的 ASP.NET 窗体中, 因此不需要自己的 <form> 标记。生成的代码如下所示:

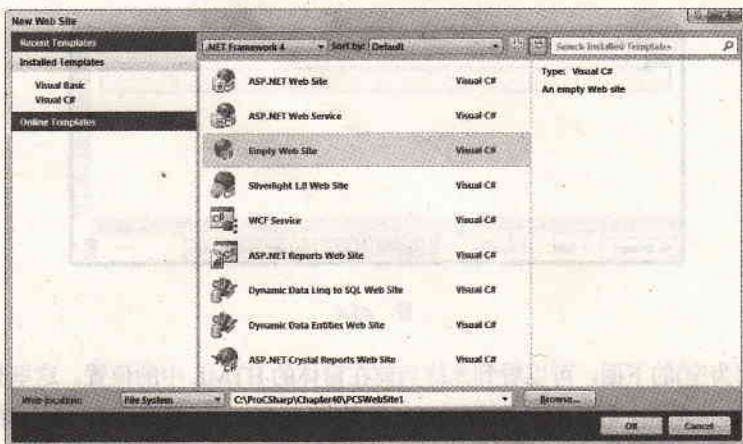


图 40-2



注意, 这里不使用 ASP.NET Web Site 模板, 为了使第一个例子简单一些。ASP.NET Web Site 模板包含 Master Pages 等技术, 下一章会讨论 Master Pages。

稍后, Visual Studio .NET 应构建如下内容:

- 新的解决方案 PCSWebApp1, 包含 C# Web 站点 PCSWebApp1
- Web.config, Web 应用程序的配置文件

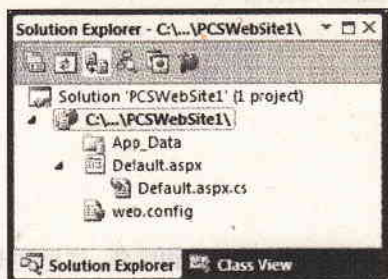


图 40-3

接着需要添加 Web 页面。这可以通过 Website | Add New Item 菜单项完成。选择 Web Form 模板, 不修改其他设置(文件名使用 Default.aspx, 并选中 Place code in separate file 选项), 再单击 Add 按钮。这会添加如下内容:

- Default.aspx, Web 应用程序中的第一个 ASP.NET 页面
- Default.aspx.cs, Default.aspx 的代码隐藏类文件

这些都可以在 Solution Explorer 窗口中看到, 如图 40-3 所示。

可以在设计视图或源代码(HTML)视图中查看.aspx 文件。这与 Windows 窗体(参见第 39 章)完全相同。Visual Studio 中的初始视图是 Default.aspx 的设计视图或源视图(使用左下角的按钮可以切换视图, 或在分隔视图中同时查看这两个视图)。设计视图如图 40-4 所示。

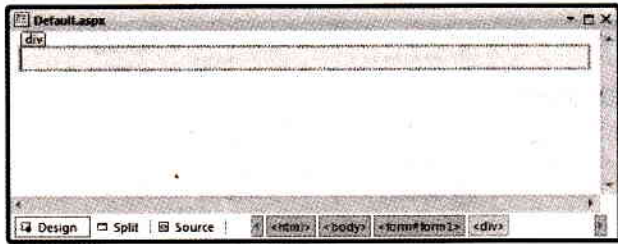


图 40-4

在窗体(当前为空)的下面,可以看到光标当前在窗体的 HTML 中的位置。这里光标在<form>元素的<div>元素中,<form>元素在页面的<body>元素中。<form>元素显示为<form#form1>,用它的id 属性标识该元素,这在后面将会看到。<div>元素也标识在设计视图中。

页面的源代码视图显示了在.aspx 文件中生成的代码:



可从
wrox.com
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
    </form>
</body>
</html>
```

代码段 PCSWebSite1/Default.aspx

如果读者熟悉 HTML 语法,就会觉得这些代码很眼熟。这里列出了 HTML 页面中遵循 XHTML 架构所需的基本代码,并包含几行额外的代码。最重要的元素是<form>,它的id 属性是 form1,这个元素将包含 ASP.NET 代码。这里最重要的属性是 runat。与本节前面的服务器端代码块相同,把这个属性设置为 server,表示窗体的处理将在服务器上进行。如果没有包含这个属性,就不会在服务器端上完成任何处理,窗体也不会执行任何操作。在 ASP.NET 页面中,只有一个服务器端<form>元素。

这段代码中另一项比较重要的内容是顶部的<%@ Page %>标记,它定义了对于 C# Web 应用程序开发人员非常重要的页面特征。首先,Language 属性指定在页面中使用 C#语言,与前面的<script>块相同(Web 应用程序默认的语言是 VB,使用 Web.config 配置文件可以修改这个属性,本章后面将讲述着一点)。下面的 3 个属性 AutoEventWireup、CodeFile 和 Inherits 用于把 Web 窗体关联到代码隐藏文件中的一个类上,这里是 Default.aspx.cs 文件中的部分类 _Default。这就需要讨论 ASP.NET 代码模型的相关内容。

40.2.1 ASP.NET 代码模型

在 ASP.NET 中, 布局(HTML)代码、ASP.NET 控件和 C# 代码用于生成用户看到的 HTML。布局和 ASP.NET 代码存储在.aspx 文件中, 也就是上一节的.aspx 文件。用于定制窗体的行为的 C# 代码包含在.aspx 文件中, 也可以同前面的例子, 放在一个单独的.aspx.cs 文件中, 该文件通常称为代码隐藏文件。

对于 Web 站点项目, 在处理 ASP.NET Web 窗体时, 一般在用户请求页面时, 尽管可以预编译站点, 但此时会发生几个事件:

- ASP.NET 进程检查页面, 确定必须创建什么对象, 以实例化页面对象模型。
- 为页面动态创建一个基类, 包括页面上控件的成员和这些控件的事件处理程序(如按钮单击事件)。
- 包含在.aspx 页面中的其他代码与这个基类合并, 从而构成完整的对象模型。
- 编译所有代码, 并缓存起来, 以备处理随后的请求。
- 生成 HTML, 并把 HTML 返回给用户。

Web 应用程序的处理过程与此类似, 尽管不需要动态编译, 因为根据定义, Web 应用程序是预编译的。

在 Web 站点 PCSWebApp1 中, 为 Default.aspx 生成的代码隐藏文件的内容最初非常少。首先, 查看可能需要在 ASP.NET Web 页面中使用的默认名称空间引用的集合:



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;
```

代码段 PCSWebSite1/Default.aspx.cs

在这些引用的下面, Default.aspx 部分类的定义几乎完全是空的:

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

这里可以使用 Page_Load() 事件处理程序添加在加载页面时需要的任何代码。在添加更多事件处理程序时, 这个类文件会包含越来越多的代码。注意没有把这个事件处理程序关联到页面上的代码, 如前所述, 事件处理程序由 ASP.NET 运行库推断。这要归功于 AutoEventWireup 属性, 把它设置为 false, 表示必须自己在代码中把事件处理程序与事件关联起来。

这个类是一个部分类的定义, 因为前面介绍的过程需要它。在预编译页面时, 会从页面的 ASP.NET 代码中创建一个单独的部分类的定义, 这包括添加到页面中的所有控件。在设计期间, 编译器会推断这个部分类的定义, 以便在代码隐藏中使用 IntelliSense, 来引用页面上的控件。

40.2.2 ASP.NET 服务器控件

因为前面生成的代码并不能完成许多工作,所以下面应添加一些内容。在 Visual Studio 中使用 Web 窗体设计器,它支持拖放操作,其方式与 Windows 窗体设计器相同。

可以添加到 ASP.NET 页面中的控件有 3 种类型:

- **HTML 服务器控件**——这些控件模拟 HTML 元素,HTML 开发人员会很熟悉它们。
- **Web 服务器控件**——这是一组新的控件,其中一些控件的功能与 HTML 控件相同,但它们的属性和其他元素有一个公共的命名模式,便于进行开发,而且可以与相似的 Windows 窗体控件保持一致。还有一些全新的、非常强大的控件,如本章后面所述。Web 服务器控件有几种类型,包括标准控件,如按钮、验证用户输入的验证控件、简化用户管理的登录控件,以及处理数据源的一些较复杂的控件。
- **自定义控件和用户控件**——由开发人员定义的控件,可以用第 41 章介绍的许多方式来创建它们。



下一节列出了许多常用 Web 服务器控件及其使用说明的完整列表。下一章将介绍其他控件。本章没有介绍 HTML 控件。这些控件提供的功能,Web 服务器也能提供,而且 Web 服务器控件为熟悉编程的开发人员提供了一个功能比 HTML 更丰富的环境。如果学会如何使用 Web 服务器控件后,使用 HTML 服务器控件就不难了。更多信息可以参阅清华大学出版社引进并出版的《ASP.NET 4 高级编程——涵盖 C#和 VB.NET(第 7 版)》。

1. 添加 Web 服务器控件

下面在上一节创建的 Web 站点 PCSWebApp1 中,添加两个 Web 服务器控件。所有 Web 服务器控件都以下述 XML 元素类型的方式使用:

```
<asp:controlName runat="server" attribute="value"> Contents </asp:controlName>
```

在上面的代码中,controlName 是 ASP.NET 服务器控件的名称,attribute="value" 是一个或多个属性说明,Contents 根据需要指定控件的内容。一些控件可以使用属性和控件元素的内容来设置属性,如 Label(用于显示简单文本),其中 Text 可以用两种方式指定。其他控件可以使用元素包含模式来定义它们的层次结构,例如,Table(定义一个表)可以包含 TableRow 元素,从而按照声明方式指定表中的行。

注意,控件的语法基于 XML(它们也可以内嵌在非 XML 代码中,如 HTML)。省略闭合标记、表示空元素的“/>”,或者重叠控件,都会产生错误。

最后,再次查看 Web 服务器控件上的 runat="server" 属性。它在这里和其他地方都是基本属性,遗漏这个属性常常也会产生错误,结果将导致 Web 窗体不能正常工作。

(1) 在源代码视图中添加 Web 服务器控件

第一个示例应简单一些。修改 Default.aspx 的 HTML 源视图,代码如下。



可从
wrox.com
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label runat="server" ID="resultLabel" /> <br />
            <asp:Button runat="server" ID="triggerButton" Text="Click Me" />
        </div>
    </form>
</body>
</html>
```

代码段 PCSWebSite1/Default.aspx

这里添加了两个 Web 窗体控件：标签和按钮。



在添加控件时，Visual Studio .NET 的 IntelliSense 会提示代码输入项，这与 C# 代码编辑器相同。另外，如果在拆分的视图中编辑代码，再同步视图，那么在源面板中编辑的元素会在设计面板中突出显示。

回过头来看看设计屏幕，其中已经添加了控件，并用它们的 ID 属性命名(ID 属性常常称为控件的标识符)。与 Windows 窗体相同，可以通过 Properties 窗口访问所有属性、事件等，无论何时进行了修改，在代码或设计中会得到即时反馈。



也可以使用 CSS 属性窗口和其他样式窗口，给控件指定样式。但除非很熟悉 CSS，否则现在不要使用这种技术，而是应关注控件的功能。

我们添加的所有服务器控件都会自动成为窗体的对象模型的一部分。Windows 窗体开发人员可以即时得到这个对象模型，并开始认识到它与 Windows 窗体的类似性。

(2) 添加事件处理程序

要让这个应用程序完成一些工作，应添加单击按钮对应的事件处理程序。可以在 Properties 窗口中为按钮输入一个方法名，也可以双击该按钮，得到默认的事件处理程序。如果双击该按钮，就可以自动添加一个事件处理方法，如下所示：



可从
wrox.com
下载源代码

```
protected void triggerButton_Click(object sender, EventArgs e)
{
}
```

代码段 PCSWebSite1/Default.aspx.cs

把一些代码添加到源 Default.aspx 中, 就可以把事件处理程序关联到按钮上:

```
<div>
    <asp:Label Runat="server" ID="resultLabel" /> <br />
    <asp:Button Runat="server" ID="triggerButton" Text="Click Me"
        onclick="triggerButton_Click" />
</div>
```

其中, onClick 属性告诉 ASP.NET 运行库, 在生成窗体的代码模型时, 把按钮的单击事件包装到 triggerButton_Click() 方法中。

修改 triggerButton_Click() 中的代码(注意因为从 ASP.NET 代码中推断标签控件类型, 所以可以直接从代码隐藏中使用它):

```
protected void triggerButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "Button clicked!";
}
```

(3) 运行项目

下面准备运行它。不需要构建项目, 只需确保保存所有内容, 把 Web 浏览器指向 Web 站点的地址。如果使用 IIS, 这很简单, 因为我们知道指向的 URL。但因为本例使用内置的 Web 服务器, 所以需要启动运行某些项目。最快捷的方式是按 Ctrl+F5 组合键, 启动服务器, 打开一个指向所需的 URL 浏览器。

在运行内置的 Web 服务器时, 系统栏中会显示一个图标。双击这个图标, 会看到 Web 服务器执行的过程, 并可以在需要时停止它, 如图 40-5 所示。

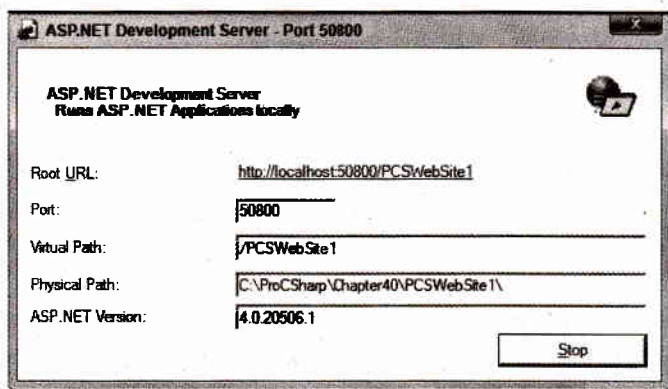


图 40-5

在图 40-5 中, 可以看到 Web 服务器在其上运行的端口和查看已创建的 Web 站点所需的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前, 使用 Page | View Source 命令(在 IE7 中)快速查看一下浏览器接收到的代码。<form>部分应如下所示:

```
<form method="post" action="Default.aspx" id="form1">
    <div>
        <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
            value="/wEPDwUKLTE2MjY5MTY1NWRkzNjRYstd1OK5KcJ9a8/X3pYTHvM=" />
    </div>
```

```

<div>
  <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
    value="/wEWAgK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />
</div>
<div>
  <span id="resultLabel"> </span> <br />
  <input type="submit" name="triggerButton" value="Click Me"
    id="triggerButton" />
</div>
</form>

```

Web 服务器控件生成了简单的 HTML，`<span>`和`<input>`分别代表`<asp.Label>`和`<asp.Button>`。还有一个名为`__VIEWSTATE`的`<input type="hidden">`字段，它把前面提到的窗体状态封装起来。在把窗体回发服务器以重新创建 UI 时使用这些信息，以便服务器可以跟踪改变的信息等。注意`<form>`元素已经为此进行了配置，通过 HTTP POST 操作(在 `method` 中指定)把数据回发给 `Default.aspx`(在 `action` 中指定)，它还被赋予了一个 `form1` 名称。

(4) 修改单击按钮时的源 HTML

在单击按钮并查看文本后，可再次检查源 HTML(清晰起见，下面添加了必要的空格)：

```

<form method="post" action="Default.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0JldHR
        vbiBjbGlja2VkiWRkZExUtMwuSlVTrzMtG7wrmj98tVn7" />
  </div>
  <div>
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
      value="/wEWAgKTpL7LBALHpP+yC0Ymqe9SgScfB2yHTGjnlQKtbudV" />
  </div>
  <div>
    <span id="resultLabel"> Button clicked! </span> <br />
    <input type="submit" name="triggerButton" value="Click Me"
      id="triggerButton" />
  </div>
</form>

```

这次，视图状态的值包含较多信息，因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体中，这确实可能是一个非常长的字符串，但因为这由系统在后台完成，所以我们几乎可以不考虑状态管理，只要回发过程之间保存字段值即可。在视图状态字符串过长时，可以禁用不需要保留状态信息的控件的视图状态。也可以禁用整个页面的视图状态。如果页面不需要在回发过程之间保留状态，就可以禁用整个页面的视图状态，以提高性能。



视图状态详见第 41 章。

为了说明不必手工对 Web 站点进行任何编译，把 `Default.aspx.cs` 中的文本“Button clicked!”改为其他内容，保存文件，再次单击按钮。Web 页面上的文本会随之改变。

2. 控件面板

本节简要介绍某些可用控件,之后把它们组合到一个更完整、更有趣的应用程序中。编辑 ASP.NET 页面时工具箱中的类别如图 40-6 所示。

注意,以下控件的描述使用了“属性”——在所有情况下,ASP.NET 代码中使用的特性与它同名。这里的介绍并不完整,只介绍了最常用的控件和属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation、Login、WebParts 类别和 AJAX Extensions 控件在第 41 章介绍,Reporting 控件可以在 Web 页面上报告信息,包括 Crystal Reports,本书不讨论它。

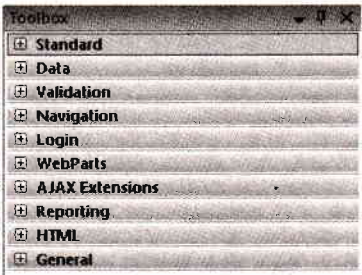


图 40-6

(1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(在这个类别和其他类别中)都继承自 System.Web.UI.WebControls.WebControl,而 System.Web.UI.WebControls.WebControl 又继承自 System.Web.UI.Control。没有使用这个继承性的 Web 服务器控件则直接派生自 Control 类或更专门化的基类,而该基类又最终派生自 Control 类。因此,Web 服务器控件有许多共同的属性和事件,如果需要,就可以使用这些属性和事件。这里不可能介绍所有属性和事件,只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式,这很容易控制,如 ForeColor、BackColor、Font 属性等,也可以使用 CSS(Cascading Style Sheet,级联样式表)类来控制。此时,应在一个独立的文件中,把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS Properties 窗口和样式管理窗口给 CSS 控件设置样式。其他重要属性包括:Width 和 Height,用于设置控件的大小;AccessKey 和 TabIndex,便于用户的交互操作;Enabled,设置控件的功能是否可以在 Web 窗体中激活。

一些控件还包含其他控件,在页面上构建建控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件,或者使用 Parent 可以访问控件的容器。

对于事件,最常用的是继承来的 Load 事件,它执行控件的初始化,PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多,下一章将详细介绍它们,尤其是下一章将介绍更高级的样式设置技术。表 40-1 详细描述了标准 Web 服务器控件。

表 40-1

控 件	说 明
Label	显示简单文本,使用 Text 属性设置和以编程方式修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据,TextChanged 事件可处理回发的选项变化。如果要求进行自动回发(而不是使用按钮),就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本,Click 事件用于响应单击(服务器回发是自动的)。也可以使用 Command 事件响应单击,该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同,但把按钮显示为超链接
ImageButton	显示一幅图像,该图像放大两倍作为一个可单击的按钮,其属性和事件继承自 Button 和 Image

(续表)

控 件	说 明
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地, 用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像, 用 Target 指定要使用的浏览器窗口。因为这个控件没有非标准事件, 所以如果在链接后要执行其他处理, 就应使用 LinkButton
DynamicHyperLink	用于在动态数据网站的预定义位置上呈显 HTML 超链接。动态数据网站参见第 42 章
DropDownList	允许用户选择一个列表项, 可以直接从列表中选择, 也可以输入前面的一或两个字母来选择。使用 Items 属性设置选项列表(这是一个包含 ListItem 对象的 ListItemCollection 类), 使用 SelectedItem 和 SelectedIndex 属性确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变, 因为这个控件也有 AutoPostBack 属性, 所以选项的改变会触发一个回发操作
ListBox	允许用户从列表选择一个或多个选项。把 SelectionMode 设置为 Multiple 或 Single, 可以确定一次选择多少个选项, Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个可以选中或取消选中的复选框。选择的状态存储在布尔属性 Checked 中, 与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回发, CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同, 如 DropDownList
RadioButton	显示一个可以打开或关闭单选按钮。一般情况下, 它们都组合在一组中, 其中只有一个 RadioButton 控件有效。使用 GroupName 属性可以把 RadioButton 控件链接到一组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮, 在这一组中, 一次只能选择一个按钮。其属性和事件与其他列表控件相同, 入 DropDownList
Image	显示一幅图像。使用 ImageUrl 进行图像引用, 如果图像加载失败, 就由 AlternateText 提供对应的文本
ImageMap	类似于 Image, 但在用户单击图像中的一个或多个热点时, 可以指定要触发的动作。要执行的动作可以是回发给服务器或重定向到另一个 URL 上。热点由派生自 HotSpot 的内嵌控件提供, 如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、TableRow 和 TableCell, 或者使用 TableRowCollection 类型的 Rows 属性编程指定表中的行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样, 这个控件有几个只能用于表的样式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同, 这个控件有一个 Click 事件, 用于确定用户在回发期间单击了哪个选项。其他属性和事件与 DropDownList 相同
HiddenField	出于任何原因, 用于提供隐藏的字段, 以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能, 但没有样式属性, 因为它派生自 Control, 而不是 WebControl。使用 Text 属性可以设置要显示的文本

(续表)

控 件	说 明
Calendar	允许用户从显示的图形日历中选择一个日期。这个控件有许多与样式相关的属性,但其基本功能的实现要使用 <code>SelectedDate</code> 和 <code>VisibleDate</code> 属性(其类型是 <code>System.DateTime</code>)来访问用户选择的日期和月份,并显示出来(总是包含 <code>VisibleDate</code>)。其关联的关键事件是 <code>SelectionChanged</code> 。这个控件的回发是自动的
AdRotator	顺序显示几幅图像。在每次服务器往返行程后,显示另一幅图像。使用 <code>AdvertisementFile</code> 属性指定描述图像的 XML 文件, <code>AdCreated</code> 事件在每幅图像发回之前执行处理操作。也可以使用 <code>Target</code> 属性在单击一幅图像时指定一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 <code>Browse</code> 按钮,以便选择要上传的文件。用户选择文件之后,就可以查看 <code>HasFile</code> 属性确定是否选择了文件,然后使用代码隐藏中的 <code>SaveAs()</code> 方法执行文件上传操作
Wizard	这个高级控件用于简化用户一次在几个页面中输入数据的常见任务。可以给向导添加多个步骤,按顺序或不按顺序显示给用户,并依赖此控件来维护状态等
Xml	这是一个更复杂的文本显示控件,用于显示用 XSLT 样式表传输的 XML 内容,这些 XML 内容是使用 <code>Document</code> 、 <code>DocumentContent</code> 或 <code>DocumentSource</code> 属性中的一个设置(取决于原始 XML 的格式)的, XSLT 样式表(可选)使用 <code>Transform</code> 或 <code>TransformSource</code> 来设置
MultiView	这个控件包含一个或多个 <code>View</code> 控件,每次只呈现一个 <code>View</code> 控件。当前显示的视图用 <code>ActiveViewIndex</code> 指定,如果视图发生改变(可能因为单击了当前视图上显示的 <code>Next</code> 链接),就可以使用 <code>ActiveViewChanged</code> 事件检测出来
Panel	添加其他控件的容器。可以使用 <code>HorizontalAlign</code> 和 <code>Wrap</code> 指定内容如何安排
Placeholder	这个控件不显示任何输出,但可以方便地把其他控件组合在一起,或者用编程方式把控件添加到给定的位置。包含的控件可以使用 <code>Controls</code> 属性来访问
View	控件的容器,类似于 <code>Placeholder</code> ,但主要用作 <code>MultiView</code> 的子控件。使用 <code>Visible</code> 属性可以指定是否显示给定的 <code>View</code> ,使用 <code>Activate</code> 和 <code>Deactivate</code> 事件检测激活状态的变化
Substitution	指定一部分不与其他输出一一起缓存的 Web 页面,这是一个与 ASP.NET 缓存行为相关的高级主题,本书不涉及
Localize	与 <code>Literal</code> 相同,但允许使用项目资源指定要在不同区域显示的文本,使文本本地化

(2) 数据 Web 服务器控件

数据 Web 服务器控件分为 3 类:

- 数据源控件(`SqlDataSource`、`AccessDataSource`、`LinqDataSource`、`EntityDataSource`、`ObjectDataSource`、`XmlDataSource` 和 `SiteMapDataSource`)
- 数据显示控件(`GridView`、`DataList`、`DetailsView`、`FormView`、`Repeater` 和 `DataPager`)
- 动态数据控件(`DynamicControl` 和 `DynamicDataManager`)

一般情况下,应把其中一个(不可见的)数据源控件放在页面上,以链接数据源;然后添加一个

绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有数据源控件要么派生自 System.Web.UI.DataSource 要么派生自 System.Web.UI.HierarchicalDataSource。这类提供的方法，如 GetView()或 GetHierarchicalView()，可以访问内部数据视图，还可以设置样式。

表 40-2 描述了各种数据源控件。注意本节没有探讨属性，这主要是因为这些控件最好通过图形方式或者向导来配置。本章后面将使用这些控件，使读者更好地理解它们的工作方式。

表 40-2

控 件	说 明
SqlDataSource	用作 SQL Server 数据库中存储的数据的管道。把这个控件放在页面上，就可以使用数据显示控件操作 SQL Server 数据。本章后面将使用这个控件
AccessDataSource	与 SqlDataSource 相似，但它处理存储在 Microsoft Access 数据库中的数据
LinqDataSource	这个控件可以处理支持 LINQ 数据模型的对象
EntityDataSource	这个控件可以处理 ADO.NET Entity 数据模型的对象
ObjectDataSource	这个控件可以处理存储在自己创建的对象中的数据，这些对象可能组合在一个集合类中。这是把自定义对象模型提供给 ASP.NET 页面的快捷方式
XmlDataSource	可以绑定到 XML 数据上。它可以绑定导航控件，如 TreeView。利用这个控件，根据需要还可以使用 XSL 样式表传输 XML 数据
SiteMapDataSource	可以绑定到层次站点地图数据上。详见第 41 章的导航 Web 服务器控件

接着是数据显示控件，如表 40-3 所示。其中几个控件可用于满足各种需求。一些控件的功能比另外一些控件更齐全，但我们常常使用最简单的控件(例如，当不需要编辑数据项时)。

表 40-3

控 件	说 明
GridView	以数据行的格式显示多个数据项(如数据库中的行)，其中每一行包含反映数据字段的列。利用这个控件的属性，可以选择、排序和编辑数据项
DataList	显示多个数据项，可以为每一项提供模板，以任意指定的方式显示数据字段。与 GridView 相同，可以选择、排序和编辑数据项
DetailsView	以表格形式显示单个数据项，表中的每一行都与一个数据字段相关。这个控件可以添加、编辑和删除数据项
FormView	使用模板显示一个数据项。与 DetailsView 相同，这个控件也可以添加、编辑和删除数据项
ListView	与 DataList 相似，但支持使用 DataPager 进行分页和更多的模板功能
Repeater	与 DataList 相似，但不能选择和编辑数据
DataPager	允许 ListView 控件分页显示
Chart	在图表中显示数据，如柱状图或饼图，这个控件参见下一章

动态数据控件参见第 42 章。

(3) 验证 Web 服务器控件

验证控件可以在不编写任何代码的前提下(在大多数情况下)下验证用户的输入。只要启动回发操作，每个验证控件就会检查控件是否有效，并相应地改变 IsValid 属性的值。如果这个属性是 false，被验证控件的用户输入就没有通过验证。包含所有控件的页面也有一个 IsValid 属性——如果页面中任意个有效性验证控件的 IsValid 属性设置为 false，该页面的 IsValid 属性也就是 false。可以在服务器端的代码上检查这个属性，并对它进行操作。

验证控件还有另一个功能。它们不仅可以在运行期间验证控件的有效性，还可以自动给用户输出有帮助的提示，把 ErrorMessage 属性设置为希望的文本意味着，在用户试图回发无效的数据时，就会看到这些文本。

存储在 ErrorMessage 中的文本可以在验证控件所在的位置输出，也可以和页面上来自所有其他验证控件的消息一起输出到一个独立的位置。第二种行为可以使用 ValidationSummary 控件来获得，并根据需要显示所有错误消息和附加文本。

在支持该控件的浏览器上，验证控件甚至可以生成客户端的 JavaScript 函数，来简化验证行为。在某些情况下，这意味着不会有回发，因为验证控件在某些环境下禁止回发，输出错误消息，而与服务器无关。

因为所有验证控件都继承自 BaseValidator，所以它们共享几个重要的属性。最重要的是上一节讨论的 ErrorMessage 属性；ControlToValidate 属性也比较重要，它指定要验证的控件的编程 ID。另一个重要的属性是 Display，它确定是把文本放在验证汇总的位置上(如果把该属性设置为 none)，还是放在验证控件的位置上。也可以给错误消息留一些空间，即使在不显示这些错误信息时(把 Display 设置为 Static)；或者按照需要给这些信息动态分配空间，这会使页面的内容有轻微的改变(把 Display 设置为 Dynamic)。表 40-4 描述了各个验证控件。

表 40-4

控 件	说 明
RequiredFieldValidator	用来检查用户是否在 TextBox 等控件中输入数据
CompareValidator	用于检查输入的数据是否满足简单的要求。利用一个运算符集合，通过 Operator 和 ValueToCompare 属性进行验证。Operator 的值可以是 Equal、GreaterThan、GreaterThanEqual、LessThan、LessThanEqual、NotEqual 或 DataTypeCheck。DataTypeCheck 可以比较 ValueToCompare 的数据类型和控件中要验证的数据。ValueToCompare 是一个字符串属性，但根据其内容可以把它解释为另一种数据类型。要进一步比较控件，可以把 Type 属性设置为 Currency、Date、Double、Integer 或 String
RangeValidator	验证控件中的数据是否落在 MaximumValue 和 MinimumValue 属性值之间，其 Type 属性类似于 CompareValidator
RegularExpressionValidator	根据存储在 ValidationExpression 中的正则表达式验证字段的内容，可以用于验证邮政编码、电话号码、IP 号码等序列

(续表)

控 件	说 明
CustomValidator	使用自定义函数验证控件中的数据。ClientValidationFunction 指定用于验证控件的客户端函数(但这表示不能使用 C#)。这个函数应返回一个布尔类型的值,表示验证是否成功。另外,还可以使用 ServerValidate 事件指定用于验证数据的服务器端函数。这个函数是一个布尔类型的事件处理程序,其参数是一个包含要验证数据的字符串,而不是 EventArgs 参数。如果验证成功,就返回 true; 否则返回 false
DynamicValidator	这个验证控件用于给动态数据站点提供业务逻辑验证,参见第 42 章
ValidationSummary	为所有设置了 ErrorMessage 的验证控件显示验证错误。通过设置 DisplayMode (BulletList、List 或 SingleParagraph) 和 HeaderText 属性,显示的内容可以格式化;把 ShowSummary 设置为 false,就会禁止显示;把 ShowMessageBox 设置为 true,内容就会显示在弹出的消息框中

3. 服务器控件的事件登记示例

在这个示例中,要为一个 Web 应用程序(会议室登记工具)创建构架。与本书的其他示例一样,可以从 Wrox 网站 www.wrox.com 上或者本书附赠光盘中找到示例应用程序和代码。现在仅介绍前端和简单的事件处理,后面将使用 ADO.NET 和数据绑定扩展这个示例,使之包含服务器端的业务逻辑。

要创建的 Web 窗体包含的字段有:用户名、事件名、会议室和与会者,以及可从中选择日期的日历(假定本例的作用是处理日常事件)。除了日历外,所有字段都使用验证控件,只有日历在服务器端验证,并提供一个默认日期,以防没有输入日期。

为了测试用户界面,窗体也提供了一个 Label 控件,使用它可以显示提交的结果。

首先,在 Visual Studio .NET 中,在 C:\ProCSharp\Chapter40\目录下创建一个新的空白 Web 站点,把它命名为 PCSWebSite2。然后,使用默认设置添加一个新的 Web 窗体,修改 Default.aspx 文件中的代码,如下所示:



```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Meeting Room Booker</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h1 style="text-align: center;">
                Enter details and set a day to initiate an event.
            </h1>
        </div>
```

代码段 PCSWebSite2/Default.aspx

页面的标题用 HTML 标记<h1>括起来,以得到大型的标题样式的文本,之后,窗体的主体放在 HTML 标记<table>中。可以使用一个 Web 服务器控件表格,但这会导致不必要的复杂性,因为使用表格的目的仅仅是为了格式化显示而已,不是用于动态的 UI 元素。时刻要牢记一个要点:在设计 Web 窗体时,不要添加不必要的服务器控件。这个表格有 3 列,第 1 列包含简单的文本标签,第 2 列包含对应于文本标签的 UI 字段(以及这些字段的验证控件),第 3 列包含一个日历控件,可以从中选择日期,这个控件跨 4 行。第 5 行包含一个跨越所有列的提交按钮,第 6 行包含一个 ValidationSummary 控件,在需要时可以显示错误信息(所有其他验证控件都设置了 Display="None",因为它们都使用这个汇总来显示错误)。在表格的下面是一个简单的标签,现在使用它可以显示结果,以后我们还将添加数据库访问。

```
<div style="text-align: center;">
  <table style="text-align: left; border-color: #000000;
    border-width: 2px; background-color: #fff99e;" cellspacing="0"
    cellpadding="8" rules="none" width="540">
    <tr>
      <td valign="top">
        Your Name: </td>
      <td valign="top">
        <asp:TextBox ID="nameBox" Runat="server" Width="160px" />
        <asp:RequiredFieldValidator ID="validateName" Runat="server"
          ErrorMessage="You must enter a name."
          ControlToValidate="nameBox" Display="None" />
      </td>
      <td valign="middle" rowspan="4">
        <asp:Calendar ID="calendar" Runat="server" BackColor="White" />
      </td>
    </tr>
    <tr>
      <td>
      </td>
    </tr>
    <tr>
      <td valign="top">
        Event Name: </td>
      <td valign="top">
        <asp:TextBox ID="eventBox" Runat="server" Width="160px" />
        <asp:RequiredFieldValidator ID="validateEvent" Runat="server"
          ErrorMessage="You must enter an event name."
          ControlToValidate="eventBox" Display="None" />
      </td>
    </tr>
    <tr>
      <td>
      </td>
    </tr>
  </table>
```

这个文件中的大多数 ASP.NET 代码都非常简单,许多代码只要浏览一遍就可以理解。特别要注意的是,在代码中用于选择会议室和多个与会者的列表项是如何附加到控件中的:

```
<tr>
  <td valign="top">
    Meeting Room: </td>
  <td valign="top">
    <asp:DropDownList ID="roomList" Runat="server" Width="160px">
      <asp:ListItem Value="1"> The Happy Room </asp:ListItem>
      <asp:ListItem Value="2"> The Angry Room </asp:ListItem>
      <asp:ListItem Value="3"> The Depressing
        Room </asp:ListItem>
      <asp:ListItem Value="4"> The Funked Out
        Room </asp:ListItem>
    </asp:DropDownList>
    <asp:RequiredFieldValidator ID="validateRoom" Runat="server"
```

```

        ErrorMessage="You must select a room."
        ControlToValidate="roomList" Display="None" />
    </td>
</tr>
<tr>
    <td valign="top">
        Attendees: </td>
    <td valign="top">
        <asp:ListBox ID="attendeeList" Runat="server" Width="160px"
            SelectionMode="Multiple" Rows="6">
            <asp:ListItem Value="1"> Bill Gates </asp:ListItem>
            <asp:ListItem Value="2"> Monica Lewinsky </asp:ListItem>
            <asp:ListItem Value="3"> Vincent Price </asp:ListItem>
            <asp:ListItem Value="4"> Vlad the Impaler </asp:ListItem>
            <asp:ListItem Value="5"> Iggy Pop </asp:ListItem>
            <asp:ListItem Value="6"> William
                Shakespeare </asp:ListItem>
        </asp:ListBox>
    </td>
</tr>

```

其中把 `Listitem` 对象与两个 Web 服务器控件关联起来。这些对象本身并不是 Web 服务器控件(它们仅继承自 `System.Object`)，因此不需要对它们使用 `Runat="server"`。在处理页面时，使用 `<asp:ListItem>` 项创建 `Listitem` 对象，再把这些对象添加到父列表控件的 `Items` 集合中，这便于初始化列表，而无需自己编写代码(必须创建一个 `ListitemCollection` 对象，添加几个 `Listitem` 对象，再把集合传递给列表控件)。当然，也可以通过编程方式完成这些工作：

```

        <asp:RequiredFieldValidator ID="validateAttendees" Runat="server"
            ErrorMessage="You must have at least one attendee."
            ControlToValidate="attendeeList" Display="None" />
    </td>
</tr>
<tr>
    <td align="center" colspan="3">
        <asp:Button ID="submitButton" Runat="server" Width="100%"
            Text="Submit meeting room request" />
    </td>
</tr>
<tr>
    <td align="center" colspan="3">
        <asp:ValidationSummary ID="validationSummary" Runat="server"
            HeaderText="Before submitting your request:" />
    </td>
</tr>
</table>
</div>
<div>
    <p>
        Results:
        <asp:Label Runat="server" ID="resultLabel" Text="None." />
    </p>
</div>
</form>
</body>
</html>

```

在设计视图中，创建的窗体如图 40-7 所示。这是一个功能全面的 UI，它可以在服务器请求之间维护它自己的状态，并验证用户输入。上述代码非常简洁，实际上，我们几乎不需要做什么工作，至少对于这个示例是这样，而只需把按钮单击事件与提交按钮关联起来。

图 40-7

实际并非如此。到目前为止我们没有验证日历控件。所需做的全部工作是给它设置一个初始值。在页面的 `Page_Load()` 事件处理程序中，在代码隐藏文件中为页面设置该值：



可从
wrox.com
下载源代码

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        calendar.SelectedDate = DateTime.Now;
    }
}
```

代码段 PCSWebSite2/Default.aspx.cs

我们把今天的日期作为初始值。注意首先检查页面的 `IsPostBack` 属性，确认是否会把调用 `Page_Load()` 作为回发操作的结果。如果正在进行回发，这个属性就应是 `true`，不必改变选中的日期(毕竟，我们不希望丢失用户的选择)。

要添加按钮单击处理程序，只需双击该按钮，并添加如下代码：

```
protected void submitButton_Click(object sender, EventArgs e)
{
    if (this.IsValid)
    {
        resultLabel.Text = roomList.SelectedItem.Text +
            " has been booked on " +
            calendar.SelectedDate.ToLongDateString() +
            " by " + nameBox.Text + " for " +
            eventBox.Text + " event. ";
        foreach (ListItem attendee in attendeeList.Items)
        {

```

```

        if (attendee.Selected)
        {
            resultLabel.Text += attendee.Text + ", ";
        }
    }
    resultLabel.Text += " and " + nameBox.Text +
        " will be attending.";
}
}

```

把 resultLabel 控件的 Text 属性设置为结果字符串，它将显示在主表格的下方。在 IE 中，这个提交结果应如图 40-8 所示，除非有错误，否则在这种情况下就应激活 ValidationSummary，如图 40-9 所示。

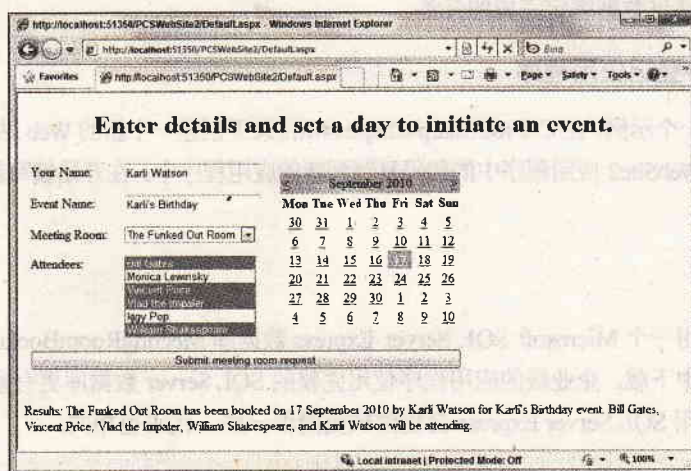


图 40-8

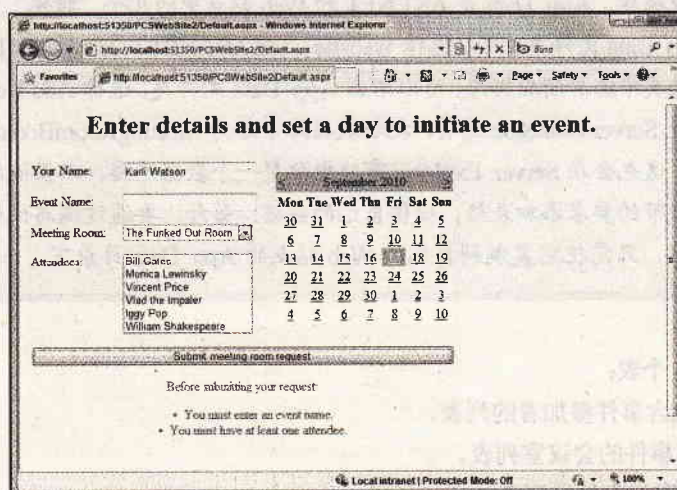


图 40-9

表 40-5

列	类 型	说 明
ID	Identity, 主键	参加者的身份标识号码
Name	varchar, 必选, 50 个字符	参加者的姓名
Email	varchar, 可选, 50 个字符	参加者的电子邮件地址

该数据库包含 20 个参加者的信息, 他们都有电子邮件地址。在一个比较高级的应用程序中, 电子邮件会在登记事件时自动发送给已登记的参加者, 我们把这个任务留给您来完成, 其中使用的技巧可以在本书的其他地方找到。

(2) 会议室

Rooms 表包含表 40-6 所示的列。

表 40-6

列	类 型	说 明
ID	Identity, 主键	房间标识号码
Room	varchar, 必选, 50 个字符	房间名

数据库提供了 20 条记录。

(3) 事件

Events 表包含表 40-7 所示的列。

表 40-7

列	类 型	说 明
ID	Identity, 主键	会议标识号码
Name	varchar, 必选, 255 个字符	会议名称
Room	int, 必选	事件对应的会议室 ID
AttendeeList	Text, 必选	参加者姓名列表
EventDate	DateTime, 必选	会议日期

可下载的数据库提供了几个事件。

2. 数据库的绑定

要绑定数据的两个控件是 attendeeList 和 roomList。在此之前, 需要添加 Web 服务器控件 SqlDataSource, 以映射到要在 MeetingRoomBooker.mdf 数据库中访问的表。最快捷的方式是把它们从工具箱拖放到 Web 窗体 Default.aspx 上, 通过配置向导配置它们。图 40-10 显示了如何为 SqlDataSource 控件访问这个向导。

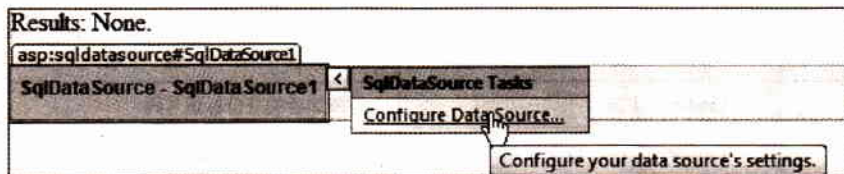


图 40-10

从数据源配置向导的第一个页面上，需要选择前面创建的数据库连接。接着，选择把连接字符串另存为 MRBConnectionString，然后从数据库的 Attendees 表中选择*(所有字段)。

之后，把 SqlDataSource 控件的 ID 改为 MRBAttendeeData。还需要添加并配置另外两个 SqlDataSource 控件，以便分别用 MRBRoomData 和 MRBEventData 的 ID 值从 Rooms 表和 Events 表中获得数据。这两个控件可以使用前面保存的 MRBConnectionString 进行连接。

添加这些数据源之后，您将会发现在代码中窗体的语法非常简单：



可从
wrox.com
下载源代码

```
<asp:SqlDataSource ID="MRBAttendeeData" runat="server"
    ConnectionString=" <%= ConnectionStrings:MRBConnectionString %> "
    SelectCommand="SELECT * FROM [Attendees]"> </asp:SqlDataSource>
<asp:SqlDataSource ID="MRBRoomData" runat="server"
    ConnectionString=" <%= ConnectionStrings:MRBConnectionString %> "
    SelectCommand="SELECT * FROM [Rooms]"> </asp:SqlDataSource>
<asp:SqlDataSource ID="MRBEventData" runat="server"
    ConnectionString=" <%= ConnectionStrings:MRBConnectionString %> "
    SelectCommand="SELECT * FROM [Events]"> </asp:SqlDataSource>
```

代码段 PCSWebSite3/Default.aspx

使用的连接字符串的定义在 web.config 文件中，本章后面将详细探讨这个文件。

接着，设置 roomList 和 attendeeList 控件的数据绑定属性。对于 roomList 控件，需要的设置如下：

- DataSourceID —— MRBRoomData
- DataTextField —— Room
- DataValueField —— ID

同样，对于 attendeeList 控件，需要的设置如下：

- DataSourceID —— MRBAttendeeData
- DataTextField —— Name
- DataValueField —— ID

也可以从代码中删除这些控件已有的硬编码列表项。

现在运行这个应用程序，从数据绑定控件中得到所有可用的参加者和会议室数据。稍后使用 MRBEventData 控件。

3. 定制日历控件

在把会议添加到数据库中之前，先修改一下日历的显示方式。最好用另一种颜色显示登记之前的所有日期，以防此类日期可被选中。这要求修改在日历中设置日期的方式，以及日期单元格的显示方式。

首先是日期选择。有 3 个地方需要查看事件登记的日期，并修改相应选项：一是在 `Page_Load()` 方法中设置初始日期时；二是在用户试图从日历中选择日期时；三是登记一个事件，并设置一个新的日期，以防用户在选择新日期前，在同一天连续登记两个事件。因为这是很常见的功能，所以也可以创建一个私有方法来执行这个计算。这个方法应接受一个试用日期作为参数，并返回要使用的日期，该日期可以与试用日期相同，也可以是试用日期之后的下一个可用日期。

在添加这个方法之前，需要让代码访问 `Events` 表中的数据。为此可以使用 `MRBEventData` 控件，因为这个控件可以填充 `DataView`。所以，添加下面的私有成员和属性(可能还需要为 `System.Data` 导入一个名称空间，才能使这些代码正常运行)：



```
private DataView eventData;
private DataView EventData
{
    get
    {
        if (eventData == null)
        {
            eventData =
                MRBEventData.Select(new DataSourceSelectArguments())
                as DataView;
        }
        return eventData;
    }
    set
    {
        eventData = value;
    }
}
```

代码段 PCSWebSite3/Default.aspx.cs

`EventData` 属性用需要的数据填充 `eventData` 成员，其结果缓存起来，供以后使用。这里使用 `SqlDataSource.Select()` 方法获得 `DataView`。

把这个 `GetFreeDate()` 方法添加到代码隐藏文件中：

```
private DateTime GetFreeDate(DateTime trialDate)
{
    if (EventData.Count > 0)
    {
        DateTime testDate;
        bool trialDateOK = false;
        while (!trialDateOK)
        {
            trialDateOK = true;
            foreach (DataRowView testRow in EventData)
            {
                testDate = (DateTime)testRow["EventDate"];
                if (testDate.Date == trialDate.Date)
                {
                    trialDateOK = false;
                    trialDate = trialDate.AddDays(1);
                }
            }
        }
    }
}
```

```

    }
}
return trialDate;
}

```

这段简单的代码使用 `EventData` 对应的 `DataRowView` 提取事件数据。首先看看一般情况：没有登记任何会议，此时返回该试用日期，以确认该日期，接着对 `Event` 表中的日期进行迭代，把该日期与试用日期比较。如果找到一个匹配的日期，就给试用日期加一天，并执行另一次搜索。

从 `DataTable` 中提取数据相当简单：

```
testDate = (System.DateTime)testRow["EventDate"];
```

把列数据强制转换为 `System.DateTime`，这样会更精确。

使用 `getFreeDate()` 方法的第一个地方是在 `Page_Load()` 方法后面。这表示只需对设置 `SelectedDate` 日历属性的代码稍加修改：

```

if (!this.IsPostBack)
{
    DateTime trialDate = DateTime.Now;
    calendar.SelectedDate = GetFreeDate(trialDate);
}

```

接着需要响应日历上的日期选择。为此，需要先为日历的 `SelectionChanged` 事件添加一个事件处理程序，并强制检查现有事件的日期。双击设计器中的日历，添加如下代码：

```

protected void calendar_SelectionChanged(object sender, EventArgs e)
{
    DateTime trialDate = calendar.SelectedDate;
    calendar.SelectedDate = GetFreeDate(trialDate);
}

```

上述代码与 `Page_Load()` 的代码相同。

执行这种检查的第三个地方是响应按下的登记按钮。后面会解释它，因为后面进行了许多改变。

接着为日历的日期单元格着色，以表示现存的事件。为此，需要给 `calendar` 对象的 `DayRender` 事件添加一个事件处理程序。每次显示一个日期时，都会触发这个事件，并允许通过处理程序函数中接收到的 `DayRenderEventArgs` 参数的 `Cell` 和 `Date` 属性，访问要显示的 `cell` 对象和这个单元格的日期。我们需要比较当前呈现的单元格中的日期和 `eventTable` 对象中的日期，如果存在一个匹配的日期，就可以使用 `Cell.BackColor` 属性为单元格着色：

```

protected void calendar_DayRender(object sender, DayRenderEventArgs e)
{
    if (EventData.Count > 0)
    {
        DateTime testDate;
        foreach (DataRowView testRow in EventData)
        {
            testDate = (DateTime)testRow["EventDate"];
            if (testDate.Date == e.Day.Date)
            {
                e.Cell.BackColor = System.Drawing.Color.Red;
            }
        }
    }
}

```

这里使用红色，得到屏幕图 40-11，其中 6 月的 12、15、22 日(2010 年)都有事件。

June 2010						
Mon	Tue	Wed	Thu	Fri	Sat	Sun
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11

图 40-11

添加了日期选择逻辑后，就不可能选择显示为红色的一天。如果试图选择这样的日期，就会选择该日期后面的某一天。例如，在图 40-11 的日历中选择 6 月 15 日，就会选择 6 月 16 日。

4. 给数据库添加事件

submitButton_Click()事件处理程序目前从事件特征中组合一个字符串，并在 resultLabel 控件中显示它。要给数据库添加一个事件，需要在一个 SQL INSERT 查询中重新格式化已创建的字符串，并执行它。



注意，在正在使用的开发环境中，不必过多地考虑安全性。通过 Web 站点解决方案添加一个 SQL Server 2008 Express 数据库，把 SqlDataSource 控件配置为使用该数据库，会自动给用户提供一个连接字符串，它可用于写入数据库。在比较高级的场合下，可以使用其他账户访问资源，例如，域账户用于访问网络上其他地方的 SQL Server 实例。ASP.NET 中有这个功能(通过模拟、COM+服务或其他方式获得)，但它超出了本书的范围。在大多数情况下，正确地配置连接字符串和需要完成的任务同样复杂。

因此下面的许多代码都很熟悉：

```
protected void submitButton_Click(object sender, EventArgs e)
{
    if (this.IsValid)
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        foreach (ListItem attendee in attendeeList.Items)
        {
            if (attendee.Selected)
            {
                sb.AppendFormat("{0} ({1}),", attendee.Text, attendee.Value);
            }
        }
    }
}
```

```

sb.AppendFormat(" and {0}", nameBox.Text);
string attendees = sb.ToString();
try
{
    System.Data.SqlClient.SqlConnection conn =
        new System.Data.SqlClient.SqlConnection(
            ConfigurationManager.ConnectionStrings[
                "MRBConnectionString"].ConnectionString);
    System.Data.SqlClient.SqlCommand insertCommand =
        new System.Data.SqlClient.SqlCommand("INSERT INTO [Events] "
            + "(Name, Room, AttendeeList, EventDate) VALUES (@Name, "
            + "@Room, @AttendeeList, @EventDate)", conn);
    insertCommand.Parameters.Add(
        "Name", SqlDbType.VarChar, 255).Value = eventBox.Text;
    insertCommand.Parameters.Add(
        "Room", SqlDbType.Int, 4).Value = roomList.SelectedValue;
    insertCommand.Parameters.Add(
        "AttendeeList", SqlDbType.Text, 16).Value = attendees;
    insertCommand.Parameters.Add(
        "EventDate", SqlDbType.DateTime, 8).Value =
        calendar.SelectedDate;
}

```

这里最有趣的是如何使用下面的语法访问前面创建的连接字符串:

```
ConfigurationManager.ConnectionStrings["MRBConnectionString"].ConnectionString
```

ConfigurationManager 类(需要为 System.Configuration 导入名称空间)可以访问所有已分类的配置信息, 它们都存储在 Web 应用程序的 Web.Config 配置文件中。该文件详见本章后面的内容。

创建 SQL 命令后, 就可以使用它插入新事件:

```

conn.Open();
int queryResult = insertCommand.ExecuteNonQuery();
conn.Close();

```

ExecuteNonQuery()方法返回一个整数, 该整数表示查询会影响表中的多少行。如果它等于 1, 插入就是成功的。此时把一条成功的消息放在 resultLabel 中, 清除 eventData, 因为它现在已过期了。把日历选项改为一个新的、空闲日期。因为 GetFreeDate()方法涉及使用 eventData, 而 eventData 属性在自身没有数据时会自动刷新它自己, 所以会刷新存储的事件数据:

```

if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    eventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
}

```

如果 ExecuteNonQuery()方法返回的数字不是 1, 就会有问题。如果返回的数字不是 1, 本例中的代码就会抛出一个异常。该异常会在一般的 catch 块中捕获, 该 catch 块位于数据库访问代码中。

该 catch 块仅在 resultLabel 中显示一个常见的故障通知:

```

else
{

```

```

        throw new System.Data.DataException("Unknown data error.");
    }
}
catch
{
    resultLabel.Text = "Event not added due to DB access "
        + "problem.";
}
}
}

```

这样就完成了能识别数据的事件登记应用程序的版本。

40.3.2 数据绑定的更多内容

如前所述, Web 服务器控件有几个处理数据显示的控件: GridView、DataList、DetailsView、FormView 和 Repeater。在把数据输出到网页上时, 这些控件都非常有用, 因为它们会自动执行许多任务, 否则将需要编写许多代码。

首先, 介绍如何使用这些控件, 在 PCSWebSite3 的底部显示一个事件列表。

把一个 GridView 控件从工具箱拖放到 Default.aspx 的底部, 选择前面添加的 MRBEventData 数据源, 如图 40-12 所示。

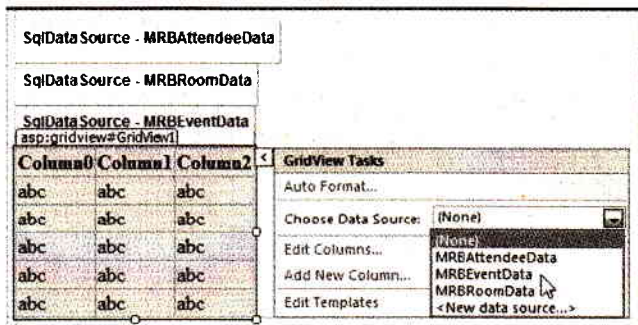


图 40-12

接着单击 Refresh Schema 按钮, 这就是在窗体底部显示事件列表所需做的全部工作, 现在尝试查看 Web 站点, 就会看到对应事件, 如图 40-13 所示。

ID	Name	Room	AttendeeList	EventDate
1	My Birthday	4	Iggy Pop (5), Sean Connery (7), Albert Einstein (10), George Clooney (14), Jules Verne (18), Robin Hood (20), and Karl Watson	17/09/2010 00:00:00
2	Dinner	1	Bill Gates (1), Monika Lewinsky (2), and Bruce Lee	05/08/2010 00:00:00
3	Discussion of darkness	6	Vlad the Impaler (4), Myra Hindley (13), and Beelzebub	29/10/2010 00:00:00
4	Christmas with Pals	9	Dr Frank N Furter (11), Bobby Davro (15), John F Kennedy (16), Stephen King (19), and Karl Watson	25/12/2010 00:00:00
5	Escape	17	Monika Lewinsky (2), Stephen King (19), and Spartacus	10/05/2010 00:00:00
6	Planetary Conquest	14	Bill Gates (1), Albert Einstein (10), Dr Frank N Furter (11), Bobby Davro (15), and Darth Vader	15/06/2010 00:00:00
7	Homecoming Celebration	7	William Shakespeare (6), Christopher Columbus (12), Robin Hood (20), and Ulysses	22/06/2010 00:00:00
8	Dalek Reunion Ball	12	Roger Moore (8), George Clooney (14), Bobby Davro (15), and Davros	12/06/2010 00:00:00
9	Romantic meal for two	13	George Clooney (14), and Donna Watson	29/03/2010 00:00:00

图 40-13

还可以对 `submitButton_Click()` 方法做进一步的修改, 确保在添加新记录时更新数据:

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
}
```

所有数据绑定控件都支持这个方法, 如果调用顶层的(this) `DataBind()` 方法, 窗体通常就会调用该方法。

注意, 在图 40-13 中, `EventDate` 字段的日期/时间显示有点麻烦。由于我们只查看日期, 因此时间总是 00:00:00 AM, 其实这个信息不需要显示。下一节将学习如何以更友好的方式在 `ListView` 控件的上下文中显示这个日期信息。果然, `DataGrid` 控件包含许多属性, 它们可以用于格式化显示的数据, 但这部分内容由您自学。

1. 使用模板显示数据

许多数据显示控件允许使用模板来格式化要显示的数据。模板在 ASP.NET 中是 HTML 的参数化部分, 它们用作某些控件的输出元素。它们允许定制如何将数据输出到浏览器上, 不需要做太多的工作就可以得到专业级的显示结果。

有几个模板可用于定制列表行为的各个方面。其中一个重要的模板是 `<ItemTemplate>`, 它可以用于显示 `Repeater`、`DataList` 和 `ListView` 控件的列表中的每个数据项。在控件声明中声明这个模板(和其他模板), 例如:

```
<asp:DataList Runat="server" ...>
    <ItemTemplate>
        ...
    </ItemTemplate>
</asp:DataList>
```

在模板声明中, 一般希望输出 HTML 的部分内容, 其中参数来自绑定到控件的数据。在输出这些参数时, 应使用一种特殊的语法:

```
<%# expression %>
```

虽然 `expression` 占位符可能仅是把参数绑定到页面或控件属性上的表达式, 但它更可能由 `Eval()` 或 `Bind()` 表达式组成。通过指定表中的列, 这个函数可用于从绑定到控件的表中输出数据。 `Eval()` 表达式使用下面的语法:

```
<%# Eval("ColumnName") %>
```

第二个可选参数允许格式化返回的数据, 它的语法与其他地方使用的字符串格式化表达式相同。该参数可以把日期字符串格式化为可读性更高的格式, 这正是前面示例所缺乏的。

`Bind()` 表达式与 `Eval()` 表达式相同, 但它可以把数据插入服务器控件的属性中, 例如:

```
<asp:Label RunAt="server" ID="ColumnDisplay" Text=' <%# Bind("ColumnName") %> ' />
```

注意, 因为双引号可在 `Bind()` 参数中使用, 所以应使用单引号把属性值括起来。

表 40-8 给出了可用的模板列表以及何时使用它们。

表 40-8

模 板	应 用 于	说 明
<ItemTemplate>	DataList、Repeater、List View	用于列表项
<HeaderTemplate>	DataList、DetailsView、FormView、Repeater	用于输出列表项前面的内容
<FooterTemplate>	DataList、DetailsView、FormView、Repeater	用于输出列表项后面的内容
<LayoutTemplate>	List View	用于指定输出周围的项
<SeparatorTemplate>	DataList、Repeater	用于列表中项之间
<ItemSeparatorTemplate>	List View	用于列表中项之间
<AlternatingItemTemplate>	DataList、List View	用于其他项, 有助于查看
<SelectedItemTemplate>	DataList、List View	用于列表中所选项
<EditItemTemplate>	DataList、FormView、List View	用于列表中正在编辑的项
<InsertItemTemplate>	FormView、List View	用于列表中正在插入的项
<EmptyDataTemplate>	GridView、DetailsView、FormView	用于显示空项, 例如, 当 GridView 中没有记录时使用它
<PagerTemplate>	GridView、DetailsView、FormView	用于格式化分页
<GroupTemplate>	List View	用于指定输出周围的项的组合
<GroupSeparatorTemplate>	List View	用于列表中不同组之间
<EmptyItemTemplate>	List View	使用分组的项时, 该模板用于为组中的空项提供输出。这个模板在组中没有足够的项时用于填充该组

了解如何使用模板最简单的方式是举一个例子。

2. 使用模板

在 PCSWebSite3 的 Default.aspx 页面顶部扩展表, 使之包含一个 List View, 以显示存储在数据库中的每个事件。使这些事件成为可选择的, 这样单击每个事件的名称, 就在 FormView 控件中显示任何事件的详细的信息。

首先需要为数据绑定控件创建新的数据源。每个数据绑定控件最好(强烈建议)有自己的数据源。

List View 控件需要 SqlDataSource 控件 MRBEEventData2, MRBEEventData2 与 MRBEEventData 相似, 但它只需返回 Name 和 ID 数据。需要的代码如下所示:

```
<asp:SqlDataSource ID="MRBEEventData2" Runat="server">
```

```
SelectCommand="SELECT [ID], [Name] FROM [Events]"
ConnectionString=" <%$ ConnectionStrings:MRBConnectionString %> ">
</asp:SqlDataSource>
```

尽管可以通过数据源配置向导方便地构建它，但 FormView 控件的数据源 MRBEventDetailData 比较复杂。这个数据源使用 ListView 控件的选中项 EventList，获取选中的项的数据。这可以使用 SQL 查询中的一个参数实现，如下所示：

```
<asp:SqlDataSource ID="MRBEventDetailData" Runat="server"
    SelectCommand="SELECT dbo.Events.Name, dbo.Rooms.Room,
        dbo.Events.AttendeeList, dbo.Events.EventDate
        FROM dbo.Events INNER JOIN dbo.Rooms
        ON dbo.Events.ID = dbo.Rooms.ID WHERE dbo.Events.ID = @ID"
    ConnectionString=" <%$ ConnectionStrings:MRBConnectionString %> ">
    <SelectParameters>
        <asp:ControlParameter Name="ID" DefaultValue="-1" ControlID="EventList"
            PropertyName="SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```

其中，ID 参数会得到在 Select 查询的 @ID 处插入的值。ControlParameter 项从 EventList 的 SelectedValue 属性中提取这个值，如果没有选中的项，就使用 -1。初看起来，这种语法有点古怪，但它非常灵活。一旦使用向导生成了其中几个对象，就不需要自己汇编它们了。

下面需要添加 DataList 和 FormView 控件。修改 PCSWebSite3 项目的 Default.aspx 中的代码，如下代码所示：

```
<tr>
    <td align="center" colspan="3">
        <asp:ValidationSummary ID="validationSummary" Runat="server"
            HeaderText="Before submitting your request:" />
    </td>
</tr>
<tr>
    <td align="left" colspan="3" style="width: 40%;">
        <table cellpadding="4" style="width: 100%;">
            <tr>
                <td colspan="2" style="text-align: center;">
                    <h2>Event details</h2>
                </td>
            </tr>
            <tr>
                <td style="width: 40%; background-color: #ccffcc;"
                    valign="top">
                    <asp:ListView ID="EventList" runat="server"
                        DataSourceID="MRBEventData2" DataKeyNames="ID"
                        OnSelectedIndexChanged="EventList_SelectedIndexChanged">
                        <LayoutTemplate>
                            <ul>
                                <asp:Placeholder ID="itemPlaceholder"
                                    runat="server" />
                            </ul>
                        </LayoutTemplate>
                        <ItemTemplate>
```

```

        <li>
            <asp:LinkButton Text='<%= Bind("Name") %>'
                runat="server" ID="NameLink" CommandName="Select"
                CommandArgument='<%= Bind("ID") %>'
                CausesValidation="false" />
        </li>
    </ItemTemplate>
    <SelectedItemTemplate>
        <li>
            <b><%= Eval("Name") %></b>
        </li>
    </SelectedItemTemplate>
</asp:ListView>
</td>
<td valign="top">
    <asp:FormView ID="FormView1" Runat="server"
        DataSourceID="MRBEventDetailData">
        <ItemTemplate>
            <h3><%= Eval("Name") %></h3>
            <b>Date:</b>
            <%= Eval("EventDate", "{0:D}") %>
            <br />
            <b>Room:</b>
            <%= Eval("Room") %>
            <br />
            <b>Attendees:</b>
            <%= Eval("AttendeeList") %>
        </ItemTemplate>
    </asp:FormView>
</td>
</tr>
</table>
</td>
</tr>
</table>

```

我们为表添加了新的一行，其中包含一个表，该表中的一列对应一个 ListView 控件，另一列对应一个 FormView 控件。

ListView 控件使用 <LayoutTemplate> 输出一个项目符号列表，使用 <ItemTemplate> 和 <SelectedItemTemplate> 作为列表项显示事件的详细信息。在 <LayoutTemplate> 中，用 ID 属性为“itemPlaceholder”的 Placeholder 控件给列表项指定一个容器元素。为了便于选项，从显示在 <ItemTemplate> 中的事件名称链接触发 Select 命令，就可以自动修改选项。我们还使用了 OnSelectedIndexChanged 事件，当 Select 命令修改选项时触发这个事件，以确保更新显示的列表，用不同的风格显示选中的项。事件处理程序如下述代码所示：

```

protected void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
    EventList.DataBind();
}

```

还需要确保把新事件添加到列表中：

```

if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
    EventList.DataBind();
}

```

现在事件的详细信息就显示在表中，如图 40-14 所示。

通常使用模板和数据绑定控件可以完成许多任务，需要用一本书的篇幅来介绍。但是，这里介绍的内容已经足够您开始试用它们了。

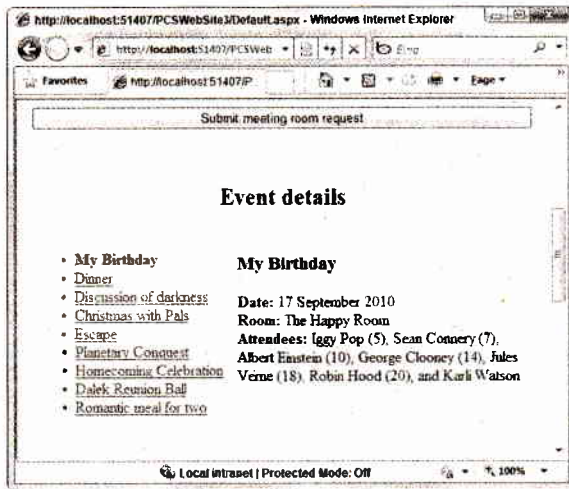


图 40-14

40.4 应用程序配置

本章一直暗示一个事情，那就是应用程序都包含 Web 页面和配置设置。这是一个要掌握的重要概念，特别是为多个并发用户配置 Web 站点时，这个概念就更加重要。

首先介绍一些术语和应用程序的生命期。应用程序定义为项目中的所有文件，并由 web.config 文件配置。在第一次启动应用程序时，将创建一个 Application 对象，即收到第一个 HTTP 请求时创建该对象。此时还将触发 Application_Start 事件，创建一个 HttpApplication 实例池。每个引入的请求都会接收到这样一个实例，它执行请求的处理过程。注意，这意味着 HttpApplication 对象不需要处理并发访问，这与全局 Application 对象不同。所有 HttpApplication 实例完成它们的任务后，就触发 Application_End 事件，应用程序终止执行，销毁 Application 对象。

上面提到的事件处理程序(以及本章讨论的所有其他事件的处理程序)可以在 global.asax 文件中定义，该文件可以添加到任意 Web 站点项目中(在给 Web 应用程序添加新项时，会看到该文件在模板中显示为 Global Application Class)。生成的文件包含空格，用户可以在这些空格中填写信息，例如：

```
void Application_Start(Object sender, EventArgs e)
```

```

{
    // Code that runs on application startup
}

```

在单个用户使用 Web 应用程序时,会启动一个会话。与应用程序类似,会话将创建一个特定于用户的 Session 对象,并触发 Session_Start 事件。在一个会话中,每个请求都将触发 Application_BeginRequest 和 Application_EndRequest 事件。在一个会话的作用域中可以多次触发这两个事件,因为这些事件会访问应用程序中的不同资源。单个会话可以手动终止,如果没有接收到更多的请求,会话也会因超时而停止。会话终止将触发 Session_End 事件,并析构 Session 对象。

对于这个过程的背景,可以执行几个操作,以简化应用程序。例如,如果应用程序的所有实例都使用一个资源密集型对象,就可以考虑在应用程序级别上实例化它。这将提高性能,减少多个用户使用的内存,因为在大多数请求中,不需要进行这种实例化。

可以使用的另一个技巧是存储会话级别的信息,以备单个用户在跨请求时使用。这些信息包括用户第一次连接(在 Session_Start 事件处理程序中)时从数据存储器中提取的用户特定信息,直到会话终止(通过超时或用户请求)时,才能使用这些信息。

这些技巧超出了本书的范围,读者可参阅《ASP.NET 4 高级编程——涵盖 C#和 VB.NET(第 7 版)》(清华大学出版社引进并已出版),它们有助于理解该过程。

最后,看看 web.config 文件。Web 站点通常在其根目录下有这个文件(但不会在默认情况下创建它),在其子目录下也有该文件,用于配置特定于该子目录的设置(如安全性)。本章开发的 Web 站点 PCSWebSite3 在添加已存储的数据库连接字符串时,会接收一个自动生成的 Web.Config 文件,如下所示:

```

<connectionStrings>
  <add name="MRBConnectionString"
    connectionString="Data Source=.\SQLEXPRESS;
    AttachDbFilename=|DataDirectory|\MeetingRoomBooker.mdf;
    Integrated Security=True;User Instance=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>

```

如果在调试模式下运行该项目,就会在 web.config 文件中看到一些额外的设置。

可以手工编辑 web.config 文件,还可以使用一个工具配置 Web 站点(及其底层的配置文件),这个工具可在 Visual Studio 的 Web Site 菜单的 ASP.NET Configuration 下面访问。该工具如图 40-15 所示。

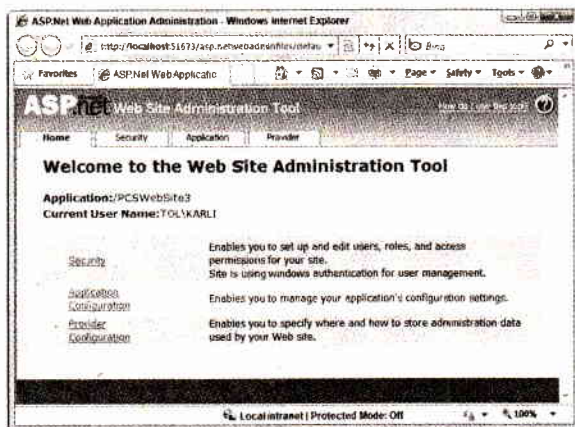


图 40-15

从图 40-15 中可以看出, 这个工具可以配置许多设置, 包括安全性。下一章将详细介绍这个工具。

40.5 小结

本章概述如何利用 ASP.NET 创建 Web 应用程序, 介绍如何使用 C# 和 Web 服务器控件提供一个功能丰富的开发环境。我们还开发了一个事件登记示例应用程序, 它阐述了许多可用的技术, 例如, 各种已存在的服务器控件、ADO.NET 的数据绑定。

主要内容如下;

- ASP.NET 概述, 以及它通常如何与 .NET 开发环境配合使用
- ASP.NET 基本语法的工作原理, 如何实现状态管理, 如何把 C# 代码集成到 ASP.NET 页面中
- 如何使用 Visual Studio 创建 ASP.NET Web 应用程序, 存在哪些可用于存储和测试 Web 站点的选项
- 总结了 ASP.NET 开发人员可以使用的 Web 控件, 以及它们如何传递动态和/或数据驱动的内容
- 如何使用事件处理程序检测并执行用户与控件的交互操作, 并通过页面和呈现事件定制控件
- 如何把数据绑定到 Web 控件上, 并使用模板和数据绑定表达式格式化已显示的数据
- 如何综合运用这些技巧构建会议室登记应用程序

掌握了这些知识, 就可以汇编功能强大的 Web 应用程序了。但这只涉及 ASP.NET 的皮毛。在开始 Web 开发之前, 应先了解更多的信息。第 41 章将扩展 ASP.NET 知识, 学习更重要的 Web 主题, 包括母版页、skinning 和个性化。



可从
wrox.com
下载源代码

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
```

代码段 PCSUserCWebApp1/PCSUserC1.ascx

这非常类似于在.aspx 文件中生成的<%@ Page %>指令，除了指定了 Control，而不是 Page。CodeFile 属性指定代码隐藏文件，Inherits 指定在代码隐藏文件中页面继承的类名。.ascx.cs 文件包含的代码与自动生成的.aspx.cs 文件一样，其中包含一个空的类定义和 Page_Load() 事件处理程序。

本例的简单控件是一个显示图形的控件，显示的图形对应于扑克牌中 4 种标准花色(即梅花、方块、红桃或黑桃)中的 1 种。这里所需的图形是以前版本的 Visual Studio 附带的图形；它们在本章的可下载代码中，位于 CardSuitImages 目录，其文件名分别是 CLUB.BMP、DIAMOND.BMP、HEART.BMP 和 SPADE.BMP。把这些位图文件复制到项目所在目录的新子目录 Images 中，以便在后面使用它们。如果不能访问这段可下载的代码，就可以使用其他任意图像，因为它们对于代码的功能并不重要。



注意，与 Visual Studio 的以前版本不同，在 Visual Studio 外部对 Web 站点结构进行的修改会自动反映在 IDE 中。只需单击 Solution Explorer 窗口中的 Refresh 按钮，就会看到新的 Images 目录和位图文件。

(1) 给控件添加内容

现在给新控件添加一些代码。在 PCSUserC1.ascx 的 HTML 视图添加下列代码：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
<table cellpadding="4">
  <tr valign="middle">
    <td>
      <asp:Image Runat="server" ID="suitPic" ImageURL="~/Images/club.bmp"/>
    </td>
    <td>
      <asp:Label Runat="server" ID="suitLabel">Club</asp:Label>
    </td>
  </tr>
</table>
```

这段代码定义了控件的默认状态，即一个梅花图形和一个标签。图像路径前面的~表示“从 Web 站点的根目录开始”。在给控件添加功能之前，先把这个控件添加到项目的 Web 页面中测试这个默认状态，因此在继续添加功能之前，把一个新的 Web 页面 Default.aspx 添加到 Web 站点中。

为了在.aspx 文件中使用自定义控件，首先需要指定如何引用该控件，也就是说，如何在 HTML 中引用表示控件的标记的名称。为此，在 Default.aspx 中代码的顶部使用<%@ Register %>指令，如下所示。



可从
wrox.com
下载源代码

```
<%@ Register TagPrefix="pcs" TagName="UserC1" Src="PCSUserC1.ascx" %>
```

代码段 PCSUserCWebApp1/Default.aspx

TagPrefix 和 TagName 特性指定要使用的标记名称(指定的格式为<TagPrefix:TagName>), 使用 Src 特性指向包含用户控件的文件。现在, 添加下面的元素, 用以下代码来替代文件中已有的<form>元素, 就可以使用控件了:

```
<form id="Form1" method="post" runat="server">
    <div>
        <pcs:UserC1 Runat="server" ID="myUserControl"/>
    </div>
</form>
```

这就是测试用户控件需要做的所有工作, 运行这些代码的结果如图 41-1 所示。

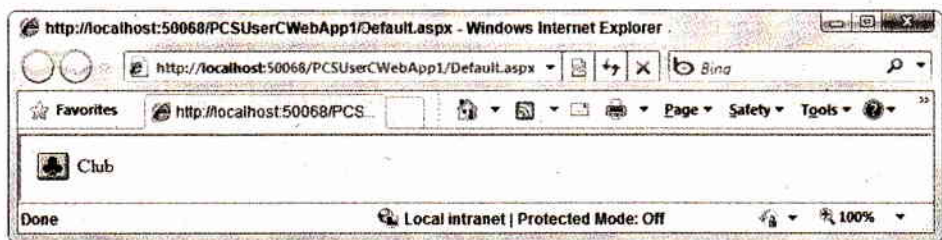


图 41-1

可以看出, 这个控件在表格布局中组合了两个现有的控件, 即图像控件和标签控件, 因此它属于合成控件一类。

(2) 添加 Suit 属性

为了控制显示的花色, 可以在<PCS:UserC1>元素上使用特性。用户控件元素上的属性会自动映射到用户控件的属性上, 因此, 只需给控件的代码隐藏 PCSUserC1.ascx.cs 添加一个特性。这个特性称为 Suit, 让它接收合适的花色值。为了便于表示控件的状态, 可以定义一个枚举, 来保存 4 个花色名称。最佳方式是在 Web 站点上添加一个目录 App_Code(App_Code 是另一个“特殊”的目录, 与 App_Data 一样, 它的功能取决于编程人员, 这里它为 Web 站点保存其他代码文件。要添加这个目录, 可以右击 Solution Explorer 窗口中的 web site 菜单, 从弹出的上下文菜单中选择 Add ASP.NET Folder | App_Code 命令), 然后在这个目录中添加一个.cs 文件 Suit.cs, 其代码如下:

可从
wrox.com
下载源代码

```
using System;

public enum Suit
{
    Club, Diamond, Heart, Spade
}
```

代码段 PCSUserCWebApp1/App_Code/Suit.cs

PCSUserC1 类需要一个成员变量, 以保存花色类型 currentSuit:

可从
wrox.com
下载源代码

```
public partial class PCSUserC1 : System.Web.UI.UserControl
{
    protected Suit currentSuit;
}
```

代码段 PCSUserCWebApp1/PCSUserC1.ascx.cs

再添加一个访问这个成员变量的 Suit 属性:

```

public Suit Suit
{
    get
    {
        return currentSuit;
    }
    set
    {
        currentSuit = value;
        suitPic.ImageUrl = "~/Images/" + currentSuit.ToString() + ".bmp";
        suitLabel.Text = currentSuit.ToString();
    }
}

```

这里的 set 存取器把图像的 URL 设置为前面复制的其中一个文件, 并把要显示的文本设置为花色名称。

(3) 访问新属性

下面需要给 Default.aspx 添加代码以访问这个新属性。使用刚才添加的属性就可以选择花色(如果编译该项目, 选项甚至就会显示在 IntelliSense 中):



```
<PCS:UserC1 Runat="server" id="myUserControl" Suit="diamond"/>
```

代码段 PCSUserCWebApp1/Default.aspx

ASP.NET 处理器足够智能可以从提供的字符串中选择正确的枚举项。但为了使该控件更有趣、互动性更强, 下面使用一个单选按钮列表来选择花色:

```

<form id="form1" runat="server">
    <div>
        <pcs:UserC1 id="myUserControl" runat="server" />
        <asp:RadioButtonList Runat="server" ID="suitList" AutoPostBack="True">
            <asp:ListItem Value="Club" Selected="True">Club</asp:ListItem>
            <asp:ListItem Value="Diamond">Diamond</asp:ListItem>
            <asp:ListItem Value="Heart">Heart</asp:ListItem>
            <asp:ListItem Value="Spade">Spade</asp:ListItem>
        </asp:RadioButtonList>
    </div>
</form>

```

还需要给列表的 SelectedIndexChanged 事件添加事件处理程序。双击设计视图中的单选按钮列表, 就可以添加处理程序。



注意, 把列表的 Autopostback 属性设置为 true, 是因为除非进行回发操作, 否则将不在服务器上执行 suitList_SelectedIndexChanged 事件处理程序, 在默认状态下, 这个控件也不会触发回发操作。

在 Default.aspx.cs 中, suitList_SelectedIndexChanged()方法需要以下代码:



可从
wrox.com
下载源代码

```
public partial class Default
{
    protected void suitList_SelectedIndexChanged(object sender, EventArgs e)
    {
        myUserControl.Suit = (Suit)Enum.Parse(typeof(Suit),
                                                suitList.SelectedItem.Value);
    }
}
```

代码段 PCSUserCWebApp1/Default.aspx.cs

我们知道, <ListItem>元素上的 Value 特性表示前面定义的 suit 枚举的有效值, 因此仅把这些值解析为枚举类型, 并把它们用作用户控件的 Suit 属性值。使用简单的数据类型强制转换语法, 就可以把返回的对象类型强制转换为 Suit, 因为这种类型不能通过隐式转换得到。

在运行 Web 站点时, 可以改变花色, 如图 41-2 所示。

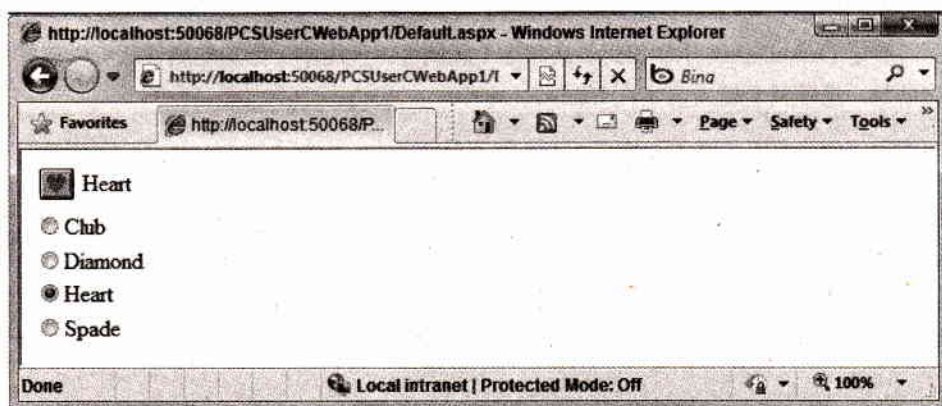


图 41-2

既然完成了用户控件的创建工作后, 使用<%@ Register %>指令和为控件创建的两个源代码文件 (PCSUserC1.ascx 和 PCSUserC1.ascx.cs), 可以在任何其他 Web 页面中使用这个用户控件。

41.1.2 PCSDemoSite 中的用户控件

在 PCSDemoSite 中, 要把上一章的会议室登记网站转换为用户控件, 以便于重用。为了查看这个控件, 必须以 User1 的身份, 用密码 User1!! 登录站点(41.4 节将介绍登录系统的工作原理), 然后导航到 Meeting Room Booker 页面, 如图 41-3 所示。

除了样式上有显著的变化之外, 本章后面的主题还对该页面进行了如下大的改动:

- 用户名自动从用户详细信息中获取。
- 在页面底部不显示额外的数据, 从代码隐藏文件中删除了相应的 DataBind()调用。
- 控件的下面没有显示结果标签, 用户查看在日历中添加的事件和事件列表, 就可以获得足够多的反馈, 无需报告事件的添加是否成功。
- 包含用户控件的页面使用了导航控件和母版页。

要进行这些修改, 代码的改动其实很简单, 但这里不介绍它们。本章后面还会介绍这个控件。

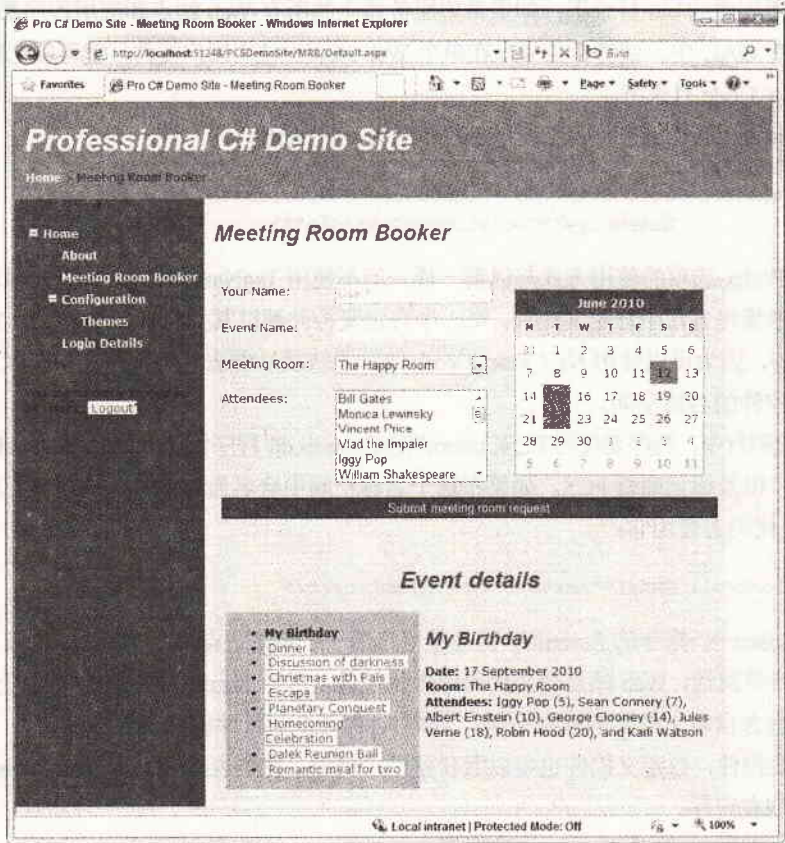


图 41-3

41.1.3 自定义控件

与用户控件相比，自定义控件又进了一步，原因是自定义控件完全自包含在 C# 程序集中，不需要单独的 ASP.NET 代码。这意味着不需要在 .ascx 文件中组装 UI。相反，完全可以控制写入输出流的内容，即控件所生成的 HTML。

一般情况下，开发自定义控件要比开发用户控件花费的时间更长，原因是自定义控件的语法更复杂一些，通常需要编写更多的代码才能得到结果。在开发用户控件时，只需简单地把几个其他控件组合在一起，而开发自定义控件就不那么简单了。

1. 创建和引用自定义控件

为了使自定义控件拥有最多可定制的行为，可以从 `System.Web.UI.WebControls.WebControl` 中派生一个类，这样就创建了一个完整的自定义控件。此外，可以扩展现有控件的功能，创建一个派生的自定义控件。最后，可以像上一节那样把几个现有的控件组合在一起，但要使用逻辑性更强的结构，创建一个合成的自定义控件。

上述 3 种方法创建的自定义控件都可以按同一种方式在 ASP.NET 页面中使用。我们只需把生成的程序集放在使用它的 Web 站点或应用程序可以找到的地方，并使用 `<%@ Register %>` 指令注册要使用的元素名称。这里“Web 站点或应用程序可以找到的地方”有两种情况：可以把程序集放在

Web 站点或应用程序的 bin 目录下；如果希望服务器上的所有 Web 站点和应用程序都可以访问它，就可以把它放在 GAC 中。或者，如果只在单个 Web 站点上使用用户控件，就可以把控件的.cs 文件放在站点的 App_Code 目录下。

<%@ Register %>指令的语法在自定义控件中有一些变化：

```
<%@ Register TagPrefix="PCS" Namespace="PCSCustomWebControls"
    Assembly="PCSCustomWebControls"%>
```

虽然 TagPrefix 选项的使用方式与以前一样，但不使用 TagName 或 Src 特性。原因是所使用的自定义控件程序集包含几个自定义控件，每一个自定义控件都以其类名来命名，所以 TagName 属性就多余了。此外，因为可以使用 .NET Framework 的动态查找功能去查找程序集，所以只需命名程序集和其中包含控件的名称空间。

在上面的代码行中，程序要使用 PCSCustomWebControls.dll 程序集和 PCSCustomWebControls 名称空间中的控件，以及标记前缀 PCS。如果在这个名称空间中有名为 Controll1 的控件，则可以通过下面的 ASP.NET 代码去使用它：

```
<PCS:Controll1 Runat="server" ID="MyControll1"/>
```

<%@ Register %>指令的 Assembly 特性是可选的——如果站点的 App_Code 目录下有自定义控件，就可以忽略该属性，Web 站点会在代码中查找控件。尽管 Namespace 属性不是可选的，但必须在代码文件中包含自定义控件的名称空间，否则 ASP.NET 运行库就找不到它们。

类似于列表控件，自定义控件也可以嵌套使用，例如，在列表控件中可以嵌套<asp:ListItem>控件，以填充该列表控件：

```
<asp:DropDownList ID="roomList" Runat="server" Width="160px">
    <asp:ListItem Value="1"> The Happy Room </asp:ListItem>
    <asp:ListItem Value="2"> The Angry Room </asp:ListItem>
    <asp:ListItem Value="3"> The Depressing Room </asp:ListItem>
    <asp:ListItem Value="4"> The Funked Out Room </asp:ListItem>
</asp:DropDownList>
```

可以以类似的方式创建一些控件，把它们解释为其他控件的子控件。这是比较高级的技术，本章不探讨。

2. 自定义控件示例

下面将理论用于实践。我们将使用 C:\ProCSharp\Chapter41\ 目录下的一个 Web 站点 PCSCustomCWebSite1(使用空白 Web 站点模板创建)，以及其 App_Code 目录下的一个自定义控件来说明这个简单的自定义控件。这个控件是已有 Label 控件的彩色版本，可以给文本中的每个字母轮流设置一组不同的颜色。

RainbowLabel 控件的代码在 App_Code\RainbowLabel.cs 文件中，首先是下述 using 语句：



可从
wrox.com
下载源代码

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

代码段 PCSCustomCWebSite1/RainbowLabel.cs

`System.Drawing` 名称空间用于 `Color` 枚举, `Web` 名称空间用于 ASP.NET 控件引用。该文件的类在私有 `Color` 数组 `colors` 中维护一个颜色数组, 这些颜色用于文本中的字母:

```
namespace PCSCustomWebControls
{
    public class RainbowLabel : Label
    {
        private Color[] colors = new Color[] {Color.Red,
                                                Color.Orange,
                                                Color.Yellow,
                                                Color.GreenYellow,
                                                Color.Blue,
                                                Color.Indigo,
                                                Color.Violet};
    }
}
```

另外注意 `PCSCustomWebControls` 名称空间用于包含控件。如前所述, 这是一个必须的名称空间, 有了它, `Web` 页面才能正确地引用控件。

为了可以循环使用不同的颜色, 还要在私有属性 `offset` 中存储一个整数偏移量:

```
private int offset
{
    get
    {
        object rawOffset = ViewState["_offset"];
        if (rawOffset != null)
        {
            return (int)rawOffset;
        }
        else
        {
            ViewState["_offset"] = 0;
            return 0;
        }
    }
    set
    {
        ViewState["_offset"] = value;
    }
}
```

注意这个属性只是在一个成员字段中存储了一个值, 这源于 ASP.NET 维护状态的方式, 如上一章所述。因为控件在每个回发操作中都会实例化, 所以要存储值, 必须使用视图状态。为了易于访问, 只需使用 `ViewState` 集合, 它可以存储能串行化的任意对象。如果没有这么做, 在每次回发期间, `offset` 就会恢复其初始值。

要修改 `offset`, 可以使用 `Cycle()` 方法:

```
public void Cycle()
{
    offset = ++offset;
}
```

这个方法只递增在视图状态中为 `offset` 存储的值。

最后, 重写自定义控件最重要的方法——Render()方法。在这个方法中输出 HTML, 而且这是一个难以实现的方法。如果考虑可以查看控件的所有浏览器, 以及可能影响呈显方式的所有变量, 这个方法就会非常庞大。幸好, 对于本例, 这个方法相当简单:

```
protected override void Render(HtmlTextWriter output)
{
    string text = Text;
    for (int pos = 0; pos < text.Length; pos++)
    {
        int rgb = colors[(pos + offset) % colors.Length].ToArgb()
            & 0xFFFFFF;

        output.Write(string.Format(
            " <font color=\"#{0:X6}\"> {1} </font> ", rgb, text[pos]));
    }
}
```

这个方法允许访问输出流, 以显示自定义控件的内容。只有两种情况不需要实现这个方法:

- 当设计一个没有可视化表示的控件(通常称为组件)时。
- 当从已有控件中派生且不需要改变其显示特征时。

自定义控件还提供自定义方法、引发自定义事件、响应子控件(如果有的话)。对于 RainbowLabel 控件, 不需要考虑这些。

下面添加一个 Web 页面 Default.aspx, 并添加代码, 以显示控件和访问 Cycle(), 如下所示:



可从
wrox.com
下载源代码

```
<%@ Register TagPrefix="pcs" Namespace="PCSCustomWebControls" %>
...
<form id="form1" runat="server">
    <div>
        <pcs:RainbowLabel runat="server" ID="rainbowLabel1"
            Text="Multicolored label!" />
        <asp:Button Runat="server" ID="cycleButton" Text="Cycle colors"
            OnClick="cycleButton_Click" />
    </div>
</form>
```

代码段 PCSCustomCWebSite1/Default.aspx

Default.aspx.cs 中需要的代码非常简单:



可从
wrox.com
下载源代码

```
protected void cycleButton_Click(object sender, EventArgs e)
{
    rainbowLabel1.Cycle();
}
```

代码段 PCSCustomCWebSite1/Default.aspx.cs

现在可以查看示例文本, 给文本中的字母循环使用不同的颜色, 如图 41-4 所示。

可以对自定义控件做许多工作, 实际上, 其可能性是无穷的, 但必须来自对这些可能性进行试验。

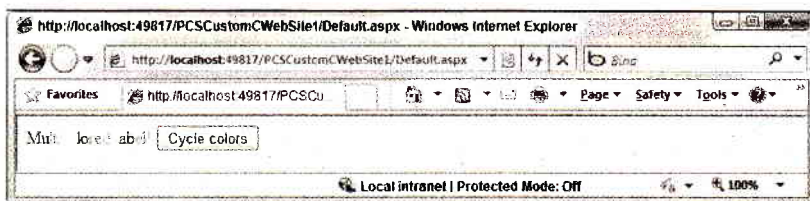


图 41-4

41.2 母版页

母版页可以使 Web 站点更容易设计。把所有(至少是大多数)页面布局都放在一个文件中,就可以重点关注站点的各个 Web 页面更重要的事情。

母版页在扩展名为 .master 的文件中创建,并可以像任何其他站点内容那样,通过 Website | Add New Item 菜单项添加。初看起来,为母版页生成的代码类似于标准.aspx 页面:

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MyMasterPage.master.cs" Inherits="MyMasterPage" %>
<!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>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server">
            </asp:ContentPlaceHolder>
        </div>
    </form>
</body>
</html>
```

其区别如下:

- 尽管属性相同,但使用<%@ Master %>指令,而不是<%@ Page %>指令。
- 在页面标题中放置一个 ID 为 head 的 ContentPlaceHolder 控件。
- 在页面正文中放置一个 ID 为 ContentPlaceHolder1 的 ContentPlaceHolder 控件。

这个 ContentPlaceHolder 控件使母版页非常有用。在一个页面中可以有任意多个 ContentPlaceHolder 控件,它们都由使用母版页的.aspx 页面用于“插入”内容。虽然可以把默认内容插入 ContentPlaceHolder 控件中,但.aspx 页面会重写这项内容。

.aspx 页面要使用母版页,需要修改<%@ Page %>指令,如下所示:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" MasterPageFile="~/MyMasterPage.master"
    Title="Page Title" %>
```

这里添加了两个新属性: `MasterPageFile` 属性表示要使用的母版页, `Title` 属性设置母版页中 `<title>` 元素的内容。

把一个包含母版页的 .aspx 页面添加到 Web 站点上时, 就可以选择使用母版页。Add New Item 向导包含 Select master page 复选框, 如果选中这个复选框, 向导的下一个页面就会列出 Web 应用程序中的一个母版页列表, 以供选择 Web 应用程序。接着, 为页面生成的代码会包含如上所示的 `MasterPageFile` 属性。

如果要使用默认的母版页内容, .aspx 页面就不必包含任何其他代码。另外, 也不能包含 Form 控件, 因为页面只能有一个 Form 控件, 而母版页已经有了一个 Form 控件。

使用母版页的 .aspx 页面可以包含不是指令的非根级内容、脚本元素和 Content 控件。可以有任意多个 Content 控件, 每个 Content 控件都会把内容插入母版页的其中一个 `ContentPlaceHolder` 控件中。唯一要注意的是, 确保 Content 控件的 `ContentPlaceHolderID` 属性匹配要在其中插入内容的 `ContentPlaceHolder` 控件的 ID。所以, 要把内容添加到前面的母版页中, 只需要下面的 .aspx 文件:

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.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">
    Custom content!
</asp:Content>
```

母版页的真正强大之处在于, 在自己的母版页中在 `ContentPlaceHolder` 控件的四周放置其他内容, 例如, 导航控件、站点徽标和 HTML。可以为主要内容、边栏内容、脚标文本等提供多个 `ContentPlaceHolder` 控件。

如果不希望为特定的 `ContentPlaceHolder` 控件提供内容, 就可以省略页面上的 Content 控件。例如, 可以从上面的代码中删除 Content1 控件, 这不会影响最终的显示结果。

41.2.1 在 Web 页面中访问母版页

给 Web 页面添加母版页时, 有时需要从 Web 页面的代码中访问母版页。为此, 可以使用 `Page.Master` 属性, 它以 `MasterPage` 对象的形式返回对母版页的一个引用。可以把这个对象的类型强制转换为母版页的类型, 该母版页的类型由母版页文件定义(在上一节的示例中, 这个类称为 `MyMasterPage`)。有了这个引用后, 就可以访问母版页类的任意公共成员了。

另外, 还可以使用 `MasterPage.FindControl()` 方法通过它们的标识符定位母版页上的控件, 以便处理母版页上内容占位符外部的内容。

一个典型用法是, 如果定义了一个用于标准窗体的母版页, 其中包含一个提交按钮, 就可以在子页面上定位提交按钮, 在母版页上为 Submit 按钮添加事件处理程序。这样, 就可以提供自定义的验证逻辑, 来响应窗体的提交了。

41.2.2 嵌套的母版页

Select master page 选项在创建新母版页时也可用。使用这个选项可以根据父母版页创建嵌套的

母版页。例如，可以创建一个母版页 `MyNestedMasterPage`，它使用如下 `MyMasterPage`：

```
<%@ Master Language="C#" MasterPageFile="~/MyMasterPage.master"
    AutoEventWireup="false" CodeFile="MyNestedMasterPage.master.cs"
    Inherits="MyNestedMasterPage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
    <!-- Disabled for child controls. -->
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
    Runat="Server">
    First nested place holder:
    <asp:ContentPlaceHolder ID="NestedContentPlaceHolder1" runat="server">

        </asp:ContentPlaceHolder>
    <br />
    <br />
    Second nested place holder:
    <asp:ContentPlaceHolder ID="NestedContentPlaceHolder2" runat="server">

        </asp:ContentPlaceHolder>
    </asp:Content>
```

使用这个母版页的页面为 `NestedContentPlaceHolder1` 和 `NestedContentPlaceHolder2` 提供内容，但不能直接访问在 `MyMasterPage` 中指定的 `ContentPlaceHolder` 控件。在这个例子中，`MyNestedMasterPage` 修正 `head` 控件的内容，为 `ContentPlaceHolder1` 控件提供一个模板。

创建一系列嵌套的母版页，可以为页面提供其他布局，且不改变基本母版页的某些方面。例如，根母版页可能包含导航和基本布局，嵌套的母版页可以为不同数量的列提供布局。接着在站点的页面中使用嵌套的母版页，在不同的页面上在这些布局之间快速切换。

41.2.3 PCSDemoSite 中的母版页

在 `PCSDemoSite` 中，使用了一个母版页 `MasterPage.master`（这是母版页的默认名称），其代码如下所示：



```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MasterPage.master.cs"
    Inherits="MasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <link rel="stylesheet" href="StyleSheet.css" type="text/css" />
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div id="header">
            <h1><asp:literal ID="Literal1" runat="server"
                text="<%$ AppSettings:SiteTitle %>" /></h1>
            <asp:SiteMapPath ID="SiteMapPath1" Runat="server"
                CssClass="breadcrumb" />
        </div>
```

```

<div id="nav">
  <div class="navTree">
    <asp:TreeView ID="TreeView1" runat="server"
      DataSourceID="SiteMapDataSource1" ShowLines="True" />
  </div>
<br />
<br />
<asp:LoginView ID="LoginView1" Runat="server">
  <LoggedInTemplate>
    You are currently logged in as
    <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
    <asp:LoginStatus ID="LoginStatus1" Runat="server" />
  </LoggedInTemplate>
</asp:LoginView>
</div>
<div id="body">
  <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
</div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
</body>
</html>

```

代码段 PCSDemoSite/MasterPage.master

这里的许多控件前面都没有见过，稍后将介绍它们。这里要注意的是，<div>元素包含各个内容部分(标题、导航栏和页面正文)，使用<%= \$AppSettings.SiteTitle %>从 web.config 文件中获得站点标题：



可从
wrox.com
下载源代码

```

<appSettings>
  <add key="SiteTitle" value="Professional C# Demo Site"/>
</appSettings>

```

代码段 PCSDemoSite/web.config

还有 StyleSheet.css 的一个样式表链接：

```
<link rel="stylesheet" href="StyleSheet.css" type="text/css" />
```

这个 CSS 样式表包含此页面上<div>元素的基本布局信息，以及会议室登记工具控件的一部分。注意，该文件中的样式信息不包含颜色、字体等。这是主题中的样式表要获得的信息，详见本章后面的内容。这里只有布局信息，如<div>的大小。



注意，本章的 Web 站点尽可能遵循最佳实践方式。布局使用 CSS 而不是表格，很快就会成为 Web 站点布局的业界标准，并且值得认真掌握。在上面的代码中，“#”符号用于格式化有指定 id 属性的<div>元素，而.mrbEventList 格式化有指定 class 属性的 HTML 元素。

41.3 站点导航

3 个 Web 导航服务器控件 SiteMapPath、Menu 和 TreeView，可以用于为 Web 站点提供 XML 站点地

图：如果实现另一个站点地图提供程序，就以另一种格式提供站点地图。一旦创建这样一个数据源，这些 Web 导航服务器控件就可以自动为用户生成位置信息和导航信息。



稍后介绍一个 XML 站点地图示例。

尽管也可以使用 TreeView 控件显示其他结构化数据，但通过站点地图它真正开始发挥作用，可以提供导航信息的另一种视图。

Web 导航服务器控件如表 41-1 所示。

表 41-1

控 件	说 明
SiteMapPath	显示痕迹导航样式的信息，允许用户查看他们在站点结构中的位置，并导航到父区域中。可以提供各种模板，如 NodeStyle 和 CurrentNodeStyle，来定制痕迹导航路径的外观
Menu	通过 SiteMapDataSource 控件链接到站点地图信息，可以查看完整的站点结构，控件的外观由模板定制
TreeView	可以在树型结构中显示层次结构的数据，如内容表。树的节点存储在 Nodes 属性中，选中的节点存储在 SelectedNode 中。有几个事件可以在服务器端处理用户交互操作，包括 SelectedNodeChanged 和 TreeNode Collapsed。这个控件一般是数据绑定的控件

41.3.1 添加站点地图文件

要为站点提供站点地图 XML 文件，可以使用 Web site | Add New Item 菜单项添加一个站点地图文件(.sitemap)。通过提供程序链接到站点地图。因为默认的 XML 提供程序会在站点的根目录下查找 Web.sitemap 文件，所以，除非要使用另一个提供程序，否则就应接受所提供的默认文件名。

站点地图 XML 文件包含一个<siteMap>根元素，这个根元素包含一个<siteMapNode>元素，<siteMapNode>元素又可以包含任意多个嵌套的<siteMapNode>元素。

每个<siteMapNode>元素都使用表 41-2 中的属性。

表 41-2

属 性	说 明
Title	页面标题，用作站点地图中显示的链接文本
url	页面位置，用作站点地图中显示的超链接位置
Roles	用户角色，允许查看菜单中的这个站点地图项等
description	可选文本，用于站点地图显示的弹出式工具提示

站点有了 Web.sitemap 文件后，添加痕迹导航路径就只需在页面上放置如下代码：

```
<asp:SiteMapPath ID="SiteMapPath1" Runat="server" />
```

这将使用默认的提供程序和当前的 URL 位置, 来格式化父页面的链接列表。
添加菜单或树型视图菜单需要 SiteMapDataSource 控件, 这也非常简单:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
```

如果使用自定义提供程序, 唯一的区别是, 可以通过 SiteMapProvider 属性指定该提供程序 ID。还可以使用 StartingNodeOffset 属性删除菜单数据的上一层(如根级的 Home 项); 使用 ShowStartingNode="False" 将只删除顶级链接; 使用 StartFromCurrentNode="True" 表示从当前位置开始; 使用 StringNodeUrl 属性会重写根节点。

只要把它们的 DataSourceID 设置为 SiteMapDataSource 的 ID, Menu 控制和 TreeView 控件就可以使用这个数据源中的数据。这两个控件都包含许多样式属性, 并可以设置主题, 详见本章后面的内容。

41.3.2 PCSDemoSite 中的导航

PCSDemoSite 的站点地图如下所示:



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf - 8" ?>
<siteMap>
  <siteMapNode url="~/Default.aspx" title="Home">
    <siteMapNode url="~/About/Default.aspx" title="About" />
    <siteMapNode url="~/MRB/Default.aspx" title="Meeting Room Booker"
      roles="RegisteredUser,SiteAdministrator" />
    <siteMapNode url="~/Configuration/Default.aspx" title="Configuration"
      roles="RegisteredUser,SiteAdministrator">
      <siteMapNode url="~/Configuration/Themes/Default.aspx" title="Themes"
        roles="RegisteredUser,SiteAdministrator"/>
    </siteMapNode>
    <siteMapNode url="~/Users/Default.aspx" title="User Area"
      roles="SiteAdministrator" />
    <siteMapNode url="~/Login.aspx" title="Login Details" />
  </siteMapNode>
</siteMap>
```

代码段 PCSDemoSite/Web.sitemap

PCSDemoSite 站点使用自定义提供程序从 Web.sitemap 中获得信息, 这是必需的, 因为默认的提供程序会忽略 roles 属性。这个自定义提供程序在 Web 站点的 web.config 文件中定义, 如下所示:



可从
wrox.com
下载源代码

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

代码段 PCSDemoSite/web.config

这个自定义提供程序和默认提供程序的唯一区别是添加了 securityTrimmingEnabled="true", 它指示提供程序, 只为当前用户允许查看的节点提供数据。这种可见性由用户的角色成员资格确定,

如下一节所述。

在 PCSDemoSite 中包含 SiteMapPath、TreeView 导航的 MasterPage。母版页和一个数据源一起显示，如下所示：



```
<div id="header">
  <h1><asp:literal ID="Literal1" runat="server"
    text="<%$ AppSettings:SiteTitle %>" /></h1>
  <asp:SiteMapPath ID="SiteMapPath1" Runat="server"
    CssClass="breadcrumb" />
</div>
<div id="nav">
  <div class="navTree">
    <asp:TreeView ID="TreeView1" runat="server"
      DataSourceID="SiteMapDataSource1" ShowLines="True" />
  </div>
  <br />
  <br />
  <asp:LoginView ID="LoginView1" Runat="server">
    <LoggedInTemplate>
      You are currently logged in as
      <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
      <asp:LoginStatus ID="LoginStatus1" Runat="server" />
    </LoggedInTemplate>
  </asp:LoginView>
</div>
<div id="body">
  <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
</div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
```

代码段 PCSDemoSite/MasterPage.master

这里唯一要注意的是，给 SiteMapPath 和 TreeView 导航提供了 CSS 类，以便于使用主题特性。

41.4 安全性

在 Web 站点中，安全性和用户管理的实现常常相当复杂，这是有原因的；我们必须考虑许多因素，包括：

- 要实现哪种用户管理系统？用户要映射到 Windows 用户账户上吗？还是实现某种独立的管理系统？
- 如何实现登录系统？
- 是让用户在站点上注册吗？如果是，如何注册？
- 如何让一些用户只查看某些内容，只执行某些操作，而给另一些用户提供额外的特权？
- 如果忘记了密码，该怎么办？

在 ASP.NET 中，有一整套可供使用的工具来处理这些问题，实际上，使用该工具只需几分钟就可以在站点上实现一个用户系统。我们有 3 种可供使用的身份验证系统：

- Windows 身份验证——用户有 Windows 账户，一般在内联网站或 WAN(Wide Area Network, 广域网)门户使用
- Forms 身份验证——Web 站点维护它自己的用户列表，完成自己的身份验证
- Microsoft Live ID 身份验证(以前称为 Passport 身份验证)——Microsoft 为用户提供的一个集中式身份验证服务

虽然完整讨论 ASP.NET 中的安全性至少需要一章的篇幅，但可以快速浏览一下实现安全性所涉及的工作。这里将只讨论 Forms 身份验证，因为这是最通用的系统，能很快建立和运转起来。

实现 Forms 身份验证最快捷的方式是使用 Web Site ASP.NET Configuration 工具，上一章简要介绍过这个工具，它有一个 Security 选项卡，其中是一个 Security Setup 向导。这个向导允许选择身份验证类型、添加角色、添加用户，以及站点的安全区域。

41.4.1 使用 Security Setup 添加 Forms 身份验证功能

为了便于说明，在 C:\ProCSharp\Chapter41\目录下创建一个新的空白 Web 站点 PCSAuthentication Demo。之后，使用如下示例步骤配置安全性：

- (1) 打开 Web Site ASP.NET Configuration 工具。
- (2) 导航到 Security 选项卡。
- (3) 单击 Use the security setup wizard to configure security step by step 链接。
- (4) 阅读其中的信息后，单击第(1)步中的 Next 按钮。
- (5) 在向导的第(2)步中，选择 From the internet，以选中 Forms 身份验证，单击 Next 按钮。
- (6) 在确认使用默认的 Advanced provider settings 提供程序存储安全信息后，再次单击 Next 按钮。



这个提供程序的信息可以通过 Provider 选项卡配置，在 Provider 选项卡中，可以选择把信息存储到其他地方，如 SQL Server 数据库中，但选择默认的 SQL Server Express 数据库将便于演示。

- (7) 选择 Enable roles for this Web site 选项，单击 Next 按钮。
- (8) 添加两个角色 Administrator 和 User，单击 Next 按钮。
- (9) 添加两个用户，也称为 Administrator 和 User。注意密码(在 machine.config 中定义)的默认安全角色是相当强的，密码至少有 7 个字符，至少包括 1 个符号字符和混合了大小写的字母。
- (10) 单击 Next 按钮，然后再次单击 Next 按钮，跳过 Add New Access 页面。



在默认情况下，所有用户和角色都可以访问站点的所有区域。在 Add New Access Rules 页面中，可以限制角色、用户或匿名用户的访问区域。可以为站点的每个目录限制访问区域，因为这可以通过目录中的 web.config 文件实现，如后面所示。

- (11) 单击 Finish 按钮。
- (12) 在 Security 选项卡的主页面上，单击 Manage users 链接。
- (13) 依次为每个用户单击 Edit Roles 链接，给两个角色添加 Administrator 用户，给 User 角色仅

添加 User 用户。

完成上述步骤后, 就有一个用户系统了, 其中包含角色和用户。

为了验证其作用, 必须给 Web 站点添加几个控件, 进行身份验证。下面几节的示例代码假定, Web 站点用本节描述的 User、Administrator 角色和用户来配置。

41.4.2 实现登录系统

如果在运行安全向导后打开 web.config 文件, 就会看到其中修改的如下内容:

```
<roleManager enabled="true" />
```

和

```
<authentication mode="Forms" />
```

这似乎不太像我们刚才做的工作, 但注意许多信息存储在一个 SQL Server Express 数据库中, 该数据库在 App_Data 目录下, 命名为 ASPNETDB.MDF(需要单击 Solution Explorer 窗口中的 refresh 按钮, 才能看到这个目录)。使用任意标准数据库管理工具都可以查看存储在这个文件中的数据, 包括 Visual Studio。如果非常小心, 甚至就可以直接在这个数据库中添加用户和角色。

在默认情况下, 登录过程通过 Web 站点根目录下的一个 Login.aspx 页面实现。如果用户试图导航到无权访问的位置, 在成功登录后, 他们就会自动重定向到这个页面, 并返回希望的位置。

在 PCSAuthenticationDemo 站点中添加两个 Web 窗体 Default.aspx 和 Login.aspx, 把一个 Login 控件从工具箱拖放到 Login.aspx 窗体上。

这就是允许用户登录到 Web 站点上所需的全部工作。在浏览器上打开站点, 导航到 Login.aspx(现在必须手工执行这个操作, 因为没有限制对 Default.aspx 的访问), 再输入在 Security Setup 向导中添加的一个用户的详细信息, 如图 41-5 所示。

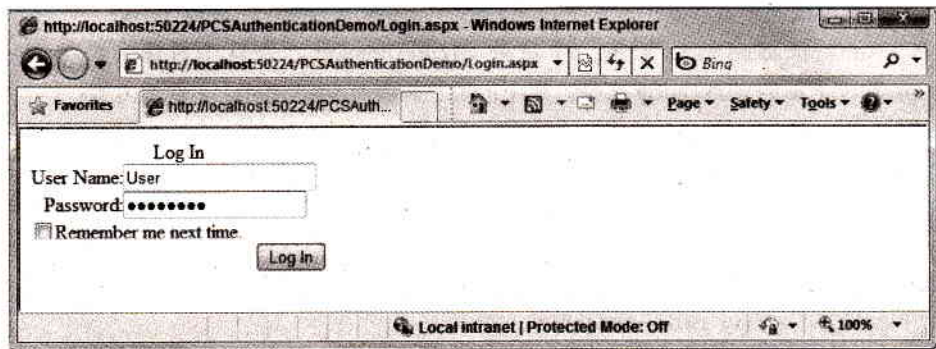


图 41-5

登录后, 就会返回 Default.aspx, 目前这是一个空页面。

41.4.3 Web 登录服务器控件

工具箱的 Login 部分包含几个控件, 如表 41-3 所示。

表 41-3

控 件	说 明
Login	如前所示，这个控件允许用户登录 Web 站点。这个控件的大多数属性都用于给所提供的模板指定样式。还可以使用 DestinationPageUrl 属性强制登录后重定向到指定的位置，使用 VisibleWhenLoggedIn 属性可以确定登录的用户是否能看到该控件。另外，可以使用各种文本属性，如 CreateUserText，可以输出对用户有帮助的消息
LoginView	这个控件可以显示各种内容，这取决于用户是否登录，或用户的角色是什么。可以把内容放在<AnonymousTemplate>和<LoggedInTemplate>，以及<RoleGroups>中来控制这个控件的输出
PasswordRecovery	这个控件允许用户把密码发送给他自己，它可以使用为用户定义的密码恢复问题。它的大多数属性也用于显示格式，但 MailDefinition-Subject 属性用于配置发送给用户的地址的电子邮件，SuccessPageUrl 属性在要求用户输入密码后重定向用户
LoginStatus	该控件显示 Login 或 Logout 链接，其文本和图像都可以定制，这取决于用户是否登录
LoginName	对于当前登录的用户输出用户名
CreateUserWizard	该控件显示一个窗体，用户可以使用该窗体注册到站点上，并添加到用户列表中。与其他登录控件一样，它也有许多与布局格式相关的属性，但默认的格式已完全可用了
ChangePassword	该控件允许用户修改密码，它有 3 个字段，分别表示旧密码、新密码和确认密码。它也有许多样式属性

这些控件将在 41.4.5 节中使用。

41.4.4 保护目录

最后要讨论的是如何限制对目录的访问。这可以通过前面介绍的 Web Site Configuration 工具实现，但手工完成也很简单。

给 PCSAuthenticationDemo 添加一个目录 SecureDirectory，在这个目录中添加 Web 页面 Default.aspx 和一个新的 web.config 文件。用下面的代码替代 web.config 文件的内容：



可从
wrox.com
下载源代码

```
<?xml version="1.0" ?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
      <allow roles="Administrator" />
      <deny roles="User" />
    </authorization>
  </system.web>
</configuration>
```

代码段 PCSAuthenticationDemo/SecureDirectory/web.config

<authorization>元素可以包含一个或多个表示权限规则的<deny>或<allow>元素，每个元素都有一个 users 或 roles 属性，表示该规则应用于什么成员。因为规则从上到下应用，所以如果规则的成员关系有重叠，那么较特殊的规则一般应放在靠上的位置。在这个例子中，“？”表示匿名用户，他们和 User 角色中的用户会被拒绝访问这个目录。注意，只有这里的<allow>规则放在 User 角色的<deny>规则之前，才表示 User 和 Administrator 角色中的用户允许访问这个目录——某个用户的所

有角色都考虑在内，但规则的顺序仍有效。

现在，在登录到 Web 站点，导航到 SecureDirectory/Default.aspx 时，只有位于 Admin 角色中，才能访问该页面。其他用户或未通过身份验证的用户都会被重定向到登录页面。

41.4.5 PCSDemoSite 中的安全性

PCSDemoSite 站点使用了前面介绍的 Login 控件、LoginView 控件、LoginStatus 控件、LoginName 控件、PasswordRecovery 控件和 ChangePassword 控件。

一个区别是其中包含 Guest 角色，其结果是 guest 用户不能修改其密码，这使用 LoginView 比较合适，如 Login.aspx 所示：



```
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
    Runat="server">
    <h2>Login Page</h2>
    <asp:LoginView ID="LoginView1" Runat="server">
        <RoleGroups>
            <asp:RoleGroup Roles="Guest">
                <ContentTemplate>
                    You are currently logged in as <b>
                        <asp:LoginName ID="LoginName1" Runat="server" /></b>.
                    <br />
                    <br />
                    <asp:LoginStatus ID="LoginStatus1" Runat="server" />
                </ContentTemplate>
            </asp:RoleGroup>
            <asp:RoleGroup Roles="RegisteredUser,SiteAdministrator">
                <ContentTemplate>
                    You are currently logged in as <b>
                        <asp:LoginName ID="LoginName2" Runat="server" /></b>.
                    <br />
                    <br />
                    <asp:ChangePassword ID="ChangePassword1" Runat="server">
                    </asp:ChangePassword>
                    <br />
                    <br />
                    <asp:LoginStatus ID="LoginStatus2" Runat="server" />
                </ContentTemplate>
            </asp:RoleGroup>
        </RoleGroups>
        <AnonymousTemplate>
            <asp:Login ID="Login1" Runat="server">
            </asp:Login>
            <asp:PasswordRecovery ID="PasswordRecovery1" Runat="Server" />
        </AnonymousTemplate>
    </asp:LoginView>
</asp:Content>
```

代码段 PCSAuthenticationDemo/Login.aspx

这里的视图会显示下述几个页面中的一个：

- 对匿名用户显示 Login 控件和 PasswordRecovery 控件。
- 对 Guest 用户显示 LoginName 控件和 LoginStatus 控件，根据需要，还会显示登录的用户名，并允许注销。

- 对 RegisteredUser 和 SiteAdministrator 用户显示 LoginName 控件、LoginStatus 控件和 Change Password 控件。

该站点在各个目录中还包含各自的 web.config 文件，以限制访问，也可以根据角色限制导航到其他地方。



注意，为站点配置的用户显示在 About 页面上，也可以添加自己的已配置用户。基本站点的用户(及其密码)是 User1(User1!!)、Admin(Admin!!)和 Guest(Guest!!)。

这里要注意，尽管站点的根目录拒绝匿名用户，但 Themes 目录(详见下一节)重写了这个设置，方法是允许匿名用户访问。这是必需的，因为如果没有重写该设置，匿名用户就会看到没有主题的站点，因为主题文件不能访问。另外，根文件 web.config 中的完整安全规范如下：



可从
wrox.com
下载源代码

```
<location path="StyleSheet.css">
  <system.web>
    <authorization>
      <allow users="?" />
    </authorization>
  </system.web>
</location>
<system.web>
  <authorization>
    <deny users="?" />
  </authorization>
  ...
</system.web>
```

代码段 PCSDemoSite/web.config

其中，<location>元素用来重写用 path 属性指定的特定文件的默认设置，这里是 StyleSheet.css 文件。根据个人爱好，<location>元素可用来把任意<system.web>设置应用于指定的文件或目录，并可以把所有特定于目录的设置集中在一个地方(替代多个 web.config 文件)。在上面的代码中，给匿名用户授予了访问 Web 站点中根样式表的权限，这是必需的，因为这个文件在母版页中定义了<div>元素的布局。不这么做，为匿名用户在登录页面上显示的 HTML 就很难读懂。

另一个要注意的地方是，在会议室登记工具用户控件的代码隐藏文件中，有如下 Page_Load() 事件处理程序：



可从
wrox.com
下载源代码

```
void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        nameBox.Text = Context.User.Identity.Name;
        DateTime trialDate = DateTime.Now;
        calendar.SelectedDate = GetFreeDate(trialDate);
    }
}
```

代码段 PCSDemoSite/MRB/MRB.ascx.cs

其中，从当前的上下文环境中提取用户名。注意，在代码隐藏文件中，还要频繁使用 Context.User。

IsInRole()方法来检查访问。

41.5 主题

在 ASP.NET 页面中合并使用母版页和 CSS 样式表后,还要把窗体和函数分开,即把页面的外观和操作方式分开定义。利用主题,可以在一步中完成这个任务,并可以把所提供的几个主题之一动态应用于页面的外观。

主题包含如下内容:

- 主题的名称
- 可选的 CSS 样式表
- 可以给各个控件类型设置样式的外观文件(.skin)

这些内容以两种不同的方式应用于页面:通过 Theme 或 StyleSheetTheme 属性。

- Theme: 所有外观属性都应用于控件,重写页面上控件已有的任何属性
- StyleSheetTheme: 已有的控件属性优先于外观文件中定义的属性

CSS 样式表的工作方式与方法的使用方式相同,因为它们都以标准的 CSS 方式应用。

41.5.1 把主题应用于页面

可以用几种声明或编程的方式把主题应用于页面,应用主题最简单的声明方式是在<%@ Page %>指令中使用 Theme 或 StyleSheetTheme 属性:

```
<%@ Page Theme="myTheme" ... %>
```

或者:

```
<%@ Page StyleSheetTheme="myTheme" ... %>
```

其中, myTheme 是给主题定义的名称。

另外,还可以在 Web 站点的 web.config 文件中使用一项,给该站点上的所有页面指定要使用的主题:

```
<configuration>
  <system.web>
    <pages Theme="myTheme" />
  </system.web>
</configuration>
```

这里也可以使用 Theme 或 StyleSheetTheme 属性。还可以使用<location>元素重写某个页面或目录的这个设置,其方式与上一节中给安全信息使用该元素的方式相同,从而更具体地设置主题。

如果采用编程方式,就可以在页面的代码隐藏文件中应用主题,但只能在 Page_PreInit()事件处理程序中应用主题,该事件处理程序在页面的生命周期的早期触发。在这个事件中,只需把 Page.Theme 或 Page.StyleSheetTheme 属性设置为要应用的主题的名称即可,例如:

```
protected override void OnPreInit(EventArgs e)
{
    Page.Theme = "myTheme";
}
```

因为通过代码应用主题,所以可以动态地应用一组主题中的一个主题文件。这个技巧将在 41.5.3 节中使用。

41.5.2 定义主题

主题在 ASP.NET 中的另一个“特殊”目录(这里是 App_Themes)中定义。App_Themes 目录可以包含任意多个子目录,每个子目录下有一个主题,其中子目录名定义主题名。

定义主题时,需要把主题的所有必要文件放在主题子目录中。对于 CSS 样式表,不需要考虑文件名;主题系统仅查找扩展名为.css 的文件。同样,尽管建议使用多个.skin 文件,每个.skin 文件用于待设置外观的一个控件类型,每个.skin 文件的名称都用该控件名指定,但.skin 文件也可以有任意文件名。

外观文件包含服务器控件的定义,其格式与标准 ASP.NET 页面中使用的格式相同。其区别是外观文件中的控件不会添加到页面中,它们只用于提取属性。按钮外观的定义一般放在 Button.skin 文件中,其内容可能如下所示:

```
<asp:Button Runat="server" BackColor="#444499" BorderColor="#000000"
    ForeColor="#ccccff" />
```

这个外观实际上是从 PCSDemoSite 的 DefaultTheme 主题中提取的,负责设置本章前面 Meeting Room Booker 页面中的按钮外观。

以这种方式为控件类型创建外观时,不需要使用 ID 属性。

41.5.3 PCSDemoSite 中的主题

Web 站点 PCSDemoSite 包含 3 个主题,可以在/Configuration/Themes/Default.aspx 页面上选择这 3 个主题——只要作为 RegisteredUser 或 SiteAdministrator 角色的一个成员登录即可。这个页面如图 41-6 所示。

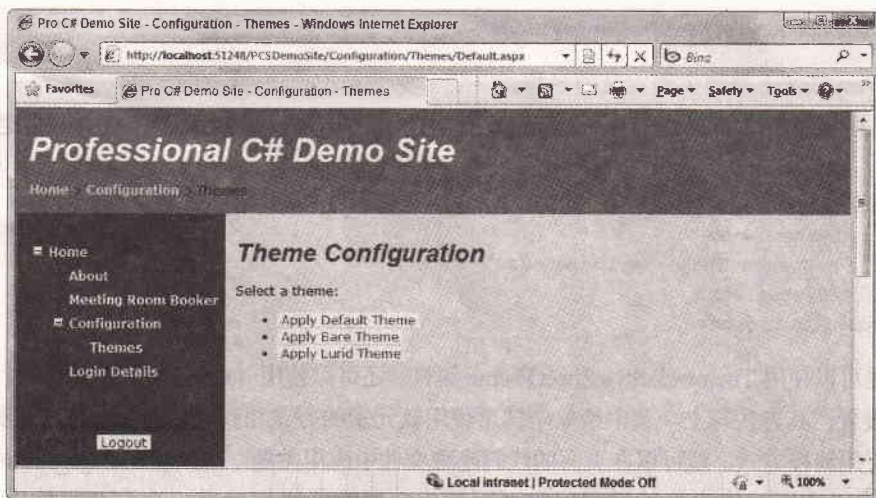


图 41-6

虽然这里使用的主题是 DefaultTheme,但也可以在这个页面上选择其他选项,图 41-7 显示了 BareTheme 主题。

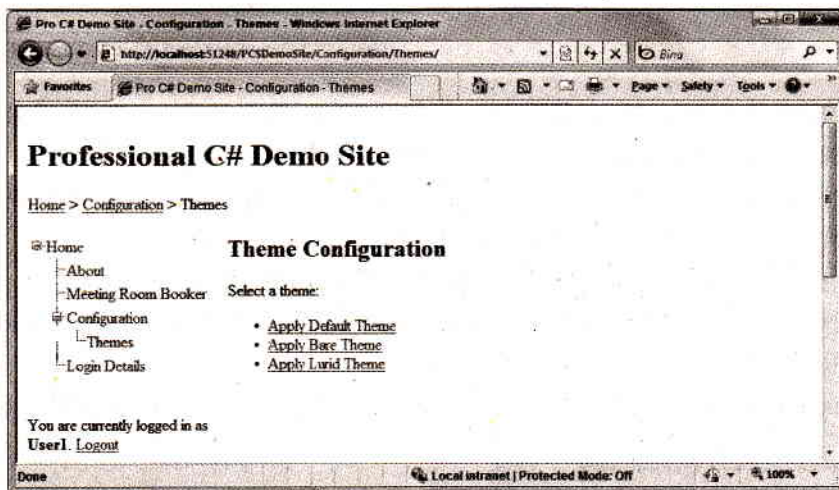


图 41-7

例如，这类主题适合于 Web 页面的可打印版本。BareTheme 目录实际上并不包含文件，这里使用的文件是根样式表 StyleSheet.css。

示例站点还包含第 3 个命名为 LuridTheme 的主题。这个主题的颜色非常鲜亮，但很难阅读，它说明了站点的外观如何可以使用主题动态改变。更重要的是，像这样的主题可以提供 Web 站点的高对比度或大文本版本，以便于访问。

在 PCSDemoSite 中，因为当前选择的主题存储在会话状态中，所以在站点四周导航时，主题会保持不变。/Configuration/Themes/Default.aspx 的代码隐藏文件如下所示：



```
public partial class _Default : MyPageBase
{
    private void ApplyTheme(string themeName)
    {
        if (Session["SessionTheme"] != null)
        {
            Session.Remove("SessionTheme");
        }
        Session.Add("SessionTheme", themeName);
        Response.Redirect("~/Configuration/Themes", true);
    }

    void applyDefaultTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("DefaultTheme");
    }

    void applyBareTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("BareTheme");
    }

    void applyLuridTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("LuridTheme");
    }
}
```

代码段 PCSDemoSite/Configuration/Themes/Default.aspx.cs

这里的关键功能在 `ApplyTheme()` 方法中，它使用 `SessionTheme` 键把所选主题的名称放在会话状态中，并确定会话状态中是否已有一个条目，如果有，就删除它。

如前所述，主题必须在 `Page_PreInit()` 事件处理程序中应用。因为不能从所有页面使用的母版页中访问它，所以，如果要把选中的主题应用于所有页面，就有两个选项：

- 在所有要应用主题的页面中，重写 `Page_PreInit()` 事件处理程序
- 为所有要应用主题的页面提供一个公共基类，再重写这个基类中的 `Page_PreInit()` 事件处理程序

PCSDemoSite 使用第二个选项，在 `Code/MyPageBase.cs` 中提供了一个公共页面基类：



```
public class MyPageBase : Page
{
    protected override void OnPreInit(EventArgs e)
    {
        // theming
        if (Session["SessionTheme"] != null)
        {
            Page.Theme = Session["SessionTheme"] as string;
        }
        else
        {
            Page.Theme = "DefaultTheme";
        }

        // base call
        base.OnPreInit(e);
    }
}
```

代码段 PCSDemoSite/App_Code/MyPageBase.cs

这个事件处理程序检查 `SessionTheme` 中条目的会话状态，如果会话状态中有一项，就应用选中的主题；否则就使用 `DefaultTheme`。

也要注意这个类继承自通用的页面基类 `Page`，这是必需的，否则页面就不能作为一个 ASP.NET Web 页面运作。

为了使程序正常工作，还要为所有 Web 页面指定这个基类。这有几种方式，最简单的方式是在页面的 `<@ Page %>` 指令或页面的代码隐藏文件中指定。前一种策略适合于简单的页面，但不能使用的页面自定义代码隐藏，因为页面不再其自定义代码隐藏文件中使用代码。另一种方式是修改页面在代码隐藏文件中继承的类。在默认情况下，新页面继承自 `Page`，但可以改变这种继承关系。在前面主题选择页面的代码隐藏文件中，注意有如下代码：



```
public partial class _Default : MyPageBase
{
    ...
}
```

代码段 PCSDemoSite/Configuration/Themes/Default.aspx.cs

因为这里把 `MyPageBase` 指定为 `_Default` 类的基类，所以使用在 `MyPageBase.cs` 中重写的方法。

41.6 Web 部件

ASP.NET 包含一组名为 Web 部件的服务器控件，它们允许用户使 Web 页面个性化。例如，在基于 SharePoint 的 Web 站点和 My MSN 主页 <http://www.msn.com/> 上都可以看到这些控件。在使用 Web 部件时，可以得到的功能如下：

- 给用户显示默认的页面布局。这个布局包含许多 Web 部件组件，每个 Web 部件都有标题和内容。
- 用户可以修改 Web 部件在页面上的位置。
- 用户可以定制页面上 Web 部件的外观，或者完全从页面中删除它们。
- 为用户提供一类 Web 部件，允许用户将此类 Web 部件添加到页面中。
- 用户可以从页面中导出 Web 部件，再把它们导入另一个页面或站点上。
- Web 部件之间可以存在连接关系。例如，显示在一个 Web 部件中的内容可以是显示在另一个 Web 部件中的内容的图形表示。
- 用户进行的任何修改都可以在访问站点的过程中保存下来。

ASP.NET 为移植 Web 部件功能提供了一个完整的架构，包括管理控件和编辑控件。

Web 部件的使用是一个复杂的主题，本节不描述 Web 部件的所有功能，也不列出 Web 部件组件提供的所有属性和方法，只概述 Web 部件，让读者体验和理解它的基本功能。

41.6.1 Web 部件应用程序组件

工具箱的 Web 部件部分包含 13 个控件，如表 41-4 所示。该表还介绍了 Web 部件页面的一些重要概念。

表 41-4

控 件	说 明
WebPartManager	每个使用 Web 部件的页面必须有一个(只能有一个)WebPartManager 控件的实例，尽管只要需要在页面上使用 Web 部件时，就应使用母版页。但根据个人爱好可以把它放在母版页上。这个控件负责管理大部分 Web 部件功能，不需要用户太多的干涉。只要根据自己需要的功能，将该控件放在 Web 页面上即可。对于更高级的功能，可以使用这个控件提供的许多属性和事件
ProxyWebPartManager	如果把 WebPartManager 控件放在母版页上，就很难在各个页面上配置它——实际上不能以声明方式这么做。这与 Web 部件之间的静态连接的定义尤其相关。ProxyWebPartManager 控件允许在 Web 页面上以声明方式定义静态连接，这避免出现不能在同一个页面上放置两个 WebPartManager 控件的问题
WebPartZone	WebPartZone 控件用于定义可以包含 Web 部件的页面区域。一般在页面上使用多个 WebPartZone 控件。例如，可以在页面的 3 列布局中使用 3 个 WebPartZone 控件。用户可以在 WebPartZone 区域之间移动 Web 部件，或者在一个 WebPartZone 区域中重新定位它们

(续表)

控 件	说 明
CatalogZone	CatalogZone 控件允许用户把 Web 部件添加到页面中。这个控件包含的控件派生自 CatalogZone, CatalogZone 提供了 3 个控件,本表后面 3 项将介绍这 3 个控件。CatalogZone 控件及其包含的控件是否可见,取决于 WebPartManager 设置的当前显示模式
DeclarativeCatalogPart	DeclarativeCatalogPart 控件允许定义内联的 Web 部件控件。之后,这些控件就可以通过 CatalogZone 控件用于用户
PageCatalogPart	用户可以删除(关闭)显示在页面上的 Web 部件。为了检索它们, PageCatalogPart 控件提供了一个可以在页面上替换的、已关闭的 Web 部件列表
ImportCatalogPart	ImportCatalogPart 控件允许通过 CatalogPart 接口把从页面中导出的 Web 部件再导入另一个页面中
EditorZone	EditorZone 控件包含的控件允许用户根据 Web 部件包含的控件,编辑 Web 部件的显示和行为的各个方面。它可以包含派生自 EditorPart 的控件,包括本表后面 4 行中的 4 个控件。与 CatalogZone 控件一样,这个控件的显示取决于当前显示模式
AppearanceEditorPart	这个控件允许用户修改 Web 部件控件的外观和大小,并能隐藏它们
BehaviorEditorPart	这个控件允许用户使用该控件的许多属性配置 Web 部件的行为,例如,Web 部件是否可以关闭,Web 部件的标题链接到什么 URL 上
LayoutEditorPart	这个控件允许用户修改 Web 部件的布局属性,例如,它包含在什么区域,在最小化状态下它是否显示
PropertyGridEditorPart	这是最一般的 Web 部件编辑器控件,它允许定义可以为自定义 Web 部件控件编辑的属性。之后,用户就可以编辑这些属性
ConnectionsZone	这个控件允许用户在提供连接功能的 Web 部件之间创建连接。与 CatalogZone 控件和 EditorZone 控件不同,在这个控件中不放置任何其他控件。这个控件生成的用户界面取决于页面上可用于连接的控件。这个控件的可见性取决于显示模式

注意,表 41-4 中的控件不包含任何 Web 部件控件。这是因为这些控件是我们自己创建的。任何放在 WebPartZone 区域中的控件会自动变成 Web 部件,包括(最重要的是)用户控件。使用用户控件,可以把其他控件组合在一起,提供 Web 部件控件的用户界面和功能。

41.6.2 Web 部件示例

为了说明 Web 部件的功能,可以查看本章可下载代码中的另一个示例 Web 站点 PCSWebParts。这个示例将使用 PCSAuthenticationDemo 示例的同一个安全数据库。它有两个用户,其用户名是 User 和 Administrator,密码都是 Pa\$\$w0rd。还可以作为一个用户登录,处理页面上的 Web 部件,注销,之后再作为另一个用户登录,以完全不同的方式处理 Web 部件。这两个用户的个性化会在站点访问的过程中保存下来。

登录到站点上后,最初显示的结果(用 User 登录)如图 41-8 所示。

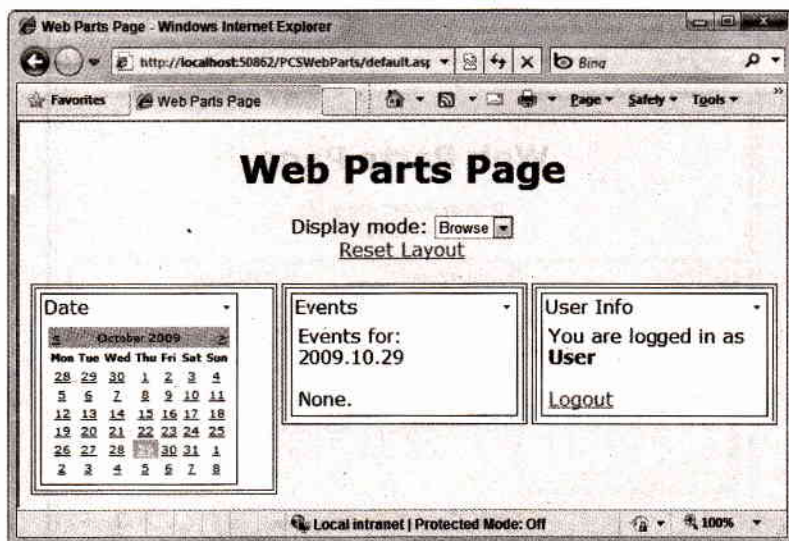


图 41-8

这个页面包含如下控件：

- 1 个 WebPartManager 控件(没有可见的组件)。
- 3 个 WebPartZone 控件。
- 3 个 Web 部件(Date、Events 和 User Info)，分别放在 3 个 WebPartZone 控件中。其中两个 Web 部件通过一个静态连接关联起来，如果修改 Date 部件中的日期，Events 部件中显示的日期就会更新。
- 改变显示模式的下拉列表。这个列表不包含所有可能的显示模式，只包含可用的显示模式。可用的模式从 WebPartManager 控件中获得，如后面所示。列出的模式有：
 - Browse —— 这种模式是默认的，允许查看和使用 Web 部件。在这种模式下，每个 Web 部件都可以使用下拉菜单最小化和关闭，下拉菜单可以从每个 Web 部件的右上角访问。
 - Design —— 在这种模式下，可以重新定位 Web 部件。
 - Edit —— 在这种模式下，可以编辑 Web 部件属性。每个 Web 部件的下拉菜单中，有另一个开始可用的菜单项 Edit。
 - Catalog —— 在这种模式下，可以给页面添加新 Web 部件。
- 一个链接，将 Web 部件布局重置为默认布局(仅用于当前用户)
- 一个 EditorZone 控件(只在 Edit 模式下可见)
- 一个 CatalogZone 控件(只在 Catalog 模式下可见)
- 类别中可以添加到页面中的另一个 Web 部件

每个 Web 部件都在用户控件中定义。

(1) Web 部件的操作

为了说明布局的修改方式，可使用下拉列表将显示模式改为 Design。注意，然后每个 WebPartZone 控件都带有一个 ID 值(分别是 LeftZone、CenterZone 和 RightZone)。还可以通过拖动标题来移动 Web 部件，在拖动过程中甚至可以看到可视化的反馈效果。如图 41-9 所示，其中显示了正在移动的名 Date 的 Web 部件。

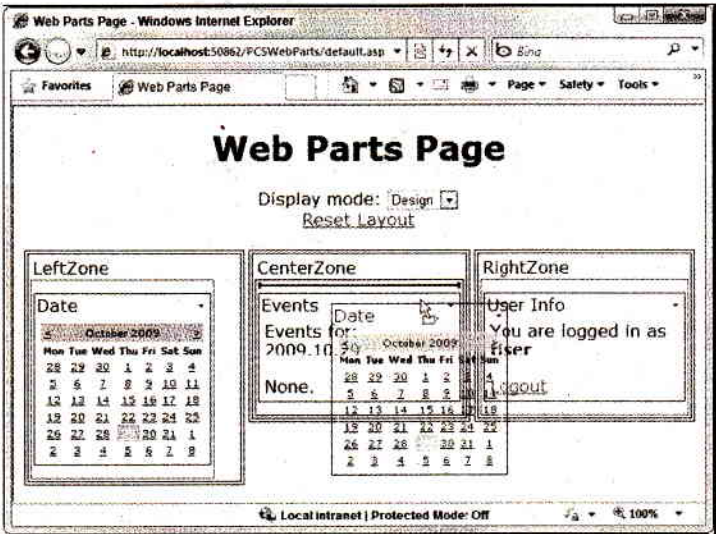


图 41-9

接着，从类别中添加一个新的 Web 部件。将显示模式改为 Catalog，注意 CatalogZone 在页面的底部开始可见。单击 Declarative Catalog 链接，就可以在页面上添加一个 Links 控件，如图 41-10 所示。

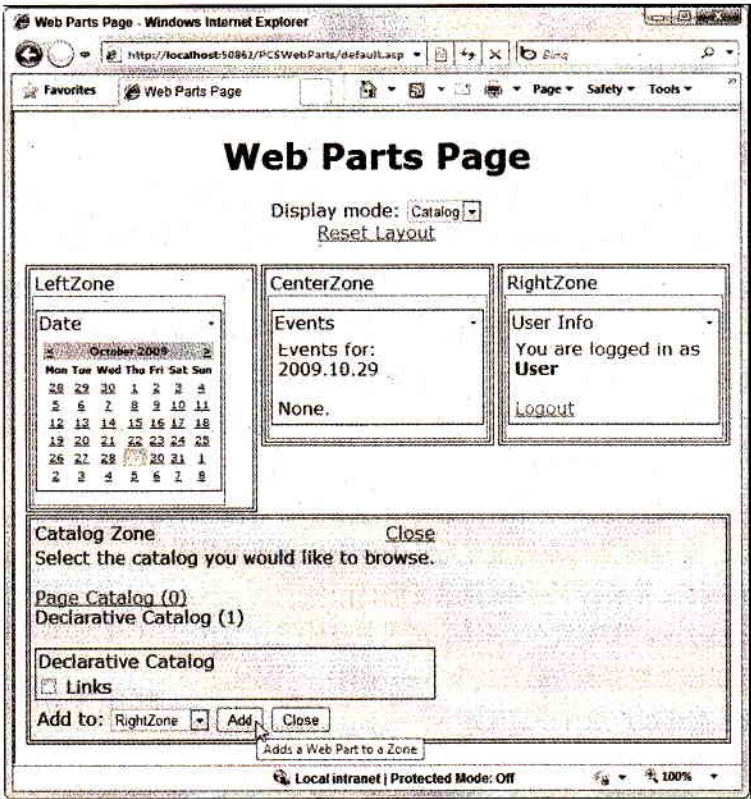


图 41-10

注意这里还有一个 Page Catalog 链接。如果对于该部件使用下拉菜单选择 Web 部件，就会看到 Page Catalog 链接，它没有完全删除，只是隐藏了。

之后，把显示模式改为 Edit，再从 Web Part 的下拉列表中选择 Edit 项，如图 41-11 所示。

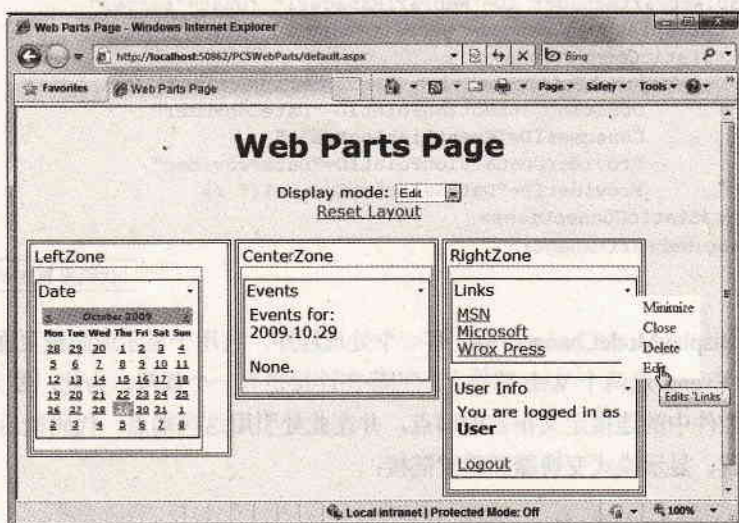


图 41-11

选择这个菜单项时，会打开 EditorZone 控件。在本例中，这个控件包含一个 AppearanceEditorPart 控件，如图 41-12 所示。

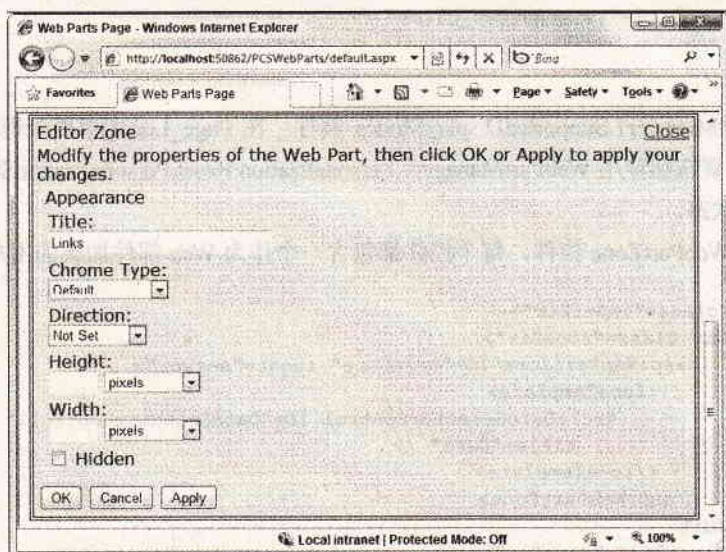


图 41-12

使用这个界面可以编辑和应用 Web 部件的属性值。

完成修改后，确认存储了它们，以便用户注销，再作为另一个用户登录，之后切换回第一个用户。

(2) Web 部件包含的页面代码

现在，读者可能以为这个功能需要许多代码。实际上，本例的代码相当简单。查看 Web 部件页面的代码。<form>元素以一个 WebPartManager 控件开头：



可从
wrox.com
下载源代码

```
<form id="form1" runat="server">
  <asp:WebPartManager ID="WebPartManager1" runat="server"
    OnDisplayModeChanged="WebPartManager1_DisplayModeChanged">
    <StaticConnections>
      <asp:WebPartConnection ID="dateConnection"
        ConsumerConnectionPointID="DateConsumer"
        ConsumerID="EventListControl1"
        ProviderConnectionPointID="DateProvider"
        ProviderID="DateSelectorControl1" />
    </StaticConnections>
  </asp:WebPartManager>
```

代码段 PCSWebParts/Default.aspx

这个控件的 `DisplayModeChanged` 事件有一个处理程序，它用于显示或隐藏页面底部的 `<div>` 编辑器。在 `Date` 和 `Events` 这两个 Web 部件之间的静态连接也有一个规范。为此，要为用于这些 Web 部件的两个用户控件中的连接定义指定的端点，并在此处引用这些端点。代码稍后列出。

接着定义标题、显示模式变换器和重置链接：

```
<div class="mainDiv">
  <h1> Web 部件 Page </h1>
  Display mode:
  <asp:DropDownList ID="displayMode" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="displayMode_SelectedIndexChanged" />
  <br />
  <asp:LinkButton runat="server" ID="resetButton" Text="Reset Layout"
    OnClick="resetButton_Click" />
  <br />
  <br />
```

使用 `WebPartManager1.SupportedDisplayModes` 属性，在 `Page_Load()` 事件处理程序中填充显示模式下拉列表。重置按钮使用 `WebPartManager1.Personalization.Reset.PersonalizationState()` 方法给当前用户重置个性化状态。

接着是 3 个 `WebPartZone` 控件，每个控件都包含一个作为 Web 部件加载的用户控件：

```
<div class="innerDiv">
  <div class="zoneDiv">
    <asp:WebPartZone ID="LeftZone" runat="server">
      <ZoneTemplate>
        <uc1:DateSelectorControl ID="DateSelectorControl1" runat="server"
          title="Date" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
  <div class="zoneDiv">
    <asp:WebPartZone ID="CenterZone" runat="server">
      <ZoneTemplate>
        <uc2:EventListControl ID="EventListControl1" runat="server"
          title="Events" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
  <div class="zoneDiv">
    <asp:WebPartZone ID="RightZone" runat="server">
      <ZoneTemplate>
        <uc4:UserInfo ID="UserInfo1" runat="server" title="User Info" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
```

```

    </asp:WebPartZone>
</div>

```

最后是 EditorZone 控件和 CatalogZone 控件,它们分别包含 AppearanceEditor 控件、PageCatalogPart 控件和 DeclarativeCatalogPart 控件:

```

<asp:PlaceHolder runat="server" ID="editorPH" Visible="false">
  <div class="footerDiv">
    <asp:EditorZone ID="EditorZone1" runat="server">
      <ZoneTemplate>
        <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
          runat="server" />
      </ZoneTemplate>
    </asp:EditorZone>
    <asp:CatalogZone ID="CatalogZone1" runat="server">
      <ZoneTemplate>
        <asp:PageCatalogPart ID="PageCatalogPart1" runat="server" />
        <asp:DeclarativeCatalogPart ID="DeclarativeCatalogPart1"
          runat="server">
          <WebPartsTemplate>
            <uc3:LinksControl ID="LinksControl1" runat="server"
              title="Links" />
          </WebPartsTemplate>
        </asp:DeclarativeCatalogPart>
      </ZoneTemplate>
    </asp:CatalogZone>
  </div>
</asp:PlaceHolder>
</div>
</div>
</form>

```

DeclarativeCatalogPart 控件包含第 4 个用户控件,这是一个 Links 控件,用户可以把它添加到页面中。

(3) Web 部件的代码

Web 部件的代码相当简单。例如,Web 部件 Links 只包含下述代码:



可从
wrox.com
下载源代码

```

<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="LinksControl.ascx.cs"
Inherits="LinksControl" %>
<a href="http://www.msn.com/"> MSN </a>
<br />
<a href="http://www.microsoft.com/"> Microsoft </a>
<br />
<a href="http://www.wrox.com/"> Wrox Press </a>

```

代码段 PCSWebParts/LinksControl.ascx

不需要额外的标记,就可以使这个用户控件作为一个 Web 部件工作。这里唯一要注意的是,用户控件(与 Default.aspx 中使用的相同)的<uc3:LinksControl>元素有一个 title 属性,尽管用户控件没有 Title 属性。DeclarativeCatalogPart 控件使用这个属性来推断 Web 部件要显示的标题(可以在运行期间用 AppearanceEditorPart 控件编辑)。

(4) 连接控件

将一个接口引用从 `DateSelectorControl` 传递给 `EventListControl`(这些 Web 部件使用的两个用户控件类), 就建立了 `Date` 控件和 `Events` 控件之间的连接。



可从
wrox.com
下载源代码

```
public interface IDateProvider
{
    SelectedDatesCollection SelectedDates
    {
        get;
    }
}
```

代码段 PCSWebParts/App Code/IDateProvider.cs

因为 `DateSelectorControl` 支持这个接口, 所以可以使用 `this` 传送 `IDateProvider` 接口的一个实例。引用通过 `DateSelectorControl` 中的端点方法来传递, 该引用用 `ConnectionProvider` 属性修饰:



可从
wrox.com
下载源代码

```
[ConnectionProvider("Date Provider", "DateProvider")]
public IDateProvider ProvideDate()
{
    return this;
}
```

代码段 PCSWebParts/DateSelectorControl.ascx.cs

这就是把 Web 部件标记为一个提供程序控件所需做的全部工作。之后, 就可以通过端点 ID 引用提供程序, 在本例中是 `DateProvider`。

要使用提供程序, 可以使用 `ConnectionConsumer` 属性在 `EventListControl` 中修饰一个使用方法:



可从
wrox.com
下载源代码

```
[ConnectionConsumer("Date Consumer", "DateConsumer")]
public void GetDate(IDateProvider provider)
{
    this.provider = provider;
    IsConnected = true;
    SetDateLabel();
}
```

代码段 PCSWebParts/EventListControl.ascx.cs

这个方法存储了一个传递过来的 `IDateProvider` 接口引用, 设置一个标记, 并修改控件中的标签文本。

这个例子没有更多需要解释的地方。示例中有几个小段装饰代码, 还有 `Page_Load()` 中事件处理程序的详细信息, 但这里不需要讨论它们。查看本章的可下载代码, 可以进一步研究它们。

但是 Web 部件可以完成更多工作。Web 部件架构非常强大, 功能非常丰富, 需要一整本书的篇幅来探讨。但本节概述了 Web 部件, 并揭示了它们的一些功能。

41.7 ASP.NET AJAX

Web 应用程序编程是一个不断变化和改进的主题。如果花点时间查看当前的 Internet, 就会注意到, 最近的 Web 站点在可用性方面比老网站好得多。许多目前最好的 Web 站点都提供了丰富的用户界面, 其响应能力与 Windows 应用程序差不多。它们使用客户端处理技术实现, 主要通过 JavaScript 代码和不断利用 Ajax 技术。

Ajax 并不是一种新技术, 它只是一组合并的标准, 这组标准可以实现当前 Web 浏览器丰富的潜在功能。

在支持 Ajax 的 Web 应用程序中, 最重要的特性是 Web 浏览器能在带外操作中与 Web 服务器通信。这称为异步回发或部分页面的回发。实际上, 这意味着用户可以与服务器端的功能和数据交互, 而无需更新整个页面。例如, 单击一个链接, 移动到表的第二页数据上时, Ajax 可以只刷新表的内容, 而不刷新整个 Web 页面。也就是说, 需要的 Internet 通信量较少, 从而使 Web 应用程序的响应比较快。

本节将在代码中使用 Ajax 的 Microsoft 实现方式, 它称为 ASP.NET AJAX。这种实现方式采用 Ajax 模型, 并将它应用于 ASP.NET 架构。ASP.NET AJAX 提供了许多服务器控件和客户端技术, 它们专门面向 ASP.NET 开发人员, 从而可以毫不费力地在 Web 应用程序中添加 Ajax 功能。

本节内容如下:

- Ajax 和实现 Ajax 的技术。
- ASP.NET AJAX 及其组成部分, 以及 ASP.NET AJAX 提供的功能。
- 如何通过服务器端和客户端代码在 Web 应用程序中使用 ASP.NET AJAX。这是本节最大的一部分。

41.8 Ajax 的概念

Ajax 允许通过异步回发和动态的客户端 Web 页面操作, 改进 Web 应用程序的用户界面。术语“Ajax”由 Jesse James Garrett 提出, 是 Asynchronous JavaScript and XML 的缩写。



注意, Ajax 不是一个缩写词, 因此它不能写作 AJAX。但是, 在产品名称 ASP.NET AJAX 中它是大写, 这是 Ajax 的 Microsoft 实现方式, 如下一节所述。

根据定义, Ajax 涉及 JavaScript 和 XML。但是, Ajax 编程也需要使用其他技术, 如表 41-5 所述。

表 41-5

技 术	说 明
HTML/XHTML	HTML(Hypertext Markup Language, 超文本标记语言)是显示和布局语言, 由 Web 浏览器用于在图形化用户界面上呈现信息。前面学习了 HTML 如何实现这个功能, ASP.NET 如何生成 HTML 代码。可扩展的 HTML(XHTML)是使用 XML 结构的一个较严谨的 HTML 版本
CSS	CSS(层叠样式表)是 HTML 元素根据一个样式表中定义的规则设置样式的方式。它允许将样式同时应用于多个 HTML 元素, 能在不修改 HTML 的情况下切换风格从而改变 Web 页面的外观。因为 CSS 包含布局和样式信息, 所以也可以使用 CSS 在页面上定位 HTML 元素
DOM	DOM(文档对象模型)是在层次结构中表示和操作(X)HTML 代码的一种方式。例如, 它允许访问 Web 页面中的“表 x 中第 3 行的第 2 列”, 且无需使用原始的文本处理方式定位这个元素
JavaScript	JavaScript 是一种客户端脚本编辑技术, 它允许在 Web 浏览器中执行代码。JavaScript 的语法规类似于其他基于 C 的语言, 包括 C#, 并提供变量、函数、分支代码、循环语句和其他熟悉的编程元素。但是与 C#不同, JavaScript 不是强类型化的, JavaScript 代码的调试比较困难。对于 Ajax 编程, JavaScript 是一种关键技术, 因为它允许利用 DOM 操作, 动态修改 Web 页面

(续表)

技 术	说 明
XML	XML 是标记数据的一种中性平台方式, 对 Ajax 非常关键, 它既是操作数据的方式, 也是客户端和服务端之间的通信语言
XmlHttpRequest	自 Internet Explorer 5 以来, 浏览器就把 XmlHttpRequest API 作为一种在客户端和服务端之间进行异步通信的方式。Microsoft 最初把它引入为一种技术, 在 Outlook Web Access 产品中, 以访问通过 Internet 存储在 Exchange 服务器中的电子邮件。后来它变成在 Web 应用程序中进行异步通信的标准方式, 是支持 Ajax 的 Web 应用程序的一个核心技术。这个 API 的 Microsoft 实现方式称为 XMLHTTP, 它利用所谓的 XMLHTTP 协议来通信

Ajax 还需要用服务器端代码处理部分页面的回发和完整页面的回发, 这包括服务器控件的事件处理程序和 Web 服务。图 41-13 显示了这些技术如何在 Ajax Web 浏览器模型中联合使用, 并与传统的 Web 浏览器模型进行比较。

“传统的” Web 浏览器模型



AjaxWeb 浏览器模型

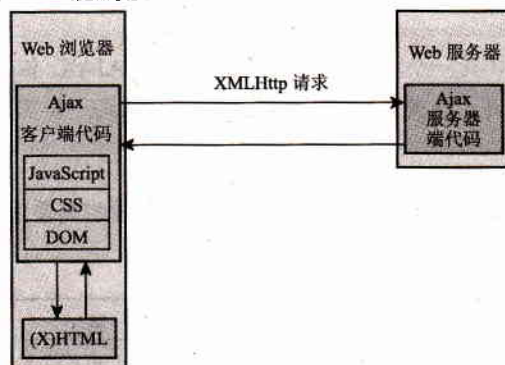


图 41-13

在 Ajax 推出之前, 表 41-1 中的列出前 4 种技术(HTML、CSS、DOM 和 JavaScript)用于创建所谓的动态 HTML(DHTML) Web 应用程序。这些应用程序比较著名出自两个原因: 它们提供的用户界面要好得多; 它们一般只能用于一种类型的 Web 浏览器。

自 DHTML 推出以来, 标准已有了改进, Web 浏览器遵循的标准的级别也提高了。但是, 它们仍有区别, Ajax 解决方案必须考虑这些区别。也就是说, 大多数开发人员实现 Ajax 解决方案还相当慢。只有开发出更抽象的 Ajax 架构(如 ASP.NET AJAX), 创建支持 Ajax 的 Web 站点才是企业级开发的一个可行选项。

41.9 ASP.NET AJAX

ASP.NET AJAX 是 Ajax 架构的 Microsoft 实现方式,专门面向 ASP.NET 开发人员。它是 ASP.NET 核心功能的一部分。Web 站点 <http://ajax.asp.net> 可以用于以前的 ASP.NET 版本,它包含相关的文档、论坛和示例代码,可以用于我们使用的任何 ASP.NET 版本。

ASP.NET AJAX 提供了如下功能:

- 服务器端架构允许 ASP.NET Web 页面响应部分页面的回发操作。
- ASP.NET 服务器控件便于实现 Ajax 功能。
- HTTP 处理程序允许 ASP.NET Web 服务在部分页面的回发操作中,使用 JSON(JavaScript Object Notation, JavaScript 对象标记)串行化功能与客户端代码通信。
- Web 服务支持客户端代码访问 ASP.NET 应用程序服务,包括身份验证和个性化服务。
- Web 站点模板可用于创建支持 ASP.NET AJAX 的 Web 应用程序。
- 客户端的 JavaScript 库对 JavaScript 语法提供了许多改进,还提供了许多代码,来简化 Ajax 功能的实现。

这些可行的服务器控件和服务器的架构统称为 ASP.NET 扩展。ASP.NET AJAX 的客户端部分称为 AJAX 库。

还可以从网站 <http://ajax.asp.net> 上下载其他软件包包括以下两个重要软件包:

- **ASP.NET AJAX Control Toolkit**——这个下载软件包包含由开发团体创建的其他服务器控件。这些控件是共享的源控件,可以查看和修改它们。
- **Microsoft AJAX Library 3.5**——这个下载软件包包含 JavaScript 客户端架构,它们由 ASP.NET AJAX 用于实现 Ajax 功能。如果开发的是 ASP.NET AJAX 应用程序,就不需要这个软件包。这个下载软件包适用于其他语言,如 PHP,使用与 ASP.NET AJAX 相同的基本代码来实现 Ajax 功能。它超出了本章的范围。



网站上还有 ASP.NET AJAX 新版本的预览下载软件包,本章不介绍它。

这两个下载软件包提供了功能丰富的架构,该架构可以用于在自己的 ASP.NET Web 应用程序中添加 Ajax 功能。下面几节将介绍 ASP.NET AJAX 的各个组成部分。

41.9.1 核心功能

ASP.NET AJAX 的核心功能分为两部分: AJAX 扩展和 AJAX 库。

1. AJAX 扩展

ASP.NET AJAX 功能包含在 GAC 中安装的两个程序集中:

- **System.Web.Extensions.dll**——这个程序集包含 ASP.NET AJAX 功能,包括 AJAX 扩展和 AJAX 库 JavaScript 文件,它们可以通过 ScriptManager 组件(稍后介绍)来获得。

- `System.Web.Extensions.Design.dll`——这个程序集包含用于 AJAX 扩展服务器控件的 ASP.NET Designer 组件，它由 ASP.NET Designer 在 Visual Studio 或 Visual Web Developer 中使用。
- ASP.NET AJAX 中的许多 AJAX 扩展组件都涉及支持部分页面的回发和用于 Web 服务的 JSON 串行化。这包括各种 HTTP 处理程序组件和对已有 ASP.NET 架构的扩展。所有这些功能都可以通过 Web 站点的 `web.config` 文件来配置。还有用于其他配置的和属性。但大多数配置都是透明的，用户很少需要改变默认设置。
- 与 AJAX 扩展的主要交互操作是使用服务器控件将 Ajax 功能添加到 Web 应用程序中。有几个服务器控件可以用各种方式增强用户的应用程序。表 41-6 列出了一些服务器端组件。本章后面将介绍它们。

表 41-6

控 件	说 明
ScriptManager	这个控件是 ASP.NET AJAX 功能的核心，使用部分页面回发功能的每个页面都需要它。它的主要作用是管理对 AJAX 库 JavaScript 文件的客户端引用，AJAX 库 JavaScript 文件位于 ASP.NET AJAX 程序集中。AJAX 库主要由 AJAX 扩展服务器控件使用，这些控件会生成自己的客户端代码。 这个控件还负责配置要在客户端代码中访问的 Web 服务。给 ScriptManager 控件提供 Web 服务信息，就可以生成客户端类和服务器端类，来透明地管理与 Web 服务的异步通信。 还可以使用 ScriptManager 控件维护对自己的 JavaScript 文件的引用
UpdatePanel	UpdatePanel 控件非常有用，也许是最常用的 ASP.NET 控件。这个控件与标准的 ASP.NET 占位符类似，可以包含任何其他控件。更重要的是，在部分页面的回发过程中，它还把页面的一部分标记为可以独立于页面的其他部分来更新的区域。 UpdatePanel 控件包含的、引发回发操作的任意控件(如按钮控件)，都不会引发整个页面的回发操作，它们只引发部分页面的回发，从而只更新 UpdatePanel 控件的内容。 在许多情况下，只需要这个控件就可以实现 Ajax 功能。例如，可以把一个 GridView 控件放在 UpdatePanel 控件中，该控件的分页、排序和其他回发功能都在部分页面的回发过程中发挥作用
UpdateProgress	在部分页面的回发过程中，这个控件可以为用户提供反馈。在更新 UpdatePanel 时，可以为要显示的 UpdateProgress 控件提供一个模板。例如，可以使用悬浮的<div>控件显示一条消息如“Updating...”，以便告诉用户应用程序正在忙。注意部分页面的回发不会干扰 Web 页面的其他区域，其他区域仍可以响应
Timer	ASP.NET AJAX 的 Timer 控件是使 UpdatePanel 控件定期更新的一种有效方式。可以把这个控件配置为定期触发回发操作。如果这个控件包含在 UpdatePanel 控件中，则每次触发 Timer 控件时，都会更新该 UpdatePanel 控件。Timer 控件也有关联的事件，以便用户可以执行定期的服务器端处理
AsyncPostBackTrigger	这个控件可以在未包含在 UpdatePanel 控件中的控件里触发 UpdatePanel 控件的更新操作。例如，可以在 Web 页面的其他地方放置一个下拉列表，来更新包含 GridView 控件的 UpdatePanel 控件

AJAX 扩展还包含 ExtenderControl 抽象基类, 来扩展已有的 ASP.NET 服务器控件。它由 ASP.NET AJAX Control Toolkit 中的各种类使用, 如后面所述。

2. AJAX 库

在支持 ASP.NET AJAX 的 Web 应用程序中, AJAX 库包含的 JavaScript 文件由客户端代码使用。在这些 JavaScript 文件中包含许多功能, 其中一些是增强 JavaScript 语言的通用代码, 一些则专用于 Ajax 功能。AJAX 库包含的功能层彼此互为基础, 如表 41-7 所示。

表 41-7

功 能 层	说 明
浏览器兼容性	AJAX 库的最底层代码根据客户机的 Web 浏览器来映射各种 JavaScript 功能, 这是必需的, 因为 JavaScript 在不同浏览器中的实现方式是有区别的。提供这个功能层, 其他层上的 JavaScript 代码就不必考虑浏览器的兼容性了, 我们也可以编写独立于浏览器、在所有客户机环境中工作的代码
核心服务	这一层包含对 JavaScript 语言的增强, 尤其是 OOP 功能。使用这一层的代码, 可以使用 JavaScript 脚本文件定义名称空间、类、派生类和接口。C#开发人员对此特别感兴趣, 因为它使 JavaScript 代码的编写非常类似于用 C#编写 .NET 代码, 且鼓励代码的重用
基类库	客户基类库(BCL)包含许多 JavaScript 类, 它们为 AJAX 库层次结构中下层的类提供了底层功能。这些类中的大多数都不能直接使用
网络	网络层上的类允许客户端代码异步地调用服务器端代码。这一层包含的基本架构可以调用 URL, 响应回调函数的结果。在大多数情况下, 这些功能都不能直接使用, 而应使用封装了该功能的类。这一层还包含用于 JSON 序列化和反序列化的类, 大多数网络类都在客户端的 Sys.Net 名称空间中
用户界面	这一层包含的类提取用户界面元素, 如 HTML 元素和 DOM 事件。可以使用这一层的方法和属性编写中性语言的 JavaScript 代码, 来操作客户端上的 Web 页面。用户界面类包含在 Sys.UI 名称空间中
控件	AJAX 库的最后一层包含最高级的代码, 它们提供 Ajax 行为和服务器控件功能。这包括可以使用动态生成代码, 例如, 用于从客户端的 JavaScript 代码中调用 Web 服务

AJAX 库可以用于扩展和定制支持 ASP.NET AJAX 的 Web 应用程序的操作, 但不一定要这么做, 注意这一点很重要。要想在应用程序中不使用任何附加的 JavaScript, 还有很长的路要走, 只有需要更高级的功能, 才需要这么做。如果要编写附加的客户端代码, 那么使用 AJAX 库提供的功能会比较容易完成任务。

41.9.2 ASP.NET AJAX Control Toolkit

AJAX Control Toolkit 是附加服务器控件的一个集合, 包括由 ASP.NET AJAX 社区编写的扩展控件。扩展控件允许在已有的 ASP.NET 服务器控件中添加功能, 一般把它关联到一个客户端行为。例如, AJAX Control Toolkit 中的一个扩展程序能在 TextBox 中放置“watermark”文本, 以扩展 TextBox

控件, 当用户还没有在文本框中添加任何内容时, 就会显示该文本。这个扩展控件在服务器控件 `TextBoxWatermark` 中实现。

使用 `AJAX Control Toolkit` 可以给站点添加许多功能, 它们超出了核心下载包的范围。这些控件可以使浏览操作更有趣, 也许能为增强 `Web` 应用程序提供许多新思路。但是, 因为 `AJAX Control Toolkit` 独立于核心下载包, 所以这些控件并没有获得与核心下载包中的控件相同级别的支持。

41.10 ASP.NET AJAX 网站示例

既然前面介绍了 `ASP.NET AJAX` 的组件部分, 下面就开始探讨如何使用它们增强网站。本节将讨论使用 `ASP.NET AJAX` 的 `Web` 应用程序如何工作, 如何使用 `ASP.NET AJAX` 中的各种功能。首先仔细研究一个简单的应用程序, 然后在后续的章节中添加其他功能。

`ASP.NET` 网站模板包含了 `ASP.NET AJAX` 的所有核心功能。也可以使用 `AJAX Control Toolkit Web Site` 模板(在安装它之后), 以包含 `AJAX Control Toolkit` 中的控件。本示例要在 `C:\ProCSharp\Chapter41` 目录中创建一个使用空白网站模板的新网站 `PCSAjaxWebSite`。

添加一个 `Web` 窗体 `Default.aspx`, 并修改其代码, 如下所示:



可从
wrox.com
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Pro C# ASP.NET AJAX Sample</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <div>
            <h1>Pro C# ASP.NET AJAX Sample</h1>
            This sample obtains a list of primes up to a maximum value.
            <br />
            Maximum:
            <asp:TextBox runat="server" id="MaxValue" Text="2500" />
            <br />
            Result:
            <asp:UpdatePanel runat="server" ID="ResultPanel">
                <ContentTemplate>
                    <asp:Button runat="server" ID="GoButton" Text="Calculate" />
                    <br />
                    <asp:Label runat="server" ID="ResultLabel" />
                    <br />
                    <small>
                        Panel render time: <% =DateTime.Now.ToLongTimeString() %>
                    </small>
                </ContentTemplate>
            </asp:UpdatePanel>
        </div>
    </form>
</body>
</html>
```

```

</asp:UpdatePanel>
<asp:UpdateProgress runat="server" ID="UpdateProgress1">
  <ProgressTemplate>
    <div style="position: absolute; left: 100px; top: 200px;
      padding: 40px 60px 40px 60px; background-color: lightyellow;
      border: black 1px solid; font-weight: bold; font-size: larger;
      filter: alpha(opacity=80);">Updating.</div>
    </ProgressTemplate>
  </asp:UpdateProgress>
  <small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
</div>
</form>
</body>
</html>

```

代码段 PCSAjaxWebSite1/Default.aspx

切换到设计视图(注意 ASP.NET AJAX 控件(如 UpdatePanel 和 UpdateProgress)有可视化的设计组件, 双击 Calculate 按钮, 添加一个事件处理程序。修改代码, 如下所示:



可从
wrox.com
下载源代码

```

protected void GoButton_Click(object sender, EventArgs e)
{
    int maxValue = 0;
    System.Text.StringBuilder resultText = new System.Text.StringBuilder();
    if (int.TryParse(MaxValue.Text, out maxValue))
    {
        for (int trial = 2; trial <= maxValue; trial++)
        {
            bool isPrime = true;
            for (int divisor = 2; divisor <= Math.Sqrt(trial); divisor++)
            {
                if (trial % divisor == 0)
                {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
            {
                resultText.AppendFormat("{0} ", trial);
            }
        }
    }
    else
    {
        resultText.Append("Unable to parse maximum value.");
    }
    ResultLabel.Text = resultText.ToString();
}

```

代码段 PCSAjaxWebSite1/Default.aspx.cs

保存修改的内容, 按 F5 键, 运行项目。如果有提示, 就在 web.config 文件中启动调试功能。在显示如图 41-14 所示的 Web 页面时, 注意显示的两个呈显时间相同。

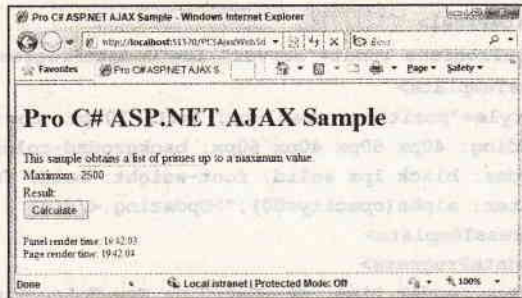


图 41-14

单击 **Calculate** 按钮，显示小于等于 2500 的质数。除非在较慢的计算机上运行，否则应立即得到结果。注意呈现的时间现在已经不同了，只有 **UpdatePanel** 控件中显示的时间改变了。

最后，在最大值中添加一些 0，引入一个处理延迟(在较快的 PC 上添加 3 个 0 就足够了)，再次单击 **Calculate** 按钮。这次在显示结果之前，注意 **UpdateProgress** 控件显示一条部分透明的反馈消息，如图 41-15 所示。

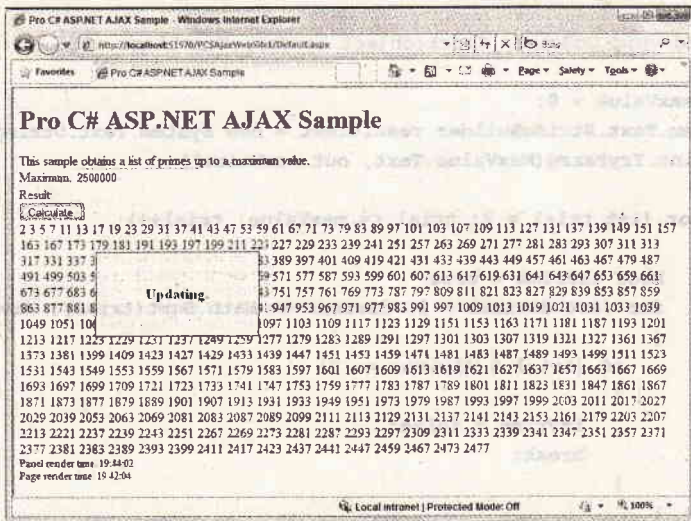


图 41-15

更新应用程序时，页面仍可以响应。例如，可以滚动页面。



注意，更新完成时，把浏览器的滚动位置设置为单击 **Calculate** 按钮之前的地方。在大多数情况下，部分页面的更新会很快执行完，这非常有利于可用性。

关闭浏览器，返回 Visual Studio。

41.11 支持 ASP.NET AJAX 的网站配置

ASP.NET AJAX 需要的大多数配置都是默认提供的，如果要修改这些默认值，那么实际上只需

把它们添加到 web.config 文件中。例如，可以添加一个<system.web.extensions>段，来提供额外的配置。可以用这个配置段添加的大多数配置都与 Web 服务相关，并包含在<webServices>元素中，该元素又放在<scripting>元素中。首先，可以添加一段，通过 Web 服务访问 ASP.NET 身份验证访问(根据个人爱好这里也可以选择强制 SSL)：

```
<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true" requireSSL="true"/>
    </webServices>
  </scripting>
</system.web.extensions>
```

接下来，通过配置文件 Web 服务，启用和配置对 ASP.NET 个性化功能的访问。

```
<profileService enabled="true"
  readAccessProperties="propertyname1,propertyname2"
  writeAccessProperties="propertyname1,propertyname2" />
```

最后一个与 Web 服务相关的设置是通过角色 Web 服务启用和配置对 ASP.NET 角色功能的访问。

```
<roleService enabled="true"/>
</webServices>
```

最后，<system.web.extensions>段包含一个元素，该元素允许配置异步通信的压缩和缓存：

```
<scriptResourceHandler enableCompression="true" enableCaching="true" />
</scripting>
</system.web.extensions>
```

AJAX Control Toolkit 的其他配置

要使用 AJAX Control Toolkit 中的控件，可以在 web.config 文件中添加如下配置：

```
<controls>
  ...
  <add namespace="AjaxControlToolkit" assembly="AjaxControlToolkit"
    tagPrefix="ajaxToolkit"/>
</controls>
```

这将工具包中的控件映射到 ajaxToolkit 标记前缀上。这些控件包含在 AjaxControlToolkit.dll 程序集中，该程序集应在 Web 应用程序的/bin 目录下。

还可以使用<%@ Register %>指令单独地在 Web 页面上注册控件。

```
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
  TagPrefix="ajaxToolkit" %>
```

41.12 添加 ASP.NET AJAX 功能

在网站上添加 Ajax 功能的第一步是在 Web 页面上添加一个 ScriptManager 控件，之后，添加服务器控件(如 UpdatePanel 控件)，以启用部分页面的呈现功能，再添加 AJAX Control Toolkit 中的动态控件，给应用程序增加可用性和功能。还可以添加客户端代码，使用 AJAX 库有助于进一步定制和增强应用程序的功能。

本节介绍可以使用服务器控件添加的功能。本章后面将讨论客户端技术。

41.12.1 ScriptManager 控件

如本章前面所述, ScriptManager 控件必须包含在使用部分页面回发和其他几个 ASP.NET AJAX 功能的所有页面上。



为了确保在 Web 应用程序中的所有页面都包含 ScriptManager 控件, 必须将这个控件添加到应用程序使用的母版页中。

除了启用 ASP.NET AJAX 功能之外, 还可以使用属性配置这个控件。在这些属性中, 最简单的是 EnablePartialRendering 属性, 其默认值是 true。如果把这个属性设置为 false, 就禁用了所有异步回发处理功能, 例如, UpdatePanel 控件提供的页面回发功能。如果要给经理做一个演示, 比较支持 AJAX 的网站和传统的网站, 就可以这么做。

使用 ScriptManager 控件有几个原因, 例如, 下面几种常见情形:

- 确定是否把服务器端代码作为部分页面回发的结果调用
- 添加对其他客户端 JavaScript 文件的引用
- 引用 Web 服务
- 给客户返回错误消息

下面几节介绍这些配置选项。

1. 检测部分页面的回发

ScriptManager 控件包含一个布尔属性 IsInAsyncPostBack。可以在服务器端代码中使用这个属性, 检测部分页面是否正在回发。注意页面的 ScriptManager 控件实际上可能在母版页上。除了通过母版页访问这个控件之外, 还可以使用静态方法 GetCurrent(), 获得对当前 ScriptManager 实例的引用。例如:

```
ScriptManager scriptManager = ScriptManager.GetCurrent(this);  
if (scriptManager != null && scriptManager.IsInAsyncPostBack)  
{  
    // Code to execute for partial - page postbacks.  
}
```

必须将对 Page 控件的引用传递给 GetCurrent() 方法。例如, 如果在 ASP.NET Web 页面的 Page_Load() 事件处理程序中使用这个方法, 就可以将 this 用作 Page 引用。另外, 注意检查 null 引用, 以避免异常。

2. 客户端 JavaScript 引用

除了在 HTML 页面的标题或页面的 <script> 元素中添加代码之外, 还可以使用 ScriptManager 类的 Scripts 属性。这可以使脚本引用集中在一起, 更便于维护它们。为此, 可以给 <UpdatePanel> 控件元素添加一个 <Scripts> 子元素, 再给 <Scripts> 添加 <asp:ScriptReference> 子控件元素。使用 ScriptReference 控件的 Path 属性引用自定义脚本。

下面的例子说明了如何在 Web 应用程序的根文件夹下, 添加对一个自定义脚本文件 MyScript.js

的引用:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Scripts>
    <asp:ScriptReference Path="~/MyScript.js" />
  </Scripts>
</asp:ScriptManager>
```

3. Web 服务引用

为了从客户端 JavaScript 代码中访问 Web 服务, ASP.NET AJAX 必须生成一个代理类。要控制这个行为, 可以使用 ScriptManager 类的 Services 属性。与 Scripts 属性一样, 也可以以声明方式指定这个属性, 但这次要使用<Services>元素。给这个元素添加<asp:ServiceReference>控件。对于 Services 属性中的每个 ScriptReference 对象, 都需要使用 Path 属性指定 Web 服务的路径。

ServiceReference 类也有一个 InlineScript 属性, 它默认为 false。当这个属性是 false 时, 客户端代码向服务器发出请求, 会得到一个代理类, 来调用 Web 服务。为了增强性能(尤其是在一个页面上使用大量 Web 服务的情况), 可以将 InlineScript 属性设置为 true, 这会在页面的客户端脚本中定义代理类。

ASP.NET Web 服务的文件扩展名是.asmx。如果不想详细阅读本章, 但希望在 Web 应用程序的根文件夹下添加对 Web 服务 MyService.asmx 的引用, 应使用下面的代码:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Services>
    <asp:ServiceReference Path="~/MyService.asmx" />
  </Services>
</asp:ScriptManager>
```

采用这种方式只能添加对本地 Web 服务的引用(即 Web 服务和主调代码在同一个 Web 应用程序中)。可以通过本地 Web 方法间接调用远程 Web 服务。

本章后面将讨论如何从客户端 JavaScript 代码中异步调用 Web 方法, 以这种方式使用代理类生成这些方法。

4. 客户端错误消息

如果在部分页面的回发过程中抛出了异常, 默认操作就是将异常包含的错误消息放在客户端 JavaScript 警报消息框中。处理 ScriptManager 实例的 AsyncPostBackError 事件, 可以定制要显示的消息。在这个事件处理程序中, 可以使用 AsyncPostBackEventArgs.Exception 属性访问抛出的异常, 使用 ScriptManager.AsyncPostBackErrorMessage 属性设置显示给客户端的消息。这么做可以对用户隐藏异常细节。

如果要重写默认行为, 以另一种方式显示消息, 就必须使用 JavaScript 处理客户端对象 PageRequestManager 的 endRequest 事件, 详见本章后面的内容。

41.12.2 使用 UpdatePanel 控件

UpdatePanel 控件是编写支持 ASP.NET AJAX 的 Web 应用程序时最常用的控件。如本章前面的简单例子所述, 这个控件可以封装 Web 页面的一部分, 从而使它能够参与部分页面的回发操作。为

此，要在页面中添加一个 UpdatePanel 控件，用它需要包含的控件填充其子元素 <ContentTemplate>。

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
    <ContentTemplate>
        ...
    </ContentTemplate>
</asp:UpdatePanel>
```

根据 UpdatePanel 控件的 RenderMode 属性值，<ContentTemplate>模板的内容呈现在<div>或元素中。这个属性的默认值是 Block，即显示在<div>元素中。要使用元素，应将 RenderMode 属性设置为 Inline。

1. 一个 Web 页面上的多个 UpdatePanel 控件

可以在一个页面上包含任意多个 UpdatePanel 控件。如果回发操作由包含在页面上的任意 UpdatePanel 控件的<ContentTemplate>模板中的控件引发，就进行部分页面的回发，而不是整个页面的回发。这会使所有 UpdatePanel 控件根据其 UpdateMode 属性值进行更新。这个属性的默认值是 Always，它表示 UpdatePanel 控件为页面上的部分页面回发操作而更新，即使这个操作在另一个 UpdatePanel 控件中引发，也是如此。如果把这个属性设置为 Conditional，UpdatePanel 控件就仅在它包含的控件引发部分页面回发操作时更新，或者在激活了已定义的触发器时更新。触发器稍后介绍。

如果把 UpdateMode 属性设置为 Conditional，那么还可以将 ChildrenAsTriggers 属性设置为 false，以禁止 UpdatePanel 控件包含的控件触发 UpdatePanel 控件的更新操作。但要注意，在这种情况下，这些控件仍会触发一个部分页面更新操作，它会使页面上的其他 UpdatePanel 控件进行更新。例如，这会使 UpdateMode 属性值为 Always 的 UpdatePanel 控件进行更新，如下面的代码所示：

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
    ChildrenAsTriggers="false">
    <ContentTemplate>
        <asp:Button runat="Server" ID="Button1" Text="Click Me" />
        <small> Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %> </small>
    </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
    <ContentTemplate>
        <small> Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %> </small>
    </ContentTemplate>
</asp:UpdatePanel>
<small> Page render time: <% =DateTime.Now.ToLongTimeString() %> </small>
```

在这段代码中，把 UpdatePanel2 控件的 UpdateMode 属性值设置为默认的 Always。单击对应按钮时，会引发一个部分页面回发操作，但只更新 UpdatePanel2 控件。注意，可看到只更新了“Panel2 render time”标签。

2. 服务器端的 UpdatePanel 更新

有时页面上有多个 UpdatePanel 控件，可能不打算更新其中一个控件，除非满足某些条件。在这种情况下，应将 UpdatePanel 控件的 UpdateMode 属性设置为 Conditional，如上一节所述，再把 ChildrenAsTriggers 属性设置为 false。接着，对于页面上引发部分页面回发操作的其中一个控件，在

服务器端的事件处理程序代码中, (有条件地)调用 UpdatePanel 控件的 Update()方法, 例如:

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (TestSomeCondition())
    {
        UpdatePanel1.Update();
    }
}
```

3. UpdatePanel 的触发器

给 Web 页面上其他地方的控件的 Triggers 属性添加触发器, 就可以通过该控件更新 UpdatePanel 控件。触发器是 Web 页面上其他地方的控件的事件与 UpdatePanel 控件之间的关联。因为所有控件都有默认事件(如 Button 控件的默认事件是 Click), 所以可以不指定事件名。有两种触发器可以添加, 它们用两个类表示:

- AsyncPostBackTrigger —— 在指定控件的指定事件发生时, 这个类会更新 UpdatePanel 控件。
- PostBackTrigger —— 在指定控件的指定事件发生时, 这个类会更新整个页面。

一般使用 AsyncPostBackTrigger 类, 但如果希望 UpdatePanel 中的一个控件引发整个页面的回发操作, 就可以使用 PostBackTrigger 类。

这两个触发器类都有两个属性 ControlID 和 EventName, ControlID 指定了通过其标识符激活触发器的控件, EventName 指定了控件中链接到触发器上的事件的名称。

为了扩展前面的例子, 考虑下面的代码:

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
    ChildrenAsTriggers="false">
    <Triggers>
        <asp:AsyncPostBackTrigger ControlID="Button2" />
    </Triggers>
    <ContentTemplate>
        <asp:Button runat="Server" ID="Button1" Text="Click Me" />
        <small> Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %> </small>
    </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
    <ContentTemplate>
        <asp:Button runat="Server" ID="Button2" Text="Click Me" />
        <small> Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %> </small>
    </ContentTemplate>
</asp:UpdatePanel>
<small> Page render time: <% =DateTime.Now.ToLongTimeString() %> </small>
```

新的 Button 控件 Button2 指定为 UpdatePanel1 中的一个触发器。单击这个按钮时, 会更新 UpdatePanel1 和 UpdatePanel2。更新 UpdatePanel1 是因为激活了触发器, 更新 UpdatePanel2 是因为它使用了 UpdateMode 的默认值 Always。

41.12.3 使用 UpdateProgress

如前面的例子所示, UpdateProgress 控件允许在部分页面的回发过程中给用户显示进度消息。

使用 `ProgressTemplate` 属性可提供显示进度的 `ItemTemplate`, 为此, 一般使用控件的 `<ProgressTemplate>` 子元素。

使用 `AssociatedUpdatePanelID` 属性将 `UpdateProgress` 控件与指定的 `UpdatePanel` 控件关联起来, 就可以在页面上放置多个 `UpdateProgress` 控件。如果没有设置该属性(默认), 无论哪个 `UpdatePanel` 控件引发了部分页面回发操作, 都显示 `UpdateProgress` 模板。

在执行部分页面回发操作时, 显示 `UpdateProgress` 模板之前有一个延迟。这个延迟可以通过 `DisplayAfter` 属性来配置, `DisplayAfter` 是一个 `int` 属性, 它指定延迟时间(单位是毫秒), 默认为 500 毫秒。

最后, 可以使用布尔属性 `DynamicLayout` 指定在显示模板之前, 是否为模板分配空间。因为这个属性的默认值是 `true`, 此时页面上的空间是动态分配的, 所以为了显示内联的进度模板, 需要删除其他控件。如果把这个属性设置为 `false`, 就在显示模板之前, 为模板分配空间, 这样页面上其他控件的布局不会改变。可以根据显示进度时要达到的效果设置这个属性。对于使用绝对坐标定位的进度模板, 如前面的例子所示, 应将这个属性设置为默认值。

41.12.4 使用扩展控件

ASP.NET AJAX 的核心软件包包含一个 `ExtenderControl` 类, 它的作用是允许扩展其他 ASP.NET 服务器控件(即增加功能)。它广泛应用于 AJAX Control Toolkit, 效果不错。可以使用 ASP.NET AJAX Server Control Extender 项目模板创建自己的扩展控件。 `ExtenderControl` 控件的工作方式都类似: 把它们放在页面上, 与目标控件关联起来, 添加进一步的配置。接着扩展程序就会发出客户端代码, 以添加功能。

为了在一个简单的例子中了解扩展控件, 在 `C:\ProCSharp\Chapter41` 目录下创建一个新的空网站 `PCSExtenderDemo`, 把 AJAX Control Toolkit 程序集添加到网站的 `bin` 目录下, 然后给一个新 Web 窗体 `Default.aspx` 添加如下代码:



可从
wrox.com
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
    TagPrefix="ajaxToolkit" %>

<!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>Color Selector</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <div>
            <asp:UpdatePanel runat="server" ID="updatePanel1">
                <ContentTemplate>
                    <span style="display: inline-block; padding: 2px;">
                        My favorite color is:
                    </span>
                    <asp:Label runat="server" ID="favoriteColorLabel" Text="green"
                        style="color: #00dd00; display: inline-block; padding: 2px;
```

```

        width: 70px; font-weight: bold;" />
<ajaxToolkit:DropDownExtender runat="server" ID="dropDownExtender1"
    TargetControlID="favoriteColorLabel"
    DropDownControlID="colDropDown" />
<asp:Panel ID="colDropDown" runat="server"
    Style="display: none; visibility: hidden; width: 60px;
        padding: 8px; border: double 4px black;
        background-color: #ffffdd; font-weight: bold;">
    <asp:LinkButton runat="server" ID="OptionRed" Text="red"
        OnClick="OnSelect" style="color: #ff0000;" /><br />
    <asp:LinkButton runat="server" ID="OptionOrange" Text="orange"
        OnClick="OnSelect" style="color: #dd7700;" /><br />
    <asp:LinkButton runat="server" ID="OptionYellow" Text="yellow"
        OnClick="OnSelect" style="color: #ddd000;" /><br />
    <asp:LinkButton runat="server" ID="OptionGreen" Text="green"
        OnClick="OnSelect" style="color: #00dd00;" /><br />
    <asp:LinkButton runat="server" ID="OptionBlue" Text="blue"
        OnClick="OnSelect" style="color: #0000dd;" /><br />
    <asp:LinkButton runat="server" ID="OptionPurple" Text="purple"
        OnClick="OnSelect" style="color: #dd00ff;" />
</asp:Panel>
</ContentTemplate>
</asp:UpdatePanel>
</div>
</form>
</body>
</html>

```

代码段 PCSExtenderDemo/Default.aspx

还需要在这个文件的代码隐藏中添加如下事件处理程序:



可从
wrox.com
下载源代码

```

protected void OnSelect(object sender, EventArgs e)
{
    favoriteColorLabel.Text = ((LinkButton)sender).Text;
    favoriteColorLabel.Style["color"] = ((LinkButton)sender).Style["color"];
}

```

代码段 PCSExtenderDemo/Default.aspx.cs

在浏览器中, 刚开始并没有显示很多内容, 扩展程序似乎没有什么作用, 如图 41-16 所示。

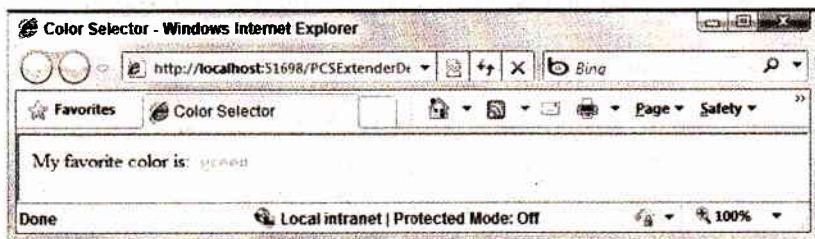


图 41-16

但是, 把鼠标悬停在文本“green”上时, 会动态地显示一个下拉框。如果单击这个下拉框, 就会显示一个列表, 如图 41-17 所示。

单击下拉列表中的一个链接时, 文本会相应地改变(一个部分页面回发操作之后)。

对于这个简单的例子，要注意两个要点：

- 非常容易将扩展程序与目标控件关联起来。
- 下拉列表用自定义代码设置了样式，这表示可以在列表中放置任意内容。这个简单的扩展程序是给 Web 应用程序添加功能的有效方式，使用起来也很简单。

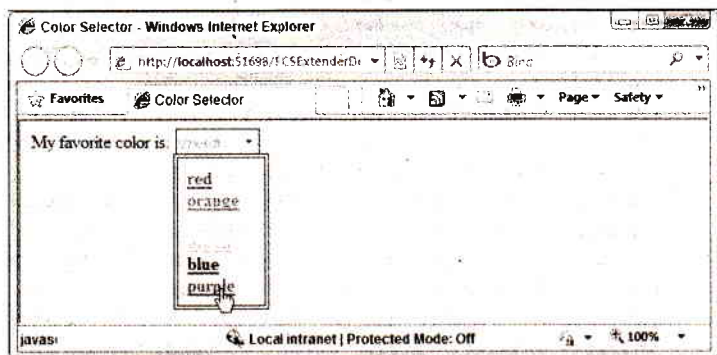


图 41-17

因为 AJAX Control Toolkit 中的扩展程序在不断地增加和更新，所以请定期访问 [http://ajax.asp.net/Ajax Control Toolkit/Samples/](http://ajax.asp.net/Ajax%20Control%20Toolkit/Samples/)。因为这个 Web 页面包含所有当前扩展程序的实时演示，所以可以看到它们的运行情况。

除了 AJAX Control Toolkit 提供的扩展控件之外，还可以创建自己的扩展控件。要创建有效的扩展程序，必须使用 AJAX 库。

41.13 使用 AJAX 库

AJAX 库有许多可进一步增强 Web 应用程序的功能。但是，为了增强 Web 应用程序，至少需要了解 JavaScript 的基本知识。尽管这不是一个全面的教程，但本节将介绍 AJAX 库提供的一些功能。

使用 AJAX 库的基本原则与在 Web 应用程序中添加任意类型的客户端脚本一样，仍使用核心语言 JavaScript，仍与 DOM 交互。但是，在许多方面，AJAX 库都使工作更容易完成。本节将学习这些内容，为用户进一步试验 AJAX 库和学习在线 AJAX 库文档打下基础。

本节介绍的技术都在 PCSLibraryDemo 项目中演示，该项目将贯穿本章的剩余内容。

41.13.1 给 Web 页面添加 JavaScript

首先需要了解的是如何给 Web 页面添加客户端 JavaScript，这里有 3 个选项：

- 使用 `<script>` 元素，在 ASP.NET Web 页面上在线添加 JavaScript
- 将 JavaScript 添加到单独的 JavaScript 文件中，其扩展名是 .js，再使用 `ScriptManager` 控件的 `<Scripts>` 子元素(首选)或从 `<script>` 元素中引用这些文件
- 从服务器端代码中生成 JavaScript，如代码隐藏或自定义扩展控件

这些技术都有自己的优点。对于原型代码，在线编码是无可替代的，因为它非常快，而且易于使用。将 HTML 元素的客户端事件处理程序和带客户端函数的服务器控件关联起来很容易，因为所有的代码都在同一个文件中。

使用单独的文件有利于代码重用，因为可以创建自己的类库，这类似于已有的 AJAX 库 JavaScript 文件。

从代码隐藏中生成代码较难实现，因为我们通常不能像使用 C# 代码那样在编写 JavaScript 代码时访问 IntelliSense。但是，可以动态地生成代码，以响应应用程序的状态，有时这是完成任务的唯一方式。

可以用 AJAX Control Toolkit 创建的扩展程序包含一个独立的 JavaScript 文件，它用于定义相应行为，从而解决从服务器提供客户端代码的一些问题。

本章将使用在线编码技术，因为它最简单，允许我们只关注 JavaScript 功能。

41.13.2 全局实用程序函数

AJAX 库提供的一个最常用的特性是封装了其他功能的全局函数集，包括：

- `$get()`——这个函数允许获得 DOM 元素的一个引用，方法是将其客户端 id 值提供为一个参数，可选的第二个参数指定了要搜索的父元素。
- `$create()`——这个函数允许同时创建对象并进行初始化。可以给这个函数提供 1~5 个参数。第一个参数是要实例化的类型，它一般是由 AJAX 库定义的一个类型。其他参数分别指定了属性的初始值、事件处理程序、其他组件的引用，以及对象要关联的 DOM 对象。
- `$addHandler()`——这个函数为给对象添加事件处理程序提供了一条捷径。

还有更多全局函数，但这些都是最常用的全局函数，尤其是 `$create()`，它可以大大减少创建和初始化对象所需的代码量。

41.13.3 使用 AJAX 库 JavaScript OOP 扩展

AJAX 库包含一个增强的架构，用于定义那些使用基于 OOP 的系统的类型，基于 OOP 的系统与 .NET Framework 技术紧密相关。可以创建名称空间，给名称空间添加类型，为类型添加构造函数、方法、属性和事件，甚至可以在类型定义上使用继承和接口。

本节将介绍如何实现这个功能的基本内容，但这里没有探讨事件和接口。这些构造超出了本章的范围。

1. 定义名称空间

要定义名称空间，应使用 `Type.registerNamespace()` 函数，例如：

```
Type.registerNamespace("ProCSharp");
```

注册名称空间后，就可以给它添加类型。

2. 定义类

定义类需要 3 步。首先，定义构造函数；然后，添加属性和方法；最后，注册该类。

要定义构造函数，需要使用名称空间和类名来定义函数，例如：

```
ProCSharp.Shape = function(color, scaleFactor) {
    this._color = color;
    this._scaleFactor = scaleFactor;
}
```

这个构造函数接受两个参数,使用它们设置本地字段(注意不一定要显式地定义这些字段,只需设置它们的值)。

要添加属性和方法,应给它们赋予类的 `prototype` 属性,如下所示:

```
ProCSharp.Shape.prototype = {
    get_Color : function() {
        return this._color;
    },
    set_Color : function(color) {
        this._color = color;
    },
    get_ScaleFactor : function() {
        return this._scaleFactor;
    },
    set_ScaleFactor : function(scaleFactor) {
        this._scaleFactor = scaleFactor;
    }
}
```

这段代码通过 `get` 和 `set` 存取器定义了两个属性。

要注册类,应调用其 `registerClass()` 函数:

```
ProCSharp.Shape.registerClass('ProCSharp.Shape');
```

3. 继承

派生类的方式与创建类相同,但略有区别。在构造函数中使用 `initializeBase()` 函数初始化基类,以数组的形式传递参数:

```
ProCSharp.Circle = function(color, scaleFactor, diameter) {
    ProCSharp.Circle.initializeBase(this, [color, scaleFactor]);
    this._diameter = diameter;
}
```

用前面的方式定义属性和方法:

```
ProCSharp.Circle.prototype = {
    get_Diameter : function() {
        return this._diameter;
    },
    set_Diameter : function(diameter) {
        this._diameter = diameter;
    },
    get_Area : function() {
        return Math.PI * Math.pow((this._diameter * this._scaleFactor) / 2, 2);
    },
    describe : function() {
        var description = "This is a " + this._color + " circle with an area of "
```

```

        + this.get_Area();
        alert(description);
    }
}

```

注册类时，作为第二个参数提供基类类型：

```
ProCSharp.Circle.registerClass('ProCSharp.Circle', ProCSharp.Shape);
```

将它们作为其他参数传递，可以实现接口，但简单起见，这里没有提供其细节。

4. 使用用户定义的类型

以这种方式定义了类之后，就可以通过简单的语法实例化和使用它们。例如：

```

var myCircle = new ProCSharp.Circle('red', 1.0, 4.4);
myCircle.describe();

```

这段代码会显示一个 JavaScript 警报框，如图 41-18 所示。

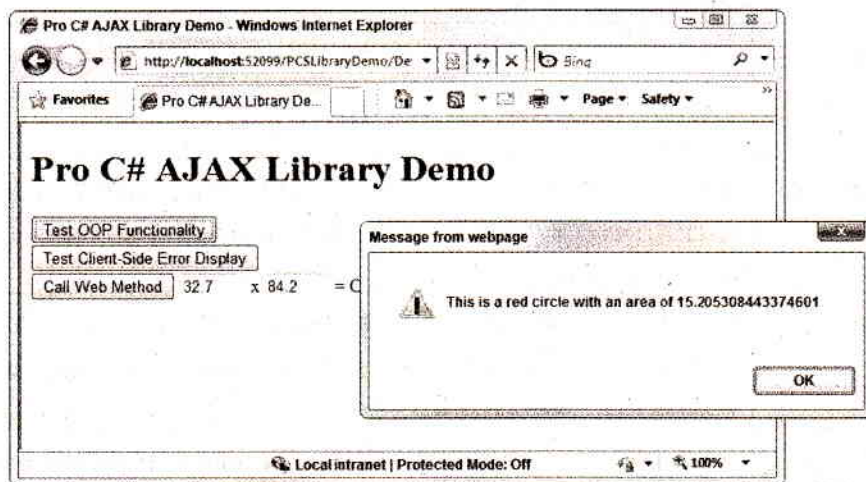


图 41-18

如果要测试一下，就可以运行 PCSLibraryDemo 项目，并单击 Text OOP Functionality 按钮。

41.13.4 PageRequestManager 对象和 Application 对象

在 AJAX 库中，最有用的类是 PageRequestManager 类和 Application 类。PageRequestManager 类在 Sys.WebForms 名称空间中，Application 类在 Sys 名称空间中。关于这两个类，重要的信息是它们提供的几个事件可以与 JavaScript 事件处理程序关联起来。这些事件在页面的生命周期(对于 Application 类)或部分页面回发过程(对于 PageRequestManager 类)中非常有趣的时刻发生，可以在这些关键时刻执行操作。

AJAX 库定义事件处理程序的方式类似于 .NET Framework 中事件处理程序的定义方式。每个事件处理程序都有类似的签名，带两个参数。第一个参数是对生成事件的对象的引用。第二个参数是 Sys.EventArgs 类的一个实例或派生自这个类的一个子类的实例。PageRequestManager 类和 Application 类提供的许多事件都包括专门的事件参数类，它可以用于确定事件的更多信息。表 41-8 按照事件的发

生顺序列出了这些事件，先加载页面，然后触发部分页面的回发操作，最后关闭页面。

表 41-8

事 件	说 明
Application .init	这个事件在页面的生命周期中第一个发生，它在加载了所有的 JavaScript 文件之后、创建应用程序中的任何对象之前发生
Application .load	这个事件在加载并初始化了应用程序中的对象后触发。这个事件经常关联一个事件处理程序，在第一次加载页面时执行操作。也可以为页面上的 <code>pageLoad()</code> 函数提供实现代码，该函数自动定义为这个事件的处理程序。使用 <code>Sys.ApplicationLoadEventArgs</code> 对象发送事件参数，该对象包含 <code>IsPartialLoad</code> 属性，用于确定是否已发生部分页面的回发操作。用 <code>get_IsPartialLoad()</code> 存取器访问这个属性
PageRequestManager .initializeRequest	这个事件在部分页面的回发操作之前、创建请求对象之前发生。可以使用 <code>Sys.WebForms.InitializedRequestEventArgs</code> 事件参数属性，访问触发回发操作的元素(<code>postBackElement</code>)和底层的请求对象(<code>request</code>)
PageRequestManager .beginRequest	这个事件在部分页面的回发操作之前、创建请求对象之后发生。可以使用 <code>Sys.WebForms.BeginRequestEventArgs</code> 事件参数属性，访问触发回发操作的元素(<code>postBackElement</code>)和底层的请求对象(<code>request</code>)
PageRequestManager .pageLoading	这个事件在部分页面的回发操作之后、后续的处理开始之前触发。这个处理过程可以包含要删除或更新的<div>元素，该元素使用 <code>panelsDeleting</code> 和 <code>panelsUpdating</code> 属性，通过 <code>Sys.WebForms.PageLoadingEventArgs</code> 对象来引用
PageRequestManager .pageLoaded	这个事件在部分页面的回发操作之后、处理 <code>UpdatePanel</code> 控件之后触发。这个处理过程可以包含要创建或更新的<div>元素，该元素使用 <code>panelsCreated</code> 和 <code>panelsUpdated</code> 属性，通过 <code>WebForms.PageLoadedEventArgs</code> 对象来引用
PageRequestManager .endRequest	这个事件完成部分页面的回发操作之后发生。传递给事件处理程序的 <code>System.WebForms.EndRequestEventArgs</code> 对象可以检测和处理服务器端的错误(使用 <code>error</code> 和 <code>errorHandled</code> 属性)，以及通过 <code>response</code> 访问响应对象
Application .unload	这个事件在释放应用程序中的对象之前触发，以便根据需要执行最后的操作或清理任务

使用静态的 `add_xxx()` 函数，可以给 `Application` 对象的事件添加事件处理程序，例如：

```
Sys.Application.add_load(LoadHandler);

function LoadHandler(sender, args)
{
    // Event handler code.
}
```

对于 `PageRequestManager` 对象的过程与此类似，但必须使用 `get_instance()` 函数获得当前对象的一个实例，例如：

```

Sys.WebForms.PageRequestManager.getInstance().add_beginRequest(
    BeginRequestHandler);

function BeginRequestHandler(sender, args)
{
    // Event handler code.
}

```

在 PCSLibraryDemo 应用程序中, 为 PageRequestManager.endRequest 事件添加一个事件处理程序。这个事件处理程序响应服务器端处理的错误, 在 id 为 errorDisplay 的元素中显示一条错误消息。要测试这个方法, 可以单击 Test Client-Side Error Display 按钮, 如图 41-19 所示。

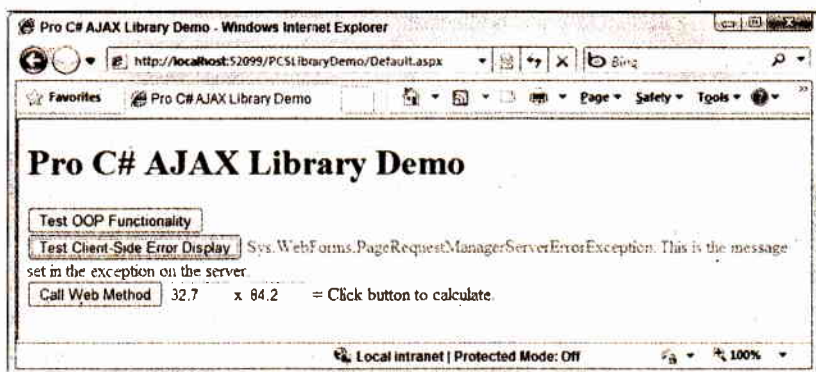


图 41-19

得到该结果的代码如下:

```

Sys.WebForms.PageRequestManager.getInstance().add_endRequest(
    EndRequestHandler);

function EndRequestHandler(sender, args)
{
    if (args.get_error() != undefined)
    {
        var errorMessage = args.get_error().message;
        args.set_errorHandled(true);
        $get('errorDisplay').innerHTML = errorMessage;
    }
}

```

注意, 把 EndRequestEventArgs 对象的 errorHandled 属性设置为 true, 这会禁止默认行为, 即, 使用 JavaScript 的 alert() 函数在对话框中显示错误消息。

在服务器上抛出一个异常, 就生成了错误, 如下所示:

```

protected void testErrorDisplay_Click(object sender, EventArgs e)
{
    throw new ApplicationException(
        "This is the message set in the exception on the server.");
}

```

还有许多情形可以使用事件处理技术, 来处理 PageRequestManager 和 Application 的事件。

41.13.5 JavaScript 的调试

过去, JavaScript 很难调试。但这在 Visual Studio 的最新版本中得到了解决。现在可以像 C# 代码那样在 JavaScript 代码中添加断点, 并单步执行代码。还可以在中断模式下查询对象状态, 改变属性值等。编写 JavaScript 代码时使用的 IntelliSense 功能也在 Visual Studio 的最新版本中得到了很大改进。

然而, 有时还希望添加调试和跟踪代码, 在代码执行过程中报告信息。例如, 使用 JavaScript 的 `alert()` 函数在对话框中显示信息。

有一些第三方工具可用于添加便于调试的客户端 UI, 包括:

- Fiddler —— 这个工具可以从 www.fiddlertool.com 上获得, 它可以记录计算机和 Web 应用程序之间的所有 HTTP 流量, 包括部分页面的回发。还有一些工具可以查看在处理 Web 页面的过程中发生的事件的详细信息。
- Nikhil 的 Web Development Helper —— 这个工具可以从 <http://projects.nikhilk.net/Projects/WebDevHelper.aspx> 上获得, 它也可以记录 HTTP 流量。另外, 这个工具包含许多专门用于 ASP.NET 和 ASP.NET AJAX 开发的实用程序, 例如, 可以查看视图状态, 执行即时的 JavaScript 代码。后者特别适合于测试在客户端上创建的对象。Web Development Helper 还在发生 JavaScript 错误时显示其他错误信息, 更便于跟踪 JavaScript 代码中的错误。

AJAX 库也提供了 `Sys.Debug` 类, 给应用程序添加额外的调试特性。该类最有用的一个功能是 `Sys.Debug.traceDump()` 函数, 它允许分析对象, 使用这个函数的一种方式是将一个 `id` 属性为 `TraceConsole` 的 `textarea` 控件放在 Web 页面上, 接着, `Debug` 的所有输出就会发送到这个控件上。例如, 可以使用 `traceDump()` 方法, 将 `Application` 对象的信息输出到控制台上:

```
Sys.Application.add_load(LoadHandler);

function LoadHandler(sender, args)
{
    Sys.Debug.traceDump(sender);
}
```

这会得到如下输出:

```
traceDump {Sys._Application}
  _updating: false
  _id: null
  _disposing: false
  _creatingComponents: false
  _disposableObjects {Array}
  _components {Object}
  _createdComponents {Array}
  _secondPassComponents {Array}
  _loadHandlerDelegate: null
  _events {Sys.EventHandlerList}
    _list {Object}
      load {Array}
        [0] {Function}
  _initialized: true
  _initializing: true
```

在这个输出中, 可以查看该对象的所有属性。这个属性特别适合于 ASP.NET AJAX 开发。

41.13.6 异步调用 Web 方法

ASP.NET AJAX 最强大的一个功能是可以从客户端脚本中调用 Web 方法, 这就允许访问数据、服务器端处理和各种其他功能。

因为第 43 章将介绍 Web 方法, 所以这里不详细介绍它, 仅讨论一些基本知识。简言之, Web 方法是可以从 Web 服务中提供并能通过 Internet 访问远程资源的方法。在 ASP.NET AJAX 中, 还可以将 Web 方法用作服务器端 Web 页面的代码隐藏中的静态方法。在 Web 方法中使用参数和返回的方式与其他方法类型中的用法相同。

在 ASP.NET AJAX 中, Web 方法是异步调用的。给 Web 方法传送参数, 并定义一个回调函数, 当 Web 方法调用完成时, 就会调用这个回调函数。该回调函数用于处理 Web 方法的响应。也可以提供另一个回调函数, 以处理调用失败的情况。

在 PCSLibraryDemo 应用程序中, 单击 Call Web Method 按钮时, 就调用一个 Web 方法, 如图 41-20 所示。

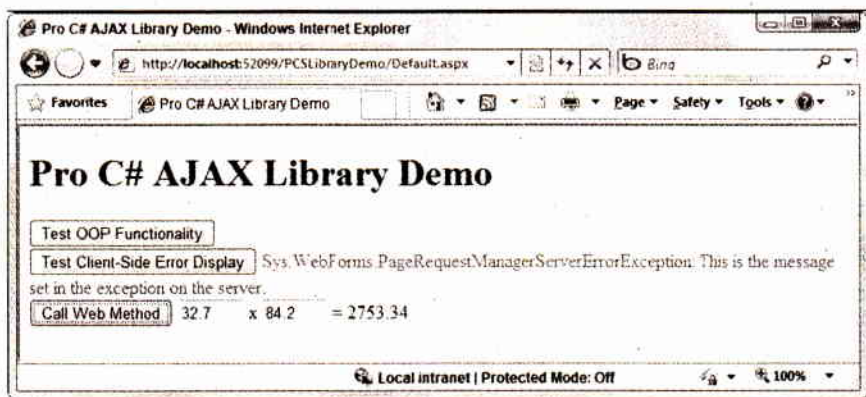


图 41-20

从客户端脚本中使用 Web 方法之前, 必须生成一个客户端代理类, 以进行通信。为此, 最简单的方式是在 ScriptManager 控件中, 引用包含 Web 方法的 Web 服务的 URL:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
  <Services>
    <asp:ServiceReference Path="~/SimpleService.asmx" />
  </Services>
</asp:ScriptManager>
```

ASP.NET Web 服务使用扩展名.asmx, 如上面的代码所示。为了使用客户端代理访问 Web 服务中的 Web 方法, 必须给 Web 服务应用 System.Web.Script.Services.ScriptService 属性。

对于 Web 页面的代码隐藏中的 Web 方法, 不需要这个属性, 或者 ScriptManager 中的这个引用, 但必须使用静态方法, 并给方法应用 System.Web.Services.WebMethod 属性。

生成了客户端存根后, 就可以通过其名称访问 Web 方法, 它定义为类中与 Web 服务同名的一个函数。在 PCSLibraryDemo 中, Web 服务 SimpleService.asmx 的一个 Web 方法是 Multiply(), 它对

两个 `double` 参数执行相乘操作。从客户端代码中调用这个方法时，要传递方法需要的两个参数(在本例中，从 HTML `<input>` 元素中获得)，并可以传递一个或两个回调函数引用。如果传递一个引用，这个回调函数就在调用成功返回时使用这个回调函数。如果传递了两个引用，第二个引用就在 Web 方法失败时用作回调函数。

在 `PCSLibraryDemo` 中，使用了一个回调函数，它提取 Web 方法调用的结果，并将它赋予 `id` 为 `webMethodResult` 的 `<span>` 元素：

```
function callWebMethod()
{
    SimpleService.Multiply(parseFloat($get('xParam').value),
        parseFloat($get('yParam').value), multiplyCallBack);
}

function multiplyCallBack(result)
{
    $get('webMethodResult').innerHTML = result;
}
```

这个方法非常简单，但演示了从客户端代码中异步调用 Web 服务的便利性。

41.13.7 ASP.NET 应用程序服务

ASP.NET AJAX 包含 3 个专用的 Web 服务，这 3 个 Web 服务用于访问 ASP.NET 应用程序服务。这些服务可以通过下面的客户端类来访问：

- `Sys.Services.AuthenticationService`——这个服务包含的方法可以登录或注销用户，或确定用户是否已登录。
- `Sys.Services.ProfileService`——这个服务可以获取和设置当前登录的用户的配置文件属性。配置文件属性在应用程序的 `web.config` 文件中配置。
- `Sys.Services.RoleService`——这个服务可以确定当前登录的用户的角色成员资格。

如果使用正确，这些类可以实现响应性非常好的用户界面，其中包含身份验证、配置文件和成员资格功能。

这些服务超出了本章的范围；但应知道它们，它们很值得研究。

41.14 小结

本章介绍了创建 ASP.NET 页面和 Web 站点的几种高级技术，并在示例 Web 站点 `PCSDemoSite` 中演示了这些技术。还介绍了如何使用 ASP.NET AJAX 增强 ASP.NET Web 站点。ASP.NET AJAX 包含的丰富功能使 Web 站点的响应更快、更动态，大大改进了用户体验。

首先探讨了如何使用 C# 创建可重用的 ASP.NET 服务器控件。其中讨论了如何从现有的 ASP.NET 页面中创建简单的用户控件，以及如何从头创建自定义控件。还研究了如何把上一章的会议室登记工具示例修改为一个用户控件。

接着介绍了母版页，如何为 Web 站点的页面提供模板，这是重用代码和简化开发的另一种方式。在本章可下载代码的 `PCSDemoSite` 中，有一个母版页包含 Web 导航服务器控件，允许用户在站点上浏览。该 `PCSDemoSite` 示例建立了主题的框架，该主题非常适合于把功能与设计分开，是一个强

大的可访问性技术。

我们还简要介绍了安全性，探讨了如何在 Web 站点上轻松地实现基于窗体的身份验证。

之后研究了 Web 部件，介绍了如何使用 Web 部件服务器控件把基本应用程序和这种技术提供的一些功能集成在一起。

本章最后一部分讨论了 ASP.NET AJAX。首先学习了 Ajax 的概念、如何使用 Ajax 的 Microsoft 实现方式 ASP.NET AJAX，包括创建支持 ASP.NET AJAX 的 Web 站点的服务器端技术，以及使用 AJAX 库实现的客户端技术。

希望本章能激发读者使用 ASP.NET 创建 Web 站点的兴趣，尤其是使用 ASP.NET AJAX 提供的新功能。Ajax 在 Web 上非常流行，ASP.NET AJAX 是将 Ajax 功能与 ASP.NET 应用程序集成起来的最佳方式。这个产品也得到了非常好的支持，同时基于社区的版本，如 AJAX Control Toolkit，提供了更酷的功能，这些功能可以在应用程序中免费使用。

尽管必须学习 JavaScript 语言，但这是值得的。使用 ASP.NET AJAX 比仅使用 ASP.NET 可以使 Web 应用程序更好、功能更强、更动态。在 Visual Studio 的最新版本中，有一些使 ASP.NET AJAX 更容易使用的工具。ASP.NET AJAX 的最新版本包含更多的功能，本章不可能全部涵盖，如客户端模板和实时数据绑定。应时刻关注开发人员博客和 <http://www.asp.net> 网站，研究这些高级功能。

下一章介绍构建 ASP.NET Web 站点和应用程序的最新技术——动态数据网站和 MVC 架构，以及一种支持技术——ASP.NET 路由。

第 42 章

ASP.NET 动态数据和 MVC

本章内容:

- 使用路由实现 URL
- 用动态数据构建网站
- 用 ASP.NET MVC 构建 Web 应用程序

前两章学习了如何使用 ASP.NET 创建网站,如何添加高级功能,以简化这些网站。本章介绍的框架可用于加速网站的创建,还包括建立最佳站点的最佳实践技术。

当前有两个框架随 ASP.NET 一起发布为 .NET 4 的一部分,经过 Microsoft 和 ASP.NET 开发团队多年的研究,它们都已成熟。这两个框架是:

- 动态数据
- MVC(Model-View-Controller, 模型-视图-控制器)

第一个框架是动态数据,它允许利用数据库中先前存在的数据构建网站。Visual Studio 提供的向导可非常快地生成动态数据网站,接着就可以定制网站的外观。在动态数据网站上包含高级模板功能,以便网站快速启动和运行。动态数据并不适用于所有类型的网站,但对于其中的许多类型,包括电子商务站点和数据操作站点,动态数据都提供了非常好的开发起点。

MVC 框架可从某个设想开始,创建所有种类的网站,这些网站的用户界面和业务逻辑更清晰地分开,并可以对业务逻辑进行独立的单元测试。这方面越来越重要,因为我们目前都已转而使用测试驱动开发(test-driven development, TDD)模型。传统上,此外 ASP.NET 网站很难测试,因为需要某种与 Web 页面直接交互的方式。把设计和功能分开简化了测试工作。

MVC 还有其他优点,如本章后面所述,而且它会使代码非常简洁、易于理解,非常适合于企业级的网站开发,而且可以简化单元测试。

动态数据和 MVC 都使用 .NET 3.5 引入的另一种方法:路由。这种方法可简化 URL,且包含额外的信息,是本章的第一个主题。在介绍了路由后,本章就依次讨论动态数据和 MVC,最后一节介绍如何合并这些技术。



动态数据和 MVC 都是大型主题,需要好几章的篇幅才能深入论述。本章仅概述这些架构的用法,以帮助读者获得开始使用它们的足够信息,如果读者想深入研究它们,随后就可以查找更详尽的资源。

42.1 路由

ASP.NET 路由技术允许为 Web 页面设置合理的、人类可读的 URL。基本上,这意味着用户在浏览器的地址栏中看到的 Web 页面的 URL 不必匹配该 Web 页面的物理地址。相反,Web 站点可使用 URL 包含的信息映射到我们定义的页面,还可以给这个页面传递参数。

初看起来,这似乎有悖常理。为什么不希望用户看到 Web 页面的地址?不提供这些信息肯定是使资源的定位更困难。实际上,如本节后面所述,映射 URL 使用户更容易定位资源,而且还有其他几个优点。

例如,考虑下面的 URL:

```
http://www.myeccommercesite.com/products/kites/reddragon
```

尽管对使用这个 URL 的 Web 站点毫不了解,也很容易猜出页面的内容。从 URI 中的单词可以看出,这是一个,命名为 Red Dragon 的风筝的产品页面。

这类 URL 的优点如下:

- 用户可以一眼看出页面是什么。
- 即使没有创建书签,用户也很容易记住 URL。
- 用户可以手工修改 URL,以进行导航(例如,用户可能用/products/balloons 替代上述 URL 的最后一部分)。
- 因为搜索引擎结果对这些描述性的 URL 进行了优化,所以页面的排名较靠前。

实现这种形式的 URL 基本上有两种方式。可以为每个可能的 URL 创建一个页面,也可以使用一个架构,把 URL 映射到可为多个 URL 显示数据的页面上。第一种方法值得一提,因为它不一定是不可可能的——可以创建从后端数据中生成的静态 HTML 站点,但其维护工作量很大,在大多数情况下,比较麻烦。现在,采用第二种方法的工具已成熟,且比较实用。

尽管“URL 重写”(在 ASP.NET 的以前版本中该功能的名称)需要深入理解 ASP.NET 体系结构且较难实现,但实际上,使用这类 URL 的功能在 ASP.NET 中已存在一段时间。有了 ASP.NET 路由技术后,事情就变得简单多了。

本节介绍如下内容:

- **查询字符串参数**——在专门介绍路由技术之前,需要先详细理解在 URL 中包含额外数据的情况。查询字符串参数为此提供了一种替代方式,且自从 Web 一出现就存在这种方式。虽然这种方式比路由技术简单,但它们并没有路由技术的所有优点。
- **定义路由**——ASP.NET 路由技术要求定义可以在网站上可用的路由,以便使用它们。
- **使用路由参数**——一旦在 URL 中找到某个路由,目标页面就可以使用路由参数。

本章的可下载代码包含一个简单的解决方案 PCRoutingDemo,本节将引用它来演示各种技术。

42.1.1 查询字符串参数

在 Web 上冲浪时,用户可能会注意到,许多 Web 页面的 URL 除了包含页面位置之外,还包含其他信息。例如,在电子商务网站上,一个 URL 如下所示:

```
http://www.myeccommercesite.com/products.aspx?id=4
```

在这个 URL 中, 目标页面是 `products.aspx`, 但在这个页面的标识符后面, 还有查询字符串形式的其他信息。“?” 字符表示查询字符串的开头, URL 的剩余内容由一个名称/值对组成, 并用一个等号(=)把名称(id)和值(4)分开。实际上, URL 可以包含几个查询字符串名称/值对, 并用&符号隔开。

用户用查询字符串导航到 ASP.NET 页面(也许是单击数据库中的数据所呈现的链接)时, 我们可以编写代码, 以呈现相应的页面。在这个例子中, 可能使用 id 查询字符串参数的值, 从数据库中检索有对应 ID 的产品数据。这意味着不必为数据库中的每个产品创建一个页面, 而可以创建单个页面, 它可以显示任意产品的详细信息。

实际上, ASP.NET 页面以 `NameValueCollection` 对象的形式接收查询字符串信息。这个对象在 `HttpRequest` 对象的 `QueryString` 属性中传递, 它可以在代码隐藏中通过继承的 `Request` 属性获得。因为查询字符串集合按名称和索引来索引, 所以要获得上述 id 查询字符串参数的值, 可以使用下面两行代码中的一行:

```
string idValue = Request.QueryString[0];
```

或

```
string idValue = Request.QueryString["id"];
```

注意, 因为所有查询字符串值都作为字符串传递, 所以要获得 id 的整数值, 需要解析该值。

查询字符串参数的用法参见本章的 `PCSRoutingDemo` 网站。如果在浏览器中查看该网站, 并单击 `Navigation with query string` 链接, 用户就会看到查询字符串参数, 如图 42-1 所示。

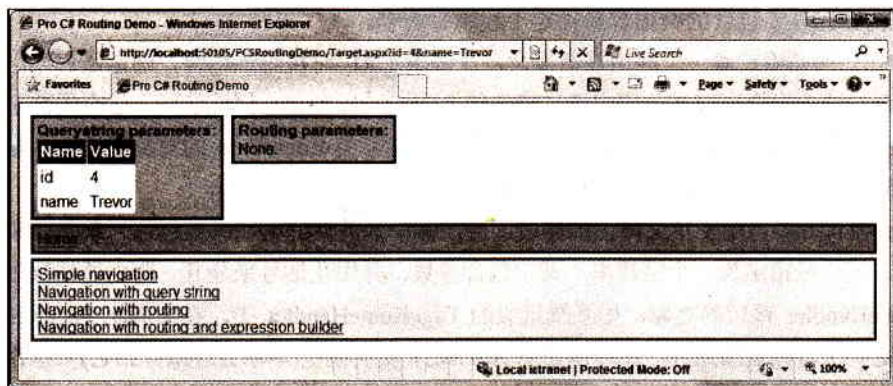


图 42-1

显示这些值的代码在母版页的代码隐藏中, 如下所示:



可从
wrox.com
下载源代码

```
if (Request.QueryString.Count == 0)
{
    this.NoQueryStringLabel.Text = "None.";
}
else
{
    GridView1.DataSource = from string key in Request.QueryString.Keys
                           select
                               new
                               {
                                   Name = key,
                                   Value = Request.QueryString[key]
                               }
    GridView1.DataBind();
}
```

```

    };
    GridView1.DataBind();
}

```

代码段 PCSRoutingDemo/MasterPage.master.cs

这里，把查询字符串参数集合转换为一个对象集合，其 Name 属性和 Value 属性可以通过数据绑定到一个 GridView 控件上，并显示出来。

这种通过查询字符串参数传递数据的方法非常好，且非常有用，但 ASP.NET 路由技术可以用更简洁的方式达到相同的目的，且具备所有优点。

42.1.2 定义路由

为了使用 ASP.NET 路由技术，需要配置在 Web 站点中可用的路由。之后，一般在事件处理程序 `HttpApplication.Application_Start` 中，用户就可以使用这些路由，Web 站点就可以使用传递过来的参数。

路由的定义包含如下内容：

- 路由的名称(可以省略)
- 路由的 URL 模式，包括必要的参数规范
- 路由的处理程序，它确定路由映射到哪个目标 URL 上
- 可选的默认值集合，用于未包含在 URL 中的路由参数
- 可选的参数约束集合，用于判断路由是否有效

果然，ASP.NET 包含的类很容易指定上述所有内容。

ASP.NET 网站维护着一个路由规范集合，在请求 URL 时检查该集合，确定是否存在一个匹配的路由规范。这个集合可以在代码中通过静态的 `RouteTable.Routes` 属性访问。这个属性是一个路由集合，该路由集合通过一个可选的名称和一个派生自抽象类 `RouteBase` 的对象来定义。除非创建自己的类来表示路由，否则就可以把 `Route` 类的实例添加到这个集合中。

`Route` 类有许多可用于定义路由的构造函数，其中最简单的构造函数包含一条路径和一个路由处理程序。该路径指定为一个字符串，其中包含参数，并用花括号来分开。路由处理程序是一个实现了 `IRouteHandler` 接口的对象，如系统定义的 `PageRouteHandler` 类。在实例化 `PageRouteHandler` 类时，指定一个目标页面路径，还可以使用 ASP.NET 安全模型(本章会详细介绍它)来检查物理路径是否获得授权。因此，添加路由最简单的方法如下：

```

RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new PageRouteHandler("target.aspx")));

```

这行代码把 `RouteName` 路由添加到路由定义中。这个路由匹配如下形式的 URL：

```
http://<domain>/path/{pathparameter}
```

把任何匹配的 URL 都映射到 `target.aspx` 文件上。

例如，如果用户请求如下 URL：

```
http://<domain>/path/oranges
```

网站就会作为一个匹配 RouteName 路由的 URL 检测它, 因此把用户重定向到 target.aspx 页面上。如果这个页面的代码请求 pathparameter 参数的值, 它就会接收到参数值 oranges。

在本章的 PCSRoutingDemo 网站中, 单击 Navigation with routing 链接, 就会看到一个路由在发挥作用, 得到的屏幕如图 42-2 所示。

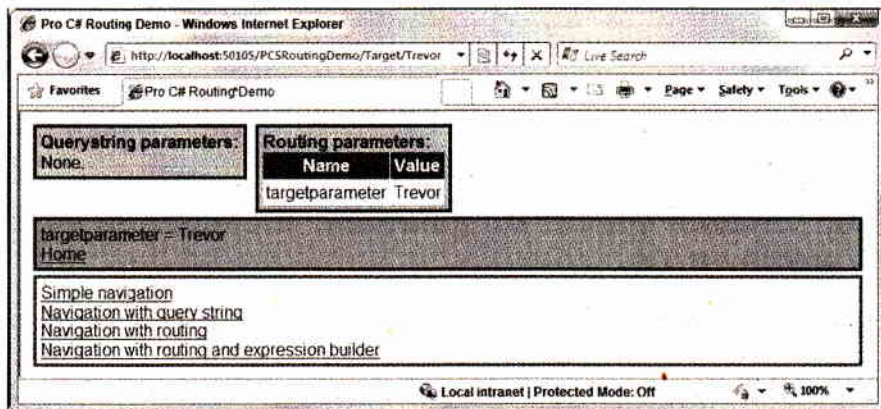


图 42-2

这个路由在 App_Code 的 Global.asax(用于 Global.cs 的代码隐藏文件)中定义; 如下所示:



可从
wrox.com
下载源代码

```
RouteTable.Routes.Add("TestRoute",
    new Route("Target/{targetparameter}",
        new PageRouteHandler("~/Target.aspx"));
```

代码段 PCSRoutingDemo/App_Code/Global.cs

稍后将说明如何把参数呈现在页面上。

1. 路由的授权

由于在路由过程中实际上有两个 URL 参与——一个由用户请求, 另一个用于生成页面的物理文件位置, 因此 ASP.NET 的授权可以在两个地方进行。默认行为是检查这两个 URL, 以确定用户是否有权访问页面。在 PageRouteHandler 类的构造函数中, 可以使用第二个参数重写这个操作, 如下所示:

```
RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new PageRouteHandler("~/target.aspx", false));
```

如果重写了这个操作, 就只检查被请求的 URL 是否获得授权, 而不检查物理 URL。这样, 就可以把物理 URL 放在用户不能直接访问的位置, 如果希望禁止直接导航到该 URL, 这就很有效。

2. 默认参数值

对于路由, 默认行为是必须指定所有参数, 才能匹配请求的 URL 和路由的路径。在上面的例子中, 这表示下面的 URL 无效:

http://<domain>/path

但可以给路由参数指定默认值,从而使这个 URL 有效,它使用 `pathparameter` 参数的默认值。

要指定参数的默认值,需要提供一个 `RouteValueDictionary` 集合,这是一个路由参数和默认值的名称/值对集合。可以把这个集合传递给 `Route` 类的构造函数,例如:

```
RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new RouteValueDictionary
        {
            { "pathparameter", "defaultValue" }
        },
        new PageRouteHandler("~/target.aspx"));
```

这段代码给 `pathparameter` 参数指定了默认值 `defaultvalue`,如果 `pathparameter` 参数在 URL 中没有匹配的值,就使用这个默认值。

3. 参数约束

路由参数默认接受来自用户的任意输入值,但有时需要限制允许接受的值。于是,如果用户传递了不允许接受的值,路径就不匹配。

参数约束也用 `RouteValueDictionary` 集合定义,该集合包含一个路由参数和约束的名称/值对集合。约束可以是正则表达式,或者实现了 `IRouteConstraints` 接口的对象。与默认情况一样,约束也可以在 `Route` 类的构造函数中指定,例如:

```
RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new RouteValueDictionary
        {
            { "pathparameter", "yes" }
        },
        new RouteValueDictionary
        {
            { "pathparameter", "yes|no" }
        },
        new PageRouteHandler("~/target.aspx"));
```

在这段代码中,对于 `pathparameter`,只有 `yes` 和 `no` 是有效值。如果指定了其他值,就不匹配路由。

4. 路由顺序和数据标记

在定义路由时,把它们添加到 `RouteTable.Routes` 集合中的顺序很重要。这是因为 ASP.NET 尝试按照路由在集合中出现的顺序,匹配 URL 和路由。如果在查看了所有路由后没有找到匹配的路由,就直接使用 URL。如果找到一个匹配的路由,就使用匹配的路由。如果有多个路由匹配 URL,就使用第一个匹配的路由。

路由顺序与约束一起使用会有很好的效果。例如,可以添加两个路由,第一个是上一节中受约束的路由,第二个路由如下所示:

```
RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new PageRouteHandler("~/alternatetarget.aspx"));
```

这表示,与前面一样,包含 `pathparameter` 值 `yes` 或 `no` 的 URL 会映射到 `target.aspx` 上,但如果传递了任何其他参数值,URL 就映射到 `alternatetarget.aspx` 上,而不是返回一个“404 not found”错误。

但是,如果这两个路由的添加顺序相反,则无论给 `pathparameter` 指定什么值,都会调用 `alternatetarget.aspx`,因为在检查受约束的路由之前,这个路由总是提供了一个匹配的值。

如果两个路由映射到同一个 URL 上,就需要确定匹配哪个路由。为此,可以给页面传送额外的数据,作为数据标记。数据标记通过另一个 `RouteValueDictionary` 集合提供,其中包含一个路由参数和数据标记的名称/值对集合。例如:

```
RouteTable.Routes.Add("RouteName",
    new Route("path/{pathparameter}",
        new RouteValueDictionary
        {
            { "pathparameter", "yes" }
        },
        new RouteValueDictionary
        {
            { "pathparameter", "yes|no" }
        },
        new RouteValueDictionary
        {
            { "customdatatoken", "yesnomatch" }
        },
        new PageRouteHandler("~/target.aspx"));
```

如果这个路由是匹配的,就给目标 URL 传递数据标记 `customdatatoken` 和字符串值 `yesnomatch`,这样页面就可以识别出使用了这个路由。

可以像传递数据标记一样为参数传递任意数据,上面仅是如何使用数据标记的一个例子。

42.1.3 使用路由参数

读取和使用通过 ASP.NET 路由功能传送给页面的参数值非常类似于读取和使用查询字符串。只是不使用 `Request.QueryString`,而使用 `Page.RouteData`,在加载页面时 ASP.NET 会填充 `Page.RouteData`,它包含提取路由参数或数据标记所需的所有信息。

通过表达式生成器和数据查询参数值,路由参数值也可用于 ASP.NET 标记。

1. RouteData 值

路由参数可通过 `Page.RouteData.Values` 属性应用于代码,`Page.RouteData.Values` 属性是 `RouteValueDictionary` 类的另一个实例。同样,数据标记可通过 `Page.RouteData.DataTokens` 来使用。在示例 Web 站点上,利用下面的代码,从 `Values` 属性中提取值,并通过 `DataGrid` 显示对应值:



可从
wrox.com
下载源代码

```
if (Page.RouteData.Values.Count == 0)
{
    this.NoRoutingLabel.Text = "None.";
}
else
{
    GridView2.DataSource = from entry in Page.RouteData.Values
                           select
```

```

new
{
    Name = entry.Key,
    Value = entry.Value
};

GridView2.DataBind();

```

代码段 PCSRoutingDemo/MasterPage.master.cs

另外，也可以按名称提取参数，因为它们在集合中建立了索引。例如：

```
string parameterValue = Page.RouteData.Values["pathparameter"] as string;
```

2. 表达式生成器

在 ASP.NET 标记文件中可以使用两个表达式生成器 `RouteValue` 和 `RouteUrl`，来提取和使用路由参数值。

`RouteValue` 可以内联使用，从而输出参数的值，如下所示：

```
<%$ RouteValue:parameterName %>
```

例如，在 Web 站点 PCSRoutingDemo 上，`Target.aspx` 文件包含如下标记代码行：

```
targetparameter = <asp:Label runat="server"
Text=" <%$ RouteValue:targetparameter %> " />
```

代码段 PCSRoutingDemo/Target.aspx



这行代码把 `Label` 控件的 `Text` 属性设置为 `targetparameter` 参数的值。

`RouteUrl` 用于构建匹配路由的 URL。把链接放在 ASP.NET 代码中时，这是一个很好的功能，因为如果路由定义不断发生变化，以这种方式创建的 URL 就会改变。要使用这个表达式生成器，可以内联使用下面的标记：

```
<%$ RouteUrl: parameterName = parameterValue %>
```

可以包含多个参数，各个名称/值对之间用逗号分隔开。这个表达式生成器尝试根据所使用的参数名，匹配指定的参数和路由。在一些情况下，可能有多个路由使用相同的参数，此时可以根据名称识别要使用的特定路由，如下所示：

```
<%$ RouteUrl: parameterName = parameterValue , routename = routeName %>
```

在 PCSRoutingDemo 中，母版页有两个不同的链接，其中一个使用了这个表达式生成器：

```

<asp:HyperLink ID="HyperLink3" runat="server" NavigateUrl="~/Target/Trevor"
>Navigation with routing</asp:HyperLink>

<asp:HyperLink ID="HyperLink4" runat="server"
NavigateUrl=" <%$ RouteUrl:targetparameter=Reginald,routename=TestRoute %>"
>Navigation with routing and expression builder</asp:HyperLink>

```

代码段 PCSRoutingDemo/MasterPage.master



这个链接的第二个版本展示了在标记中包含链接的最佳方式，因为它允许以后改变路由规范，如上所述。

3. 数据查询参数

在代码中使用路由参数的另一种方式是直接给数据查询提供它们。这个技巧允许从标识代码中直接提取数据库中的数据，而无需提取参数值，再在代码隐藏中使用它们。

为此，应在查询的参数中使用 `<asp:routeparameter>`。例如，可以在 `<asp:sqldatasource>` 数据源中使用下面的代码：

```
<asp:sqldatasource id="SqlDataSource1" runat="server"
    connectionstring="<%$ ConnectionStrings:ProductDatabase %%"
    selectcommand="SELECT ProductName,ProductId,ProductDescription FROM Products
        WHERE ProductName = @productname"
    <selectparameters>
        <asp:routeparameter name="productname" RouteKey="productnameparameter" />
    </selectparameters>
</asp:sqldatasource>
```

这段代码从路由参数 `productnameparameter` 中提取值，并把这个值传递给 SQL 查询中的 `@productname` 参数。

42.2 动态数据

如果仔细思考一下数据驱动的 Web 站点(目前大多数 Web 站点都是数据驱动的)，就会发现它们的许多功能都非常类似。这种 Web 站点包含如下一个或多个概念：

- 根据底层数据源(如数据库表或单个数据库行)中的数据，呈现动态生成的 HTML
- 站点地图中包含的页面会直接或间接映射到数据源(如数据库表)中的数据项
- 结构直接或间接与底层数据源的结构相关(站点的一部分可能映射到数据库表上，如 `About` 或 `Products`)
- 允许修改反射在页面上的底层数据源

如果希望构建数据驱动的站点，就可以使用相当标准化的代码来获得上述特性。可能把 ASP.NET 元素(如数据表)直接绑定到数据库表上，或者包含一个数据对象的中间层，来表示数据库中的数据，并绑定到这些数据上。前面的章节有许多用于这个目的的代码。

但是，因为这种情况非常常见，所以有一种替代方式：使用一个架构来提供许多代码，而不是自己完成繁琐的编码工作。ASP.NET 动态数据就是这样一个架构，它能非常容易地创建数据驱动的网站。除了提供上述代码(在动态数据网站中称为“搭框架(scaffolding)”)之外，动态数据网站还提供了许多额外的功能，如下所述。

本节将介绍如何创建动态数据站点，并了解它提供一些功能。

42.2.1 创建动态数据网站

了解动态数据网站提供了什么特性的最佳方式是在 Visual Studio 中构建这样一个网站，其过程非常简单。为了创建动态数据网站，需要有一些源数据。虽然可以使用任意数据，但也可以使用本

章可下载代码中提供的样本数据。这些数据是一个 SQL Server Express 数据库，其文件名是 MagicShop.mdf。图 42-3 显示了这个数据库包含的表。

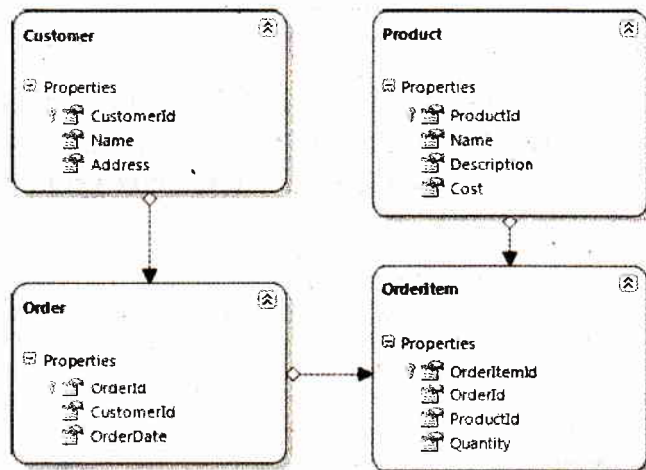


图 42-3

MagicShop.mdf 数据库表示一个简单的结构，该结构可用于电子商务网站。数据的类型和相互关系用来阐述动态数据站点的工作原理。

1. 选择数据访问模型

通过 File | New | Web Site 菜单项创建动态数据站点时，可以使用两个模板：

- Dynamic Data Linq to SQL Web Site
- Dynamic Data Entities Web Site

这两个模板基本相同，除了访问数据的方式不同：分别通过 LINQ to SQL 或 ADO.NET Entity Framework 来访问。选择哪个模板完全取决于个人喜好。动态数据网站的核心功能对于这两个模板相同。

本章的可下载代码包含两个网站 PCSDynamicDataDemoLinq 和 PCSDynamicDataDemo Entities，它们包含使用 MagicShop.mdf 数据库的站点的未定制版本。

2. 添加数据源

使用任一个可用的模板创建了站点后，下一步是添加数据源。这表示使用 LINQ to SQL 类或 ADO.NET Entity Model 模板给项目添加一个新项(最好添加到网站的 App_Code 目录中)。在添加之前，还可以把数据库的本地副本添加到网站的 App_Code 目录下，除非使用的是远程数据源。

如果 MagicShop.mdf 数据库用作一个测试，就可以根据所使用的站点模板执行如下操作(在把数据库添加到 App_Code 目录中之后)：

- 对于 LINQ to SQL，添加 LINQ to SQL 类，其文件名是 MagicShop.dbml。一旦添加了该文件后，就把 MagicShop.mdf 数据库中的所有表添加到设计器中。
- 对于 ADO.NET Entities，添加一个实体模型 MagicShop.edmx，在 Add New Item 向导中，使用默认设置，给数据库中的所有表添加实体。

3. 配置 Scaffolding

在完成动态数据网站的初始构建过程之前，还有一步要完成：必须在网站的 Global.asax 文件中为“搭框架”配置数据模型。除了解释性的注释中提到的区别之外，这个文件在两个站点模板类型中相同。如果查看该文件，会发现通过一个模型配置网站的框架，该模型在应用程序级别上定义，如下所示：

```
private static MetaModel s_defaultModel = new MetaModel();
public static MetaModel DefaultModel
{
    get
    {
        return s_defaultModel;
    }
}
```

Global.asax 文件在 RegisterRoutes() 方法中访问这个模型，该方法在 Application_Start() 处理程序中调用。这个方法还配置了网站中的动态数据路由(参见本章后面的内容)。该方法包含如下被注释掉的代码(简明起见，这里把代码放在两行上)：

```
//DefaultModel.RegisterContext(typeof(YourDataContextType),
// new ContextConfiguration() { ScaffoldAllTables = false });
```

配置模型仅需要取消注释这行代码，并给数据模型提供合适的数据上下文类型。还可以一开始就把 ScaffoldAllTables 属性改为 true，以指示模型为所有可用的表提供框架。以后可以撤销这个改变，以便更精细地控制创建什么框架(包括在网站上哪些数据是可见的，哪些数据是可编辑的等)，参见本章后面的内容。

利用上一节描述的 LINQ to SQL 类访问 MagicShop.mdf 数据库，LINQ to SQL 站点需要下面的代码：



可从
wrox.com
下载源代码

```
DefaultModel.RegisterContext(typeof(MagicShopDataContext),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

代码段 PCSDynamicDataDemoLinq/Global.asax

对于这个站点的 ADO.NET Entities 版本，代码如下：



可从
wrox.com
下载源代码

```
DefaultModel.RegisterContext(typeof(MagicShopModel.MagicShopEntities),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

代码段 PCSDynamicDataDemoEntities/Global.asax

4. 查看结果

现在，一切准备就绪，可以测试默认的动态数据网站了。无论使用什么模板，最终结果都相同。如果在浏览器中查看 Default.aspx 页面，屏幕将如图 42-4 所示。

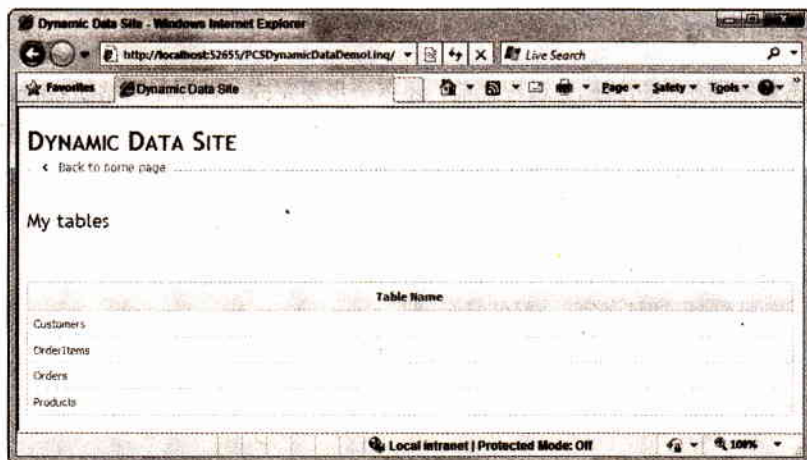


图 42-4

这个页面显示了到数据库中的每个表的一个链接列表，还显示了在 `Default.aspx` 页面中定义的其他一些信息，如下所示：



可从
wrox.com
下载源代码

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<asp:Content ID="headContent" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>

<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
    Runat="Server">
    <asp:ScriptManagerProxy ID="ScriptManagerProxy1" runat="server" />

    <h2 class="DDSubHeader">My tables</h2>

    <br /><br />

    <asp:GridView ID="Menu1" runat="server" AutoGenerateColumns="false"
        CssClass="DDGridView" RowStyle-CssClass="td" HeaderStyle-CssClass="th"
        CellPadding="6">
        <Columns>
            <asp:TemplateField HeaderText="Table Name" SortExpression="TableName">
                <ItemTemplate>
                    <asp:DynamicHyperLink ID="HyperLink1" runat="server"><%#
                        Eval("DisplayName") %></asp:DynamicHyperLink>
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
</asp:Content>
```

代码段 PCSDynamicDataDemoEntities/Default.aspx 和
PCSDynamicDataDemoLinq/Default.aspx

在网站的母版页和 CSS 文件中包含了许多显示代码，为了节省篇幅，这里没有列出来。上述代码中重要的部分是 `GridView` 控件，它包含一个 `DynamicHyperLink` 控件，后一个控件用于呈现表的链接。从代码隐藏中，把数据绑定到 `GridView` 控件上，如下所示(简明起见，略微进行了重新格式化)：



可从
wrox.com
下载源代码

```
protected void Page_Load(object sender, EventArgs e)
{
    System.Collections.IList visibleTables =
        ASP.global_asax.DefaultModel.VisibleTables;
    if (visibleTables.Count == 0)
    {
        throw new InvalidOperationException(
            "There are no accessible tables. Make sure that at least one data"
            + " model is registered in Global.asax and scaffolding is enabled"
            + " or implement custom pages.");
    }
    Menu1.DataSource = visibleTables;
    Menu1.DataBind();
}
```

代码段 PCSDynamicDataDemoEntities/Default.aspx.cs 和
PCSDynamicDataDemoLinq/Default.aspx.cs

这段代码从模型中提取可见表对应的一个列表(这里提取所有表, 因为如前所述, 为所有表都提供框架)。每个表都用一个 `MetaTable` 对象描述。DynamicHyperLink 控件可根据这些对象的属性, 智能地呈现表的页面链接。例如, Customers 表的链接如下所示:

/PCSDynamicDataDemoLinq/Customers/List.aspx

显然, 网站并没有定义这个页面, 而是如本章第一部分所述, 使用路由生成这个链接的内容。如果单击该链接, 就会看到 Customers 表的页面, 如图 42-5 所示。

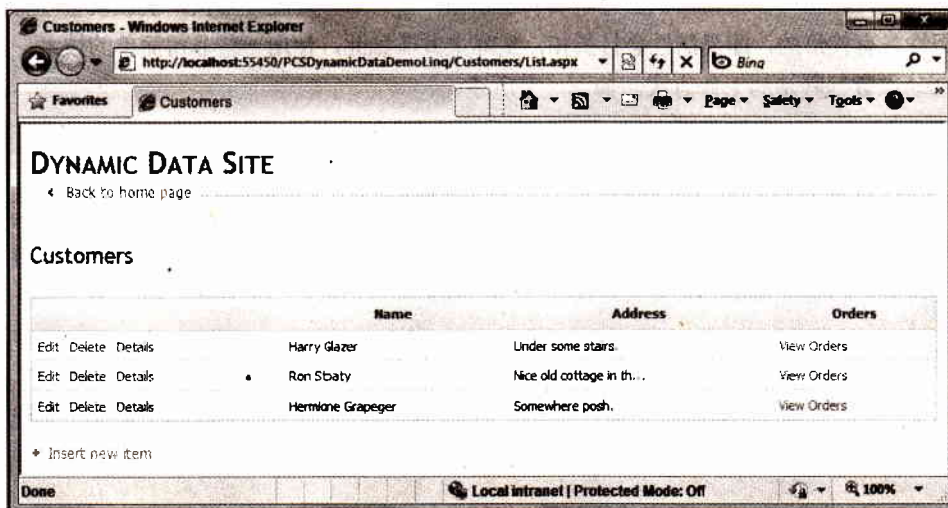


图 42-5

这个页面从模板中生成模板存储在站点的代码中, 其讨论参见下一节。可以使用这个页面上的链接检查、编辑、插入和删除记录, 还可以遍历表之间的关系。每个客户的 View Orders 链接会显示该客户的订单——页面上包含了一个客户下拉列表, 根据需要该下拉列表可用于查看不同客户的订单。

如果单击客户的 Edit 链接, 就会打开一个编辑客户的视图, 如图 42-6 所示。

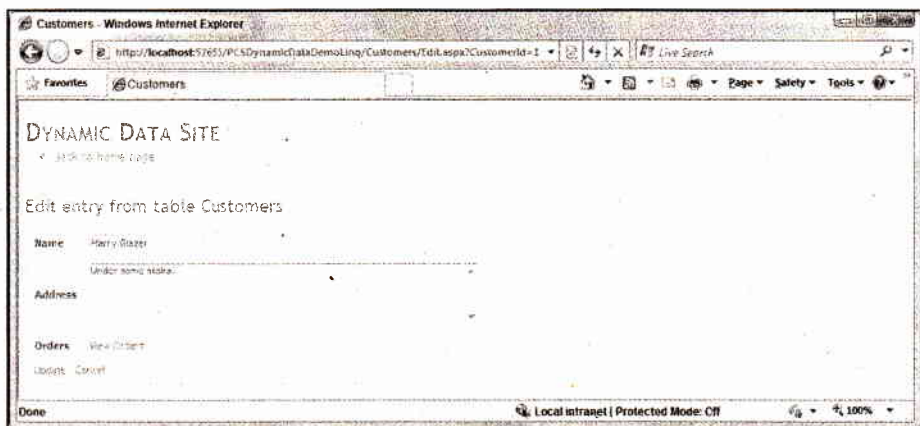


图 42-6

动态数据网站使用的模板系统允许根据列的数据类型，定制给表中的每一列显示什么信息。或者给指定的列进行进一步的、独特的定制：这里有非常大的灵活性。



MagicShop.mdf 数据库中使用的 ID 值存储为整数值。数据库也常常使用 GUID ID 值(使用 `uniqueidentifier` SQL 类型)。动态数据网站目前不能很好地处理上述 ID 值，尽管有变通方案，但目前没有理想的解决方案。

本节提到，甚至可以使用默认设置启动动态数据，而且运行速度非常快。现在读者可能会想，这种易用性是有代价的，即灵活性会打折扣。实际上并非如此，使用一些非常简单的技术，就可以完全定制动态数据网站的工作方式，如下一节所述。

42.2.2 定制动态数据网站

有许多方式可以定制动态数据网站，以达到预期的效果。这包括仅修改模板的 HTML 和 CSS，以及定制通过代码呈现数据的方式和特性修改方式。本节将介绍这些方式，从如何定制框架开始——此外，定制框架会影响主键值的可见性，如上一节所述。

1. 控制框架

在本章前面的示例动态数据网站中，为所有表(和这些表的所有列)自动配置了框架，为此，把网站的 `ContextConfiguration` 的 `ScaffoldTables` 属性设置为 `true`，对于 LINQ to SQL 站点，代码如下所示：



可从
wrox.com
下载源代码

```
DefaultModel.RegisterContext(typeof(MagicShopDataContext),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

代码段 PCSDynamicDataDemoLinq/Global.asax

如果把该值改为 `false`，默认就不为任何表或列提供任何框架。为了指示动态数据架构给表中的列搭建框架，必须在数据模型中提供元数据。动态数据运行库再读取这些元数据，来生成框架。元数据还可以提供其他事物的信息，如有效性验证逻辑——稍后讨论。

为表提供元数据需要执行两个步骤：

- 为要提供元数据的每个表创建一个元数据类定义，该类的成员映射到表中的列。
- 把元数据类关联到数据模型表类上。

在 LINQ to SQL 和 ADO.NET Entities 中，生成所有数据模型项，作为部分类定义。例如，在示例代码中，有一个 Customer 类(包含在 ADO.NET Entities 版本的 MagicShopModel 名称空间中)用于 Customer 表中的行。为了给 Customer 表中的行提供元数据，需要创建一个 CustomerMetadata 类。之后，就可以给 Customer 类提供第二个部分类定义，并使用 MetadataType 特性把这些类链接起来。

在 .NET Framework 中，通过数据注解支持元数据。MetadataType 特性和其他元数据特性都位于 System.ComponentModel.DataAnnotations 名称空间中。MetadataType 特性使用 Type 参数指定元数据的类型。控制框架的两个特性是 ScaffoldTable 和 ScaffoldColumn。这两个特性都有一个布尔参数，用于指定是否为表或列生成框架。

下面的代码通过示例演示了如何实现这个过程：



可从
wrox.com
下载源代码

```
using System.ComponentModel.DataAnnotations;

...

[MetadataType(typeof(CustomerMetadata))]
public partial class Customer { }
```

代码段 PCSDynamicDataDemoEntities/App_Code/Customer.cs 和
PCSDynamicDataDemoLinq/App_Code/Customer.cs



可从
wrox.com
下载源代码

```
using System.ComponentModel.DataAnnotations;

...

[ScaffoldTable(true)]
public class CustomerMetadata
{
    [ScaffoldColumn(false)]
    public object Address { get; set; }
}
```

代码段 PCSDynamicDataDemoEntities/App_Code/CustomerMetadata.cs 和
PCSDynamicDataDemoLinq/App_Code/CustomerMetadata.cs

其中，ScaffoldTable 特性指定，为 Customer 表生成框架，ScaffoldColumn 特性用于确保不给 Address 列搭建框架。注意，无论其类型是什么，该列都用 object 类型属性表示。只需确保属性名匹配列名即可。

这里的代码用于上一节介绍的示例网站，以隐藏客户的地址。

2. 定制模板

如本章前面所述，通过一系列模板生成动态数据。页面模板用于在不同类型的列表和细目页面中布置控件，字段模板用于在显示、编辑和外键选择模式下显示不同的数据类型。

所有这些模板都位于动态数据网站的 DynamicData 子文件夹中，该文件夹有表 42-1 所示的嵌套子文件夹：

表 42-1

目 录	描 述
Content	这个文件夹包含用于其他模板的图像和用于分页的模板(当数据列表很长时,会自动应用分页模板)
CustomPages	如果需要提供定制页面,就使用这个目录
EntityTemplates	这个目录包含用户控件模板,该模板用于在页面上以视图、编辑或插入模式显示单行数据
FieldTemplates	这个目录包含用于显示单列数据的用户控件模板
Filters	这个目录包含用户控件模板,该模板用于显示外键关系的筛选器。在默认代码中,这些都是下拉列表。例如, Orders 的列表页面包含一个筛选器,用于仅显示与某个特定用户相关联的订单
PageTemplates	这个目录包含主模板,用于在各种操作模式下显示单行或多行数据。这些页面的正文使用其他目录中的用户控件模板构建其内容

下面讨论这些模板及其代码隐藏,看看它们如何契合在一起。例如,使用 FieldTemplates 目录下的两个字段模板显示文本列。第一个字段模板 Text.ascx 如下:

```
<asp:Literal runat="server" ID="Literal1" Text="<%# FieldValueString %>" />
```

这行代码很简单——仅在 Literal 控件中输出列的文本值。但如果该列处于可编辑模式,就使用 Text_Edit.ascx:

```
<asp:TextBox ID="TextBox1" runat="server" Text="<%# FieldValueEditString %>"
  CssClass="DDTextBox"></asp:TextBox>

<asp:RequiredFieldValidator runat="server" ID="RequiredFieldValidator1"
  CssClass="DDControl" ControlToValidate="TextBox1" Display="Dynamic"
  Enabled="false" />

<asp:RegularExpressionValidator runat="server" ID="RegularExpressionValidator1"
  CssClass="DDControl" ControlToValidate="TextBox1" Display="Dynamic"
  Enabled="false" />

<asp:DynamicValidator runat="server" ID="DynamicValidator1" CssClass="DDControl"
  ControlToValidate="TextBox1" Display="Dynamic" />
```

这里使用 TextBox 控件呈现 Literal 控件,这样该控件就是可编辑的,根据需要,还可以使用 3 个验证控件提供验证功能。这些验证控件如何工作由数据模型及其关联的元数据确定。例如,不可空白列将导致 RequiredFieldValidator 控件有效。DynamicValidator 控件与元数据属性(如 StringLength)一起使用,StringLength 属性可用于设置字符串允许的最大长度。

3. 配置路由

在处理动态数据站点时,一个需要掌握的重要概念是,页面是根据动作生成的。动作是定义页面应如何响应的方式,例如,用户单击了某个链接后页面应如何响应。默认定义了 4 个页面动作: List、Details、Edit 和 Insert。

为动态数据站点定义每个页面模板(也称为视图)可以根据当前执行的动作做出不同的响应。网站的路由配置把动作和视图关联起来,每个路由都可以通过它应用的表选择性地约束。例如,可以创建一个用于列出客户的新视图。该视图可能执行与默认 List.aspx 视图不同的操作。要创建新视图,必须配置路由,以便使用正确的视图。

动态数据网站的默认路由在 Global.aspx 中配置,如下所示:



可从
wrox.com
下载源代码

```
routes.Add(new DynamicDataRoute("{table}/{action}.aspx")
{
    Constraints = new RouteValueDictionary(
        new { action = "List|Details|Edit|Insert" }),
    Model = DefaultModel
});
```

代码段 PCSDynamicDataDemoEntities/Global.asax 和
PCSDynamicDataDemoLinq/Global.asax

这段代码使用本章前面介绍的路由架构，尽管这里的路由使用 `DynamicDataRoute` 类型。这个类派生自 `Route` 类，为处理动作、视图和表提供了特定的功能。

在这段代码中注意，这个路由包含表名和动作名——动作的值限制为 4 个预定义的页面动作类型。要使用另一个视图列出客户，可以添加下面的路由(在已有路由的前面，或者这个路由应优先)：

```
routes.Add(new DynamicDataRoute("Customers/List.aspx")
{
    Table = "Customers",
    Action = PageAction.List,
    ViewName = "ListCustomers",
    Model = DefaultModel
});
```

这个路由把 `/Customers/List.aspx` 对应的 URL 与 `ListCustomers.aspx` 视图关联起来，为了使代码能正常运行，必须在 `PageTemplates` 目录中提供该名称的文件。这里还指定了 `Table` 和 `Action` 属性，因为它们在 URL 中不再可用。动态数据路由的工作方式是：使用 `{table}` 和 `{action}` 路由参数填充 `Table` 和 `Action` 属性，在这个 URL 中，没有显示这些参数。

以这种方式，可以构建非常复杂的路由系统，为表和动作提供专业化页面。还可以使用 `ListDetails.aspx` 视图，它是数据的主从视图，允许选择行和直接编辑。要使用这个视图，可以提供其他路由，或者仅取消注释下面的路由(由动态数据站点模板提供)：

```
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
//{
//    Action = PageAction.List,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
//{
//    Action = PageAction.Details,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
```

只要使用 `List` 或 `Details` 页面动作，这些路由就会使用 `ListDetails.aspx` 视图。不需要 `Edit` 和 `Insert` 页面动作，因为如前所述，编辑功能是直接提供的。它的一个例子如图 42-7 所示。

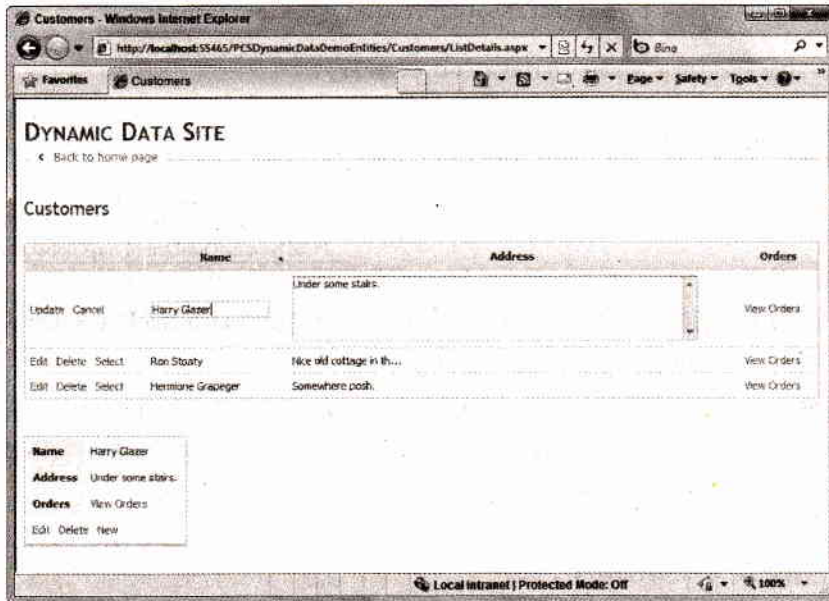


图 42-7

42.2.3 进一步开发

本节仅涉及使用动态数据的皮毛。把自定义代码和所提供的架构合并起来，能实现很多功能。使用架构可以极大地减少访问数据的工作量，开发人员就可以集中精力考虑网站的其他方面。关于这个主题的深入探讨有许多资源，希望本节为读者提供了足够的信息，使读者可以开始使用动态数据，知道其优点和可能的用法。

42.3 MVC

网站开发在继续演变，令人激动的新方法层出不穷。最近开发的一个激动人心的新方法是网站的 MVC(Model-View-Controller 模型-视图-控制器)体系结构模式。这不是一种新模式——实际上这个概念至少已有 30 年的历史了。但是，它长时间以来没有用在 Web 站点中，并且(对于本章的上下文比较重要)在 .NET 4 中才与 .NET Framework 集成起来。.NET 4 中包含的版本是 ASP.NET MVC 2，这个名称暗示，ASP.NET MVC 已经出现一段时间了，尽管在 .NET 4 之前单独下载它。

在介绍 MVC 的 ASP.NET 专用实现方式之前，先解释一下 MVC 的含义以及其用途。本节内容如下：

- MVC 的含义
- ASP.NET MVC 的含义
- 如何构建和定制基于 ASP.NET MVC 的网站

42.3.1 MVC 的含义

MVC 是一种编程方式，它把代码分为 3 个不同的部分：模型、视图和控制器。这不是 Web 特有的概念，而可以应用于涉及用户交互的任意系统。这些术语的定义如下：

- **模型**——表示应用程序的数据或状态的代码。注意这不包括数据存储器,例如,是否包含某数据是数据库的工作。但是,它封装了数据的所有操作,包括业务逻辑,如验证规则。它还包含与底层数据库交互操作所需的代码(如果存在底层数据存储器)。
- **视图**——视图是用户界面,它给用户提模型,允许用户在模型上执行操作。在视图和模型之间不必是一对一关系,给定的模型可以通过多个视图可视化。对模型的任意改变都应通过所关联视图中的变化反映出来。
- **控制器**——负责在模型和视图之间斡旋。用户在视图上执行操作时,控制器会响应该操作,根据需要,还通过与模型直接交互,把该操作传递给模型。

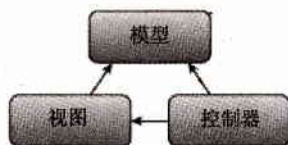


图 42-8

这 3 部分的关系如图 42-8 所示。

控制器可以访问视图和模型,因为它必须在模型和视图之间斡旋。因为视图必须呈现模型中的信息,并反映模型中的变化,所以它也需要模型的访问权限。

以这种方式编程有许多优点。第一个也是最明显的优点是,它清晰地区分了应用程序不同单元的功能。假定这些部分之间直接存在明确定义的合同,那么甚至可以独立地开发这些单元。这意味着,MVC 应用程序非常适用于许多开发人员同时开发一个项目的情况。

另一个优点是单元测试变得非常简单,因为每个部分可以在需要时进行独立的单元测试。有用户界面的应用程序很难测试,因为很难或不可能设计出准确模拟用户操作的测试,用户界面的自动化也是一个很大的挑战。因为 MVC 通过控制器准确地定义了用户可能执行的操作,所以可以肯定,如果这些控制器进行了充分的测试,就不会出问题。毕竟,视图只能与控制器交互,且其方式与任何其他客户端代码和控制器(包括测试代码)的交互方式相同。

MVC 的主要缺点是复杂性有所上升,而且要编写的代码可能较多。对于小型应用程序,这可能不太理想,它肯定会增加开发时间。另外,较难使用适合于应用程序的主要技术提供的所有特性。例如,当这种技术与 ASP.NET 一起使用时,就会看到,不能使用基于事件的用户交互和视图状态。

42.3.2 ASP.NET MVC 的含义

ASP.NET MVC 把 MVC 的原则应用于 ASP.NET 开发环境。这表示,ASP.NET 页面和控件用作视图,.NET 类用于控制器,数据架构(如 LINQ to SQL 或 ADO.NET Entity Framework)用于模型。

如果愿意,现在就可以停止阅读下去,并开始使用 ASP.NET 编写 MVC 网站。但是,.NET 4 中尤其令人激动的是,.NET 4 包含了 MVC 模式的完整实现,它支持的许多类可以处理许多管道(用于连接对象)和框架(减少重复或样板代码)。这就是 ASP.NET MVC 2 架构。

了解该架构提供了什么功能的最佳方式是构建一个应用程序(或者查看本章可下载代码对应的版本)。

42.3.3 简单的 ASP.NET MVC 应用程序

本节将使用本章前面介绍的 MagicShop.mdf 数据库构建一个 ASP.NET MVC 应用程序。在这个过程中,我们将讨论负责处理模型、视图和控制器的代码,它们是如何合并起来的,之后添加一些额外的定制。

1. 创建应用程序

首先创建一个新项目，在 Visual Studio 中选择 File | New | Project 菜单项，再选择 ASP.NET MVC 2 Web Application 模板，把项目命名为 PCSMNCMagicShop，把它保存在 C:\ProcSharp\Chapter42 目录下。虽然此时提示用户是否同时创建单元测试应用程序，但现在不选择这个选项(单元测试超出了本章的范围)。



注意，如果使用 File | New | Web Site 菜单项，就没有用于 ASP.NET MVC 的模板。ASP.NET MVC 应用程序必须是 Web 应用程序，不是 Web 站点，因为它们需要在部署前完全编译站点，才能工作。这是由于构成 ASP.NET MVC 应用程序的模块需要相互连接，且视图是动态生成的。

2. 查看应用程序

向导完成后，注意 Visual Studio 会自动创建许多目录和文件，如表 42-2 所示。

表 42-2

项	描 述
Content 目录	这个目录与动态数据网站的 Content 目录一样，也用于保存应用程序使用的图像和其他杂项资源。最初它包含 Web 应用程序使用的 CSS 文件 Site.css
Controllers 目录	这个目录包含站点使用的控制器类。模板包括两个用于开始工作的控制器 HomeController 和 AccountController
Models 目录	这个目录最初是空的，其中应放置模型。例如，可以添加 LINQ to SQL 类、ADO.NET Entity Framework 类，或者适合放在这里的任何自定义类
Scripts 目录	这个目录包含应用程序使用的脚本文件，如 JavaScript 文件。这里默认包含许多 JavaScript 文件，包括 Microsoft jquery 库和 AJAX 库需要的 JavaScript 文件
Views 目录	这个目录包含应用程序的视图，即 ASP.NET Web 窗体和用户控件。因为每个控制器都有自己的视图子目录，所以该目录默认包含 Home 和 Account 子目录的内容。其中还有一个 Shared 目录，它包含了通用视图和母版页，包括站点的默认母版页 Site.Master。这个目录还有一个 Web.config 文件，它配置为禁止直接访问视图文件
Global.asax	包含在应用程序根目录下的 Global.asax 文件(及其代码隐藏文件)会建立如本章前面所述的路由。ASP.NET MVC 在 RouteCollection 类上定义了扩展方法，以便于以它需要的方式添加路由
Web.config	这个配置文件引用 ASP.NET MVC 需要的所有额外资源，包括必要的程序集。它还配置 HTTP 处理程序，该程序用于根据所请求的路由路径动态地生成视图，并(根据项目模板)配置基于窗体的身份验证

此时可以运行应用程序，查看运行结果。这是一个功能非常全面的主页，还包括一个 About 页面和系统注册/登录。但是，如果尝试修改页面上的内容，就很容易了解该应用程序的工作方式。

3. 添加模型

根据要达到的目标的不同，添加模型、视图和控制器代码的顺序也不同。但一般先添加模型，

尤其是要访问数据库中的数据时，更应添加模型。如果先添加模型，可以从一开始就访问强类型化的模型类，ASP.NET MVC 向导可以使用模型类型信息自动生成许多必要的框架搭建代码。

在这个示例中，要使用本章前面介绍的 MagicShop.mdf 数据库。因为示例代码是 Web 应用程序，而不是 Web 站点，所以不能仅把这个文件复制到 App_Data 目录下，而必须右击该目录，选择 Add | Existing Item 菜单项，找到该数据库文件，再单击 Add 按钮，把它添加到 App_Data 目录中。或者通过 Visual Studio 构建一个数据库，连接到另一个数据库，或者选择一个完全不同的数据源。

一旦有了数据源，就可以使用自己喜欢的技术添加模型。这包括使用 LINQ to SQL 类、ADO.NET Entity Data Model，或者从头设计的自定义类。无论使用什么方法，都应把代码放在应用程序的 Models 文件夹中。简单起见，右击 Models 目录，选择 Add | New Item 菜单项，添加 ADO.NET Entity Data Model，其名称是 MagicShop。这个模板位于 Data 模板部分。因为有了数据源后，向导中的大多数默认设置就已经配置好了，所以只需选择要建模的表(所有表)。

添加模型代码后，在继续前必须构建解决方案，从而使后续的向导能够识别新代码。

4. 添加控制器

要给新模型添加控制器，可以使用 ASP.NET MVC Add Controller 向导。要访问这个向导，可以右击 Controllers 文件夹，选择 Add New Controller 菜单项。

在 MagicShop 数据库中为产品添加一个控制器，命名为 ProductsController，如图 42-9 所示，确保选中如图 42-9 所示的复选框。

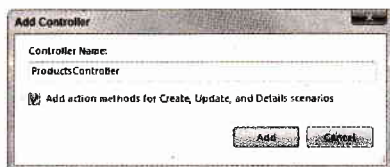


图 42-9

所添加的控制器包含几个方法，每个方法都返回一个 ActionResult。这些方法封装了用户(或其他代码)可以执行的操作，通过 URL 并使用路由技术(详见 42.3.4 节)提供每个方法。向导添加的方法如表 42-3 所示。

表 42-3

方 法	描 述
Index()	这是控制器的默认操作，如果没有指定其他方法，该方法确定使用哪些视图。例如，第一次显示视图时要显示什么内容。对于产品控制器，希望视图显示一个产品列表
Create()	为创建新项添加两个方法。第一个方法不带参数，在触发添加新项(也许单击了 Add new item 链接)的请求时使用。第二个方法负责实际添加项
Details()	这个方法负责显示单项的详细视图
Edit()	有两个编辑方法，这与两个创建方法相同。第一个方法接收一个 ID 值，并开始编辑已有的特定项。第二个方法接收两个参数，一个是 ID，另一个参数指定要修改的值

注意，添加的这些方法都没有实现代码，我们需要添加实现代码，后面各节讲述如何实现这些方法。

5. 添加索引操作

要实现的最简单的控制器方法是 `Index()`。这个方法只需根据项的集合返回一个视图。这需要在代码中引用模型，并编写获取和返回视图的代码，如下所示：



```
using PCSMVMagicShop.Models;

namespace PCSMVMagicShop.Controllers
{
    public class ProductsController : Controller
    {
        private MagicShopEntities entities = new MagicShopEntities();

        public ActionResult Index()
        {
            return View(entities.Products.ToList());
        }
        ...
    }
}
```

代码段 PCSMVMagicShop/Controllers/ProductsController.cs

这段代码告诉 ASP.NET MVC 框架，为 `Products` 项的集合(即 `IEnumerable<Product>` 实例)构建一个视图。为此，必须创建一个合适的视图。

6. 添加产品视图

添加视图并不像听上去那么困难。实际上，ASP.NET MVC 包含一个向导，该向导可以创建许多简单的视图。

为了使用这个向导，只需右击返回视图的方法，从上下文菜单中选择 `Add View` 命令，如图 42-10 所示。

这会打开 `Add View` 向导，通过它可以定制要添加的视图。在这个例子中，可以用默认值 `Index` 作为视图名称，但应为数据模型类型选择一个强类型化视图，并把 `View content` 改为 `List`，如图 42-11 所示。

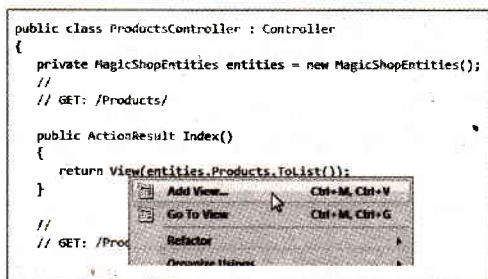


图 42-10

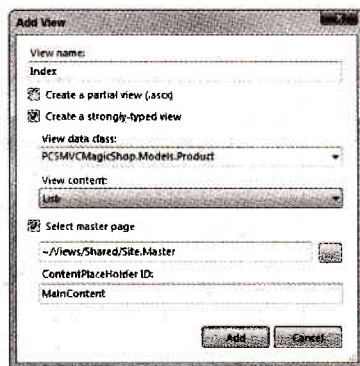


图 42-11

这个向导在 `Views` 文件夹中添加了一个新文件夹 `Products` 和一个 Web 窗体 `Index.aspx`。添加到视图中的代码如下所示：



可从
wrox.com
下载源代码

```
//<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable
        <PCSMVMagicShop.Models.Product>>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>
        Index</h2>
    <table>
        <tr>
            <th>
            </th>
            <th>
                ProductId
            </th>
            <th>
                Name
            </th>
            <th>
                Description
            </th>
            <th>
                Cost
            </th>
        </tr>
        <% foreach (var item in Model)
        { %>
        <tr>
            <td>
                <%= Html.ActionLink("Edit", "Edit", new { id=item.ProductId }) %>
                |
                <%= Html.ActionLink("Details", "Details", new { id=item.ProductId }) %>
            </td>
            <td>
                <%= Html.Encode(item.ProductId) %>
            </td>
            <td>
                <%= Html.Encode(item.Name) %>
            </td>
            <td>
                <%= Html.Encode(item.Description) %>
            </td>
            <td>
                <%= Html.Encode(String.Format("{0:F}", item.Cost)) %>
            </td>
        </tr>
        <% } %>
    </table>
    <p>
        <%= Html.ActionLink("Create New", "Create") %>
    </p>
</asp:Content>
```

代码段 PCSMVMagicShop/Views/Products/Index.aspx

这个视图没有代码隐藏文件，它继承了 `System.Web.Mvc` 名称空间中的 ASP.NET MVC 泛型基类 `ViewPage<T>`。所使用的泛型参数是 `IEnumerable<Product>`。

这段代码一目了然，但一些特性需要注意。首先，Web 窗体使用站点母版页，并填充内容区域。其次，ASP.NET MVC 架构使用模型类型信息，根据可用的属性智能地填充视图，包括输出多项信息循环结构。最后，使用几个辅助方法输出模型信息，添加动作链接。其中一个动作链接通过导航到合适的 URL 上，来调用一个控制器方法(详见 42.3.4 节)。

可以随意编辑这个视图。例如，在这个应用程序中，因为其实不需要提供 `ProductId` 值，所以可以删除下面的部分：

```
<th>
    ProductId
</th>
```

和

```
<td>
    <%= Html.Encode(item.ProductId) %>
</td>
```

不必担心，这不会破坏视图中的任何内容。

7. 测试产品视图的功能

现在该测试视图是否工作了。构建应用程序并运行它(出现提示时启用调试功能)。我们还没有添加给新视图提供链接的任何代码，但这不是个问题，因为所有管道都设置好了。看到主页时，手工编辑 URL，方法是把 `Products` 添加到最后面，并导航到该页面。显示器应更新为如图 42-12 所示的内容。

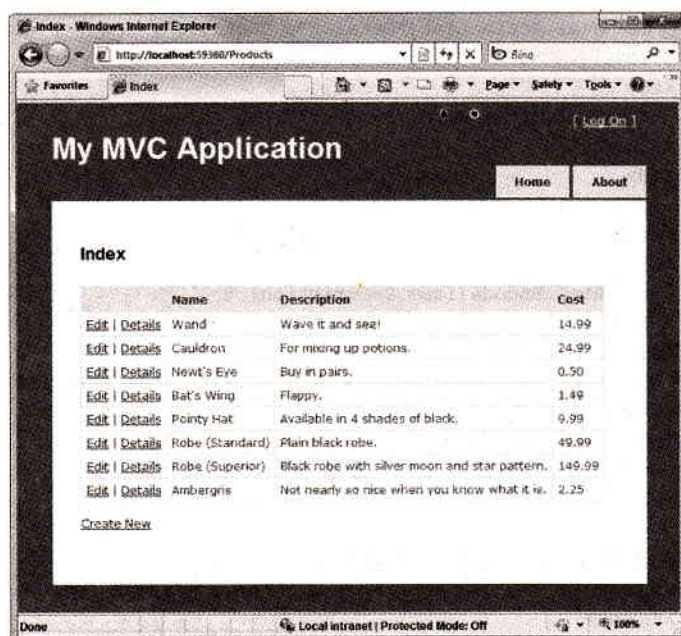


图 42-12

这样就实现了列表视图，而且工作量最少。

注意所提供的链接还没有激活。例如，如果尝试单击第一个 Details 链接，页面就会导航到 <http://localhost:59360/Products/Details/1>，并接收到如下错误消息：

```
The view 'Details' or its master was not found. The following locations were searched:
~/Views/Products/Details.aspx
~/Views/Products/Details.ascx
~/Views/Shared/Details.aspx
~/Views/Shared/Details.ascx
```

这段代码的作用是，视图提供一个 URL，该 URL 映射到 ProductsController 中的 Details() 方法上，而这个方法还没有配置。可以在这个方法中放置一个断点，并刷新页面，来确认这一点。这个方法当前调用无参数的 View() 方法，并返回一个 ActionResult，因为 View() 方法没有定义关联的视图，所以出错。

42.3.4 定制 ASP.NET MVC 应用程序

现在有了一个可以运行的 ASP.NET MVC 应用程序，尽管它肯定不是“功能齐全”的。此时，应用程序仅创建了一个简单的产品列表视图，但如上一节末尾所述，还需要其他视图，才能使应用程序正常工作。另外，因为通过手工输入 URL 来导航到产品列表肯定不是 Web 站点导航的好方法，所以也需要修改。

本节将详细讨论 URL 如何映射到控制器操作上，接着修改 Web 应用程序 PCSMVCMagicShop，为 Products 页面添加辅助功能。

1. URL 操作路由

如前所述，应用程序中的 Global.asax 文件配置了路由。这个文件的代码隐藏的内容如下所示：



```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            "Default", // Route name
            "{controller}/{action}/{id}", // URL with parameters
            new { controller = "Home", action = "Index", id = "" } // Parameter defaults
        );
    }

    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
}
```

代码段 PCSMVCMagicShop/Global.asax.cs

这段代码看起来很熟悉，参见 42.1 节。IgnoreRoute() 和 MapRoute() 方法是 ASP.NET MVC 定义的扩展方法，以便于定制路由。

这段代码只定义了一个默认路由，其形式是 {Controller}/{action}/{id}。还给 3 个路由参数 controller、action 和 id 提供了默认值 Home、Index 和空字符串。对于用于进入产品列表的 /Products 路由，这 3 个参数分别设置为 Products、Index 和空字符串。

现在，一切应步入正轨了。在开始测试前，添加一个控制器 ProductsController，它有一个命名为 Index() 的方法，目前它是默认的控制方法。目前可以推断出，输入 URL 时会发生什么：

(1) 路由系统解释输入的 URL，给前面提到的 controller、action 和 id 赋值。

(2) ASP.NET MVC 搜索并找到用于 Products 的控制器(注意 URL 不必是 ProductsController，会自动添加 Controller 后缀)。

(3) ASP.NET MVC 搜索并再次找到 Index 动作，它是 ProductsController 类中的 Index() 方法。

(4) 调用 Index() 方法时，ASP.NET MVC 搜索并找到 Products 视图文件夹中的视图，它可以显示 Products 对象集合。

(5) ASP.NET 呈显视图。

同样，以后尝试查看某项的细节时，会先解释 URL 路由 /Products/Details/1，再定位对应的方法 Details()。这里的最后一个形参作为实参传递给 Details() 方法，尽管此时似乎有点问题，因为该方法还没有实现，也没有可用的视图。

这个路由行为和 URL 直接映射到控制器方法上的方式，是理解 ASP.NET MVC 的关键。它表示，从给定的 URL 到代码有一条直接的路由，当然，代码执行什么操作完全取决于我们。因此“简单的”操作，如列出产品，仅是一个开始，我们可以用相同的构想方式编写任意动作。

然而，还必须考虑其他含义。实际上可以直接给用户代码以及传递给代码的参数。这表示，防备性的代码是最基本的，因为谁知道方法会从手工输入的 URL 中接收什么参数。

2. 动作链接

知道 URL 如何映射到动作上后，就应查看如何在应用程序中包含链接。可以在站点中硬编码链接，例如，要进入前面添加的 Products 页面，可以使用下面的代码：

```
<asp:HyperLink runat="server" NavigateUrl="~/Products">Products</asp:HyperLink>
```

但是，这个链接假定站点的结构不一定是一成不变的。如果在某一时刻改变了路由规范，这个链接就无效。

本章前面提到，ASP.NET 路由系统允许根据路由参数动态地生成链接。同样，ASP.NET MVC 允许根据控制器、动作、需要使用的其他值以及要显示的文本，生成链接。这可以通过 Html.ActionLink() 扩展方法来实现，根据要达到的目标，该方法有许多重载版本。

例如，如果查看 Views\Shared 文件夹中的母版页 Site.Master，就会看到下述定义站点菜单的代码：



可从
wrox.com
下载源代码

```
<div id="menucontainer">
  <ul id="menu">
    <li><%= Html.ActionLink("Home", "Index", "Home") %></li>
    <li><%= Html.ActionLink("About", "About", "Home") %></li>
  </ul>
</div>
```

代码段 PCSMVCMagicShop/Views/Shared/Site.Master

这里使用的重载版本有 3 个字符串参数 `linkText`、`actionName` 和 `controllerName`。第一个参数指定要显示的文本，另外两个参数指定要使用的动作和控制器。例如，上述代码段中的第一个调用映射到 `HomeController` 控制器类的 `Index()` 方法上。

用默认的路由设置进行呈现时，这些 `ActionLink()` 调用会输出如下 HTML：

```
<div id="menucontainer">
  <ul id="menu">
    <li><a href="/"> Home</a></li>
    <li><a href="/Home/About">About</a></li>
  </ul>
</div>
```

知道了 URL 如何映射到控制器方法上，以及如何在 `Global.asax` 中为路由设置默认值，这很有意义。另外优点是，如果路由改变了，也不必自己设计这些链接或重写它们。

为了以相同的方式给产品列表页面添加链接，应给母版页添加如下代码：



可从
wrox.com
下载源代码

```
<div id="menucontainer">
  <ul id="menu">
    <li><%= Html.ActionLink("Home", "Index", "Home") %></li>
    <li><%= Html.ActionLink("About", "About", "Home") %></li>
    <li><%= Html.ActionLink("Products", "Index", "Products") %></li>
  </ul>
</div>
```

代码段 PCSMVCMagicShop/Views/Shared/Site.Master

`ActionLink()` 的各种重载版本允许执行其他操作，如把参数传送给控制器方法。在前面创建的产品列表视图中，用如下代码呈现编辑项链接：



可从
wrox.com
下载源代码

```
<%= Html.ActionLink("Edit", "Edit", new { id=item.ProductId }) %>
```

代码段 PCSMVCMagicShop/Views/Products/Index.aspx

这个重载版本使用 `linkText`、`actionName` 和 `routeValues` 参数(当前控制器的默认控制器名称)。在这个例子中，使用一个路由值 `id`，它映射到 `ProductsController` 类中的 `Edit()` 方法的 `id` 形参上。这个调用会呈现链接，如下面的示例 HTML 所示：

```
<a href="/Products/Edit/1"> Edit </a>
```

3. 用 `Details()` 查看数据项

用户单击应用程序中的 `Details` 链接时，会请求 `ProductsController` 类中的 `Details()` 动作方法。这会执行该方法，并作为形参传递产品 ID。我们只需定位所请求的产品，如果该产品存在，就返回一个新视图，如下所示：

```
public ActionResult Details(int id)
{
    var product = (from e in entities.Products
                   where e.ProductId == id
                   select e).FirstOrDefault();
```

```
if (product != null)
{
    return View(product);
}
else
{
    return RedirectToAction("Index");
}
```

如果没有找到该产品,就使用 `RedirectToAction()` 方法,顾名思义,这个方法把用户重定向到另一个视图上,在本例中是 Index 视图。

要添加新视图,应右击 `Details()` 方法,与以前一样,再单击 Add View 菜单项。这次给 Product 类创建一个强类型化的视图,包含 Details 视图的内容,如图 42-13 所示。

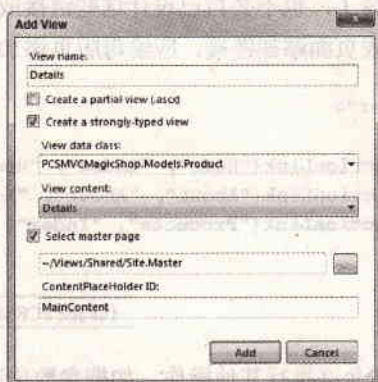


图 42-13

现在运行应用程序时,就可以通过上一节添加的 Products 菜单项导航到产品列表上,并查看任意项的细节,如图 42-14 所示。

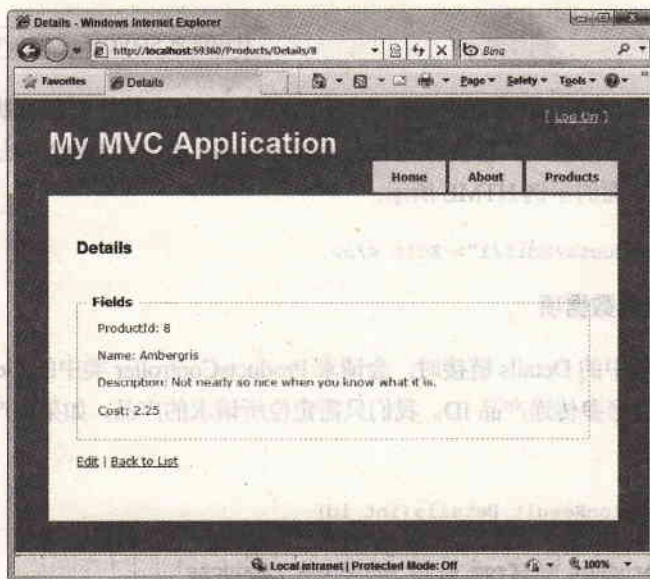


图 42-14

如果把 URL 手工编辑为某个不存在产品的 ID，就把用户重定向到产品清单上。

4. 用 Create()添加数据项

创建新数据项有两个关联的动作，它们在代码中的定义如下：



可从
wrox.com
下载源代码

```
public ActionResult Create()
{
    return View();
}
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection collection)
{
    try
    {
        // TODO: Add insert logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

代码段 PCSMVCMagicShop/Controllers/ProductsController.cs

第一个动作没有参数，仅重定向到创建视图上，该视图可以用与其他视图相同的方式添加(在 Add View 向导中把 View content 值设置为 Create)。第二个动作用于在数据源中创建一个新项，我们必须自己实现这个动作。

注意 Create()方法的第二个重载版本包含一个 AcceptVerbs 特性。它限制了用于 POST 的 HTTP 动词，在创建数据项时，如果要提交的信息比用于 GET 请求的 URL 包含的信息多，这就是有意义的。

为第一个 Create()方法创建的视图包含如下自动生成的代码：



可从
wrox.com
下载源代码

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>
        Create</h2>
    <%= Html.ValidationSummary(
        "Create was unsuccessful. Please correct the errors and try again.") %>
    <% using (Html.BeginForm())
        {%>
        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="ProductId">
                    ProductId: </label>
                <%= Html.TextBox("ProductId") %>
                <%= Html.ValidationMessage("ProductId", "**") %>
            </p>
            <p>
                <label for="Name">
                    Name:</label>
                <%= Html.TextBox("Name") %>
                <%= Html.ValidationMessage("Name", "**") %>
            </p>
        }
    %>
```

```

</p>
<p>
    <label for="Description">
        Description: </label>
    <%= Html.TextBox("Description") %>
    <%= Html.ValidationMessage("Description", "**") %>
</p>
<p>
    <label for="Cost">
        Cost: </label>
    <%= Html.TextBox("Cost") %>
    <%= Html.ValidationMessage("Cost", "**") %>
</p>
<p>
    <input type="submit" value="Create" />
</p>
</fieldset>
<% } %>
<div>
    <%=Html.ActionLink("Back to List", "Index") %>
</div>
</asp:Content>

```

代码段 PCSMVCMagicShop/Views/Products/Create.aspx

这里包含几个辅助方法，以创建和编辑窗体。这里再次说明了如何询问数据模型来提取字段要使用的信息和需要的验证。窗体本身包含在下面的代码块中：

```

using (Html.BeginForm())
{
    ...
}

```

这个辅助方法创建了一个窗体，该窗体使用标准的 HTML<input type="submit" />控件提交，使窗体值传递给动作方法。接着访问这些值，以便添加一个新数据项。

可以删除下面的代码段，以删除用于产品 ID 的文本框，因为这是自动生成的：

```

<p>
    <label for="ProductId">
        ProductId: </label>
    <%= Html.TextBox("ProductId") %>
    <%= Html.ValidationMessage("ProductId", "**") %>
</p>

```

添加数据项所需的代码如下所示：

```

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "ProductId")] Product productToCreate)
{
    try
    {
        if (!ModelState.IsValid)
        {
            return View();
        }
    }
}

```



可从
wrox.com
下载源代码

```

        entities.AddToProducts(productToCreate);
        entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

```

代码段 PCSMVCMagicShop/Controllers/ProductsController.cs

这段代码需要注意几点。第一，可以修改方法签名，以便接受一个强类型化的 **Product** 值。这比遍历一个窗体值集合简单，但为了便于操作，需要排除 **ProductId** 字段，因为不能传递它的值。**Bind** 特性可以达到这个目的。

其次，注意 **ModelState.IsValid** 属性用于检查传递给方法的模型是否有效，如果无效，方法就仅返回无参数的创建视图。此时，会显示验证汇总和验证错误，如图 42-15 所示。

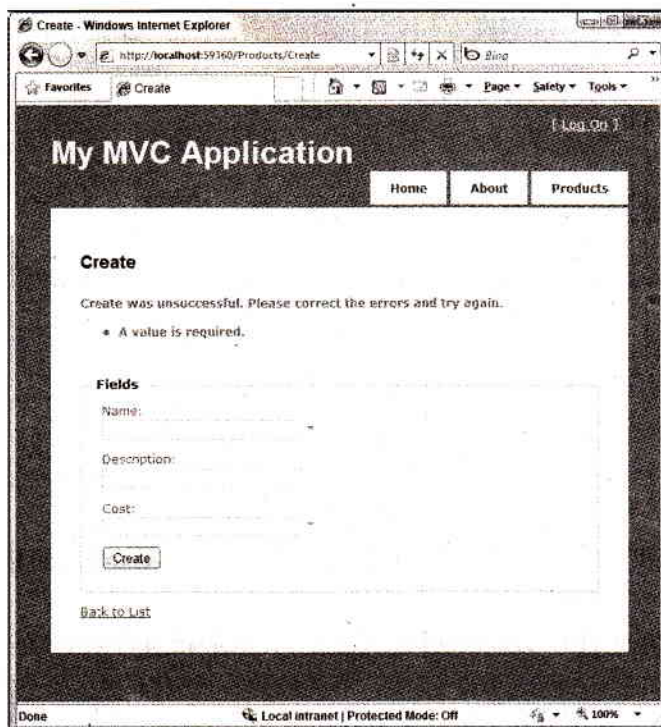


图 42-15

如果没有错误，就通过 ADO.NET Entity Framework 定义的 **AddToProducts()** 方法添加新数据项，并把页面重定向到列表视图上，如下所示：

```

        entities.AddToProducts(productToCreate);
        entities.SaveChanges();
        return RedirectToAction("Index");
    }
}

```

这里要注意的一个要点是，验证可以通过本章前面介绍的数据注解来控制。大多数默认验证都自动来自于数据库中的数据约束，但可以改变这个行为。例如，可以用下面两个类限制产品名称的

字符数:



可从
wrox.com
下载源代码

```
using System.ComponentModel.DataAnnotations;

namespace PCSMVCMagicShop.Models
{
    [MetadataType(typeof(ProductMetadata))]
    public partial class Product
    {
    }
}
```

代码段 PCSMVCMagicShop/Models/Product.cs



可从
wrox.com
下载源代码

```
using System.ComponentModel.DataAnnotations;

namespace PCSMVCMagicShop.Models
{
    public class ProductMetadata
    {
        [StringLength(20)]
        public object Name { get; set; }
    }
}
```

代码段 PCSMVCMagicShop/Models/ProductMetadata.cs

这段代码会验证数据, 如图 42-16 所示。

Create was unsuccessful. Please correct the errors and try again.

- The field Name must be a string with a maximum length of 20.

Fields

Name:

A really long name, too long in fact *

图 42-16

显然, 在控制器中需要使用的代码实质上是当前使用的模型, 这个选项可能并不总是可用, 但在正确的场合下使用会非常有效。

也可以编辑所创建的视图, 以提供更有意义的文本, 但前面创建的框架适合于大多数需求。在这个例子中, 可以改变窗体的标题和验证消息, 并为数据项描述提供更大的空间。

5. 用 Edit()方法编辑数据项

编辑数据项使用的代码非常类似于创建新数据项, 这里不再具体说明如何编辑数据项, 而只查看控制器中的代码:



可从
wrox.com
下载源代码

```
public ActionResult Edit(int id)
{
    var productToEdit = (from p in entities.Products
                        where p.ProductId == id
                        select p).First();

    return View(productToEdit);
}
```

```

    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(Product productToEdit)
    {
        try
        {
            var originalProduct = (from p in entities.Products
                                   where p.ProductId == productToEdit.ProductId
                                   select p).First();

            if (!ModelState.IsValid)
            {
                return View(originalProduct);
            }

            entities.ApplyCurrentValues(originalProduct.EntityKey.EntitySetName,
                                       productToEdit);

            entities.SaveChanges();

            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }
}

```

代码段 PCSMVCMagicShop/Controllers/ProductsController.cs

这次需要实现 Edit() 方法的两个重载版本。第一个版本从 ID 值中获取一个 Product 项，第二个版本使用标准的 ADO.NET Entity Framework 代码更新该项。

注意，因为这次在 Create() 方法的第二个重载版本中需要产品 ID，所以不能像细节视图和创建视图那样，在编辑视图中删除为 ProductId 字段自动生成的 HTML。另外，为了禁止编辑该字段，可以把它包含为一个隐藏字段，如下所示：



可从
wrox.com
下载源代码

```

<fieldset>
    <legend> Fields </legend>
    <%= Html.Hidden("ProductId", Model.ProductId) %>

```

代码段 PCSMVCMagicShop/Views/Products/Edit.aspx

这将确保该字段会正确传送给动作方法。

42.3.5 进一步开发

与动态数据网站一样，本章仅简要介绍了 ASP.NET MVC，它还有更多的内容值得关注，包括：

- 添加自己的动作
- 自定义动作返回类型
- 单元测试控制器
- 自定义 HTML 辅助方法

如果对此感兴趣，首先应关注 ASP.NET MVC 网站 <http://asp.net/mvc>，其中包含教程、示例和进

一步研究这种令人激动的新技术所需的所有内容。互联网上还有许多其他很好的资源,包括 Scott Guthrie 的博客 <http://weblogs.asp.net/scottgu>。Scott 还和 ASP.NET MVC 方面的权威作者 Rob Conery、Scott Hanselman、Phil Haack 为 Wrox 编写一本很好的书 *Professional ASP.NET MVC 1.0*(ISBN: 9780470384611)。尽管这本书仅涵盖 ASP.NET MVC 的 1.0 版本,但仍值得一读,希望 ASP.NET MVC 2 的更新版本不久就会面世。

42.4 小结

本章完成了 ASP.NET 之旅,介绍了这种语言和建立在该架构上的工具的某些最新内容,这些新增内容使 ASP.NET 成为 Web 开发的一种令人激动的方式。

本章首先介绍了路由如何使友好的 URL 更容易实现,如何控制路由,以用户希望的方式调整 URL 映射。

接着讨论了如何创建动态数据网站,提供数据与强大的前端 ASP.NET UI 之间的无缝集成。这种技术默认情况下提供非常多的代码,有时让人感觉是网站自己编写自己。我们还论述了动态数据站点支持的两个不同的数据访问方法——LINQ to SQL 和 ADO.NET Entity Framework,以及如何定制框架,从而使站点以希望的方式工作。

最后探讨了使用 ASP.NET 的最新(也许是最先进的)Web 技术: ASP.NET MVC 2 架构。它提供了一个健壮的结构,非常适用于需要合理单元测试的大型应用程序。它很容易提供高级功能,且工作量最小,这个架构提供了逻辑结构,把功能分隔开,使代码容易理解和维护。

下一章将学习 Windows Communication Foundation,它允许以灵活、安全的方式在应用程序之间进行远程调用。它是许多 Web 开发的基础,因为我们常常要为 Web 应用程序从不同的源中提取数据,而不是像本章这样就地包含数据库。

第VI部分

通 信

- 第43章 WCF
- 第44章 Windows WF 4
- 第45章 对等网络
- 第46章 消息队列
- 第47章 Syndication

第 43 章

WCF

本章内容:

- WCF 概述
- 简单的服务和客户端
- 服务、操作、数据和消息协定
- 服务的实现
- 给通信使用绑定
- 服务的不同宿主选项
- 通过服务引用和编程方式创建客户端
- 双工通信

在.NET 3.0 推出之前, 一个企业解决方案需要几种通信技术。对于独立于平台的通信, 则使用 ASP.NET Web 服务。对于比较高级的 Web 服务——可靠性、独立于平台的安全性和原子事务等技术——Web Services Enhancements 增加了 ASP.NET Web 服务的复杂性。如果要求通信比较快, 客户和服务都是.NET 应用程序, 就应使用.NET Remoting 技术。.NET Enterprise Services 支持自动事务处理, 它默认使用 DCOM 协议, 比用.NET Remoting 快。DCOM 也是允许传递事务的唯一协议。所有这些技术都有不同的编程模型, 这些模型都需要开发人员有许多技巧。

.NET Framework 3.0 引入了一种的通信技术 WCF, 它包含上述技术的所有功能, 把它们合并到一个编程模型中: Windows Communication Foundation(WCF)。

43.1 WCF 概述

WCF 合并了 ASP.NET Web 服务、.NET Remoting、消息队列和 Enterprise Services 的功能, WCF 的功能包括:

- **存储组件和服务**——与联合使用自定义主机、.NET Remoting 和 WSE 一样, 也可以将 WCF 服务存放在 ASP.NET 运行库、Windows 服务、COM+进程或 Windows 窗体应用程序中, 进行对等计算。
- **声明行为**——不要求派生自基类(这个要求存在于.NET Remoting 和 Enterprise Services 中), 而可以使用属性定义服务。这类似于用 ASP.NET 开发的 Web 服务。

- **通信信道**——在改变通信信道方面，.NET Remoting 非常灵活，WCF 也不错，因为它提供了相同的灵活性。WCF 提供了用 HTTP、TCP 和 IPC 信道进行通信的多条信道。也可以创建使用不同传输协议的自定义信道。
- **安全结构**——为了实现独立于平台的 Web 服务，必须使用标准化的安全环境。所提出的标准用 WSE 3.0 实现，这在 WCF 中被继承下来。
- **可扩展性**——.NET Remoting 有丰富的扩展功能。它不仅能创建自定义信道、格式化程序和代理，还能将功能注入客户端和服务上的消息流。WCF 提供了类似的可扩展性。但是，WCF 的扩展性用 SOAP 标题创建。
- **支持以前的技术**——要使用 WCF，根本不需要完全重写分布式解决方案，因为 WCF 可以与已有的技术集成。WCF 提供的信道使用 DCOM 与服务组件通信。用 ASP.NET 开发的 Web 服务也可以与 WCF 集成。

最终目标是通过进程或不同的系统、通过本地网络或通过 Internet 收发客户和服务之间的消息。如果需要以独立于平台的方式尽快收发消息，就应这么做。在远程视图上，服务提供了一个端点，它用协定、绑定和地址来描述。协定定义了服务提供的操作，绑定给出了协议和编码信息，地址是服务的位置。客户需要一个兼容的端点来访问服务。

图 43-1 显示了参与 WCF 通信的组件。

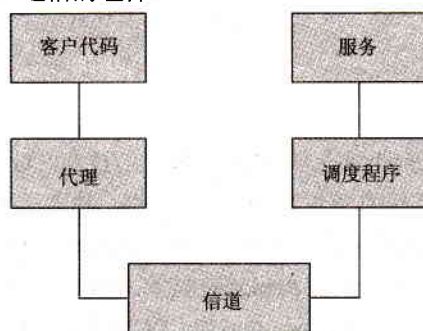


图 43-1

客户调用代理上的一个方法。代理提供了服务定义的方法，但把方法调用转换为一条消息，并把该消息传输到信道上。信道有一个客户端部分和一个服务器端部分，它们通过一个网络协议来通信。在信道上，把消息传递给调度程序，调度程序再把消息转换为用服务调用的方法调用。

WCF 支持几个通信协议。为了进行独立于平台的通信，需要支持 Web 服务标准。要在 .NET 应用程序之间通信，可以使用较快的通信协议，其系统开销较小。

下面几节介绍用于独立于平台的通信的核心服务的功能。

- **SOAP(Simple Object Access Protocol, 简单对象访问协议)**: 一个独立于平台的协议，它是几个 Web 服务规范的基础，支持安全性、事务、可靠性
- **WSDL(Web Services Description Language, Web 服务描述语言)**: 提供描述服务的元数据
- **REST(Representational State Transfer, 代表性状态传输)**: 由支持 REST 的 Web 服务用于在 HTTP 上通信
- **JSON(JavaScript Object Notation, JavaScript 对象标记)**: 便于在 JavaScript 客户端上使用

43.1.1 SOAP

为了进行独立于平台的通信, 可以使用 SOAP 协议, 它得到 WCF 的直接支持。SOAP 最初是 Simple Object Access Protocol 的缩写, 但自从 SOAP 1.2 以来, 就不再是这样了。SOAP 不再是一个对象访问协议, 因为可以发送用 XML 架构定义的消息。

· 服务从客户中接收 SOAP 消息, 并返回一条 SOAP 响应消息。SOAP 消息包含信封, 信封包含标题和正文。

```
<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
  xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
  </s:Header>
  <s:Body>
    <ReserveRoom xmlns="http://www.wrox.com/ProCSharp/2010">
      <roomReservation
        xmlns:d4pl="http://schemas.datacontract.org/2009/10/Wrox.ProCSharp.WCF"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <d4pl:RoomName> Hawelka </d4pl:RoomName>
        <d4pl:StartDate> 2010-06-21T08:00:00</d4pl:StartDate>
        <d4pl:EndDate>2010-06-21T14:00:00</d4pl:EndDate>
        <d4pl:Contact>Georg Danzer</d4pl:Contact>
        <d4pl:Event>White Horses</d4pl:Event>
      </roomReservation>
    </ReserveRoom>
  </s:Body>
</s:Envelope>
```

标题是可选的, 可以包含寻址、安全性和事务信息。正文包含消息数据。

43.1.2 WSDL

WSDL(Web Services Description Language, Web 服务描述语言)文档描述了服务的操作和消息。WSDL 定义了服务的元数据, 这些元数据可用于为客户端应用程序创建代理。

WSDL 包含如下信息:

- 消息的类型——用 XML 架构描述。
- 从服务中收发的消息——消息的各部分是用 XML 架构定义的类型。
- 端口类型——映射服务协定, 列出了用服务协定定义的操作。操作包含消息, 例如, 与请求和响应序列一起使用的输入和输出消息。
- 绑定信息——包含用端口类型列出的操作和用 SOAP 变体定义的操作。
- 服务信息——把端口类型映射到端点地址。



在 WCF 中, WSDL 信息由 MEX(Metadata Exchange, 元数据交换)端点提供。

43.1.3 REST

WCF 还提供了使用 REST 进行通信的功能。REST 并不是一个协议, 但定义了使用服务访问资

源的几条规则。支持 REST 的 Web 服务是基于 HTTP 协议和 REST 规则的简单服务。规则按 3 个类别来定义：可以用简单的 URI 访问的服务，支持 MIME 类型，以及使用不同的 HTTP 方法。支持 MIME 类型，就可以从服务中返回不同的数据格式，如普通 XML、JSON 或 AtomPub。HTTP 请求的 GET() 方法从服务中返回数据。其他方法有 PUT()、POST() 和 DELETE()。PUT() 方法用于更新服务端，POST() 方法可创建一个新资源，DELETE() 方法删除资源。

REST 允许给服务发送的请求比 SOAP 小。如果不需要 SOAP 提供的事务、安全消息(例如，安全通信仍可通过 HTTPS 进行)和可靠性，则利用 REST 构建的服务可以减小系统开销。

使用 REST 体系结构时，服务总是无状态的，服务的响应可以缓存。

43.1.4 JSON

除了发送 SOAP 消息之外，从 JavaScript 中访问服务最好使用 JSON。NET 3.5 包含一个数据协定序列化程序，可以用 JSON 标记创建对象。

JSON 的系统开销比 SOAP 小，因为它不是 XML，而是为 JavaScript 客户端进行了优化。这使之非常适用于从 Ajax 客户端使用。Ajax 详见第 41 章。JSON 没有提供通过 SOAP 标题发送所具备的可靠性、安全性和事务功能，但这些通常是 JavaScript 客户端不需要的功能。

43.2 简单的服务和客户端

在详细介绍 WCF 之前，首先看一个简单的服务。该服务用于预订会议室。

要存储会议室预订信息，应使用一个简单的 SQL Server 数据库和 RoomReservation 表。可以从 www.wrox.com 和本书附赠光盘中找到这个数据库和本章的示例代码。

创建一个空白的解决方案 RoomReservation，在其中添加一个新的组件库项目 RoomReservationData。第一个实现的项目只包含访问数据库的代码。因为 ADO.NET Entity Framework 使数据库访问代码非常简单，所以这里使用这个 .NET 4 技术。



ADO.NET Entity Framework 详见第 31 章。

添加一个新项，即 ADO.NET Entity Data Model，把它命名为 RoomReservationModel.edmx。选择 RoomReservations 数据库，再选择 RoomReservations 表，创建一个实体数据模型。之后，在设计器上就可以看到 RoomReservation 实体，如图 43-2 所示。

这个设计器会创建一个 RoomReservation 实体类，它为表中的每一列和 RoomReservationSEneities 类包含属性。RoomReservation Seneities 类连接到数据库上，并用注册实体类，以获得所有变更通知。

要使用 ADO.NET Entity Framework 读写数据库中的数据，添加 RoomReservationData 类。ReserveRoom() 方法将一条会议室预订信息写入数据库。GetReservations() 方法从指定的数据范围内返回一个会议



图 43-2

室预订数组。



可从
wrox.com
下载源代码

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

namespace Wrox.ProCSharp.WCF
{
    public class RoomReservationData
    {
        public void ReserveRoom(RoomReservation roomReservation)
        {
            using (var data = new RoomReservationsEntities())
            {
                data.RoomReservations.AddObject(roomReservation);
                data.SaveChanges();
            }
        }

        public RoomReservation[] GetReservations(DateTime fromDate,
                                                    DateTime toDate)
        {
            using (var data = new RoomReservationsEntities())
            {
                return (from r in data.RoomReservations
                        where r.StartDate > fromDate && r.EndDate < toDate
                        select r).ToArray();
            }
        }
    }
}
```

代码段 RoomReservationData/RoomReservationData.cs

现在开始创建服务。

43.2.1 服务协定

在解决方案中添加一个 WCF Service Library 类型的新项目，把项目命名为 RoomReservationService。把生成的文件 IService1.cs 重命名为 IRoomService.cs，把 Service1.cs 重命名为 RoomReservationService.cs。把生成的文件中的名称空间改为 Wrox.ProCSharp.WCF.Service。还需要引用 RoomReservationData 程序集，使实体类型和 RoomReservationData 类可用。

服务提供的操作可以通过接口来定义。IRoomService 接口定义了 ReserveRoom() 和 GetRoomReservations() 方法。服务协定用 ServiceContract 属性定义。由服务定义的操作应用了 OperationContract 属性。



可从
wrox.com
下载源代码

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
    [ServiceContract()]
    public interface IRoomService
    {
    }
```

```

[OperationContract]
bool ReserveRoom(RoomReservation roomReservation);
[OperationContract]
RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate);

```

代码段 RoomReservationService/IRoomService.cs

43.2.2 服务的实现

RoomReservationService 服务类实现 IRoomService 接口。实现服务时,只需调用 RoomReservationData 类的相应方法。



可从
wrox.com
下载源代码

```

using System;
using System.Linq;

namespace Wrox.ProCSharp.WCF
{
    public class RoomReservationService : IRoomService
    {
        public bool ReserveRoom(RoomReservation roomReservation)
        {
            var data = new RoomReservationData();
            data.ReserveRoom(roomReservation);
            return true;
        }

        public RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate)
        {
            var data = new RoomReservationData();
            return data.GetReservations(fromDate, toDate);
        }
    }
}

```

代码段 RoomReservationService/RoomReservationService.cs

43.2.3 WCF 服务宿主和 WCF 测试客户端

WCF Service Library 项目模板创建了一个应用程序配置文件 App.Config, 它需要适用于新类名和新接口名。service 元素引用了包含名称空间的服务类型 RoomReservationService, 协定接口需要用 endpoint 元素定义。

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- When deploying the service library project, the content of the config file
  must be added to the host's app.config file. System.Configuration does not support config
  files for libraries. -->
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.RoomReservationService">

```

```

<host>
  <baseAddresses>
    <add baseAddress =
      "http://localhost:8732/Design_Time_Addresses/RoomReservationService
        /Service1/"
    />
  </baseAddresses>
</host>
<!-- Service Endpoints -->
<!-- Unless fully qualified, address is relative to base address supplied above -->
<endpoint address="" binding="wsHttpBinding"
  contract="Wrox.ProCSharp.WCF.IRoomService">
  <!--
    Upon deployment, the following identity element should be removed or replaced to
    reflect the identity under which the deployed service runs. If removed, WCF will
    infer an appropriate identity automatically.
  -->
  <identity>
    <dns value="localhost"/>
  </identity>
</endpoint>
<!-- Metadata Endpoints -->
<!-- The Metadata Exchange endpoint is used by the service to describe itself to
  clients. -->
<!-- This endpoint does not use a secure binding and should be secured or removed
  before deployment -->
<endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
</service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior>
      <!-- To avoid disclosing metadata information,
        set the value below to false and remove the metadata endpoint above before
        deployment -->
      <serviceMetadata httpGetEnabled="True"/>
      <!-- To receive exception details in faults for debugging purposes,
        set the value below to true. Set to false before deployment
        to avoid disclosing exception information -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

代码段 RoomReservationService/App.config



服务地址 `http://localhost:8731/Design_Time_Addresses` 有一个关联的访问控制列表 (ACL), 它允许交互式用户创建一个侦听端口。默认情况下, 非管理员用户不允许在监听模式下打开端口。使用命令行实用程序 `netsh http show urlacl` 可以查看 ACL, 用 `netsh http add url=http://+8080/ MyURI user=someUser` 添加新项。

从 Visual Studio 2010 中启动这个库，会启动 WCF 服务宿主，它显示为任务栏的注意区域中的一个图标。单击这个图标会打开 WCF 服务宿主窗口，如图 43-3 所示。在其中可以查看服务的状态。WCF 库应用程序的项目属性包含 WCF 选项的选项卡，在其中可以选择运行同一个解决方案中的项目时，是否启动 WCF 服务宿主。默认打开这个选项。另外在项目属性的调试配置中，会发现已定义了命令行参数/client: "WcfTestClient.exe"。WCF 服务主机使用这个选项，会启动 WCF 测试客户端，如图 43-4 所示，该测试客户端可用于测试应用程序。双击一个操作，输入字段就会显示在应用程序的右边，可以在其中填充要发送给服务的数据。单击 XML 选项卡，可以看到已收发的 SOAP 消息。

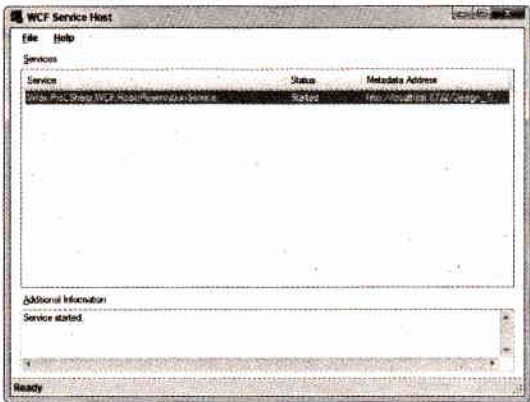


图 43-3

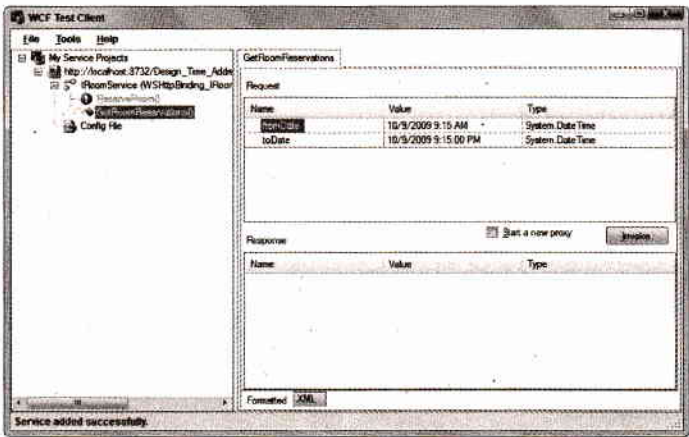


图 43-4

43.2.4 自定义服务宿主

使用 WCF 可以在任意宿主上运行服务。可以为对等服务创建一个 Wondows 窗体或 WPF 应用程序，或创建一个 Windows 服务，或用 Windows Activation Services(WAS)存放该服务。控制台应用程序也适合于演示简单的主机。

对于服务主机，必须引用 RoomReservationService 库。该服务从实例化和打开 ServiceHost 类型的对象开始。这个类在 System.ServiceModel 名称空间中定义。实现该服务的 RoomReservationService 类在构造函数中定义。调用 Open()方法会启动服务的监听器信道，该服务准备用于侦听请求。Close()

方法会停止信道。



可从
WTOX.COM
下载源代码

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
    class Program
    {
        internal static ServiceHost myServiceHost = null;

        internal static void StartService()
        {
            myServiceHost = new ServiceHost(typeof(RoomReservationService));
            myServiceHost.Open();
        }

        internal static void StopService()
        {
            if (myServiceHost.State != CommunicationState.Closed)
                myServiceHost.Close();
        }

        static void Main()
        {
            StartService();

            Console.WriteLine("Server is running. Press return to exit");
            Console.ReadLine();

            StopService();
        }
    }
}
```

代码段 RoomReservationServiceHost/Program.cs

对于 WCF 配置，需要把用服务库创建的应用程序配置文件复制到宿主应用程序中。使用 WCF Service Configuration Editor 可以编辑这个配置文件，如图 43-5 所示。

使用自定义服务宿主，可以在 WCF 库的项目设置中取消用来启动 WCF 服务宿主的 WCF 选项。

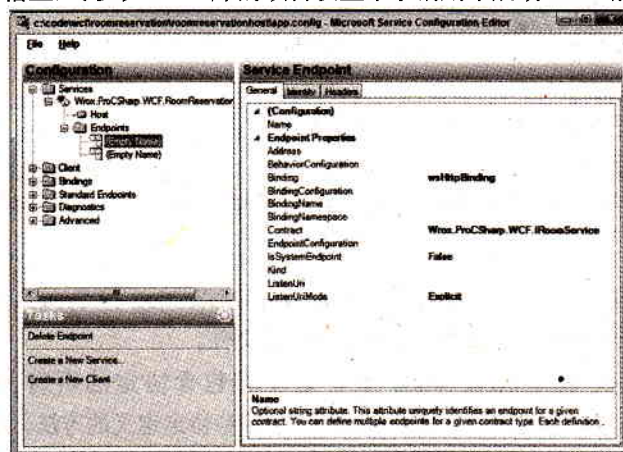


图 43-5

43.2.5 WCF 客户端

对于客户端，WCF 也可以灵活选择所使用的应用程序类型。客户端也可以是一个简单的控制台应用程序。但是，对于预订会议室，应创建一个包含控件的 Windows 窗体应用程序，如图 43-6 所示。

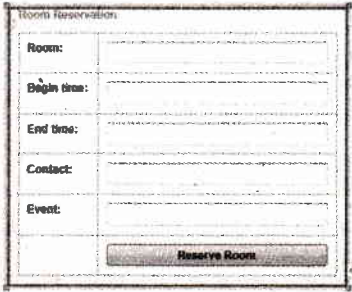


图 43-6

因为服务用 mexHttpBinding 绑定提供了一个 MEX 端点，并且通过行为配置来启用元数据的访问，所以可以从 Visual Studio 中添加一个服务引用。在添加服务引用时，会弹出如图 43-7 所示的对话框。单击 Discover 按钮时，可以在同一个解决方案中找到服务。

进入服务的链接，将服务引用名设置为 RoomReservationService。服务引用名定义了所生成的代理类的名称空间。

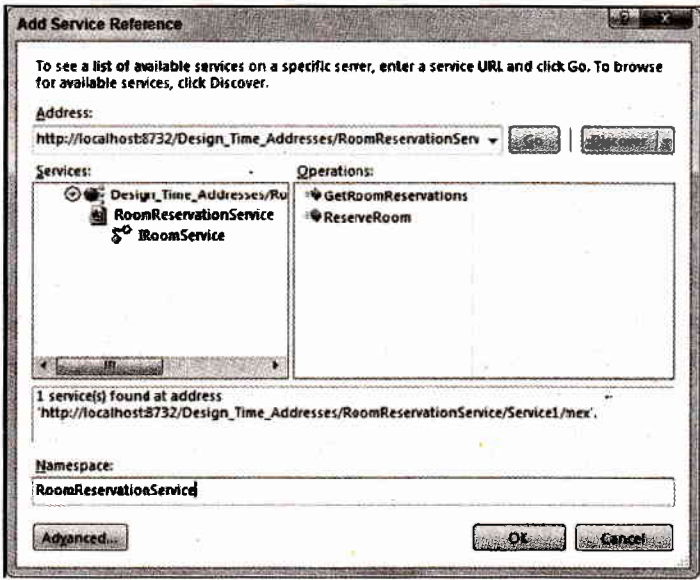


图 43-7

添加服务引用，会在服务中添加对 System.Runtime.Serialization 和 System.ServiceModel 程序集的引用，还会添加一个包含绑定信息和端点地址的配置文件。

从数据协定中生成 RoomReservation 类。这个类包含协定的所有 [DataMember] 元素。RoomServiceClient 类是客户端的代理，该客户端包含由服务协定定义的方法。使用这个客户端，可以将会议室预订信息发送给正在运行的服务。



可从
wrox.com
下载源代码

```
private void OnReserveRoom(object sender, RoutedEventArgs e)
{
    var reservation = new RoomReservation()
    {
        RoomName = textRoom.Text,
        Event = textEvent.Text,
        Contact = textContact.Text,
        StartDate = DateTime.Parse(textStartTime.Text),
        EndDate = DateTime.Parse(textEndTime.Text)
    };

    var client = new RoomServiceClient();
    bool reserved = client.ReserveRoom(reservation);
    client.Close();
    if (reserved)
        MessageBox.Show("reservation ok");
}
```

代码段 RoomReservationClient/MainWindow.xaml.cs

运行服务和客户端，就可以将会议室预订信息添加到数据库中。在 RoomReservation 解决方案的设置中，可以配置多个启动项目，在本例中是 RoomReservationClient 和 RoomReservationHost。

43.2.6 诊断

运行客户端和服务应用程序时，知道后台发生了什么很有帮助。为此，WCF 使用一个需要配置的跟踪源。可以使用 Service Configuration Editor，选择 Diagnostics 节点，启用 Tracing and Message Logging 功能，来配置跟踪。把跟踪源的跟踪级别设置为 Verbose 会生成非常详细的信息。这个配置更改把跟踪源和监听器添加到应用程序配置文件中，如下所示：



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="System.ServiceModel.MessageLogging"
              switchValue="Verbose,ActivityTracing">
        <listeners>
          <add type="System.Diagnostics.DefaultTraceListener" name="Default">
            <filter type="" />
          </add>
          <add name="ServiceModelMessageLoggingListener">
            <filter type="" />
          </add>
        </listeners>
      </source>
      <source name="System.ServiceModel" switchValue="Verbose,ActivityTracing"
              propagateActivity="true">
        <listeners>
          <add type="System.Diagnostics.DefaultTraceListener" name="Default">
            <filter type="" />
          </add>
          <add name="ServiceModelTraceListener">
            <filter type="" />
          </add>
        </listeners>
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

```
</listeners>
</source>
</sources>
<sharedListeners>
  <add initializeData="c:\code\wcf\roomreservation\roomreservationhost\
                                     app_messages.svclog"
        type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
        name="ServiceModelMessageLoggingListener" traceOutputOptions="Timestamp">
    <filter type="" />
  </add>
  <add initializeData="c:\code\wcf\roomreservation\roomreservationhost\
                                     app_tracelog.svclog"
        type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
        name="ServiceModelTraceListener" traceOutputOptions="Timestamp">
    <filter type="" />
  </add>
</sharedListeners>
<trace autoflush="true" />
</system.diagnostics>
<!-->
```

代码段 RoomReservationHost/App.config

WCF 类的实现代码使用 System.ServiceModel 和 System.ServiceModel.MessageLogging 跟踪源来写入跟踪消息。跟踪和配置跟踪源及监听器的更多内容详见第 19 章。

启动应用程序时，使用 verbose 跟踪设置的跟踪文件会很快变得很大。为了分析 XML 日志文件中的信息，.NET SDK 包含一个 Service Trace Viewer 工具 svctraceviewer.exe。图 43-8 显示了这个工具选择跟踪和消息日志文件后的视图。在默认配置下，可以看到互换了几条消息，其中的许多消息都与安全性相关。根据安全性需求，可以选择其他配置选项。

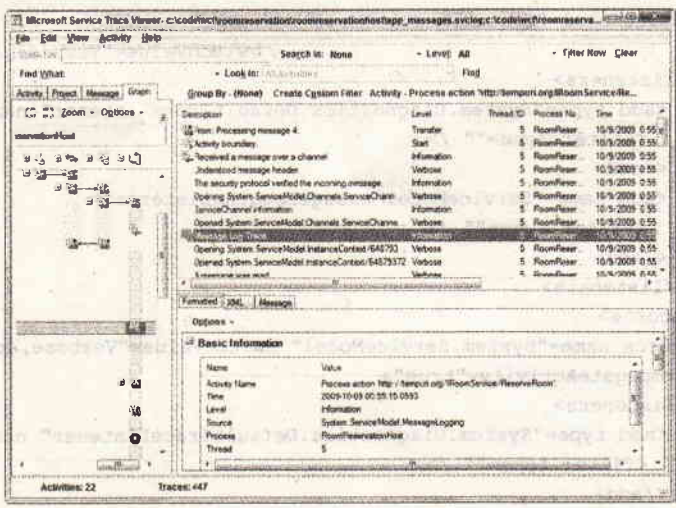


图 43-8

下面详细介绍 WCF 的细节和不同的选项。

43.3 协定

协定定义了服务提供的功能和客户端可以使用的功能。协定可以完全独立于服务的实现代码。

由 WCF 定义的分合同可以分为 3 种不同的类型:数据协定、服务协定和消息协定。协定可以用.NET 属性来指定:

- 数据协定——数据协定定义了从服务中接收和返回的数据。用于收发消息的类关联了数据协定属性。
- 服务协定——服务协定用于定义描述了服务的 WSDL。这个协定用接口或类定义。
- 消息协定——如果需要完全控制 SOAP 消息,消息协定就可以指定应放在 SOAP 标题中的数据以及放在 SOAP 正文中的数据。

下面几节将详细探讨这些,并进一步讨论定义协定时应考虑的版本问题。

43.3.1 数据协定

在数据协定中,把 CLR 类型映射到 XML 架构。数据协定不同于其他.NET 序列化机制。在运行时序列化中,所有字段都会序列化(包括私有字段),而在 XML 序列化中,只序列化公共字段和属性。数据协定要求用 `DataMember` 特性显式标记要序列化的字段。无论字段是私有或公共的,还是应用于属性,都可以使用这个特性。

```
[DataContract(Namespace="http://www.thinktecture.com/SampleServices/2010")
public class RoomReservation
{
    [DataMember] public string Room { get; set; }
    [DataMember] public DateTime StartDate { get; set; }
    [DataMember] public DateTime EndDate { get; set; }
    [DataMember] public string ContactName { get; set; }
    [DataMember] public string EventName { get; set; }
}
```

为了独立于平台和版本,如果要求用新版本修改数据,且不破坏旧客户端和服务,使用数据协定是指定要发送哪些数据的最佳方式。还可以使用 XML 序列化和运行时序列化。XML 序列化是 ASP.NET Web 服务使用的机制,.NET Remoting 使用运行时序列化。

使用 `DataMember` 特性,可以指定表 43-1 中的属性。

表 43-1

用 DataMember 指定的属性	说 明
Name	序列化元素的名称默认与应用了[DataMember]特性的字段或属性同名。使用 Name 属性可以修改该名称
Order	Order 属性指定了数据成员的序列化顺序
IsRequired	使用 IsRequired 属性,可以指定元素必须经过序列化,才能接收。这个属性可以用于解决版本问题。 如果在已有的协定中添加了成员,协定不会被破坏,因为在默认情况下字段是可选的(IsRequired=false)。将 IsRequired 属性设置为 true,就可以破坏已有的协定

(续表)

用 DataMember 指定的属性	说 明
EmitDefaultValue	EmitDefaultValue 属性指定有默认值的成员是否应序列化。如果把 EmitDefaultValue 属性设置为 true，具有该类型的默认值的成员就不序列化

43.3.2 版本问题

创建数据协定的新版本时，注意更改的种类，如果应同时支持新旧客户端和新旧服务，就应执行相应的操作。

在定义协定时，应使用 DataContractAttribute 的 Namespace 属性添加 XML 名称空间信息。如果创建了数据协定的新版本，破坏了兼容性，就应改变这个名称空间。如果只添加了可选的成员，就没有破坏协定——这就是一个可兼容的改变。旧客户端仍可以给新服务发送消息，因为不需要其他数据。新客户端可以给旧服务发送消息，因为旧服务仅忽略额外的数据。

删除字段或添加需要的字段会破坏协定。此时还应改变 XML 名称空间。名称空间的名称可以包含年份和月份，如 <http://thinkecture.com/SampleServices/2009/10>。每次做了破坏性的修改时，都要改变名称空间，如把年份和月份改为实际值。

43.3.3 服务协定

服务协定定义了服务可以执行的操作。ServiceContract 特性与接口或类一起使用，来定义服务协定。由服务提供的方法通过 IRoomService 接口应用 OperationContract 特性，如下所示：

```
[ServiceContract]
public interface IRoomService
{
    [OperationContract]
    bool ReserveRoom(RoomReservation roomReservation);
}
```

可能用 ServiceContract 特性设置的属性如表 43-2 所示。

表 43-2

用 ServiceContract 设置的属性	说 明
ConfigurationName	这个属性定义了配置文件中服务配置的名称
CallbackContract	当服务用于双工消息传递时，CallbackContract 属性定义了了在客户端中实现的合同
Name	这个 Name 属性定义了 WSDL 中 <portType> 元素的名称
Namespace	Namespace 属性定义了 WSDL 中 <portType> 元素的 XML 名称空间
SessionMode	使用 SessionMode 属性，可以定义调用这个协定的操作所需的会话。其值用 SessionMode 枚举定义，包括 Allowed、NotAllowed 和 Required
ProtectionLevel	ProtectionLevel 属性确定了绑定是否必须支持保护通信。其值用 ProtectionLevel 枚举定义，包括 None、Sign、EncryptAndSign

使用 OperationContract 特性可以指定如表 43-3 所示的属性。

表 43-3

用 OperationContract 指定的属性	说 明
Action	WCF 使用 SOAP 请求的 Action 属性, 把该请求映射到相应的方法上。Action 属性的默认值是协定 XML 名称空间、协定名和操作名的组合。该消息如果是一条响应消息, 就把 Response 添加到 Action 字符串中。指定 Action 属性可以重写 Action 值。如果指定值 “*”, 服务操作就会处理所有消息。
ReplyAction	Action 属性设置了引入的 SOAP 请求的 Action 名, 而 ReplyAction 属性设置了响应消息的 Action 名。
AsyncPattern	如果使用异步模式来实现操作, 就把 AsyncPattern 属性设置为 true。异步模式详见第 20 章。
IsInitiating IsTerminating	如果协定由一系列操作组成, 且初始化操作本应把 IsInitiating 属性赋予它, 该系列的最后一个操作就需要指定 IsTerminating 属性。初始化操作启动一个新会话, 服务器用终止操作来关闭会话。
IsOneWay	设置 IsOneWay 属性, 客户端就不会等待响应消息。在发送请求消息后, 单向操作的调用者无法直接检测失败。
Name	操作的默认名称是指定了操作协定的方法名。使用 Name 属性可以修改该操作的名称。
ProtectionLevel	使用 ProtectionLevel 属性可以确定消息是应只签名, 还是应加密后签名。

在服务协定中, 也可以用 [DeliveryRequirements] 特性定义服务的传输要求。RequireOrderedDelivery 属性指定所发送的消息必须以相同的顺序到达。使用 Queued DeliveryRequirements 属性可以指定, 消息以断开连接的模式发送, 例如, 使用消息队列(参见第 46 章)。

43.3.4 消息协定

如果需要完全控制 SOAP 消息, 就可以使用消息协定。在消息协定中, 可以指定消息的哪些部分要放在 SOAP 标题中, 哪些部分要放在 SOAP 正文中。下面的例子显示了 ProcessPersonRequestMessage 类的一个消息协定。该消息协定用 MessageContract 特性指定。SOAP 消息的标题和正文用 MessageHeader 和 MessageBodyMember 属性指定。指定 Position 属性, 可以确定正文中的元素顺序。还可以为标题和正文字段指定保护级别。

```
[MessageContract]
public class ProcessPersonRequestMessage
{
    [MessageHeader]
    public int employeeId;

    [MessageBodyMember(Position=0)]
    public Person person;
}
```

ProcessPersonRequestMessage 类与用 IProcessPerson 接口定义的服务协定一起使用:

```
[ServiceContract]
public interface IProcessPerson
{
}
```

```
[OperationContract]
public PersonResponseMessage ProcessPerson(
    ProcessPersonRequestMessage message);
}
```

与 WCF 服务相关的另一个重要协定是错误协定，这个协定参见 43.4.2 节。

43.4 服务的实现

服务的实现代码用 ServiceBehavior 特性标记，如下面的 RoomReservationService 类所示：

```
[ServiceBehavior]
public class RoomReservationService: IRoomService
{
    public bool ReserveRoom(RoomReservation roomReservation)
    {
        // implementation
    }
}
```

ServiceBehavior 特性用于描述 WCF 服务提供的操作，以截获所需功能的代码，如表 43-4 所示。

表 43-4

用 ServiceBehavior 指定的属性	说 明
TransactionAutoComplete- OnSessionClose	当前会话正确完成时，就自动提交该事务。这类似于第 51 章讨论的 Enterprise Services 中的 AutoComplete 属性
TransactionIsolationLevel	要定义服务中事务的隔离级别，可以把 TransactionIsolationLevel 属性设置为 IsolationLevel 枚举中的一个值。事务信息的隔离级别详见第 23 章
ReleaseServiceInstanceOn- TransactionComplete	完成事务处理后，可回收服务的实例
AutomaticSessionShutdown	如果在客户端关闭连接时没有关闭会话，就可以把 AutomaticSessionShutdown 属性设置为 false。在默认情况下，会关闭会话
InstanceContextMode	使用 InstanceContextMode 属性，可以确定应使用有状态的对象还是无状态的对象。默认设置为 InstanceContextMode.PerCall，为每个方法调用创建一个新对象。其他可能的设置有 PerSession 和 Single。这两个设置都使用有状态的对象。但是，PerSession 会为每个客户端创建一个新对象，而 Single 允许在多个客户端上共享同一个对象
ConcurrencyMode	因为有状态的对象可以由多个客户端(或同一个客户的多个线程)使用，所以必须注意这种对象类型的并发问题。如果把 ConcurrencyMode 属性设置为 Multiple，多个线程就可以访问对象，但必须处理同步问题。如果把该属性设置为 Single，一次就只有一个线程能访问对象，但不必处理同步问题；如果客户端较多，就可能出现可伸缩性问题。Recntrant 值表示只有从调用返回的线程可以访问对象。对于无状态的对象，这个设置没有任何意义，因为每个方法调用都会实例化一个新对象，所以不共享状态

(续表)

用 ServiceBehavior 指定的属性	说 明
UseSynchronizationContext	Windows 窗体和 WPF 控件的成员都只能从创建线程中调用。如果服务位于 Windows 应用程序中, 其方法调用了控件成员, 就把 UseSynchronizationContext 属性设置为 true。这样, 服务就运行在 SynchronizationContext 属性定义的线程中
IncludeExceptionDetailInFaults	在 .NET 中, 错误被看作异常。SOAP 指定, SOAP 错误返回客户端, 以防服务器出问题。出于安全考虑, 最好不要把服务器端异常的细节返回客户端。因此, 异常默认转换为未知错误。要返回特定的错误, 可抛出 FaultException 类型的异常。为了便于调试, 返回真实的异常信息很有帮助。此时应把 IncludeExceptionDetailInFaults 属性的设置改为 true。这里抛出 FaultException <TDetail> 异常, 其中原始异常包含详细信息
MaxItemsInObjectGraph	使用 MaxItemsInObjectGraph 属性, 可以限制要序列化的对象数。如果序列化一个对象树型结构, 则默认的限制过低
ValidateMustUnderstand	把 ValidateMustUnderstand 属性设置为 true, 表示必须理解 SOAP 标题(默认)

为了演示服务行为, IStateService 接口定义了一个服务协定, 其中的两个操作用于获取和设置状态。有状态的服务协定需要一个会话。这就是把服务协定的 SessionMode 属性设置为 SessionMode.Required 的原因。服务协定还将 IsInitiating 和 IsTerminating 特性应用于操作协定, 以定义了启动和关闭会话的方法。



可从
wrox.com
下载源代码

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IStateService
{
    [OperationContract(IsInitiating=true)]
    void Init(int i);

    [OperationContract]
    void SetState(int i);

    [OperationContract]
    int GetState();

    [OperationContract(IsTerminating=true)]
    void Close();
}
```

代码段 StateServiceSample/IStateService.cs

服务协定由 StateService 类实现。服务的实现代码定义了 InstanceContextMode.PerSession, 使状态与实例保持同步。



可从
wrox.com
下载源代码

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class StateService: IStateService
{
    int i = 0;

    public void Init(int i)
```

```

    {
        this.i = i;
    }

    public void SetState(int i)
    {
        this.i = i;
    }

    public int GetState()
    {
        return i;
    }

    public void Close()
    {
    }
}

```

代码段 StateServiceSample/StateService.cs

现在必须定义对地址和协议的绑定。其中，将 `basicHttpBinding` 赋予服务的端点：



可从
wrox.com
下载源代码

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="StateServiceSample.Service1Behavior"
        name="Wrox.ProCSharp.WCF.StateService">
        <endpoint address="" binding="basicHttpBinding"
          bindingConfiguration=""
          contract="Wrox.ProCSharp.WCF.IStateService">
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8731/Design_Time_Addresses/
              StateServiceSample/Service1/" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="StateServiceSample.Service1Behavior">
          <serviceMetadata httpGetEnabled="True"/>
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

代码段 StateServiceSample/app.config

如果用定义的配置启动服务宿主，就会抛出一个 `InvalidOperationException` 类型的异常。该异常

的错误消息是“协定需要会话，但绑定 ‘BasicHttpBinding’ 不支持它，或者没有正确配置为支持它。”

并不是所有绑定都支持所有服务。因为服务协定需要用 [ServiceContract(ServiceMode=ServiceMode.Required)] 特性指定一个会话，主机可能会失败，这是因为所配置的绑定不支持会话。

只要修改对绑定的配置，使之支持会话(如 wsHttpBinding)，服务器就会成功启动。



```
<endpoint address="" binding="wsHttpBinding"
    bindingConfiguration=""
    contract="Wrox.ProCSharp.WCF.IStateService">
</endpoint>
```

代码段 StateServiceSample/app.config

43.4.1 以编程方式创建客户端

现在可以创建客户端应用程序了。在前面的例子中，通过添加服务引用来创建客户端应用程序。除了添加服务引用之外，还可以直接访问包含协定接口的程序集，使用 ChannelFactory<TChannel> 类实例化连接到服务的信道。

ChannelFactory<TChannel> 类的构造函数接受绑定配置和端点地址这两个参数。绑定参数必须与用服务主机创建的绑定兼容，用 EndpointAddress 类定义的地址引用所运行的服务的 URI。

CreateChannel() 方法创建一条连接到服务的信道。接着调用服务的方法，可以看出，该服务实例包含状态，直到调用指定了 IsTerminating 操作行为的 Close() 方法为止。



```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
    class Program
    {
        static void Main()
        {
            var binding = new WSHttpBinding();
            var address = new EndpointAddress("http://localhost:8731/" +
                "Design_Time_Addresses/StateServiceSample/Service1/");

            var factory = new ChannelFactory<IStateService>(binding, address);

            IStateService channel = factory.CreateChannel();
            channel.Init(1);
            Console.WriteLine(channel.GetState());
            channel.SetState(2);
            Console.WriteLine(channel.GetState());
            channel.Close();

            factory.Close();
        }
    }
}
```

代码段 StateClient/program.cs

在服务的实现代码中，可以通过 OperationBehavior 特性将服务方法应用于如表 43-5 所示的属性。

表 43-5

通过 OperationBehavior 应用的属性	说 明
AutoDisposeParameters	默认情况下，所有可释放的参数都自动释放。如果参数不应释放，就可以把 AutoDisposeParameters 属性设置为 false。接着，发送方将负责释放该参数
Impersonation	使用 Impersonation 属性，可以模拟调用者，以调用者的身份运行方法
ReleaseInstanceMode	InstanceContextMode 使用服务行为设置定义对象实例的生命周期。使用操作行为设置，可以根据操作重写设置。ReleaseInstanceMode 用 ReleaseInstanceMode 枚举定义实例发布模式。其 None 值使用 InstanceContextMode 设置。BeforeCall、AfterCall 和 BeforeAndAfterCall 值定义了操作的循环时间
TransactionScopeRequired	使用 TransactionScopeRequired 属性可以指定操作是否需要一个事务。如果需要 一个事务，且调用者已经发出一个事务，就使用同一个事务。如果调用者没有 发出事务，就创建一个新的事务
TransactionAutoComplete	TransactionAutoComplete 属性指定事务是否自动完成。如果把该属性设置为 true，在抛出异常的情况下就终止事务。如果这是一个根事务，且没有抛出异常，就提交事务

43.4.2 错误处理

默认情况下，在服务中出现的详细的异常消息不返回客户端应用程序。其原因是安全性。不应把详细的异常消息提供给使用服务的第三方。而异常应记录到服务上(为此可以使用跟踪和事件日志功能)，包含有用信息的错误应返回调用者。

可以抛出一个 FaultException 异常，来返回 SOAP 错误。抛出 FaultException 异常会创建一个非类型化的 SOAP 错误。返回错误的首选方式是生成强类型化的 SOAP 错误。

应与强类型化的 SOAP 错误一起传递的信息用数据协定定义，如下面的 StateFault 类所示：



可从
wrox.com
下载源代码

```
[DataContract]
public class StateFault
{
    [DataMember]
    public int BadState { get; set; }
}
```

代码段 StateServiceSample/IStateService.cs

SOAP 错误的类型必须用 FaultContractAttribute 和操作协定定义：

```
[FaultContract(typeof(StateFault))]
[OperationContract]
void SetState(int i);
```

在实现代码中，抛出一个 FaultException<TDetail>异常。在构造函数中，可以指定一个新的 TDetail 对象，在本例中就是 StateFault。另外，FaultReason 中的错误信息可以赋予构造函数。FaultReason 支持多种语言的错误信息。



可从
wrox.com
下载源代码

```
public void SetState(int i)
{
    if (i == -1)
    {
        FaultReasonText[] text = new FaultReasonText[2];
        text[0] = new FaultReasonText("Sample Error",
            new CultureInfo("en"));
        text[1] = new FaultReasonText("Beispiel Fehler",
            new CultureInfo("de"));
        FaultReason reason = new FaultReason(text);

        throw new FaultException<StateFault>(
            new StateFault() { BadState = i }, reason);
    }
    else
    {
        this.i = i;
    }
}
```

代码段 StateServiceSample/StateService.cs

在客户端应用程序中，可以捕获 `FaultException<StateFault>` 类型的异常。出现该异常的原因由 `Message` 属性定义。`StateFault` 用 `Detail` 属性访问。



可从
wrox.com
下载源代码

```
try
{
    channel.SetState(-1);
}
catch (FaultException<StateFault> ex)
{
    Console.WriteLine(ex.Message);
    StateFault detail = ex.Detail;
    Console.WriteLine(detail.BadState);
}
```

代码段 StateServiceClient/Program.cs

除了捕获强类型化的 SOAP 错误之外，客户端应用程序还可以捕获 `FaultException<Detail>` 异常的基类的异常：`FaultException` 异常和 `CommunicationException` 异常。捕获 `CommunicationException` 异常还可以捕获与 WCF 通信相关的其他异常。

43.5 绑定

绑定描述了服务的通信方式。使用绑定可以指定如下特性：

- 传输协议
- 安全性
- 编码格式
- 事务流
- 可靠性
- 形状变化
- 传输升级

绑定包含多个绑定元素，它们描述了所有绑定要求。可以创建自定义绑定，也可以使用表 43-6 中的其中一个预定义绑定：

表 43-6

标 准 绑 定	说 明
BasicHttpBinding	BasicHttpBinding 绑定用于最广泛的交互操作的第一代 Web 服务。所使用的传输协议是 HTTP 或 HTTPS，其安全性仅由协议保证
WSHttpBinding	WSHttpBinding 绑定用于下一代 Web 服务，用 SOAP 扩展确保安全、可靠性和事务处理的平台。所使用的传输协议是 HTTP 或 HTTPS，为了确保安全，实现了 WS-Security 规范的安全性。使用 WS-Coordination、WS-AtomicTransaction 和 WS-BusinessActivity 规范支持事务，通过 WS-ReliableMessaging 的实现方式支持可靠的消息传送。WS-Profile 也支持用于发送附件的 MTOM(Message Transmission Optimization Protocol，消息传输优化协议)编码。WS-*标准的规范可参见 http://www.oasis-open.org
WS2007HttpBinding	WS2007HttpBinding 派生自基类 WSHttpBinding，支持 OASIS(Organization for the Advancement of Structured Information Standards，结构化信息标准促进组织)定义的安全性、可靠性和事务规范
WSHttpContextBinding	WSHttpContextBinding 派生自基类 WSHttpBinding，开始支持没有使用 cookie 的上下文。这个绑定会添加 ContextBindingElement，交换上下文信息
WebHttpBinding	这个绑定用于通过 HTTP 请求(而不是 SOAP 请求)提供的服务，它对于脚本客户端很有用，如 ASP.NET AJAX
WSFederationHttpBinding	WSFederationHttpBinding 是一种安全、可交互操作的绑定，支持在多个系统上共享身份，以进行身份验证和授权
WSDualHttpBinding	与 WSHttpBinding 相反，WSDualHttpBinding 绑定支持双工的消息传送
NetTcpBinding	所有用 Net 作为前缀的标准绑定都使用二进制编码在 .NET 应用程序之间通信。这个编码比 WSxxx 绑定使用的文本编码快。NetTcpBinding 绑定使用 TCP/IP 协议
NetTcpContextBinding	类似于 WSHttpContextBinding，NetTcpContextBinding 会添加 ContextBindingElement，与 SOAP 标题交换上下文信息
NetPeerTcpBinding	NetPeerTcpBinding 为对等通信提供绑定
NetNamedPipeBinding	NetNamedPipeBinding 为同一系统上的不同进程之间的通信进行了优化
NetMsmqBinding	NetMsmqBinding 为 WCF 引入了排队通信。这里消息会发送到消息队列中
MsmqIntegrationBinding	MsmqIntegrationBinding 是用于使用消息队列的已有应用程序的绑定。而 NetMsmqBinding 绑定需要位于客户端和服务器的 WCF 应用程序
CustomBinding	使用 CustomBinding，可以完全定制传输协议和安全要求

不同的绑定支持不同的功能。以 WS 开头的绑定独立于平台，支持 Web 服务规范。以 Net 开头的绑定使用二进制格式，使 .NET 应用程序之间的通信有很高的性能。其他功能支持会话、可靠的会话、事务和双工通信。表 43-7 列出了支持这些功能的绑定。

表 43-7

功 能	绑 定
会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding
可靠的会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding
事务	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetMsmqBinding、MsmqIntegrationBinding
双工通信	WSDualHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetPeerTcpBinding

除了定义绑定之外，服务还必须定义端点。端点依赖于合同、服务的地址和绑定。在下面的代码示例中，实例化了一个 `ServiceHost` 对象，将地址 `http://localhost:8080/RoomReservation`、一个 `WSHttpBinding` 实例和协定添加到服务的一个端点上。

```
static ServiceHost host;

static void StartService()
{
    var baseAddress = new Uri("http://localhost:8080/RoomReservation");
    host = new ServiceHost(
        typeof(RoomReservationService));

    var binding1 = new WSHttpBinding();
    host.AddServiceEndpoint(typeof(IRoomService), binding1, baseAddress);
    host.Open();
}
```

除了以编程方式定义绑定之外，还可以在应用程序配置文件中定义它。WCF 的配置放在 `<system.serviceModel>` 元素中，`<service>` 元素定义了所提供的服务。同样，如代码所示，服务需要一个端点，该端点包含地址、绑定和协定信息。`WSHttpBinding` 的默认绑定配置用 XML 属性 `bindingConfiguration` 修改，该属性引用了绑定配置 `WSHttpConfig1`。这个绑定配置在 `<bindings>` 段中，它用于修改 `WSHttpBinding` 配置，以启用 `reliableSession`。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address="http://localhost:8080/RoomReservation"
          contract="Wrox.ProCSharp.WCF.IRoomService"
          binding="wsHttpBinding" bindingConfiguration="wsHttpBinding" />
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
        <binding name="wsHttpBinding">
          <reliableSession enabled="true" />
        </binding>
      </wsHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

```

        </bindings>
    </system.serviceModel>
</configuration>

```

43.6 宿主

在选择运行服务的宿主时, WCF 非常灵活。宿主可以是 Windows 服务、COM+应用程序、WAS(Windows Activation Services, Windows 激活服务)或 IIS、Windows 应用程序, 或简单的控制台应用程序。在用 Windows 窗体或 WPF 创建自定义主机时, 很容易创建对等的解决方案。

43.6.1 自定义宿主

先从自定义宿主开始。下面的示例代码列出了控制台应用程序中的服务宿主。但在其他自定义主机类型中, 如 Windows 服务或 Windows 应用程序, 可以用相同的方式编写服务。

在 Main()方法中, 创建一个 ServiceHost 实例。之后, 读取应用程序配置文件, 来定义绑定。也可以通过编程方式定义绑定, 如前面所示。接着, 调用 ServiceHost 类的 Open()方法, 使服务接受客户调用。在控制台应用程序中, 必须在关闭服务之前, 不能关闭主线程。这里实际上在调用 Close()方法时, 要求用户“按回车键”, 以结束(退出)服务。

```

using System;
using System.ServiceModel;

public class Program
{
    public static void Main()
    {
        using (var serviceHost = new ServiceHost())
        {
            serviceHost.Open();

            Console.WriteLine("The service started. Press return to exit");
            Console.ReadLine();

            serviceHost.Close();
        }
    }
}

```

要终止服务宿主, 可以调用 ServiceHost 类的 Abort()方法。要获得服务的当前状态, State 属性会返回 CommunicationState 枚举定义的一个值, 该枚举的值有 Created、Opening、Opened、Closing、Closed 和 Faulted。



如果从 Windows 窗体或 WPF 应用程序中启动服务, 该服务的代码调用 Windows 窗体控件的方法, 就必须确保, 只有控件的创建线程才能访问该控件的方法和属性。在 WCF 中, 设置 [ServiceBehavior] 特性的 UseSynchronizationContext 属性, 就可以实现该行为。

43.6.2 WAS 宿主

在 WAS 宿主中, 可以使用 WAS 工作进程中的功能, 如自动激活服务、健康监控和进程。

要使用 WAS 宿主, 只需创建一个 Web 站点和一个 .svc 文件, 其中的 ServiceHost 声明包含服务类的语言和名称。下面的代码使用 Service1 类。另外, 还必须指定包含服务类的文件。这个类的实现方式与前面定义 WCF 服务库的方式相同。

```
<%@ServiceHost language="C#" Service="Service1" CodeBehind="Service1.svc.cs" %>
```

如果使用 WAS 宿主中可用的 WCF 服务库, 就可以创建一个 .svc 文件, 它只包含类的引用:

```
<%@ ServiceHost Service="Wrox.ProCSharp.WCF.Services.RoomReservationService" %>
```

自从引入 Windows Vista 和 Windows Server 2008 以来, WAS 允许定义 .NET TCP 和 Message Queue 绑定。如果使用以前的版本, Windows Server 2003 和 Windows XP 中的 IIS 6 或 IIS 5.1, 就只能使用 HTTP 绑定从 .svc 文件中激活服务。



还可以将 WCF 服务添加到 Enterprise Service 组件中, 详见第 51 章。

43.6.3 预配置的宿主类

为了减少配置的必要性, WCF 还提供了一些带预配置绑定的宿主类。一个例子是 System.ServiceModel.Web 名称空间中的 System.ServiceModel.Web 程序集中的 WebServiceHost 类。如果没有用 WebHttpBinding 配置默认端点, 这个类就为 HTTP 和 HTTPS 基址创建一个默认端点。另外, 如果没有定义另一个行为, 这个类就会添加 WebHttpBehavior。利用这个行为, 可以执行简单的 HTTP GET 操作和 POST、PUT、DELETE (使用 WebInvoke 属性) 操作, 而无需额外的设置。



可从
wrox.com
下载源代码

```
Uri baseAddress = new Uri("http://localhost:8000/RoomReservation");
var host = new WebServiceHost(typeof(RoomReservationService), baseAddress);
host.Open();

Console.WriteLine("service running");
Console.WriteLine("Press return to exit...");
Console.ReadLine();

if (host.State == CommunicationState.Opened)
    host.Close();
```

代码段 RoomReservationWebServiceHost/Program.cs

要使用简单的 HTTP GET 请求接收预定信息, GetRoomReservation() 方法需要一个 WebGet 特性, 把方法参数映射到 GET 请求的输入上。在下面的代码中, 定义一个 UriTemplate, 这需要待添加到基址中的 Reservations 后跟 From 和 To 参数。From 和 To 参数依次映射到 fromDate 和 toDate 变量上。



可从
wrox.com
下载源代码

```
[WebGet(UriTemplate = "Reservations?From={fromDate} & To={toDate}")]
public RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate)
{
    var data = new RoomReservationData();
```

```
return data.GetReservations(fromDate, toDate);
```

代码段 RoomReservationService/RoomReservationService.cs

现在可以使用简单的请求来调用服务了，如下所示。返回给定时间段的所有预定信息。

```
http://localhost:8000/RoomReservation/Reservations?From=2010/1/1&To=2010/8/1
```



System.Data.Services.DataServiceHost 是另一个带预配置特性的类。这个类派生自 **WebServiceHost**，提供了第 32 章讨论的数据服务。

43.7 客户端

客户端应用程序需要一个代理来访问服务。给客户端创建代理有 3 种方式：

- **Visual Studio 添加服务引用**——这个实用程序会从服务的元数据中创建代理类。
- **ServiceModel 元数据实用工具(Svcutil.exe)**——使用 **SvcUtil** 实用程序可以创建代理类。该实用程序从服务中读取元数据，以创建代理类。
- **ChannelFactory 类**——这个类由 **Svcutil** 实用程序生成的代理使用，然而，它也可以用于以编程方式创建代理。

从 Visual Studio 中添加服务引用，需要访问 WSDL 文档。WSDL 文档由 MEX 端点创建，MEX 端点需要用服务配置。在下面的配置中，带相对地址 **mex** 的端点使用 **mexHttpBinding**，并实现 **IMetadataExchange** 协定。为了通过 HTTP GET 请求访问元数据，应把 **behaviorConfiguration** 配置为 **MexServiceBehavior**。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration=" MexServiceBehavior "
        name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address="Test" binding="wsHttpBinding"
          contract="Wrox.ProCSharp.WCF.IRoomService" />
        <endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" />
      </service>
    </services>
    <host>
      <baseAddresses>
        <add baseAddress=
          "http://localhost:8732/Design_Time_Addresses/RoomReservationService/" />
      </baseAddresses>
    </host>
  </configuration>
  <behaviors>
    <serviceBehaviors>
      <behavior name="MexServiceBehavior">
        <!--To avoid disclosing metadata information,
          set the value below to false and remove the metadata endpoint above
          before deployment-->
```

```

    <serviceMetadata httpGetEnabled="True"/>
  </behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

类似于 Visual Studio 中的 Add 服务引用, Svcutil 实用程序需要元数据来创建代理类。Svcutil 实用程序可以从 MEX 元数据端点、程序集的元数据、WSDL 和 XSD 文档中创建代理。

```

svcutil http://localhost:8080/RoomReservation?wsdl /language:C# /out:proxy.cs
svcutil CourseRegistration.dll
svcutil CourseRegistration.wsdl CourseRegistration.xsd

```

生成的代理类后, 它需要从客户端代码中实例化, 需要调用方法, 最后必须调用 Close() 方法:

```

var client = new RoomServiceClient();
client.RegisterForCourse(roomReservation);
client.Close();

```

生成的代理类派生自基类 ClientBase<TChannel>, 该基类封装 ChannelFactory<TChannel> 类。除了使用生成的代理类之外, 还可以直接使用 ChannelFactory<TChannel> 类。构造函数需要绑定和端点地址; 之后, 就可以创建信道, 调用服务协定定义的方法。最后, 必须关闭该工厂。

```

var binding = new WsHttpBinding();
var address = new EndpointAddress("http://localhost:8080/RoomService");

var factory = new ChannelFactory<IStateService>(binding, address);

IRoomService channel = factory.CreateChannel();
channel.ReserveRoom(roomReservation);

//.
factory.Close();

```

ChannelFactory<TChannel> 类有几个属性和方法, 如表 43-8 所示。

表 43-8

ChannelFactory 类的成员	说 明
Credentials	Credentials 是一个只读属性, 可以访问 ClientCredentials 对象, 该对象被赋予信道, 对服务进行身份验证。Credentials 可以用端点来设置
Endpoint	Endpoint 是一个只读属性, 可以访问与信道相关联的 ServiceEndpoint。端点可以在构造函数中指定
State	State 属性的类型是 CommunicationState, 它返回信道的当前状态。CommunicationState 是一个枚举, 其值是 Created、Opening、Opened、Closing、Closed 和 Faulted
Open()	该方法用于打开信道
Close()	该方法用于关闭信道
Opening、Opened、Closing、Closed 和 Faulted	可以指定事件处理程序, 从而确定信道的状态变化。这些事件分别在信道打开前后、信道关闭前后和出错时触发

43.8 双工通信

下面的示例程序说明了如何在客户端和服务之间直接进行双工通信。客户端会启动到服务的连接。之后，服务就可以回调客户端了。

为了进行双工通信，必须指定一个在客户端中实现的协定。这里用于客户端的协定由 `IMyMessageCallback` 接口定义。由客户端实现的方法是 `OnCallback()`。操作应用了 `IsOneWay=true` 操作协定设置。这样，服务就不必等待方法在客户端上成功调用了。在默认情况下，服务实例只能从一个线程中调用(服务行为的 `ConcurrencyMode` 属性默认设置为 `ConcurrencyMode.Single`)。

如果服务的实现代码回调客户端，等待获得客户端的结果，则从客户端中获得回应的线程就必须等待，直到锁定服务对象为止。因为服务对象已经由客户端的请求锁定，所以出现了死锁。WCF 检测到这个死锁，就抛出一个异常。为了避免这种情况，可以将 `ConcurrencyMode` 属性的值改为 `Multiple` 或 `Reentrant`。使用 `Multiple` 设置，多个线程可以同时访问实例。这里必须自己实现锁定。使用 `Reentrant` 设置，服务实例将只使用一个线程，但允许将回调请求的回应重新输入到上下文中。除了改变并发模式之外，还可以用操作协定指定 `IsOneWay` 属性。这样，调用者就不会等待回应了。当然，只有不需要返回值，才能使用这个设置。

服务协定由 `IMyMessage` 接口定义。回调协定用服务协定定义的 `CallbackContract` 属性映射到服务协定上。



可从
wrox.com
下载源代码

```
public interface IMyMessageCallback
{
    [OperationContract(IsOneWay=true)]
    void OnCallback(string message);
}

[ServiceContract(CallbackContract=typeof(IMyMessageCallback))]
public interface IMyMessage
{
    [OperationContract]
    void MessageToServer(string message);
}
```

代码段 `MessageService/IMyMessage.cs`

`MessageService` 类实现了服务协定 `IMyMessage`。服务将来自客户端的消息写入控制台。要访问回调协定，可以使用 `OperationContext` 类。`OperationContext.Current` 返回与客户端中当前请求关联的 `OperationContext`。使用 `OperationContext` 可以访问会话信息、消息标题和属性，在双工通信的情况下还可以访问回调信道。泛型方法 `GetCallbackChannel()` 将信道返回客户端实例。接着调用由回调接口 `IMyMessageCallback` 定义的 `OnCallback()` 方法，可以使用这个信道将消息发送给客户端。为了演示这些操作，还可以从服务中使用独立于方法的完成的回调信道，创建一个接收回调信道的新线程。这个新线程再次使用回调信道，将消息发送给客户。



可从
wrox.com
下载源代码

```
public class MessageService: IMyMessage
{
    public void MessageToServer(string message)
    {
        Console.WriteLine("message from the client: {0}", message);
    }
}
```

```

IMyMessageCallback callback =
    OperationContext.Current.
        GetCallbackChannel<IMyMessageCallback> ();

callback.OnCallback("message from the server");

new Thread(ThreadCallback).Start(callback);
}

private void ThreadCallback(object callback)
{
    IMyMessageCallback messageCallback = callback as IMyMessageCallback;
    for (int i = 0; i < 10; i++)
    {
        messageCallback.OnCallback("message " + i.ToString());
        Thread.Sleep(1000);
    }
}
}

```

代码段 MessageService/MessageService.cs

存放服务的方式与前面的例子相同，这里不再赘述。但是对于双工通信，必须配置一个支持双工通信的绑定。支持双工信道的一个绑定是 `wsDualHttpBinding`，它在应用程序的配置文件中配置。



可从
wrox.com
下载源代码

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.MessageService">
        <endpoint contract="Wrox.ProCSharp.WCF.IMyMessage"
          binding="wsDualHttpBinding"/>
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8732/Service1" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

代码段 MessageService/app.config

在客户端应用程序中，必须用 `ClientCallback` 类实现回调协定，该类实现了 `IMyMessageCallback` 接口，如下所示：



可从
wrox.com
下载源代码

```

class ClientCallback: IMyMessageCallback
{
    public void OnCallback(string message)
    {
        Console.WriteLine("message from the server: {0}", message);
    }
}

```

代码段 MessageClient/Program.cs

在双工信道中,不能像前面那样使用 `ChannelFactory` 启动与服务的连接。要创建双工信道,可以使用 `DuplexChannelFactory` 类。这个类有一个构造函数,除了绑定和地址配置之外,它还有一个参数,这个参数指定 `InstanceContext`,它封装 `ClientCallback` 类的一个实例。把这个实例传递给工厂,服务就可以通过信道调用对象。客户端只需使连接一直处于打开状态。如果关闭连接,服务就不能通过它发送消息。

```
var binding = new WSDualHttpBinding();
var address = new EndpointAddress("http://localhost:8732/Service1");

ClientCallback clientCallback = new ClientCallback();
InstanceContext context = new InstanceContext(clientCallback);

DuplexChannelFactory<IMyMessage> factory =
    new DuplexChannelFactory<IMyMessage>(context, binding, address);

IMyMessage messageChannel = factory.CreateChannel();

messageChannel.MessageToServer("From the client");
```

启动服务主机和客户端应用程序,就可以实现双工通信。

43.9 小结

本章学习了如何使用 Windows Communication Foundation 在客户端和服务端之间通信。WCF 与 ASP.NET Web 服务一样,也独立于平台,但它提供了与 .NET Remoting、Enterprise Services 和消息队列类似的功能。

WCF 主要利用服务协定、数据协定和消息协定,来简化客户端和服务的独立开发,并支持独立的平台。可以使用几个属性定义服务的行为和对应操作。

我们还探讨了如何从服务提供的元数据中创建客户端,如何使用 .NET 接口协定来创建客户端。本章介绍了不同绑定选项的功能。WCF 不仅提供了独立于平台的绑定,还提供了在 .NET 应用程序之间快速通信的绑定。本章还探讨了如何创建自定义主机和如何使用 WAS 主机。如何定义回调接口,应用服务协定,在客户端应用程序中实现回调协定,实现双工通信。

后面几章继续介绍 WCF 的功能。第 44 章讨论 Windows Workflow Foundation,如何使用 WCF 与工作流实例通信。第 46 章解释如何通过 WCF 绑定使用断开连接的消息队列功能,第 47 章学习 WCF 的联合功能,第 51 章(参见光盘)学习如何集成 Enterprise Services 和 WCF。

第 44 章

Windows WF 4

本章内容:

- 可以创建的不同类型的工作流
- 描述一些内置活动
- 如何创建自定义活动
- 工作流概述

本章将概述 Windows Workflow Foundation 4(本章称之为 Workflow 4), 它提供了一个模型, 在该模型中, 可以使用一组构建块(称为活动)定义和执行进程。WF 还提供了一个设计器, 在默认情况下, 该设计器位于 Visual Studio 中, 允许将工具箱中的活动拖放到设计界面上, 从而创建一个工作流模板。

接着, 这个模板就可以以许多不同的方式执行, 本章将介绍这些方式。在工作流执行时, 它需要访问外界, 一般可以使用几个方法来访问外部世界。另外, 工作流也需要保存和还原其状态, 例如, 需要长时间等待时。

工作流由许多活动构成, 这些活动在运行时执行。活动可以发送电子邮件、更新数据库中的一行, 或在后端系统上执行一个事务。有许多内置活动, 它们用于一般性的工作, 也可以创建自己的自定义活动, 根据需要将它们插入工作流中。

在 Visual Studio 2010 中, 实际上有两个版本的 Workflow, 3.x 版本随 .NET Framework 3 一起发布, 版本 4 随 .NET Framework 4 一起发布。本章介绍 Workflow 的最新版本, 前一个版本的信息可参见第 57 章(见光盘)。

这两个版本有许多相同的功能, 但也有许多微妙的差别。如果是第一次使用 Workflow, 那么最好使用最新版本; 但如果已经在使用 3.x 版本, 第 57 章的一些提示将有助于您迁移到 Workflow 4 上。

本章从一个规范的例子 Hello World 开始, 每个人在面对一种新技术时都要使用这个例子。

44.1 Hello World 示例

Visual Studio 2010 包含在 .NET Framework 3.x 和 4 版本中创建工作流的内置支持。打开 New Project 对话框, 会看到一个工作流项目类型列表, 如图 44-1 所示。

确保从版本组合框中选中 .NET Framework 4, 再从可用的模板中选择 Workflow Console Application, 这会构建一个简单的控制台应用程序, 它包含工作流模板和执行这个模板的主程序。

接着, 把 WriteLine 活动从工具箱拖放到设计界面上, 就会得到如图 44-2 所示的工作流。

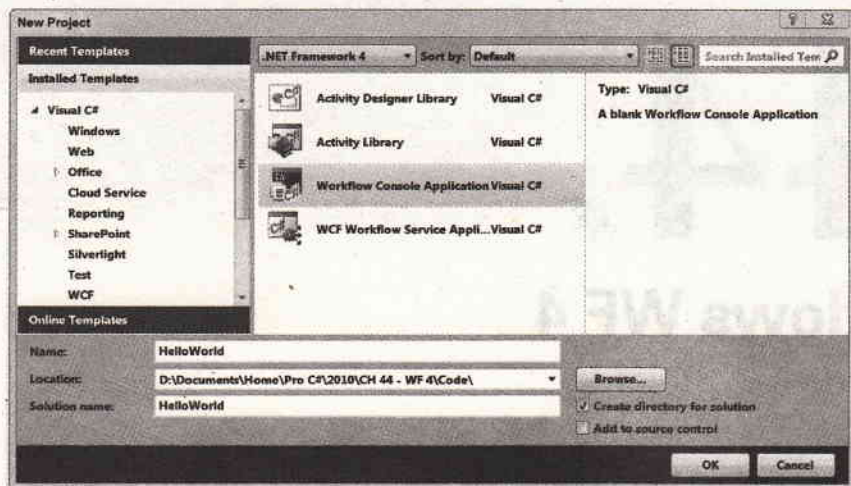


图 44-1

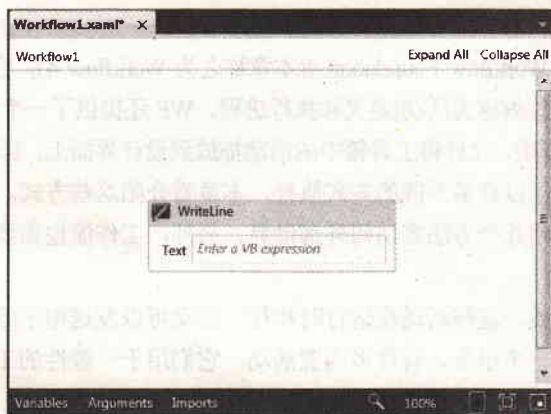


图 44-2

WriteLine 活动包含一个 Text 属性，在设置它时，可以在设计界面上直接输入文本，也可以显示属性网格。本章后面将介绍如何定义自定义活动，以使用这个相同行为。

Text 属性不仅仅是简单的字符串，实际上它定义为实参类型，该参数类型可以把一个表达式作为其源。在运行时计算表达式得到一个结果。这个文本结果会用作 WriteLine 活动的输入。要输入简单的文本表达式，必须使用双引号——如果在 Visual Studio 中按照上面的步骤操作，就在 Text 属性中输入“Hello World”。如果省略了引号，就会得到一个编译错误，因为没有引号，所以它不是一个合法的表达式。

如果构建并运行程序，就会在控制台上看到输出的文本。程序执行时，会在 Main() 方法中创建工作流的一个实例，它使用 WorkflowInvoker 类的一个静态方法来执行该实例。这个示例的代码在 01_HelloWorld 解决方案中。

WorkflowInvoker 类是 Workflow 4 新增的，它允许同步调用工作流。还有另外两个执行工作流的方法可以异步地执行工作流，参见本章后面的内容。在 Workflow 3.x 中也可以进行同步执行，但有时启动比较困难，而且系统开销比较大。

WorkflowInvoker 类的同步功能非常适用于运行短时间的工作流，以响应某些 UI 动作——可以使用工作流启用或禁用 UI 的某些元素。在 **Workflow 3.x** 中这也是可行的，但同步执行给定的工作流实例比较困难。

44.2 活动

工作流中的内容是一个活动，包括工作流本身。工作流是特殊类型的活动，它一般允许在其中定义其他活动，这称为复合活动，本章后面将介绍其他复合活动。活动是一个最终派生自 **Activity** 抽象类的类。

类层次结构比 **Workflow 3.x** 的定义更深，主要类如图 44-3 所示。

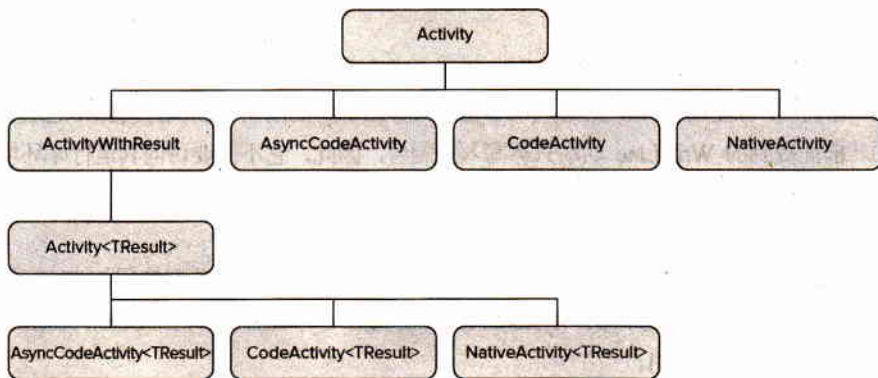


图 44-3

Activity 类是所有工作流活动的根，一般从第二层派生自定义活动。要创建一个简单的活动，如上面提到的 **WriteLine** 活动，应从 **CodeActivity** 中派生，因为这个类有足够的功能可以输出数据行。执行并返回某种形式的结果的活动应派生自 **ActivityWithResult** 类——注意这里最好使用泛型类 **Activity<TResult>**，因为它提供了一个强类型化的 **Result** 属性。

确定从哪个基类中派生是构建自定义活动时的主要问题，本章将通过一些示例来说明如何选择正确的基类。

为了让活动执行某个操作，一般应重写 **Execute()** 方法，它根据所选择的基类有许多不同的签名，这些签名如表 44-1 所示。

表 44-1

基 类	Execute() 方法
AsyncCodeActivity	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) void EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity	void Execute (CodeActivityContext)
NativeActivity	void Execute (NativeActivityContext)

(续表)

基 类	Execute() 方法
AsyncCodeActivity<TResult>	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) TResult EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity<TResult>	TResult Execute (CodeActivityContext)
NativeActivity<TResult>	void Execute (NativeActivityContext)

注意，传送给 Execute()方法的参数不同，因为它使用了特定于类型的执行上下文参数。在 Workflow 3.x 中，只使用了一个 ActivityExecutionContext 类，而在 Workflow 4 中，可以为不同类别的活动使用不同的上下文。

主要区别是，与 NativeActivityContext 相比，CodeActivityContext 和派生的 AsyncCodeActivityContext 的功能有限。这说明，派生自 CodeActivity 和 AsyncCodeActivity 的活动可以执行的操作远远少于它们的容器。例如，前面提到的 WriteLine 活动只需写入控制台，因此，它不需要访问其运行库环境。更复杂的活动需要调度其他子活动，或与其他系统通信，此时就需要从 NativeActivity 中派生，以访问完整的运行库。在创建自己的自定义活动时，会再次探讨这个主题。

WF 提供了许多标准活动，下面几节将介绍其中一些活动的示例和使用这些活动的场合。Workflow 4 使用 3 个主要的程序集：System.Activities.dll、System.Activities.Core.Presentation.dll 和 System.Activities.Presentation.dll。

44.2.1 If 活动

顾名思义，这个活动的操作类似于 C#中的 If-Else 语句。

把一个 if 活动拖放到设计界面上时，就会看到一个标志符号，如图 44-4 所示。If 是一个复合活动，它包含两个子活动，把两个子活动拖放到屏幕的 Then 和 Else 部分。



图 44-4

图 44-4 显示的 If 活动也包含一个图标，表示需要定义 Condition 属性。在执行活动时判断这个条件，如果它返回 true，就执行 Then 分支；否则执行 Else 分支。

因为 Condition 属性是一个表达式，它等于一个布尔值，所以可以在这里包含任意有效的表达式。

Workflow 4 包含一个使用 Visual Basic 语法的表达式引擎，这对于 C#程序员似乎很奇怪，因为 Visual Basic 与 C#大不相同。但因为目前是这样，所以要使用内置活动，必须学习 Visual Basic。只要记住 Visual Basic 的语法要非常详细，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，可以定义传递到工作流中的实参，然后就可以在 If 活动中由表达式计算该实参。本章后面会讨论实参和变量。

44.2.2 InvokeMethod 活动

这是最有用的活动之一，它允许执行已有的代码，有效地把这些代码封装在工作流的执行语义中。

我们一般有许多预先已有的代码，这个活动允许直接从工作流中调用这些代码。

使用 `InvokeMethod` 调用代码有两种方式，使用哪种方式取决于调用静态方法还是实例方法。如果要调用静态方法，就需要定义 `TargetType` 和 `MethodName` 参数。但如果调用实例方法，就使用 `TargetObject` 和 `MethodName` 属性，在这个实例中，`TargetObject` 可以内联创建，它也可以是在工作流的某个地方定义的变量。02_ParallelExecution 示例中的代码显示了使用 `InvokeMethod` 活动的两种模式。

如果需要给调用的方法传递参数，就可以使用 `Parameters` 集合定义它们。集合中参数的顺序必须匹配方法中参数的顺序。另外，还有一个 `Result` 属性，它设置为函数调用的返回值。可以在工作流中把它绑定到一个变量上，以恰当地使用该值。

44.2.3 Parallel 活动

`Parallel` 活动名不符实，因为初看起来，可能认为这个活动似乎在多处理器计算机上以并行方式调度其子活动，实际上并非如此，但一些特殊情况除外。

把一个 `Parallel` 活动拖放到设计界面上后，就可以拖放其他子活动，如图 44-5 所示。



图 44-5

这些子活动可以是单一的活动，如图 44-5 所示，它们也可以来自一个复合活动，如 `Sequence` 或另一个 `Parallel` 活动。

在运行期间，`Parallel` 活动会调度每个直接的子活动，以执行它们。底层的运行库执行引擎会以 FIFO(先进先出)方式调度这些子活动，因此提供了并行执行的假象，但它们仅运行在一个线程上。

为了包含真正的并行执行方式，拖放到 `Parallel` 活动中的活动必须派生自 `AsyncCodeActivity` 类。02_ParallelExecution 中的示例代码包含一个例子，它演示了如何在 `Parallel` 活动的两个分支中异步地处理代码。图 44-6 显示了一个 `Parallel` 活动中使用两个 `InvokeMethod` 活动的情况。

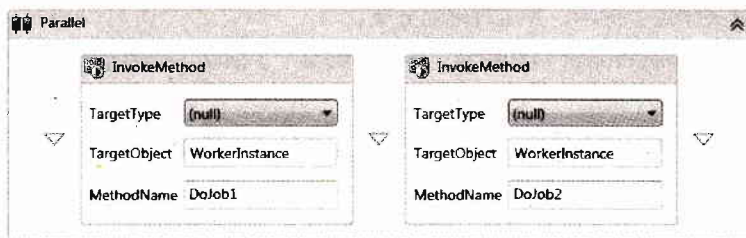


图 44-6

这里使用的 `InvokeMethod` 活动调用了两个简单的方法 `DoJob1` 和 `DoJob2`，它们分别休眠 2 分钟和 3 分钟。为了异步地运行这些方法，需要进行最后一个修改。`InvokeMethod` 活动的布尔属性 `RunAsynchronously` 默认为 `False`。在 UI 中把这个属性设置为 `True`，就会异步地调用目标方法，因此

允许 Parallel 活动同时执行多个活动。在单处理器的计算机中,会执行两个线程,造成了同时执行的假象。而在多处理器的计算机上,这些线程可能在不同的内核上调度,所以提供了真正的并行执行方式。如果创建自己的活动,就应把它们创建为异步活动,这样最终用户就可以获得并行执行的好处。

44.2.4 Delay 活动

业务进程常常需要等待一段时间才能完成——考虑使用 workflow 进行费用申请的过程。workflow 给经理发送一封电子邮件,要求他批准某个费用申请。之后 workflow 进入等待状态,等待直接经理批准(或者不批准),这里最好定义一个超时时间。如果在 1 天时间内没有返回响应,费用申请就路由给命令链中的下一个经理。

Delay 活动可以实现这个情形的一部分(另一个部分是下一节定义的 Pick 活动),其任务是等待预定义的时间,之后继续执行 workflow。

Delay 活动包含一个 Duration 属性,它可以设置为一个离散的 TimeSpan 值,但因为该属性定义为一个表达式,所以其值可以在 workflow 中链接到一个变量上,或者根据需要从其他值中计算出来。

执行 workflow 时,workflow 会进入 Idle 状态,在这个状态下,workflow 会运行 Delay 活动。空闲的 workflow 将等待持久化——此时把 workflow 实例数据存储在永久介质中(如 SQL Server 数据库),workflow 本身可以从内存中卸载。这会节省系统资源,因为在任意给定时刻,内存中都只需要正在运行的 workflow。任何延迟的 workflow 会持久化到磁盘上。

44.2.5 Pick 活动

一个常见的编程结构是等待一组可能的事件中的某个事件,如 System.Threading 名称空间中 WaitHandle 类的 WaitAny() 方法。Pick 活动是 workflow 中等待事件发生的一种方式,因为它可以定义任意多个分支,每个分支在运行之前都可以等待一个触发器动作的发生。引发触发器后,就会执行 workflow 中的其他活动。

作为一个具体的示例,考虑上一节的费用申请 workflow 过程。这里的 Pick 活动有 3 个分支:第一个分支处理所接受的申请,第二个分支处理被拒绝的申请,第三个分支处理超时情况。

这个例子的代码在下载代码的 03_PickDemo 文件夹下。它包含一个示例 workflow,该 workflow 由一个 Pick 活动和 3 个分支组成。在运行该 workflow 时,会提示用户接受或拒绝申请。如果过了 10 秒钟或更长时间,它就关闭这个提示,并运行延迟分支。

在这个例子中,DisplayPrompt 活动用作 workflow 中的第一个活动,它会调用在接口上定义的方法,该接口会提示经理批准或不批准——因为这个功能定义为接口,所以该提示可以是电子邮件、IM 消息或用其他方式通知经理,需要处理一个费用申请。之后,workflow 执行 Pick 活动,等待这个外部接口的输入(批准或不批准),同时也在等待某个延迟。

在执行 Pick 活动时,它实际上会将一个等待操作放在每条分支的第一个活动中。在触发一个事件时,会取消其他所有等待事件,并处理在其中触发了事件的其他分支。所以,在批准了费用报告的情况下,WaitForAccept 活动就完成了,接着下一个动作是输出一条配置消息。但如果经理没有批准该费用申请,WaitForReject 活动就完成了,然后输出一条拒绝消息。

最后,如果 WaitForAccept 和 WaitForReject 活动都没有完成,WaitForTimeout 活动就会在延迟时间过后完成,并且费用报告可以路由给下一个经理——可能在 Active Directory 中查找这个人。在

本示例中, 执行 `DisplayPrompt` 活动时, 会给用户显示一个对话框。所以如果执行延迟活动, 则还需要关闭这个对话框, 而这就是图 44-7 中 `ClosePrompt` 活动的作用。

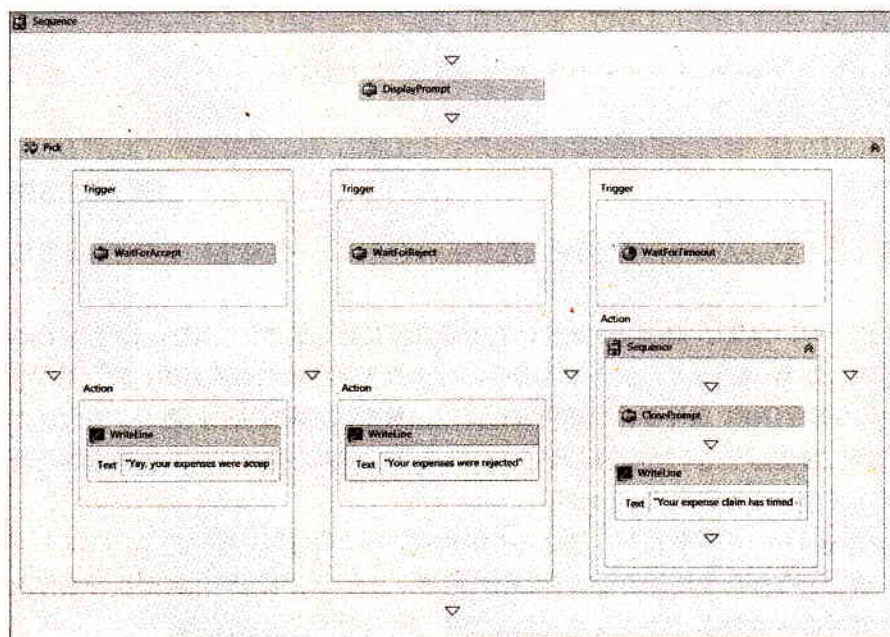


图 44-7

该例使用了一些前面没有介绍的概念, 如如何编写自定义活动或等待外部事件, 这些主题将在本章后面探讨。

44.3 自定义活动

前面使用的都是在 `System.Activities` 名称空间中定义的活动。本节将学习如何创建自定义活动, 扩展它们, 在设计时和运行时为用户提供更好体验。

首先, 创建 `DebugWrite` 活动, 在调试版本中将一行文本输出到控制台上。这是一个简单的示例, 后面将扩展它, 使用这个示例介绍自定义活动的所有可用选项。在创建自定义活动时, 可以仅在工作流项目中构建一个类, 但最好在一个独立的程序集中构建自定义活动, 因为 Visual Studio 的设计环境(特别是和工作流项目)会从程序集中加载活动, 并能在试图更新程序集时锁定它。所以, 应创建一个简单的类库项目, 在其中构建自定义活动。本例的代码在 `04_CustomActivities` 项目中。

简单的活动, 如 `DebugWrite` 活动, 直接派生自 `CodeActivity` 基类。下面的代码构建一个活动类, 并定义一个 `Message` 属性, 在执行活动时会显示该属性:



可从
wrox.com
下载源代码

```
using System;
using System.Activities;
using System.Diagnostics;

namespace Activities
{
    public class DebugWrite : CodeActivity
    {
```

```

[Description("The message output to the debug stream")]
public InArgument<string> Message { get; set; }

protected override void Execute(CodeActivityContext context)
{
    Debug.WriteLine(Message.Get(context));
}

```

代码下载 04 CustomActivities

在调度并执行 `CodeActivity` 时，会调用其 `Execute()` 方法——活动实际上应在这个方法中执行一些操作。

在这个例子中，定义了 `Message` 属性，它看起来像是普通的 .NET 属性，但它在 `Execute()` 方法中的用法不同。在 `Workflow 4` 中的许多改变中，一个改变是存储状态数据的位置。在 `Workflow 3.x` 中，常常使用标准的 .NET 属性并在活动内部存储活动数据。这种方法的问题是，因为这个存储器对工作流运行库引擎实际上是不透明的，所以为了持久化工作流，必须在所有已构建的活动上实现二进制持久性，才能忠实地还原它们的数据。

在 `Workflow 4` 中，所有数据都存储在各个活动的外部。所以这里的模型是，应从上下文中获得实参的值，给上下文提供要设置的实参的新值。这样，工作流引擎就可以跟踪执行工作流时状态的变化，且仅存储在持久性点之间有变化的值，而不是存储整个工作流数据。

在 `Message` 属性上定义的 `[Description]` 特性将在 Visual Studio 的属性网格中使用，以提供该特性的额外信息，如图 44-8 所示。

显然，这个活动肯定是可以使用的，但是，有几个地方应该调整从而使这个活动更友好。与本章前面的 `Pick` 活动一样，它有一些强制的属性，若没有定义，就会在设计界面上生成错误。为了使活动有相同的行为，需要扩展代码。



图 44-8

44.3.1 活动的验证

把活动放在设计界面上时，`Designer` 就会在两个地方查找验证信息。最简单的验证形式是给参数属性添加一个 `[RequiredArgument]` 特性。如果没有定义该参数，就在活动名的右边显示一个感叹号标志符号，如图 44-9 所示。



图 44-9

如果把鼠标悬停在感叹号上，就会显示一个工具提示“没有给活动的必选参数‘`Message`’提供值”。因为这是一个编译错误，所以需要为这个参数定义一个值，之后就可以执行应用程序了。

如果有相关的多个属性，就可以重写 `CacheMetadata()` 方法，以添加一些额外的验证代码。在执行活动之前调用这个方法，在其中可以检查所定义的必选参数，还可以给传递过来的参数选择性的添加额外的元数据。也可以在传递给 `CacheMetadata()` 方法的 `CodeActivityMetadata` 对象上调用 `AddValidationError()` 方法的一个重写版本，从而添加额外的验证错误(或警告)。

既然完成了活动的验证后,接下来就要修改活动的呈现行为,当前的 Designer 提供了这个用户体验,而且这个功能会使活动变得更有趣。

44.3.2 设计器

当活动呈现在屏幕上时,一般会把一个设计器关联到活动上。设计器的工作是为活动提供屏幕上的表示形式,在 Workflow 4 中,这种表示形式放在 XAML 中。如果以前没有使用过 XAML,在继续之前应阅读第 35 章。

活动在设计期间的体验一般在一个独立于活动的程序集中创建,因为这个设计体验在运行期间是不需要的。Visual Studio 包含一个 Activity Designer Library 项目类型,这是一个理想的起点,因为在使用这个模板创建项目时,会得到一个默认的活动设计器,接着就可以根据需要修改它。

在设计器的 XAML 中,可以提供任何内容,包括动画。在用于用户界面时,少通常意味着多,建议查看先前已有的活动,了解应提供什么内容合适。

首先,创建一个简单的设计器,把它关联到 DebugWrite 活动上。下面的代码显示了在项目中添加活动设计器(或构建新的活动设计器库项目)时创建的模板。这些代码也在 04_CustomActivities 解决方案中。



```
<sap:ActivityDesigner x:Class="Activities.Presentation.DebugWriteDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr - namespace:System.Activities.Presentation;
        assembly=System.Activities.Presentation"
    xmlns:sapv="clr - namespace:System.Activities.Presentation.View;
        assembly=System.Activities.Presentation">

    <Grid>

    </Grid>

</sap:ActivityDesigner>
```

代码下载 04_CustomActivities

所创建的 XAML 仅构建了一个网格,还包含一些导入的名称空间,本例的活动需要这些名称空间。显然,因为在模板中没有什么内容,所以首先添加用于定义消息的标签和文本框:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBlock Text="Message" Margin="0,0,5,0"/>
    <TextBox Text="{Binding Path=ModelItem.Message, Mode=TwoWay}"
        Grid.Column="1"/>
</Grid>
```

这些 XAML 在活动的 Message 属性和文本框之间构建了一个绑定。在设计器的 XAML 中,总是可以使用 ModelItem 引用,来引用正在设计的活动。

为了把上述定义的设计器和 DebugWrite 活动关联起来,还需要修改活动,添加 Designer 属性(也可以实现 IRegisterMetadata 接口,但本章没有进一步介绍这种方式):

```
[Designer("Activities.Presentation.DebugWriteDesigner, Activities.Presentation")]
```

```
public class DebugWrite : CodeActivity
{
    ...
}
```

这里使用[Designer]特性定义设计器和活动之间的链接。最好使用这个特性的字符串版本，因为这样可以确保在活动程序集内部不引用设计程序集。

现在，在 Visual Studio 中使用 DebugWrite 活动的一个实例时，会得到如图 44-10 所示的结果。



图 44-10

但其问题是 Message 属性——它没有显示在属性网格中定义的值，如果尝试在文本框中输入它的值，就会接收到一个异常。原因是这表示我们试图把一个简单的文本值绑定到 `InArgument<string>` 类型上，为了使这个绑定成功，需要使用 Workflow 4 中的另外一对内置类 `ExpressionTextBox` 和 `ArgumentToExpressionConverter`。现在，设计器的完整 XAML 如下所示。添加或修改的代码行用粗体表示。



可从
wrox.com
下载源代码

```
<sap:ActivityDesigner x:Class="Activities.Presentation.DebugWriteDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr-namespace:System.Activities.Presentation;
        assembly=System.Activities.Presentation"
    xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
        assembly=System.Activities.Presentation"
    xmlns:sadc="clr-namespace:System.Activities.Presentation.Converters;
        assembly=System.Activities.Presentation"
>
  <sap:ActivityDesigner.Resources>
    <sadc:ArgumentToExpressionConverter x:Key="argConverter"/>
  </sap:ActivityDesigner.Resources>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBlock Text="Message" Margin="0,0,5,0" />
    <sapv:ExpressionTextBox Grid.Column="1"
      Expression="{Binding Path=ModelItem.Message, Mode=TwoWay,
        Converter={StaticResource argConverter},
        ConverterParameter=In}"
      OwnerActivity="{Binding ModelItem}"/>
  </Grid>
</sap:ActivityDesigner>
```

代码下载 04_CustomActivitiesFirst

文件包含一个新的名称空间 `System.Activities.Presentation.View`，其中包含的转换器可用于在屏幕上的表达式和活动的 Message 属性之间进行转换。这里是 `ArgumentToExpressionConverter`，把它添加到 XAML 文件的资源中。

接着用 `ExpressionTextBox` 替换标准的 `TextBox` 控件。除了简单的文本之外，这个控件还允许用户输入表达式，这样 DebugWrite 活动就可以包含一个表达式，该表达式合并了正在运行的工作流中

的许多值，而不仅仅是简单的文本字符串。进行了这些修改后，DebugWrite 活动就比较类似于内置活动。

44.3.3 自定义复合活动

活动的一个常见需求是创建复合活动，即包含其他子活动的活动。例如，前面的 Pick 活动和 Parallel 活动。复合活动的执行完全由程序员负责，例如可以执行一个随机活动，它仅调度自身的其中一个子活动，或者根据当前日期是星期几调度绕过某些子活动的一个活动。最简单的执行模式是执行所有子活动，但开发人员可以确定如何执行子活动，以及活动何时完成。

创建的第一种类型的复合活动是“重试”活动。我们常常会尝试一个操作，如果它失败，就重试若干次。这个活动的伪代码如下所示：

```
int iterationCount = 0;
bool looping = true;
while ( looping )
{
    try
    {
        // Execute the activity here
        looping = false;
    }
    catch (Exception ex)
    {
        iterationCount += 1;
        if ( iterationCount >= maxRetries )
            rethrow;
    }
}
```

我们会作为一个活动复制上面的代码，再把要执行的活动插入注释所在的位置。这些工作会在自定义活动的 Execute() 方法中进行。还有另一种方式——可以使用其他活动编写整个活动。此时需要创建一个自定义活动，其中包含一个“洞”，最终用户可以把要重试的活动放在这里，再设置最多重试次数对应的一个属性。下面的代码演示了这个过程：



可从
wrox.com
下载源代码

```
public class Retry : Activity
{
    public Activity Body { get; set; }

    [RequiredArgument]
    public InArgument<int> NumberOfRetries { get; set; }

    public Retry()
    {
        Variable<int> iterationCount =
            new Variable<int> ( "iterationCount", 0 );
        Variable<bool> looping = new Variable<bool> ( "looping", true );

        this.Implementation = () =>
        {
            return new While
            {
                Variables = { iterationCount, looping },
            }
        }
    }
}
```

第VI部分 通信

代码下载 04 CustomActivities

首先定义 Activity 类型的 Body 属性——这是在重试循环中执行的活动。接着定义 RetryCount

属性，它用于指定尝试操作的次数。

这个自定义活动直接派生自 `Activity` 类，它把实现代码提供一个函数。执行包含这个活动的工作流时，实际上它会执行该函数，该函数提供了一个类似于前面伪代码定义的运行时执行路径。在构造函数中，创建活动使用的局部变量，再构建一组匹配伪代码的活动。这个示例的代码也在 04_CustomActivities 解决方案中。

有了上面的代码，还可以推断出，也可以创建没有 XAML 的工作流——没有设计体验(即，不能拖放活动，以生成代码)，但是，如果我们只需要代码，就没有理由不使用它来替代 XAML。

既然有自定义复合活动，就还需要定义一个设计器。这里需要一个活动，它包含一个可拖放另一个活动的占位符。如果查看其他标准活动，就会发现有几个体现类似行为的标准活动，如 `If` 和 `Pick` 活动。理想情况下，因为自定义活动的工作方式应与内置活动类似，所以现在看看它们的实现方式。

如果使用 `Reflector` 查看工作流库，就会发现其中缺乏设计器的 XAML。这是因为它作为一组资源编译到程序集中。使用 `Reflector` 可以查看这些资源，但首先需要下载 BAML 查看器插件。因为 BAML 查看器会反编译二进制的 BAML 格式，并生成文本，所以可以使用这些文本查看标准活动使用的 XAML。要查找 BAML 查看器，可在线搜索 `Reflector BAML viewer`，很快就会找到它。

把插件加载到 `Reflector` 中时，会加载 `System.Activities.Presentation` 程序集，再单击 `Tools` 菜单上的 `BAML Viewer` 菜单项，这会显示当前加载的程序集中所有 BAML 资源对应的一个列表，接着就可以查看对应的示例，从而找到一些示例 XAML。

我使用这个方法了解用于内置活动的 XAML，这有助于构建如图 44-11 所示的 `Retry` 活动示例。

这个活动的关键是 `WorkflowItemPresenter` 类，它在 XAML 中用于定义子活动的占位符，其定义如下：

```
<sap:WorkflowItemPresenter IsDefaultContainer="True"
    AllowedItemType="{x:Type sa:Activity}"
    HintText="Drop an activity here" MinWidth="100" MinHeight="60"
    Item="{Binding Path=ModelItem.Body, Mode=TwoWay}"
    Grid.Column="1" Grid.Row="1" Margin="2">
```

把这个控件绑定到 `Retry` 活动的 `Body` 属性上，`HintText` 定义了没有给控件添加子活动时显示的帮助文本。XAML 还包含一些样式，这些样式用于显示设计器的展开或折叠版本——这确保自定义活动的工作方式与内置活动相同。这个示例的所有代码和 XAML 都在 04_CustomActivities 解决方案中。

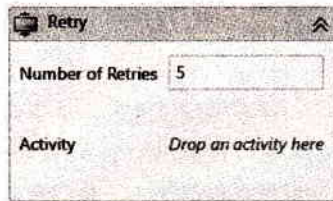


图 44-11

44.4 工作流

到目前为止，本章主要探讨了活动，但没有讨论工作流。工作流只是一个活动列表，实际上工作流本身是活动的另一个类型。使用这个模型可以简化运行库引擎，因为运行库引擎只需知道如何执行一种对象——派生自 `Activity` 类的任何对象。

前面介绍了 `WorkflowExecutor` 类，它可以用于执行工作流。但如前所述，这只是执行工作流的一种方式，执行工作流有 3 个不同的选项，每个选项都有不同的功能。在学习执行工作流的其他方

式之前，先探讨一下实参和变量。

44.4.1 实参和变量

工作流可以看作程序，任何编程语言的其中一方面是可以创建变量，把实参传入传出程序。自然，Workflow 4 也支持这些结构，本节将介绍如何定义实参和变量。

首先，假定工作流处理一个保险单，所以要传递给工作流的一个实参是保险单 ID。为了定义工作流的实参，需要进入 Designer，单击左下角的 Arguments 按钮，这会列出为工作流定义的实参列表，如图 44-12 所示，也可以在这里添加自己的实参。

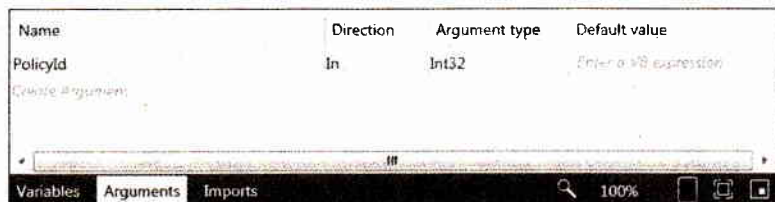


图 44-12

要定义实参，需要指定实参的 Name、Direction(可以是 In、Out 或 InOut)和数据类型。还可以选择指定默认值，如果没有提供实参，就使用默认值。

实参的方向用于确定该实参是用作工作流的输入还是输出，如果使用 InOut 方向，该实参就用作工作流的输入和输出。

本章第一节描述了如何使用 WorkflowInvoker 类执行工作流。Invoke()方法有几个重写版本，可它们用于给工作流传递实参。它们作为一个名称/值对字典传递，其中名称必须精确匹配实参的名称，这个匹配区分大小写。下面的代码用于给工作流传递一个 PolicyId 值——这个示例的代码在 05_ArgsAndVars 解决方案中。



可从
wrox.com
下载源代码

```
Dictionary<string, object> parms = new Dictionary<string, object> ();
parms.Add("PolicyId", 123);
WorkflowInvoker.Invoke(new PolicyFlow(), parms);
```

代码下载 05_ArgsAndVars

接着调用工作流，从字典中把 PolicyId 传递给指定的参数。如果在字典中提供的名称没有对应的实参，就抛出一个 ArgumentException 异常。相反，如果没有给 In 实参提供值，就不抛出异常。这好像是错误的——我们希望对于任何未定义的 In 实参抛出一个 ArgumentException 异常，而如果传递了过多的实参，则不抛出异常。

工作流完成时，可能希望检索输出实参。为此，WorkflowInvoker.Invoke()方法有一个特定的重写版本，它返回一个字典。这个字典只包含 Out 或 InOut 实参。

在工作流中，可能希望定义变量。在 Workflow 3.x 的 XAML 工作流中，这可不容易实现，但在 Workflow 4 中，已经解决了这个问题，可以轻松地在 XAML 中定义形参。

与任何编程语言一样，工作流变量也有作用域。在工作流的根活动中定义变量，就可以定义全局变量，这些变量可用于工作流中的所有活动，它们的生命周期与工作流的生命周期联系在一起。

还可以在各个活动中定义变量，此时这些变量就只能用于定义该变量的活动，以及该活动的子

活动。一旦该活动完成了，其变量就超出了作用域，且不再能访问了。

44.4.2 WorkflowApplication

尽管 `WorkflowInvoker` 类对于同步执行工作流很有用，但我们可能需要长时间运行的工作流，它们可以持久化到数据库中，在将来的某个时刻需要提取出来。此时应使用 `WorkflowApplication` 类。

`WorkflowApplication` 类类似于 `Workflow 3` 中存在的 `WorkflowRuntime` 类，在 `Workflow 3` 允许运行工作流，也允许响应在该工作流实例上发生的事件。可以使用 `WorkflowApplication` 类编写的最简单的程序如下所示：



可从
wrox.com
下载源代码

```
WorkflowApplication app = new WorkflowApplication(new Workflow1());

ManualResetEvent finished = new ManualResetEvent(false);

app.Completed = (completedArgs) => { finished.Set(); };

app.Run();

finished.WaitOne();
```

代码下载 06 WorkflowApplication

这个程序构造一个工作流应用程序实例，并关联到该实例的 `Completed` 委托上，以设置一个手工重置的事件。调用 `Run()` 方法启动工作流的执行，最后代码等待触发事件。

这揭示了 `WorkflowExecutor` 和 `WorkflowApplication` 之间的一个主要区别——后者是异步的。调用 `Run()` 方法时，系统会使用线程池中的一个线程执行工作流，而不使用主调线程。因此，需要某种形式的同步，才能确保包含工作流的应用程序在工作流完成之后退出。

长时间运行的工作流一般有许多休眠阶段——大多数工作流的执行行为可以描述为间歇执行的阶段。在工作流的开头一般要做某项工作，接着等待某个输入或延迟，接收到这个输入后，就进行处理，再进入下一个等待状态。

所以，工作流休眠时，最好从内存中卸载它，再在事件触发工作流继续执行时，再重新加载工作流。为此，需要给 `WorkflowApplication` 类添加一个 `InstanceStore` 对象，再对前面的代码进行其他一些小的修改。在架构中，抽象类 `InstanceStore` 有一个实现方式 `SqlWorkflowInstanceStore`。为了使用这个类，首先需要有一个数据库，其脚本默认位于 `C:\Windows\Microsoft.NET\Framework\v4.0.21006\SQL\en` 目录下。注意版本号会改变。

在这个目录下有许多 SQL 文件，我们需要的两个是 `SqlWorkflowInstanceStoreSchema.sql` 和 `SqlWorkflowInstanceStoreLogic.sql`。可以对已有的数据库运行这两个文件，也可以根据需要创建全新的数据库。还可以使用完整的 SQL Server 安装或 SQL Express 安装。

一旦有了数据库，就需要对宿主代码进行一些修改。首先，需要构造 `SqlWorkflowInstanceStore` 的一个实例，再把它添加到工作流应用程序中：



可从
wrox.com
下载源代码

```
SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore
(ConfigurationManager.ConnectionStrings["db"].ConnectionString);

AutoResetEvent finished = new AutoResetEvent(false);

WorkflowApplication app = new WorkflowApplication(new Workflow1());
app.Completed = (e) => { finished.Set(); };
```

```

app.PersistableIdle = (e) => { return PersistableIdleAction.Unload; };
app.InstanceStore = store;

app.Run();

finished.WaitOne();

```

代码下载 06 WorkflowApplication

粗体的代码行是添加到前面示例中的新代码。另外注意，在工作流应用程序中还给 **PersistableIdle** 委托添加了一个事件处理程序。执行工作流时，它会运行尽可能多的活动，直到没有工作可做为止。此时，工作流就会转换到 **Idle** 状态，空闲的工作流是持久化的备用工作流。**PersistableIdle** 委托用于确定应对空闲的工作流进行什么操作。默认什么都不做，然而，也可以指定 **PersistableIdleAction.Persist**，它会接受工作流的一个副本，并把它存储在数据库中，但工作流仍位于内存中，还可以指定 **PersistableIdleAction.Unload**，它会持久化工作流，之后卸载工作流。

也可以使用 **Persist** 活动请求持久化工作流，但如果通过调用 **NativeActivityContext** 的 **RequestPersist()** 方法，使自定义活动派生自 **NativeActivity**，自定义活动编写器还可以请求持久化工作流。

现在有一个问题：可以从内存中卸载工作流，并把它存储在持久性存储器中，但如何从存储器中检索工作流，再次执行它？

1. 书签

书签的传统用法是标记图书中的一页，以便从同一个地方开始继续阅读。在工作流的上下文中，书签指定了一个位置，以便从这个位置开始继续运行工作流，等待外部输入时，一般会使用书签。

例如，我们可能要等待应用程序处理保险报价。最终用户在线生成一个报价单，显然，存在与这个报价单相关联的一个工作流。因为报价单可能仅在 30 天内有效，所以在 30 天后应使该报价失效。同样，也可能请求无索赔的证明，如果该证明没有在指定时间内提供，就取消保险单。于是，这个工作流有许多执行阶段，在其他时间，工作流会处于休眠状态，此时就可以从内存中卸载工作流。但在卸载之前，需要在工作流中定义一点，可以从该点处恢复处理，此时就应使用书签。

要定义书签，需要一个派生自 **NativeActivity** 的自定义活动。接着在 **Execute()** 方法中创建一个书签，在恢复该书签时，就继续执行代码。下面的示例活动定义一个简单的 **Task** 活动，该活动创建一个书签，在书签处恢复执行后，活动就完成了。



可从
wrox.com
下载源代码

```

public class Task : NativeActivity<Boolean>
{
    [RequiredArgument]
    public InArgument<string> TaskName { get; set; }

    protected override bool CanInduceIdle
    {
        get { return true; }
    }

    protected override void Execute(NativeActivityContext context)
    {
        context.CreateBookmark(TaskName.Get(context),
            new BookmarkCallback(OnTaskComplete));
    }
}

```

```

private void OnTaskComplete(NativeActivityContext context,
    Bookmark bookmark, object state)
{
    bool taskOK = Convert.ToBoolean(state);

    this.Result.Set(context, taskOK);
}
}

```

代码下载 06 WorkflowApplication

对 `CreateBookmark` 的调用传递了书签名和一个回调函数。这个回调函数在恢复书签时执行。给回调函数传递了一个任意对象，在本例中是一个布尔值，因为每个任务都应报告其成功或失败，所以使用这个布尔值可以确定工作流的后续步骤。可以给工作流传递任意对象，它可以是带许多字段的复杂类型。

这就是编写好的活动。现在需要修改宿主代码，以便在书签处恢复执行。但这有另一个问题：宿主代码如何知道工作流创建了一个书签？如果主机应从书签处恢复执行，主机就需要知道存在一个书签。

上面创建的 `Task` 活动需要多做一些工作，告诉外界已经创建了一个任务。在产品系统中，这一般导致在队列表中存储一项，这个队列会对调用中心人员显示为一个作业列表。

与宿主的通信参见下一节。

2. 扩展

扩展仅是添加到工作流应用程序的运行环境中的一个类或接口。在 `Workflow 3.x` 中，这些扩展称为服务，但因为该服务与 `WCF` 服务冲突，所以在 `Workflow 4` 中，把它们重命名为扩展。

一般给扩展定义一个接口，再提供该接口在运行期间的实现代码。自定义活动只需调用该接口，并允许根据需要修改该实现代码。扩展的一个例子是发送电子邮件。可以创建一个 `SendEmail` 活动，它在其 `Execute()` 方法中调用扩展，再定义一个基于 `SMTP` 的电子邮件扩展或基于 `Exchange` 的 `Outlook` 扩展，以便在运行期间实际发送电子邮件。自定义活动不需要改为使用任意电子邮件提供程序，只需修改应用程序配置文件，插入一个新的电子邮件提供程序即可。

对于 `task` 示例，需要一个扩展，在 `Task` 活动在其书签处等待时获得通知。这可以把书签名和其他相关信息写入一个数据库中，这样任务队列就可以显示给用户。使用下面的接口定义这个扩展：

```

public interface ITaskExtension
{
    void ExecuteTask(string taskName);
}

```

接着，就可以修改 `Execute()` 方法，以更新 `Task` 活动，从而通知任务扩展，它正在执行：

```

protected override void Execute(NativeActivityContext context)
{
    context.CreateBookmark(TaskName.Get(context),
        new BookmarkCallback(OnTaskComplete));
    context.GetExtension<ITaskExtension>().
        ExecuteTask(TaskName.Get(context));
}

```

在传递给 `Execute()` 方法的 `context` 对象中查询 `ITaskExtension` 接口, 接着代码调用 `ExecuteTask()` 方法。因为 `WorkflowApplication` 维护着一个扩展集合, 所以可以创建一个实现这个扩展接口的类, 该类可用于维护任务列表。然后构建并执行一个新的工作流, 每个任务都通知该扩展执行它的时间。另一个进程会查看任务列表, 并显示给最终用户。

为了使示例代码简单一些, 我们只创建了一个工作流实例。这个实例包含一个 `Task` 活动和后面的一个 `If` 活动, 根据用户接受或拒绝任务来输出相应的消息。

3. 组合在一起

现在可以运行、持久化和卸载工作流, 通过书签把事件发送给工作流。最后一部分是重新加载工作流。使用 `WorkflowApplication` 时, 可以调用 `Load`, 并传递工作流的唯一 ID。每个工作流都有唯一的 ID, 调用 `ID` 属性可以从 `WorkflowApplication` 对象中检索该 ID。所以在伪代码中, 包含工作流的应用程序如下所示:

```
WorkflowApplication app = BuildApplication();
Guid id = app.Id;
app.Run();
// Wait for a while until a task is created, then reload the workflow
app = BuildApplication();
app.Load(id);
app.ResumeBookmark()
```

所提供的示例代码比前面的代码复杂一些, 因为它还包含 `ITaskExtension` 接口的实现代码, 但代码遵循前面的模式。注意对 `BuildApplication()` 方法的两个调用。它们在代码中用于构建一个 `WorkflowApplication` 实例, 并设置所有需要的属性, 例如, `InstanceStore` 以及用于 `Completed` 和 `PersistableIdle` 的委托。在第一个调用之后, 执行 `Run()` 方法, 这会开始执行工作流的一个新实例。

因为第二次加载应用程序是在一个持久性点后, 所以在这个持久性点之前卸载了工作流, 于是, 应用程序实例也删除了。于是我们构建一个新的 `WorkflowApplication` 实例, 但不是调用 `Run()` 方法, 而调用 `Load()`, 它使用持久性提供程序从数据库中加载已有的实例。调用 `ResumeBookmark()` 函数来恢复执行这个实例。

如果运行该示例, 就会在屏幕上看到一个提示。尽管这个提示显示在屏幕上, 但会持久化并卸载工作流。运行 `SQL Server Management Studio`, 执行如图 44-13 所示的命令, 就可以验证这一点。

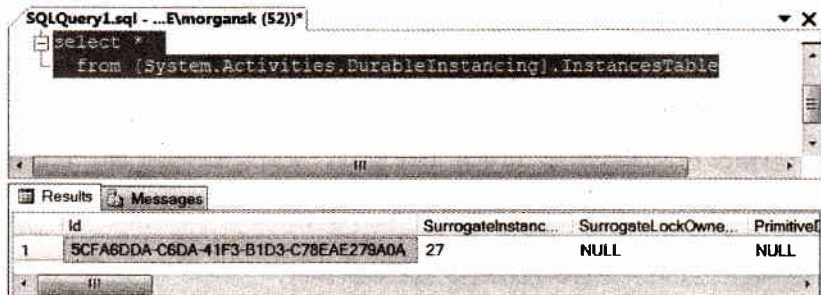


图 44-13

工作流实例存储在 `System.Activities.DurableInstancing` 模式的 `InstancesTable` 中。如图 44-13 所示

的项是运行在某台计算机上的工作流的持久化实例。

继续执行这个工作流，它最终会执行完毕，此时工作流会从实例表中删除，因为关于实例存储提供了一个 `InstanceCompletionAction` 选项，它默认设置为 `DeleteAll`。这将确保一旦工作流完成，就删除在数据库中为给定的工作流实例存储的任何数据。这是一个合理的默认值，因为一旦工作流完成，通常就不应保留该工作流的任何数据。把工作流实例的完成动作设置为 `DeleteNothing`，就可以在定义实例存储时改变这个选项。

如果现在继续运行测试应用程序，重试图 44-13 中的 SQL 命令，就会发现工作流实例已删除。

44.4.3 WorkflowServiceHost

本章前面提到，存放工作流有 3 种方式，最后一种是使用 `WorkflowServiceHost` 类，它通过 WCF 来提供工作流。工作流的一个主要使用场合是用作 WCF 服务的后端。考虑一下典型 WCF 服务执行的操作，它通常以某种顺序调用一组相关的方法。这里的主要问题是任意顺序调用这些方法，而且通常需要定义该顺序，例如，订单信息就不会在下订单之前上传。

使用工作流很容易提供那些也需要考虑方法调用顺序的 WCF 服务。这里使用的主要类是 `Receive` 和 `Send` 活动。在这个示例的代码中(在 `07_WorkflowsAsServices` 解决方案中)，所使用的场景是房地产代理(在美国是房地产经纪人)，他希望给某网站上传属性信息。

对服务的第一次调用是构建一个新的工作流实例，这个调用会返回一个唯一的 ID，在服务的后续调用中将使用它，因为它会标识我们要与服务器上的哪个工作流实例通信。工作流使用“关联”功能，把传入的请求映射到一个工作流实例上。这里要使用基于消息的关联，把传入的消息的一个属性链接到已有的工作流实例上。本例使用的工作流如图 44-14 所示。

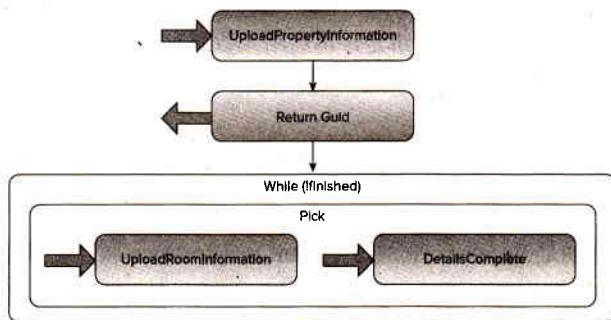


图 44-14

1. 定义 WorkflowServiceHost

工作流从 `Receive` 活动开始，该活动定义了 `UploadPropertyInformation` 的初始服务操作。从服务调用中检索的数据存储在工作流中，接着通过 `Send` 活动把一个 `Guid` 返回给调用者。这个 `Guid` 唯一地定义这个工作流实例，在其他操作中回调该实例时使用该实例。

一旦完成这两个初始的活动，工作流就会运行一个 `While` 活动，在该活动中有一个带两个分支的 `Pick` 活动。第一个分支有一个带 `UploadRoomInformation` 操作的 `Receive` 活动，第二个分支有一个带 `DetailsComplete` 操作的 `Receive` 活动。

下面的代码段用于存放工作流。它利用 `WorkflowServiceHost` 类，该类非常类似于 WCF 中的 `ServiceHost` 类。



可从
wrox.com
下载源代码

```
string baseAddress = "http://localhost:8080/PropertyService";

using (WorkflowServiceHost host =
    new WorkflowServiceHost(GetPropertyWorkflow(), new Uri(baseAddress)))
{
    host.AddServiceEndpoint(XName.Get("IProperty", ns),
        new BasicHttpBinding(), baseAddress);

    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
    smb.HttpGetEnabled = true;
    host.Description.Behaviors.Add(smb);
    try
    {
        host.Open();

        Console.WriteLine("Service host is open for business...");
        Console.ReadLine();
    }
    finally
    {
        host.Close();
    }
}
```

代码下载 07 WorkflowsAsServices

这段代码构建 `WorkflowServiceHost` 类的一个实例，它提供的工作流由 `GetPropertyWorkflow()` 方法调用提供。还可以使用一个基于代码的工作流，就像本例这样，或者使用基于 XAML 的工作流。然后添加一个端点，指定一个基本的 HTTP 绑定，再启用元数据发布功能，这样客户端就可以指向这个端点上的 `svcutil.exe`，以下载服务定义。之后，它就可以在服务主机类上调用 `Open()` 方法，准备接收引入的调用。

2. 确保可以创建工作流

工作流中的第一个活动是 `Receive`，它通过把 `CanCreateInstance` 属性设置为 `True` 来定义。这很重要，因为它通知主机，如果调用者调用了这个操作，就根据该调用构建一个新的工作流实例。在后台，`WorkflowServiceHost` 会探测工作流定义来查找 `Receive` 活动，然后根据这些活动构建一个服务协定，它还会查找把 `CanCreateInstance` 设置为 `True` 的任何类似活动，因为它知道在接收到消息时该做什么。

示例代码包含的客户端应用程序通过调用 `UploadPropertyInformation` 来创建一个新的工作流实例，它接着使用 `UploadRoomInformation` 操作上传 3 个房间的信息，最后它调用 `DetailsComplete`。这会设置 `While` 活动使用的 `Finished` 标记，从而完成这个工作流。

工作流代码相当复杂，因为在后台做了许多工作。但因为了解它如何工作很有意义，所以下面解释其中的一些代码。

工作流中的第一个活动定义如下：

```
Variable<string> address = new Variable<string> ();
Variable<string> owner = new Variable<string> ();
```

```

Variable<double> askingPrice = new Variable<double> ();

// Initial receive - this kicks off the workflow
Receive receive = new Receive
{
    CanCreateInstance = true,
    OperationName = "UploadPropertyInformation",
    ServiceContractName = XName.Get("IProperty", ns),
    Content = new ReceiveParametersContent
    {
        Parameters =
        {
            {"address", new OutArgument<string> (address)},
            {"owner", new OutArgument<string> (owner)},
            {"askingPrice", new OutArgument<double> (askingPrice)}
        }
    }
};

```

这个 **Receive** 活动把 **CanCreateInstance** 标记设置为 **true**，并定义操作协定名和服务协定名。**ServiceContractName** 包含一个名称空间(在本例中设置为 **http://pro-csharp/**)，以便唯一地标识这个服务。**Content** 属性指定在调用该操作时，从客户端传递什么数据，在本例中传递地址、拥有者，并询问价格。该服务操作完全通过代码定义——导出的元数据使用这些代码构建操作定义。

从这个操作中接收的实参绑定到工作流定义的变量上。这是把外部实参传递给 WCF 包含的工作流的方式。

在工作流中，创建唯一的 **ID**，在回调这个工作流实例时，会使用这个 **ID**。为此，需要定义一个工作流变量并赋值。在下面的代码中，使用 **Assign** 活动来完成这个操作。

```

return new Sequence
{
    Variables = { propertyId, operationHandle, finished,
                  address, owner, askingPrice },
    Activities =
    {
        receive,
        new WriteLine { Text = "Assigning a unique ID" },
        new Assign<Guid>
        {
            To = new OutArgument<Guid> (propertyId),
            Value = new InArgument<Guid> (Guid.NewGuid())
        },
        new SendReply
        {
            Request = receive,
            Content = SendContent.Create
                (new InArgument<Guid> (env => propertyId.Get(env))),
            CorrelationInitializers =
            {
                new QueryCorrelationInitializer
                {
                    CorrelationHandle = operationHandle,
                    MessageQuerySet = extractGuid
                }
            }
        }
    }
};

```

```

    }
    }
    // Other activities omitted for clarity
};

```

接着,使用 **SendReply** 活动,通过工作流定义的 **Guid** 来响应调用者。这部分相当复杂。

SendReply 活动把其 **Request** 属性设置为接收活动实例,以链接到最初的 **Receive** 活动上。该操作的返回值由 **Content** 属性定义。可以从操作中返回 3 种响应:第 1 种是一个离散值,如这里使用的 **Guid**,第 2 种是通过一个名称/值对字典来定义的一组参数,第 3 种是一个消息类型。**SendContent** 类是构建对应对象的辅助类。

最后一部分最复杂。**CorrelationInitializers** 属性用于从当前消息中提取关联标记,并在工作流的后续调用中使用,以唯一地标识工作流。**MessageQuerySet** 用于进行提取,对于这条输出消息,其定义如下。一旦定义了查询,就通过一个句柄值链接。这个句柄由任何其他希望位于同一个关联组的活动来使用。

```

MessageQuerySet extractGuid = new MessageQuerySet
{
    { "PropertyId",
      new XPathMessageQuery ( "sm:body()/ser:guid", messageContext ) }
};

```

这个查询使用 **XPath** (也称为 **voodoo 魔术**)从消息中提取要发送回客户端的数据。这里读取 **body** 元素,并在其中查找消息中 **Guid** 元素的值。

3. 通过 **Pick** 活动等待事件

前面已经构建了一个新的工作流实例,并利用一个唯一的操作 **ID** 来回应调用者。下一部分等待来自客户端的更多数据,这通过另外两个 **Receive** 活动完成。第一个 **Receive** 活动对应于 **UploadRoomInformation** 操作。

```

Variable<string> roomName = new Variable<string>();
Variable<double> width = new Variable<double>();
Variable<double> depth = new Variable<double>();

// Receive room information
Receive receiveRoomInfo = new Receive
{
    OperationName = "UploadRoomInformation",
    ServiceContractName = XName.Get("IProperty", ns),
    CorrelatesWith = operationHandle,
    CorrelatesOn = extractGuidFromUploadRoomInformation,
    Content = new ReceiveParametersContent
    {
        Parameters =
        {
            {"propertyId", new OutArgument<Guid>()},
            {"roomName", new OutArgument<string>(roomName)},
            {"width", new OutArgument<double>(width)},
            {"depth", new OutArgument<double>(depth)},

```

```
};
```

最初定义的变量用于记录从客户端传递给工作流的数据。Receive 活动又指定了 OperationName 和 ServiceContractName, 以便从这个操作中生成元数据。在这里 CorrelatesWith 值很重要, 因为它把工作流的不同操作链接在一起。在 SendReply 活动中, 创建这个句柄, 这样就可以在以后的操作中引用它。如果没有这个句柄, 就不可能确定传入的消息用于哪个工作流实例。

CorrelatesOn 属性指定从传入的消息中提取什么数据, 以便验证该消息是用于这个工作流实例的。这里又一次使用了 MessageQuerySet, 这次是从传入的消息中提取唯一 ID:

```
MessageQuerySet extractGuidFromUploadRoomInformation =
    new MessageQuerySet
    {
        { "PropertyId",
          new XPathMessageQuery
          ( @"sm:body()/local:UploadRoomInformation/local:propertyId",
            messageContext ) }
    };
```

这里 XPath 在 UploadRoomInformation 调用中查找 propertyId 元素。

这个 Receive 活动的最后一部分使用传递给服务操作的输入参数确定执行什么操作。这里的数据是唯一的 propertyId、房间名和维度大小。这些数据提取到变量中, 在示例代码中仅把这些数据输出到控制台上。在实际应用程序中, 显然可以使用这些数据更新数据库或执行类似的操作。

最后的 Receive 活动用于等待来自客户端的 DetailsComplete 消息, 它遵循与其他接收活动类似的模式, 因为它在其中定义了服务协定名和操作名, 所使用的关联句柄与最初用于 UploadRoomInformation 操作的 Send 活动和 Receive 活动相同, 接着使用 MessageQuerySet 从引入的消息中检索唯一的属性 ID。

有了这些活动, 工作流剩下的操作是一个 While 循环, 其中包含 Pick 活动及其分支。

```
new While
{
    Condition = ExpressionServices.Convert<bool>
        (env => !finished.Get(env)),
    Body = new Pick
    {
        Branches =
        {
            new PickBranch
            {
                Variables = { roomName, width, depth },
                Trigger = receiveRoomInfo,
                Action = new WriteLine { Text = "Room Info Received" },
            },
            new PickBranch
            {
                Trigger = receiveDetailsComplete,
                Action = new Sequence
                {
                    Activities =
                    {
```

```

new Assign<bool>
{
    To = new OutArgument<bool> (finished),
    Value = new InArgument<bool> (true)
},
new WriteLine { Text = "Finished" }

```

这里 While 循环定义了一个 Condition, 以判断 finished 变量的值。因为把 Condition 属性定义为一个 Activity<bool>, 所以使用 ExpressionServices 辅助类把变量表达式转换为一个活动。

Pick 活动包含两个分支, 我们需要等待每个分支的 Trigger 活动, 才能执行 Action 活动。构建工作流的代码有点长, 但与使用基于 XAML 的工作流相比, 使用 Workflow 4 更容易看出发生了什么。

44.4.4 驻留设计器

我们常常希望把最好的东西留到最后。不要打破常规, 这就是本章所做的工作。Visual Studio 中使用的工作流设计器也可以驻留在自己的应用程序中, 允许最终用户创建自己的工作流, 而无需查看 Visual Studio 的副本。这是 Workflow 4 目前的最佳特性。传统的应用程序扩展机制总是需要某种开发人员——或者编写扩展 DLL 再把它插入系统的某个地方; 或者编写宏或脚本。Windows Workflow 允许最终用户仅把活动拖放到设计界面上, 来定制应用程序。

在 Workflow 3.x 中重新驻留设计器比较麻烦, 但在 Workflow 4 中它就非常容易。因为设计器本身是一个 WPF 控件, 所以我们使用 WPF 项目作为主应用程序。这个示例的代码在 08_DesignerRehosting 项目中。

首先需要包含工作流的程序集, 接着定义主窗口的 XAML。在构建 WPF 用户界面时, 总是使用 MVVM(Model-View-ViewModel, 模型-视图-视图模型)模式, 因为它简化了编码, 并允许在需要时用不同的 XAML 覆盖相同的视图模型。主窗口的 XAML 如下所示:



```

<Window x:Class="HostApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*/>
        </Grid.RowDefinitions>
        <Menu IsMainMenu="True">
            <MenuItem Header="_File">
                <MenuItem Header="_New" Command="{Binding New}"/>
                <MenuItem Header="_Open" Command="{Binding Open}"/>
                <MenuItem Header="_Save" Command="{Binding Save}"/>
            <Separator/>
            <MenuItem Header="_Exit" Command="{Binding Exit}"/>
        </MenuItem>
        <MenuItem Header="Workflow">

```

```

        <MenuItem Header="_Run" Command="{Binding Run}"/>
    </MenuItem>
</Menu>
<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="**"/>
        <ColumnDefinition Width="4**"/>
        <ColumnDefinition Width="**"/>
    </Grid.ColumnDefinitions>
    <ContentControl Content="{Binding Toolbox}" />
    <ContentControl Content="{Binding DesignerView}"
        Grid.Column="1"/>
    <ContentControl Content="{Binding PropertyInspectorView}"
        Grid.Column="2"/>
</Grid>
</Grid>
</Window>

```

代码下载 08 DesignerRehosting

主窗口的布局相当简单，再使用一个网格来定义用于工具箱、设计器和属性表的占位符。注意所有对象都是绑定的，包括命令在内。

前面创建的 ViewModel 包含用于每个主 UI 元素(工具箱、设计器和属性网格)的属性，除了这些属性之外，还有用于每条命令的属性，如 New、Save 和 Exit。

```

public class ViewModel : BaseViewModel
{
    public ViewModel()
    {
        // Ensure all designers are registered for inbuilt activities
        new DesignerMetadata().Register();
    }

    public void InitializeViewModel(Activity root)
    {
        _designer = new WorkflowDesigner();
        _designer.Load(root);

        this.OnPropertyChanged("DesignerView");
        this.OnPropertyChanged("PropertyInspectorView");
    }

    public UIElement DesignerView
    {
        get { return _designer.View; }
    }

    public UIElement PropertyInspectorView
    {
        get { return _designer.PropertyInspectorView; }
    }

    private WorkflowDesigner _designer;
}

```

首先，ViewModel 类派生自 BaseViewModel，每次构建视图模型时，都会使用这个基类，因为

它提供了 `INotifyPropertyChanged` 的实现方式。它来自于 Josh Twist 编写的一组代码段，可以从 www.thejoyofcode.com 上下载。

构造函数确保注册所有内置活动的元数据。没有这个构造函数的调用，就不会在用户界面上显示任何类型的特定设计器。在 `InitializeViewModel()` 方法中，构建 workflow 设计器的一个实例，并给它加载一个活动。`WorkflowDesigner` 类比较奇怪，因为一旦给它加载了一个 workflow，就不能加载另一个 workflow 了，所以只要创建新 workflow，就需要重新创建这个类。

`InitializeViewModel()` 方法的最后一个操作是调用属性更改通知函数，告诉用户界面，`DesignerView` 和 `PropertyInspectorView` 都更新了。由于 UI 绑定到这些属性上，因此它们是必选的，会从新的 workflow 设计器实例中加载新值。

用户界面中下一个要创建的部分是工具箱。在 `Workflow 3.x` 中，必须自己构建这个控件，而在 `Workflow 4` 中有一个 `ToolboxControl`，它使用起来非常容易。

```
public UIElement Toolbox
{
    get
    {
        if (null == _toolbox)
        {
            _toolbox = new ToolboxControl();

            ToolboxCategory cat = new ToolboxCategory
                ("Standard Activities");
            cat.Add(new ToolboxItemWrapper(typeof(Sequence),
                "Sequence"));
            cat.Add(new ToolboxItemWrapper(typeof(Assign), "Assign"));
            _toolbox.Categories.Add(cat);

            ToolboxCategory custom = new ToolboxCategory
                ("Custom Activities");
            custom.Add(new ToolboxItemWrapper(typeof(Message),
                "MessageBox"));
            _toolbox.Categories.Add(custom);
        }

        return _toolbox;
    }
}
```

这里构建了工具箱控件，接着在第一个类别中添加两个文本框项，在第二个类别中添加一个工具箱项。`ToolboxItemWrapper` 类用于简化给工具箱添加给定活动所需的代码。

有了这些代码，就有了一个正在运行的应用程序。现在所需做的全部工作是把 `ViewModel` 关联到 XAML 上。这在主窗口的构造函数中完成。

```
public MainWindow()
{
    InitializeComponent();

    ViewModel vm = new ViewModel();
    CH044.indd 1334 1/28/10 2:17:40 PM
    vm.InitializeViewModel(new Sequence());
}
```

```

    this.DataContext = vm;
}

```

这里构建了视图模型，并添加到默认的 Sequence 活动中。这样，在应用程序运行时，就会在屏幕上显示一些内容。图 44-15 显示了此时运行应用程序时屏幕的外观。

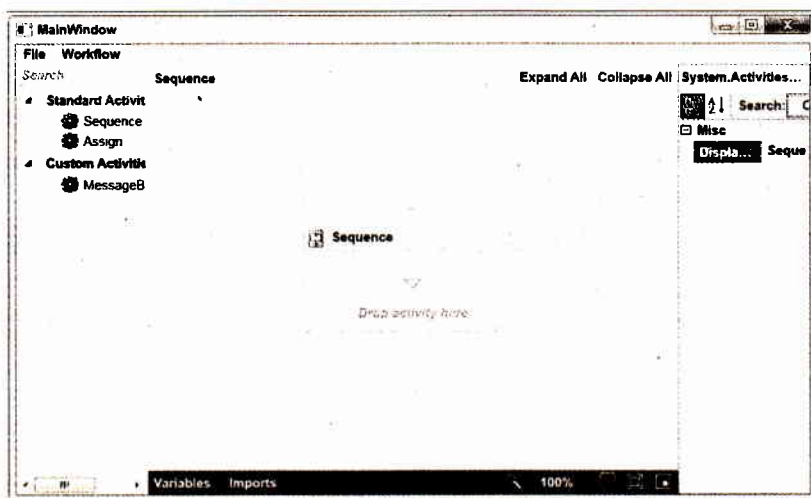


图 44-15

现在，唯一遗漏的部分是一些命令。我们使用 `DelegateCommand` 类给 WPF 编写基于 `ICommand` 的命令，这样视图模型中的代码比较容易理解。命令的实现非常简单，如下面的 `New` 命令所示：

```

public ICommand New
{
    get
    {
        return new DelegateCommand(used =>
        {
            InitializeViewModel(new Sequence());
        });
    }
}

```

因为把这条命令绑定到 `New` 菜单项上，所以单击这条命令执行该委托时，在这种情况下就仅通过一个新的 `Sequence` 活动调用 `InitializeViewModel()` 方法。因为这个方法也会触发设计器和属性网格的属性更改通知，所以设计器和属性网格也会更新。

Open 命令有点复杂：

```

public ICommand Open
{
    get
    {
        return new DelegateCommand(used =>
        {
            OpenFileDialog ofn = new OpenFileDialog();
            ofn.Title = "Open Workflow";
            ofn.Filter = "Workflows (*.xaml)|*.xaml";
            ofn.CheckFileExists = true;

```

```

        ofn.CheckPathExists = true;

        if (true == ofn.ShowDialog())
            InitializeViewModel(ofn.FileName);
    });
}
}

```

这里使用了 `InitializeViewModel()` 方法的另一个重写版本，在这个例子中，该重写版本的参数是文件名，而不是活动。这里没有列出这些代码，但可以从代码下载中获得代码。这条命令会显示一个 `OpenFileDialog`，选择这条命令时，它会把工作流加载到设计器中。对应的 `Save` 命令会调用 `WorkflowDesigner.Save()` 方法，把工作流的 XAML 存储到磁盘上。

视图模型中的最后一段代码是 `Run` 命令。如果不能执行这些代码，就不会有设计精良的工作流，所以在视图模型中也包含这个功能。它很简单——设计器包含一个 `Text` 属性，它是工作流中活动的 XAML 表示。我们只需把它转换为一个 `Activity`，再使用 `WorkflowInvoker` 类执行它即可。

```

public ICommand Run
{
    get
    {
        return new DelegateCommand(unused =>
        {
            Activity root = _designer.Context.Services.
                GetService<ModelService> ().Root.
                GetCurrentValue() as Activity;

            WorkflowInvoker.Invoke(root);
        },
        unused => { return !HasErrors; }
        );
    }
}

public bool HasErrors
{
    get { return (0 != _errorCount); }
}

public void ShowValidationErrors(IList<ValidationErrorInfo> errors)
{
    _errorCount = errors.Count;
    OnPropertyChanged("HasErrors");
}

private int _errorCount;

```

必须调整上述代码，以便放在页面上，因为从设计器中检索根活动的委托命令的第一行代码太长。接着只需使用 `WorkflowInvoker.Invoke()` 方法执行工作流。

WPF 中的命令结构包含一种禁用不能访问的命令的方式，它就是关于 `DelegateCommand` 的第二个 `Lambda` 函数。这个函数返回 `HasErrors` 的值，`HasErrors` 是一个已添加到视图模型中的布尔属性。这个属性表示在工作流中是否找到验证错误，因为视图模型实现 `IVValidationErrorService`，只要工作流的有效状态改变了，`IVValidationErrorService` 就会得到通知。