

图 33-8

除了仅处理根元素之外，还可以把名称空间应用于所有元素，如下例所示：



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XNamespace ns1 = "http://www.lippperweb.com/ns/root";
            XNamespace ns2 = "http://www.lippperweb.com/ns/sub";

            XElement xe = new XElement(ns1 + "Company",
                new XElement(ns2 + "CompanyName", "Lipper"),
                new XElement(ns2 + "CompanyAddress",
                    new XElement(ns2 + "Address", "123 Main Street"),
                    new XElement(ns2 + "City", "St. Louis"),
                    new XElement(ns2 + "State", "MO"),
                    new XElement(ns2 + "Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.xml

这会生成如图 33-9 所示的结果。

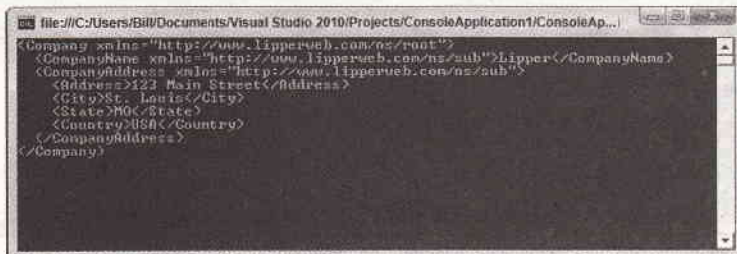


图 33-9

在这个例子中，子名称空间应用于指定的所有对象，但<Address>、<City>、<State>和<Country>元素除外，因为它们继承自其父对象<CompanyAddress>，而<CompanyAddress>有名称空间声明。

33.10.4 XComment 对象

XComment 对象可以轻松地吧 XML 注释添加到 XML 文档中。下面的例子说明了如何把一条注释添加到文档的开头:



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = new XDocument();

            XComment xc = new XComment("Here is a comment.");

            xdoc.Add(xc);

            XElement xe = new XElement("Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XComment("Here is another comment."),
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));
            xdoc.Add(xe);

            Console.WriteLine(xdoc.ToString());

            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.sln

这个例子把包含两条 XML 注释的 XDocument 对象写到控制台中, 其中一条注释写到文档的开头, 另一条注释写到<CompanyAddress>元素内部, 其结果如图 33-10 所示。

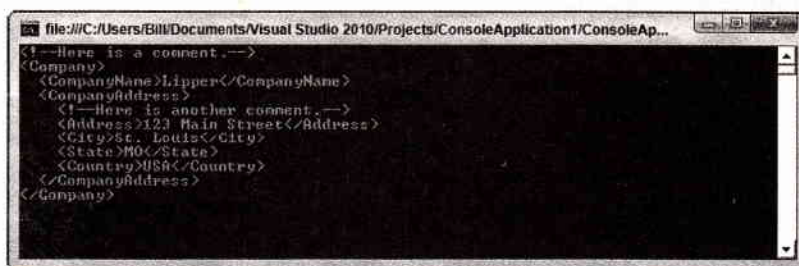


图 33-10

33.10.5 XAttribute 对象

除了元素之外, XML 的另一个要素是特性。通过 XAttribute 对象添加和使用特性。下面的例子

说明了给根节点<Customers>添加一个特性的过程:

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XElement xe = new XElement("Company",
                new XAttribute("MyAttribute", "MyAttributeValue"),
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}
```

这里把 MyAttribute 特性及 MyAttributeValue 的值添加到 XML 文档的根元素中, 结果如图 33-11 所示。

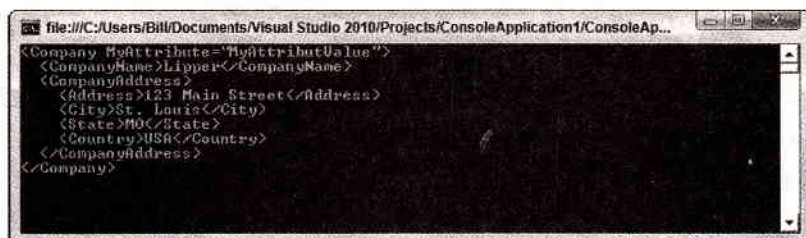


图 33-11

33.11 使用 LINQ 查询 XML 文档

现在可以把 XML 文档放在 XDocument 对象中, 操作这个文档的各个部分。还可以使用 LINQ to XML 查询 XML 文档, 操作其结果。

33.11.1 查询静态的 XML 文档

使用 LINQ to XML 查询静态的 XML 文档几乎不需要做任何工作。下面的例子就使用 hamlet.xml 文件和查询获得戏剧中的所有演员。每位演员都在 XML 文档中用<PERSONA>元素定义:



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            var query = from people in xdoc.Descendants("PERSONA")
                        select people.Value;

            Console.WriteLine("{0} Players Found", query.Count());
            Console.WriteLine();

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

            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.sln

在这个例子中，XDocument 对象加载了一个物理 XML 文件 hamlet.xml，对文档的内容执行一个 LINQ 查询：

```
var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;
```

people 对象表示在文档中找到的所有<PERSONA>元素。接着 select 语句获取这些元素的值。之后，使用 Console.WriteLine() 方法输出通过 query.Count() 方法找到的所有演员的总数。再在 foreach 循环中把每一项写到屏幕上。结果如下所示：

```
26 Players Found

CLAUDIUS, king of Denmark.
HAMLET, son to the late king, and nephew to the present king.
POLONIUS, lord chamberlain.
HORATIO, friend to Hamlet.
LAERTES, son to Polonius.
LUCIANUS, nephew to the king.
VOLTIMAND
CORNELIUS
ROSENCRANTZ
GUILDENSTERN
OSRIC
A Gentleman
A Priest.
MARCELLUS
BERNARDO
FRANCISCO, a soldier.
```

REYNALDO, servant to Polonius.
 Players.
 Two Clowns, grave-diggers.
 FORTINBRAS, prince of Norway.
 A Captain.
 English Ambassadors.
 GERTRUDE, queen of Denmark, and mother to Hamlet.
 OPHELIA, daughter to Polonius.
 Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other Attendants.
 Ghost of Hamlet's Father.

33.11.2 查询动态的 XML 文档

目前, Internet 上有许多动态的 XML 文档。给指定的 URL 端点发送一个请求, 就会找到博客种子、播客种子等许多提供 XML 文档的内容。这些种子可以在浏览器上查看, 或者通过 RSS 聚合器查看, 或用作纯粹的 XML。下面的示例说明了如何直接从代码中使用 RSS 种子:



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc =
                XDocument.Load(@"http://geekswithblogs.net/evjen/Rss.aspx");

            var query = from rssFeed in xdoc.Descendants("channel")
                        select new
                        {
                            Title = rssFeed.Element("title").Value,
                            Description = rssFeed.Element("description").Value,
                            Link = rssFeed.Element("link").Value,
                        };

            foreach (var item in query)
            {
                Console.WriteLine("TITLE: " + item.Title);
                Console.WriteLine("DESCRIPTION: " + item.Description);
                Console.WriteLine("LINK: " + item.Link);
            }

            Console.WriteLine();

            var queryPosts = from myPosts in xdoc.Descendants("item")
                            select new
                            {
                                Title = myPosts.Element("title").Value,
                                Published =
                                    DateTime.Parse(
                                        myPosts.Element("pubDate").Value),
                                Description =
                                    myPosts.Element("description").Value,
```

```

        Url = myPosts.Element("link").Value,
        Comments = myPosts.Element("comments").Value
    };
    foreach (var item in queryPosts)
    {
        Console.WriteLine(item.Title);
    }

    Console.ReadLine();
}
}

```

代码下载 ConsoleApplication1.sln

在这段代码中, XDocument 对象的 Load()方法指向一个 URL, 从该 URL 中检索 XML 文档。第一个查询提取种子中<channel>元素的所有主要子元素, 新建 Title、Description 和 Link 对象, 以获取这些子元素的值。

之后, 运行一条 foreach 语句, 迭代该查询找到的所有项, 结果如下:

```

TITLE: Bill Evjen's Blog
DESCRIPTION: Code, Life and Community
LINK: http://geekswithblogs.net/evjen/Default.aspx

```

第二个查询遍历它找到的所有<item>元素和不同子元素(这些都是在博客中找到的博客项)。尽管找到的许多项都出现在属性中, 但在 foreach 循环中只使用了 Title 属性。这个查询的部分结果如下所示:

```

AJAX Control Toolkit Controls Grayed Out-HOW TO FIX
Welcome .NET 4.0!
Visual Studio
IIS 7.0 Rocks the House!
Word Issue-Couldn't Select Text
Microsoft Releases XML Schema Designer CTP1
Silverlight Book
Microsoft Tafiti as a beta
ReSharper on Visual Studio
Windows Vista Updates for Performance and Reliability Issues
First Review of Professional XML
Go to MIX07 for free!
Microsoft Surface and the Future of Home Computing?
Alas my friends-I'm *not* TechEd bound

```

33.12 XML 文档的更多查询技术

如果正在处理 XML 文档 hamlet.xml, 就会注意到该文件相当大。在本章中, 查询 XML 文档有两种方式, 但下一节介绍 XML 文档的读写操作。

33.12.1 读取 XML 文档

可以发现使用 LINQ 查询语句查询 XML 文档多么简单, 如下所示:

```
var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;
```

这个查询返回在文档中找到的所有演员。使用 `XDocument` 对象的 `Element()` 方法，还可以获取 XML 文档中的特定值。例如，下面的 XML 片段说明了如何在 `hamlet.xml` 文档中表示标题：

```
<?xml version="1.0"?>
<PLAY>
  <TITLE> The Tragedy of Hamlet, Prince of Denmark</TITLE>

  <!--XML removed for clarity-->
</PLAY>
```

可以看出，`<TITLE>` 元素是 `<PLAY>` 元素的一个嵌套元素。在控制台应用程序中使用下面的代码可以轻松地获得标题：

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(xdoc.Element("PLAY").Element("TITLE").Value);
```

这段代码把标题 `The Tragedy of Hamlet, Prince of Denmark` 输出到控制台屏幕上。在代码中，可以使用两个 `Element()` 方法调用进入 XML 文档的层次结构，第一个 `Element()` 方法调用 `<PLAY>` 元素，然后第二个 `Element()` 方法调用嵌套在 `<PLAY>` 元素中的 `<TITLE>` 元素。

仔细查看 `hamlet.xml` 文档，会发现它使用 `<PERSONA>` 元素定义了一个很大的演员列表：

```
<?xml version="1.0"?>
<PLAY>
<TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>

<!--XML removed for clarity-->

<PERSONAE>
  <TITLE>Dramatis Personae</TITLE>

  <PERSONA>CLAUDIUS, king of Denmark.</PERSONA>
  <PERSONA>HAMLET, son to the late king,
    and nephew to the present king.</PERSONA>
  <PERSONA>POLONIUS, lord chamberlain.</PERSONA>
  <PERSONA>HORATIO, friend to Hamlet.</PERSONA>
  <PERSONA>LAERTES, son to Polonius.</PERSONA>
  <PERSONA>LUCIANUS, nephew to the king.</PERSONA>

  <!--XML removed for clarity-->

</PERSONAE>
</PLAY>
```

现在看看下面这个 C# 查询：

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(
  xdoc.Element("PLAY").Element("PERSONAE").Element("PERSONA").Value);
```

这段代码首先访问<PLAY>元素,再访问<PERSONA>元素,最后使用<PERSONA>元素。但是,其结果如下:

CLAUDIUS, king of Denmark

原因是尽管有一个<PERSONA>元素集合,但我们只处理使用 Element().Value 调用遇到的第一个<PERSONA>元素。

33.12.2 写入 XML 文档

除了读取 XML 文档之外,还可以同样轻松地写入该文档。例如,如果要改变《哈姆雷特》剧本的第一个演员的名称,就可以使用下面的代码:



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            xdoc.Element("PLAY").Element("PERSONAE").
                Element("PERSONA").SetValue("Bill Evjen, king of Denmark");

            Console.WriteLine(xdoc.Element("PLAY").
                Element("PERSONAE").Element("PERSONA").Value);

            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.sln

在这个例子中,使用 Element()对象的 SetValue()方法把<PERSONA>元素的第一个实例重写为 Bill Evjen, king of Denmark。调用 SetValue()方法并将该值应用于 XML 文档后,就使用与前面相同的方法检索该值。运行这段代码,会发现第一个<PERSONA>元素的值改变了。

修改文档的另一种方式(在这个例子中是给文档添加项)是把需要的元素创建为 XElement 对象,再把它们添加到文档中:



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
```

```

XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

XElement xe = new XElement("PERSONA",
    "Bill Evjen, king of Denmark");

xdoc.Element("PLAY").Element("PERSONAE").Add(xe);

var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;

Console.WriteLine("{0} Players Found", query.Count());
Console.WriteLine();

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

Console.ReadLine();

```

代码下载 [ConsoleApplication1.sln](#)

在这个例子中，创建了一个 XElement 文档 xe。xe 的构造会提供如下 XML 结果：

```
<PERSONA>Bill Evjen, king of Denmark</PERSONA>
```

接着使用 XDocument 对象的 Element().Add() 方法，可以添加所创建的元素：

```
xdoc.Element("PLAY").Element("PERSONAE").Add(xe);
```

现在查询所有演员时，会找到 27 个演员，而不是 26 个，新加的一个演员在列表的底部。除了 Add() 方法之外，还可以使用 AddFirst() 方法，顾名思义，它会把元素添加到列表的开头，而不是默认的末尾。

33.13 小结

本章探讨了 .NET Framework 的 System.Xml 名称空间中的许多内容，其中包括如何使用基于 XMLReader 和 XmlWriter 的类快速读写 XML 文档，如何在 .NET 中实现 DOM，如何使用 DOM 的强大功能。XML 和 ADO.NET 实际上有非常密切的关系。DataSet 和 XML 文档仅是相同底层体系结构的两个不同视图而已。另外，我们还介绍了 XPath 和 XSL 转换，以及添加到 VS 中的调试功能。

最后，可以把对象序列化到 XML 中，还可以通过两个方法调用对其进行反序列化。

XML 是以后几年中应用程序开发的一个重要部分。.NET Framework 提供了操作 XML 的丰富而强大的工具集。

本章介绍了如何使用 LINQ to XML 和读写 XML 文件及 XML 源(无论是静态还是动态的)的一些选项。

使用 LINQ to XML 可以通过一系列强类型化的操作对 XML 文件及 XML 源执行 CRUD 操作。

也可以联合使用 XmlReader、XmlWriter 和 LINQ to XML 功能编写代码。

本章还介绍了新的 LINQ to XML 帮助对象 XDocument、XElement、XNamespace、XAttribute 和 XComment。这些都是使 XML 操作比以前更方便的重要新对象。

第34章

.NET 编程和 SQL Server

本章内容:

- 用 SQL Server 驻留 .NET 运行库
- System.Data.SqlServer 名称空间中的类
- 创建用户定义的类型和聚合函数
- 存储过程
- 用户定义的函数
- 触发器
- 使用 XML 数据类型

SQL Server 2005 是这个数据库产品驻留 .NET 运行库的第一个版本。实际上,它是 Microsoft 的 SQL Server 产品在近 6 年中的第一个新版本,它允许在 SQL Server 进程中运行 .NET 程序集。而且,SQL Server 2005 允许用 .NET 编程语言,如 C# 和 Visual Basic,创建存储过程、函数和数据类型。



本章需要能驻留 CLR 的 SQL Server 版本,SQL Server 2005 或更高版本就满足这个要求。SQL Server 2005 和 2008 需要 .NET 3.5。所以对于本章的服务器端项目,CLR 版本必须设置为 3.5。

本章的示例使用新建的 SqlServerSampleDB 数据库和 AdventureWorks 数据库,SqlServerSampleDB 数据库可以与代码示例一起下载。AdventureWorks 数据库是 Microsoft 的一个样本数据库,可以在安装 SQL Server 时作为组件选择安装它。

SQL Server 有许多新特性不能直接与 CLR 关联起来,如许多对 T-SQL 的改进,但本章没有介绍它们。要了解这些特性的更多信息,可参阅清华大学出版社引进并出版的《SQL Server 2008 高级程序设计》(ISBN: 978-7-302-22272-9)

34.1 .NET 运行库的宿主

SQL Server 是 .NET 运行库的一个宿主。在 CLR 2.0 以前的版本中,要运行 .NET 应用程序已经

有多个主机,如 Windows Forms 的宿主和 ASP.NET 的宿主。Internet Explorer 是另一个能运行 Windows 窗体控件的运行库宿主。

SQL Server 允许在 SQL Server 进程中运行 .NET 程序集,在该进程中,可以用 CLR 代码创建存储过程、函数、数据类型和触发器。

每个使用 CLR 代码的数据库都创建了它自己的应用程序域。这将保证一个数据库的 CLR 代码不影响其他数据库。



应用程序域的内容详见第 18 章。

给 SQL Server 编程的一个重要因素是安全性。SQL Server 作为 .NET 运行库的宿主,定义了一些权限级别: safe、external 和 unsafe。

- **Safe**——在 safe 安全级上,只能使用计算的 CLR 类。程序集只能进行本地数据访问。这些类的功能类似于 T-SQL 存储过程。代码访问安全机制指定,只有具备 .NET 许可,才能执行 CLR 代码。
- **External**——在 external 安全级上,可以使用客户端的 ADO.NET 访问网络、文件系统、注册表或其他数据库。
- **Unsafe**——在 unsafe 安全级上,可以执行任何操作,因为这个安全级允许调用本地代码。有 unsafe 权限级别的程序集只能由数据库管理员安装。

在 SQL Server 2008 中,需要把目标架构设置为 2.0 版本,因为 SQL Server 2008 信任这个版本中的程序集。

为了可以在 SQL Server 中运行自定义 .NET 代码,必须用 sp_configure 存储过程启用 CLR:

```
sp_configure [clr enabled], 1
reconfigure
```

System.Security.Permissions 名称空间中的 HostProtectionAttribute 属性类适用于保护宿主环境。利用这个属性,可以指定方法是否使用共享状态、提供同步,或者控制宿主环境。因为这种操作在 SQL Server 代码中通常是不需要的(且会影响 SQL Server 的性能),所以应用了这些设置的程序集不允许加载到安全级是 safe 和 external 的 SQL Server 上。

为了通过 SQL Server 使用程序集,程序集可以使用 CREATE ASSEMBLY 命令安装。通过这条命令,可以在 SQL Server 中应用程序集的名称、程序集的路径和安全级别:

```
CREATE ASSEMBLY mylibrary FROM c:/ProCSharp/SqlServer/Demo.dll
WITH PERMISSION SET = SAFE
```

在 Visual Studio 2010 中,所生成程序集的权限级别可以用项目属性的 Database 选项卡定义,如图 34-1 所示。

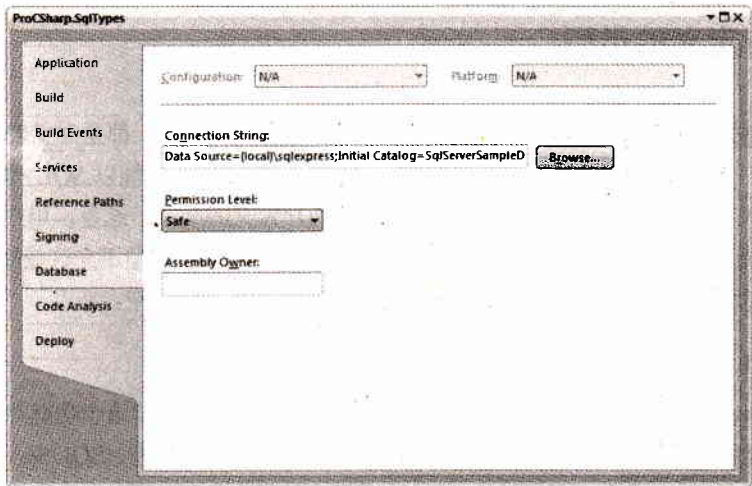


图 34-1

34.2 Microsoft.SqlServer.Server

第 30 章讨论了 System.Data.SqlClient 名称空间中的类。本节讨论另一个名称空间 Microsoft.SqlServer.Server。它包含专用于 .NET Framework 的类、接口和枚举。但是，System.Data.SqlClient 名称空间中的许多类也需要服务器端的代码。

表 34-1 列出了 Microsoft.SqlServer.Server 名称空间中的主要类及其功能。

表 34-1

类	说 明
SqlContext	与 HTTP 环境一样，SQL 环境也与客户端的请求相关联。使用 SqlContext 类的静态成员，可以访问 SqlPipe、SqlTriggerContext 和 WindowsIdentity
SqlPipe	使用 SqlPipe 类，可以把结果或信息发送给客户端。这个类提供了 ExecuteAndSend()、Send() 和 SendResultsRow() 方法。Send() 方法有不同的重载版本，分别发送 SqlDataReader、SqlDataRecord 和 string
SqlDataRecord	SqlDataRecord 表示一行数据。这个类和 SqlPipe 类一起使用，收发来自客户端的信息
SqlTriggerContext	SqlTriggerContext 类在触发器中使用。这个类提供了已激活的触发器的相关信息

这个名称空间还包含几个特性类——SqlProcedureAttribute、SqlFunctionAttribute、SqlUserDefinedAttribute 和 SqlTriggerAttribute。这些类用于在 SQL Server 中部署存储过程、函数、用户定义的类型和触发器。从 Visual Studio 中部署时，需要应用这些属性。在使用 SQL 语句部署数据库对象时，不需要这些特性，但它们也很有帮助，因为这些特性的一些属性会影响数据库对象的特征。

本章后面编写存储过程和用户定义的函数时，会介绍这些类。但下面先看看如何使用 C# 创建用户定义的类型。

34.3 用户定义的类型

用户定义的类型(UDT)的用法与一般的 SQL Server 数据类型类似,都是定义表中一列的类型。通过 SQL Server 的旧版本,已经可以定义 UDT。当然,这些 UDT 只能以 SQL 类型为基础,如下面代码中的 ZIP 类型。sp_addtype 存储过程允许创建用户定义的类型。这里,用户定义的类型 ZIP 以 CHAR 数据类型为基础,其长度为 5。NOT NULL 指定 ZIP 数据类型不允许使用 NULL。把 ZIP 用作一个数据类型,就不再需要记住,它应有 5 个字符长,且不能为空:

```
EXEC sp_addtype ZIP 'CHAR(5)', 'NOT NULL'
```

在 SQL Server 2005 和以后的版本中,UDT 可以用 CLR 类定义。但这个功能并不意味着在数据库中添加面向对象功能,例如,把 Person 类创建为包含 Person 数据类型。SQL Server 是一个关系数据存储,这对 UDT 也正确。不能创建 UDT 的类层次结构,也不能用 SELECT 语句引用 UDT 类型的字段或属性。如果必须访问某个人的属性(如 Firstname 或 Lastname)或对一个 Person 对象列表排序(如按 Firstname 或 Lastname 排序),那么最好在 Person 表中为姓或名分别定义列,或使用 XML 数据类型。

UDT 是非常简单的数据类型。在推出 .NET 之前,还可以创建自定义数据类型,如 ZIP 数据类型。不能利用 UDT 创建类层次结构,也不能把复杂的数据类型放在数据库中。UDT 的一个要求是,它必须能转换为字符串,因为字符串表示形式用于显示值。

可以定义数据在 SQL Server 中的存储方式:或者使用自动机制以本地格式存储数据;或者把数据转换为字节流以定义数据的存储方式。

34.3.1 创建 UDT

下面的示例创建一个 SqlCoordinate 类型,它表示用于显示经度和纬度的世界坐标系,以便于定义地点、城市等的位置。要使用 Visual Studio 创建 CLR 对象,可以在 Database Project | SQL Server 类别中新建一个 Visual C# SQL CLR Database Project,再使用 Solution Explorer 窗口中的 User-Defined Type 模板添加一个 UDT,指定 UDT 的名称 SqlCoordinate。

在模板中,已经定义了自定义类型的基本功能:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct SqlCoordinate: INullable
{
    public override string ToString()
    {
        // Replace the following code with your code
        return "";
    }
}
```

```

public bool IsNull
{
    get
    {
        // Put your code here
        return m_Null;
    }
}

public static SqlCoordinate Null
{
    get
    {
        SqlCoordinate h = new SqlCoordinate();
        h.m_Null = true;
        return h;
    }
}

public static SqlCoordinate Parse(SqlString s)
{
    if (s.IsNull)
        return Null;
    SqlCoordinate u = new SqlCoordinate();
    // Put your code here
    return u;
}

// This is a place-holder method
public string Method1()
{
    //Insert method code here
    return "Hello";
}

// This is a place-holder static method
public static SqlString Method2()
{
    // Insert method code here
    return new SqlString("Hello");
}

// This is a placeholder field member
public int var1;
// Private member
private bool m_Null;
}

```

因为这个类型也可以直接从客户端代码中使用,所以最好添加一个名称空间,它不是自动实现的。

SqlCoordinate 结构实现 **INullable** 接口。**INullable** 接口是 UDT 必须实现的,因为数据库类型也可以为空。**[SqlUserDefinedType]** 属性由 Visual Studio 用于对 UDT 自动部署。**Format.Native** 参数指定要使用的序列化格式。可以使用两种序列化格式: **Format.Native** 和 **Format.UserDefined**。**Format.Native** 是简单的序列化格式,其中引擎会对实例进行序列化和反序列化。这种序列化只对值类型有效(预定

义的值类型和值类型的结构)。引用类型(包括 `string` 类型)需要 `UserDefined` 序列化格式。对于 `SqlCoordinate` 结构, 要序列化的数据类型是 `int` 和 `bool`, 它们都是值类型。

使用 `Format.UserDefined` 格式需要实现 `IBinarySerialize` 接口。该接口提供了用户定义的类型自定义实现方式。在这个接口中, 必须实现 `Read()`和 `write()`方法, 才能把数据序列化到 `BinaryReader` 和 `BinaryWriter` 中。



可从
wrox.com
下载源代码

```
namespace Wrox.ProCSharp.SqlServer
{
    [Serializable]
    [SqlUserDefinedType (Format.Native)]
    public struct SqlCoordinate: INullable
    {
        private int longitude;
        private int latitude;
        private bool isNull;
```

代码段 `SqlTypes/SqlCoordinate.cs`

`SqlUserDefinedType` 特性允许设置几个属性, 如表 34-2 所示。

表 34-2

SqlUserDefinedType 特性的属性	说 明
Format	Format 属性指定数据类型在 SQL Server 中的存储方式, 目前支持的格式有 <code>Format.Native</code> 和 <code>Format.UserDefined</code>
IsByteOrdered	如果把 <code>IsByteOrdered</code> 属性设置为 <code>true</code> , 就可以创建数据类型的索引, 它可以由 SQL 语句 <code>GROUP BY</code> 和 <code>ORDER BY</code> 使用。磁盘的表示可以用于二元比较。因为每个实例都只能有一个序列化表示, 所以二元比较可以成功。其默认值是 <code>false</code>
IsFixedLength	如果所有实例的磁盘表示都有相同的大小, <code>IsFixedLength</code> 就可以设置为 <code>true</code>
MaxByteSize	存储数据所需的最大字节数用 <code>MaxByteSize</code> 设置。这个属性只能在用户定义的序列化中指定
Name	利用 <code>Name</code> 属性, 可以设置类型的另一个名称。默认情况下使用类名
ValidationMethodName	利用 <code>ValidationMethodName</code> 属性, 可以把方法名定义为在进行反序列化时验证实例

为了表示坐标的方向, 可以定义 `Orientation` 枚举:

```
public enum Orientation
{
    NorthEast,
    NorthWest,
    SouthEast,
    SouthWest
}
```

这个枚举只能在 `SqlCoordinate` 结构的方法中使用, 不能用作成员字段, 因为尽管枚举基于数字类型, 但本地序列化格式不支持枚举。以后的版本可能支持在 `SQL Server` 中使用带本地格式的枚举。

SqlCoordinate 结构指定一些构造函数来初始化 longitude、latitude 和 isNull 变量。如果没有给 longitude 和 latitude 赋值,就把 isNull 变量设置为 true,这是默认构造函数的执行情况。UDT 需要默认构造函数。

在世界坐标系统中,经度和纬度都定义为度、分、秒。奥地利的维也纳位于经度 48° 14', 纬度 16° 20'。符号°、'、"分别表示度、分、秒。

在 longitude 和 latitude 变量中,经度和纬度值用秒来存储。包含 7 个整型参数的构造函数把度、分、秒转换为秒,如果坐标基于南和西,就把经度和纬度设置为负值:

```
public SqlCoordinate(int longitude, int latitude)
{
    isNull = false;
    this.longitude = longitude;
    this.latitude = latitude;
}

public SqlCoordinate(int longitudeDegrees, int longitudeMinutes,
    int longitudeSeconds, int latitudeDegrees, int latitudeMinutes,
    int latitudeSeconds, Orientation orientation)
{
    isNull = false;
    this.longitude = longitudeSeconds + 60 * longitudeMinutes + 3600 *
        longitudeDegrees;
    this.latitude = latitudeSeconds + 60 * latitudeMinutes + 3600 *
        latitudeDegrees;
    switch (orientation)
    {
        case Orientation.SouthWest:
            longitude = -longitude;
            latitude = -latitude;
            break;
        case Orientation.SouthEast:
            longitude = -longitude;
            break;
        case Orientation.NorthWest:
            latitude = -latitude;
            break;
    }
}
```

INullable 接口定义 IsNull 属性,必须实现该接口,才能支持可空性。静态属性 Null 用于创建一个表示空值的对象。在 get 访问器中,创建一个 SqlCoordinate 对象,并把 isNull 字段设置为 true:

```
public bool IsNull
{
    get
    {
        return isNull;
    }
}

public static SqlCoordinate Null
{

```

```

get
{
    return new SqlCoordinate,{ isNull = true };
}

```

UDT 必须转换为字符串,也必须能从字符串转换回来。要转换为字符串,必须重写 Object 类的 ToString()方法。把以下代码中的 longitude 和 latitude 变量转换为一个字符串表示,以显示度、分和秒标记:

```

public override string ToString()
{
    if (this.isNull)
        return null;
    char northSouth = longitude > 0 ? 'N': 'S';
    char eastWest = latitude > 0 ? 'E': 'W';

    int longitudeDegrees = Math.Abs(longitude) / 3600;
    int remainingSeconds = Math.Abs(longitude) % 3600;
    int longitudeMinutes = remainingSeconds / 60;
    int longitudeSeconds = remainingSeconds % 60;

    int latitudeDegrees = Math.Abs(latitude) / 3600;
    remainingSeconds = Math.Abs(latitude) % 3600;
    int latitudeMinutes = remainingSeconds / 60;
    int latitudeSeconds = remainingSeconds % 60;

    return String.Format("{0} ° {1}'{2}\"{3},{4} ° {5}'{6}\"{7}",
        longitudeDegrees, longitudeMinutes, longitudeSeconds,
        northSouth, latitudeDegrees, latitudeMinutes,
        latitudeSeconds, eastWest);
}

```

使用 Parse()方法把字符串转换为其他类型。用户输入的字符串在静态方法 Parse()的 SqlString 参数中表示。首先, Parse()方法检查该字符串是否表示空值,如果表示空值,就调用 Null 属性,返回一个空的 SqlCoordinate 对象。如果 SqlString s 不表示空值,就转换字符串的文本,从而把经度值和纬度值传递给 SqlCoordinate 对象的构造函数:

```

public static SqlCoordinate Parse(SqlString s)
{
    if (s.IsNull)
        return SqlCoordinate.Null;

    try
    {
        string[] coordinates = s.Value.Split(',');
        char[] separators = { '\'', '\'', '\"' };
        string[] longitudeVals = coordinates[0].Split(separators);
        string[] latitudeVals = coordinates[1].Split(separators);

        if (longitudeVals.Length != 4 && latitudeVals.Length != 4)
            throw new ArgumentException(
                "Argument has a wrong syntax. " +
                "This syntax is required: 37°47'0\"N,122°26'0\"W");
    }
}

```

```

Orientation orientation;
if (longitudeVals[3] == "N" && latitudeVals[3] == "E")
    orientation = Orientation.NorthEast;
else if (longitudeVals[3] == "S" && latitudeVals[3] == "W")
    orientation = Orientation.SouthWest;
else if (longitudeVals[3] == "S" && latitudeVals[3] == "E")
    orientation = Orientation.SouthEast;
else
    orientation = Orientation.NorthWest;

return new SqlCoordinate(
    int.Parse(longitudeVals[0]), int.Parse(longitudeVals[1]),
    int.Parse(longitudeVals[2]),
    int.Parse(latitudeVals[0]), int.Parse(latitudeVals[1]),
    int.Parse(latitudeVals[2]), orientation);
}

catch (FormatException ex)
{
    throw new ArgumentException(
        "Argument has a wrong syntax. " +
        "This syntax is required: 37'47'\0'N,122'26'\0'W",
        ex.Message);
}
}
}

```

34.3.2 通过 SQL 使用 UDT

在构建程序集后，就可以用 SQL Server 部署它。在 SQL Server 中，或者在 Visual Studio 2010 中用 Build | Deploy Project 菜单配置 UDT，或者用下面的 SQL 命令配置 UDT：

```

CREATE ASSEMBLY SqlTypes FROM
    'c:\ProCSharp\SqlServer\SqlTypes.dll'
CREATE TYPE Coordinate EXTERNAL NAME
    [SqlTypes].[Wrox.ProCSharp.SqlServer.SqlCoordinate]

```

使用 EXTERNAL NAME 时，必须设置程序集名和类名，以及名称空间。

现在可以创建一个 Cities 表，其中包含数据类型 SqlCoordinate，如图 34-2 所示。用数据填充该表，如图 34-3 所示。

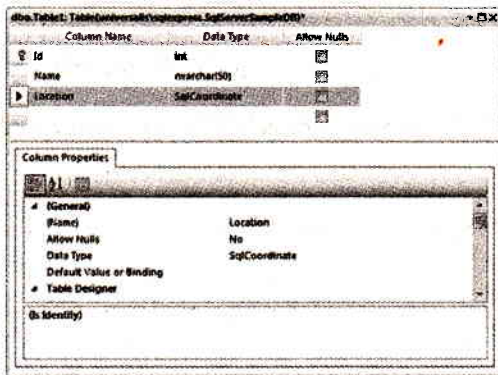


图 34-2

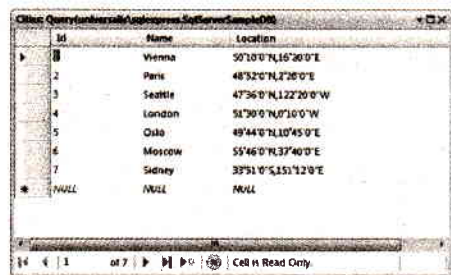


图 34-3

34.3.3 从客户端代码中使用 UDT

必须引用 UDT 的程序集，才能从客户端代码中使用 UDT。它的用法与客户端上的任何其他类型一样。



因为包含 UDT 的程序集用于客户端和 SQL 服务器中，所以最好把 UDT 放在一个独立的程序集中，而不是放在其他 SQL Server 扩展中，如存储过程和函数。

在示例代码中，SqlCommand 对象的 SELECT 语句引用 Cities 表的列，该表包含类型为 SqlCoordinate 的 Location 列。调用 ToString() 方法，会调用 SqlCoordinate 类的 ToString() 方法，以字符串格式显示坐标值：



可从
wrox.com
下载源代码

```
// UDTClient
using System;
using System.Data;
using System.Data.SqlClient;
using Wrox.ProCSharp.SqlServer;

class Program
{
    static void Main()
    {
        string connectionString =
            @"server=(local);database=ProCSharp;trusted_connection=true";
        var connection = new SqlConnection(connectionString);
        var command = connection.CreateCommand();
        command.CommandText = "SELECT Id, Name, Location FROM Cities";
        connection.Open();

        SqlDataReader reader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        while (reader.Read())
        {
            Console.WriteLine("{0,-10} {1}", reader[1].ToString(),
                reader[2].ToString());
        }
        reader.Close();
    }
}
```

代码段 UDTClient/Program.cs

当然，还可以把返回的对象从 SqlDataReader 强型转换为 SqlCoordinate 类型，以使用 SqlCoordinate 类型的任何其他方法。

```
SqlCoordinate coordinate = (SqlCoordinate)reader[2];
```

运行应用程序，结果如下：

```
Vienna 50°10'0"N,16°20'0"E
Paris 48°52'0"N,2°20'0"E
Seattle 47°36'0"N,122°20'0"W
London 51°30'0"N,0°10'0"W
```

Oslo 59°55'0"N,10°45'0"E
 Moscow 55°46'0"N,37°40'0"E
 Sydney 33°51'0"S,151°12'0"E



具备了 UDT 的所有强大功能之后,还必须注意一个重要的限制。在部署 UDT 的新版本之前,必须删除已有版本。只有删除所有使用 UDT 类型的列,才能这么做。不要对于频繁修改的类型使用 UDT。

34.4 用户定义的聚合函数

聚合函数是根据多行返回一个值的函数。内置的聚合函数有 COUNT、AVG 和 SUM:

- COUNT 返回所有选中记录的记录个数
- AVG 返回选中行的一列的平均值
- SUM 返回一系列的所有值的总和

所有内置的聚合函数都只能用于内置的值类型。

内置的聚合函数 AVG 的一种简单用法如下, 它把 AdventureWorks 样本数据库中的 ListPrice 列传递给 SELECT 语句中的 AVG 聚合函数, 以返回所有产品的平均单价:

```
SELECT AVG(ListPrice) AS 'average list price'
FROM Production.Product
```

SELECT 语句的结果返回所有产品的平均单价:

```
average list price
438,6662
```

SELECT 语句返回 ListPrice 列的平均值。聚合函数还可以用于组。在下面的例子中, AVG 聚合函数与 GROUP BY 子句一起使用, 返回每个产品系列的平均单价列表:

```
SELECT ProductLine, AVG(ListPrice) AS 'average list price'
FROM Production.Product
GROUP BY ProductLine
```

平均单价列表按照产品系列进行组合:

ProductLine	average list price
NULL	16,8429
M	827,0639
R	965,3488
S	50,3988
T	840,7621

对于自定义值类型, 如果要基于选中的一些行执行某个特定的计算操作, 就可以创建用户定义的聚合函数。

34.4.1 创建用户定义的聚合函数

要用 CLR 代码编写用户定义的聚合函数，必须实现一个简单的类，它的方法有 Init()、Accumulate()、Merge()和 Terminate()。这些方法的功能如表 34-3 所示。

表 34-3

UDT 方法	说 明
Init()	为要处理的每组行调用 Init()方法。在这个方法中，可以为要计算的每组行进行初始化
Accumulate()	为所有组中的每个值调用 Accumulate()方法。这个方法的参数必须是正确的累加类型，还可以是用户定义的类型
Merge()	聚合的结果必须与另一个聚合结果合并起来时，调用 Merge()方法
Terminate()	在处理完每一组的最后一行后，调用 Terminate()方法。这里，聚合的结果必须用正确的数据类型返回

下面的代码示例说明了如何实现一个简单的用户定义聚合函数，以计算每一组中所有行的总和。为了用 Visual Studio 进行部署，把 SqlUserDefinedAggregate 特性应用于 SampleSum 类。与 UDT 一样，用户定义的聚合函数为存储聚合结果使用的格式也必须用 Format 枚举中的值定义。Format.Native 用于对直接复制到本地结构数据类型进行自动序列化。

在示例代码中，sum 变量用于累加一组中的所有值。在 Init()方法中，给每个要累加的新组初始化变量 sum。为每个值调用的 Accumulate()方法把参数的值加到 sum 变量中。在 Merge()方法中，将聚合后的一组添加到当前组中。最后，Terminate()方法返回一组的结果。



可从
wrox.com
下载源代码

```
[Serializable]
[SqlUserDefinedAggregate (Format.Native)]
public struct SampleSum
{
    private int sum;

    public void Init()
    {
        sum = 0;
    }

    public void Accumulate(SqlInt32 Value)
    {
        sum += Value.Value;
    }

    public void Merge(SampleSum Group)
    {
        sum += Group.sum;
    }

    public SqlInt32 Terminate()
    {
        return new SqlInt32(sum);
    }
}
```

代码段 SqlSamplesUsingAdventureWorks/SampleSum.cs



可以使用 Visual Studio 中的 Aggregate 模板, 为构建用户定义的聚合函数创建核心代码。Visual Studio 中的模板创建一个结构, 它使用 SqlString 类型作为参数, 并用 Accumulate() 和 Terminate() 方法返回对应类型。可以把这个类型改为表示聚合要求的类型。在本例中, 使用 SqlInt32 类型。

34.4.2 使用用户定义的聚合函数

用户定义的聚合函数可以用 Visual Studio 或 CREATE AGGREGATE 语句部署。在下面的语句中, CREATE AGGREGATE 语句后面是聚合函数名, 参数是 (@value int) 和返回类型。EXTERNAL NAME 需要程序集名和包含名称空间的 .NET 类型。

```
CREATE AGGREGATE [SampleSum] (@value int) RETURNS [int] EXTERNAL NAME
    [Demo].[SampleSum]
```

在安装用户定义的聚合函数后, 就可以在下面的 SELECT 语句中使用它, 在这条语句中, 把 Products 表和 PurchaseOrderDetails 表连接起来, 以返回已订购的产品数量。对于用户定义的聚合函数, 把 PurchaseOrderDetails 表的 OrderQty 列定义为一个参数:

```
SELECT Purchasing.PurchaseOrderDetail.ProductID AS Id,
    Production.Product.Name AS Product,
    dbo.SampleSum(Purchasing.PurchaseOrderDetail.OrderQty) AS Sum
FROM Production.Product INNER JOIN
    Purchasing.PurchaseOrderDetail ON
    Purchasing.PurchaseOrderDetail.ProductID = Production.Product.ProductID
GROUP BY Purchasing.PurchaseOrderDetail.ProductID, Production.Product.Name
ORDER BY Id
```

返回的抽查结果使用聚合函数 SampleSum 显示已订购的产品数量:

Id	Product	Sum
1	Adjustable Race	154
2	Bearing Ball	150
4	Headset Ball Bearings	153
317	LL Crankarm	44000
318	ML Crankarm	44000
319	HL Crankarm	71500
320	Chainring Bolts	375
321	Chainring Nut	375
322	Chainring	7440

34.5 存储过程

SQL Server 可以用 C# 创建存储过程。存储过程是一个子例程, 它们以物理方式存储在数据库中。但它们绝对不能替代 T-SQL。在过程主要基于数据驱动时, T-SQL 还是有优势的。

下面是 T-SQL 存储过程 GetCustomerOrders, 它返回 AdventureWorks 数据库中的客户订单信息。这个存储过程返回用 CustomerID 参数指定的顾客的订单:

```

CREATE PROCEDURE GetCustomerOrders
(
    @CustomerID int
)
AS
SELECT SalesOrderID, OrderDate, DueDate, ShipDate FROM Sales.SalesOrderHeader
WHERE (CustomerID = @CustomerID)
ORDER BY SalesOrderID

```

34.5.1 创建存储过程

在下面的代码清单中，用 C# 实现相同的存储过程比较复杂。SqlProcedure 特性用于把存储过程标记为部署。通过这种实现方式，创建一个 SqlCommand 对象。在 SqlConnection 对象的构造函数中，传递字符串“Context Connection=true”，以使用调用该存储过程的客户端打开的连接。与第 30 章的代码类似，设置了 SQL 的 SELECT 语句，添加了一个参数。ExecuteReader() 方法返回一个 SqlDataReader 对象。通过调用 SqlPipe 的 Send() 方法，把这个读取器对象返回给客户端：



可从
wrox.com
下载源代码

```

using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [SqlProcedure]
    public static void GetCustomerOrdersCLR(int customerId)
    {
        SqlConnection connection = new SqlConnection("Context Connection=true");
        connection.Open();
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText = "SELECT SalesOrderID, OrderDate, DueDate, " +
            "ShipDate " +
            "FROM Sales.SalesOrderHeader " +
            "WHERE (CustomerID = @CustomerID) " +
            "ORDER BY SalesOrderID";

        command.Parameters.Add("@CustomerID", SqlDbType.Int);
        command.Parameters["@CustomerID"].Value = customerId;

        SqlDataReader reader = command.ExecuteReader();
        SqlPipe pipe = SqlContext.Pipe;
        pipe.Send(reader);
        connection.Close();
    }
};

```

代码段 SqlSamplesUsingAdventureWorks/GetCustomerOrdersCLR.cs

CLR 存储过程在 SQL Server 中或者使用 Visual Studio 或者使用 CREATE PROCEDURE 语句部署。通过这条 SQL 语句，定义存储过程的参数，以及程序集、类和方法的名称：

```

CREATE PROCEDURE GetCustomerOrdersCLR
(
    @CustomerID nchar(5)
)

```

AS EXTERNAL NAME Demo.StoredProcedures.GetCustomerOrdersCLR

34.5.2 使用存储过程

CLR 存储过程可以像一般的 T-SQL 存储过程那样, 也使用 `System.Data.SqlClient` 名称空间中的类调用。首先, 创建一个 `SqlConnection` 对象, `CreateCommand()` 方法返回一个 `SqlCommand` 对象。通过这个命令对象, 把存储过程的名称 `GetCustomerOrdersCLR` 设置为 `CommandText` 属性。与所有存储过程一样, `CommandType` 属性必须设置为 `CommandType.StoredProcedure`。 `ExecuteReader()` 方法返回一个 `SqlDataReader` 对象, 用于逐个记录地读取:



可从
wrox.com
下载源代码

```
using System;
using System.Data;
using System.Data.SqlClient;

//...

string connectionString =
    @"server=(local);database=AdventureWorks;trusted_connection=true";
var connection = new SqlConnection(connectionString);
SqlCommand command = connection.CreateCommand();
command.CommandText = "GetCustomerOrdersCLR";
command.CommandType = CommandType.StoredProcedure;
var param = new SqlParameter("@customerId", 3);
command.Parameters.Add(param);
connection.Open();
SqlDataReader reader =
    command.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
    Console.WriteLine("{0} {1:d}", reader["SalesOrderID"], reader["OrderDate"]);
}
reader.Close();
```

代码段 UsingSP/Program.cs



System.Data.SqlClient 名称空间中的类详见第 30 章。

调用 T-SQL 和 C# 编写的存储过程没有任何区别。调用存储过程的代码完全相同; 从调用者的代码中, 不知道存储过程是用 T-SQL 还是 CLR 实现的。上述代码的抽检结果是 ID 为 3 的客户的订单日期:

```
44124 9/1/2001
44791 12/1/2001
45568 3/1/2002
46377 6/1/2002
47439 9/1/2002
48378 12/1/2002
```

如前所述, 主要基于数据驱动的存储过程用 T-SQL 编写比较好, 代码比较短。用 CLR 编写存储过程的优点是, 可以进行一些特殊的数据处理, 如使用 .NET 加密类处理。

34.6 用户定义的函数

用户定义的函数有点类似于存储过程。其主要区别是，用户定义的函数可以在 SQL 语句中调用。

34.6.1 创建用户定义的函数

CLR 用户定义的函数可以用 `SqlFunction` 特性来定义。示例函数 `CalcHash()` 把传递进来的字符串转换为散列字符串。用于散列字符串的 MD5 算法通过 `System.Security.Cryptography` 名称空间中的 `MD5CryptoServiceProvider` 类实现。`ComputeHash()` 方法从输入的字节数组中计算散列，并返回一个计算出来的散列字节数组。该散列字节数组会使用 `StringBuilder` 类转换回 `string`。



可从
wrox.com
下载源代码

```
using System.Security.Cryptography;
using System.Text;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [SqlFunction]
    public static SqlString CalcHash(SqlString value)
    {
        byte[] source = ASCIIEncoding.ASCII.GetBytes(value.ToString());
        byte[] hash = new MD5CryptoServiceProvider().ComputeHash(source);

        var output = new StringBuilder(hash.Length);

        for (int i = 0; i < hash.Length - 1; i++)
        {
            output.Append(hash[i].ToString("X2"));
        }

        return new SqlString(output.ToString());
    }
}
```

代码段 `SqlSamplesUsingAdventureWorks/CalcHash.cs`

34.6.2 使用用户定义的函数

用户定义的函数可以在 SQL Server 中部署，与其他 .NET 扩展插件类似：或者使用 Visual Studio 2010 或者使用 `CREATE FUNCTION` 语句：

```
CREATE FUNCTION CalcHash
(
    @value nvarchar
)
RETURNS nvarchar
AS EXTERNAL NAME Demo.UserDefinedFunctions.CalcHash
```

`CalcHash()` 函数的一个简单用法如下面的 `SELECT` 语句所示，其中从 `AdventureWorks` 数据库的 `CreditCard` 表中访问信用卡号，具体方法是只从信用卡号中返回散列代码：

```
SELECT Sales.CreditCard.CardType AS [Card Type],
```

```

    dbo.CalcHash(Sales.CreditCard.CardNumber) AS [Hashed Card]
FROM Sales.CreditCard INNER JOIN Sales.ContactCreditCard ON
    Sales.CreditCard.CreditCardID = Sales.ContactCreditCard.CreditCardID
WHERE Sales.ContactCreditCard.ContactID = 11

```

返回的结果显示 ID 为 11 的联系人的散列信用卡号:

```

Card          Type Hashed Card
ColonialVoice 7482F7B4E613F71144A9B336A3B9F6

```

34.7 触发器

触发器是一种特殊的存储过程, 在修改表(例如插入、更新或删除一行)时调用它。触发器与表和激活它们的动作(例如, 行的插入/更新/删除)关联在一起。

有了触发器, 行的修改就可以通过相关的表来级联处理, 或强制更复杂的数据完整性操作。

在触发器中, 因为可以访问行的当前数据和原始数据, 所以可以把表重置回以前的状态。因为触发器会自动与激活触发器的命令所对应的事务关联起来, 所以会得到正确的事务处理结果。

下面的触发器 `uCreditCard` 属于 `AdventureWorks` 样本数据库。这个触发器在更新 `CreditCard` 表中的一行时激活。使用这个触发器, 将 `CreditCard` 表中的 `ModifiedDate` 列更新为当前日期。要访问已修改的数据, 可以使用插入的临时表。

```

CREATE TRIGGER [Sales].[uCreditCard] ON [Sales].[CreditCard]
AFTER UPDATE NOT FOR REPLICATION AS
BEGIN
    SET NOCOUNT ON;

    UPDATE [Sales].[CreditCard]
    SET [Sales].[CreditCard].[ModifiedDate] = GETDATE()
    FROM inserted
    WHERE inserted.[CreditCardID] = [Sales].[CreditCard].[CreditCardID];
END;

```

34.7.1 创建触发器

本节的例子说明了在 `Users` 表中插入新记录时触发器实现数据完整性。要使用 CLR 创建触发器, 必须定义一个简单的类, 它包含应用了 `SqlTrigger` 特性的静态方法。`SqlTrigger` 特性指定了与触发器相关联的表和何时启动触发器的事件。

在下面的例子中, 相关联的表是 `Target` 属性指定的 `Person.Contact` 表, `Event` 属性定义触发器启动的时间。这里把事件字符串设置为 `FOR INSERT`, 它表示触发器在把新的一行插入 `Users` 表中时启动。

`SqlContext.TriggerContext` 属性在 `SqlTriggerContext` 类型的对象中返回触发器环境, `SqlTriggerContext` 类提供了 3 个属性:

- `ColumnsUpdated` 返回一个布尔数组, 以标记已修改的每一列
- `EventData` 包含 XML 格式的更新数据和原始数据
- `TriggerAction` 返回 `TriggerAction` 类型的枚举, 以标记触发器启动的原因

下面的代码通过比较确定触发器环境的 `TriggerAction` 是否设置为 `TriggerAction.Insert`, 之后再

继续。

触发器可以访问临时表,例如,以下代码清单访问 INSERTED 表。通过 SQL 语句 INSERT、UPDATE 和 DELETE,可以创建临时表。INSERT 语句创建 INSERTED 表,DELETE 语句创建 DELETED 表。通过 UPDATE 语句,可以使用 INSERTED 和 DELETED 表。临时表的列与触发器关联的表相同。SQL 语句 SELECT Username, Email FROM INSERTED 用于访问用户名和电子邮件,检查电子邮件地址的语法是否正确。SqlCommand.ExecuteNonQuery()方法返回在 SqlDataReader 中表示的一行。用户名和电子邮件从数据记录中读取。使用正则表达式类 Regex 和 IsMatch()方法,检查电子邮件地址是否遵循有效的电子邮件语法。如果不遵循,就抛出一个异常,且不插入记录,因为事务会回滚:



```
using System;
using System.Data.SqlClient;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;

public partial class Triggers
{
    [SqlTrigger(Name = "InsertContact", Target="Person.Contact",
        Event="FOR INSERT")]
    public static void InsertContact()
    {
        SqlTriggerContext triggerContext = SqlContext.TriggerContext;

        if (triggerContext.TriggerAction == TriggerAction.Insert)
        {
            var connection = new SqlConnection("Context Connection=true");
            var command = new SqlCommand();
            command.Connection = connection;
            command.CommandText = "SELECT EmailAddress FROM INSERTED";
            connection.Open();
            string email = (string)command.ExecuteScalar();
            connection.Close();

            if (!Regex.IsMatch(email,
                @"([\w-]+\.)?[\w-]+@[ \w-]+\.[( \w-]+\.)?[\w-]+$"))
            {
                throw new FormatException("Invalid email");
            }
        }
    }
}
```

代码段 SqlSamplesUsingAdventureWorks/InsertContact.cs

34.7.2 使用触发器

使用 Visual Studio 2010 的部署功能,可以把触发器部署到数据库中。可以使用 CREATE TRIGGER 命令手工创建触发器:

```
CREATE TRIGGER InsertContact ON Person.Contact
FOR INSERT
AS EXTERNAL NAME Demo.Triggers.InsertContact
```

用不正确的电子邮件把行插入 Users 表中，会抛出一个异常，且插入操作不会执行。

34.8 XML 数据类型

SQL Server 一个主要的编程特性是 XML 数据类型。在 SQL Server 的旧版本中，XML 数据存储在字符串或 blob 中。现在 XML 是一个受到支持的数据类型，它允许把 SQL 查询和 XQuery 表达式合并起来，在 XML 数据中搜索。XML 数据类型可以用作变量、参数、列或 UDF 的返回值。

在 Microsoft Office 中，可以把 Word 和 Excel 文档存储为 XML。Word 和 Excel 也允许使用自定义的 XML 架构，其中只用 XML 存储内容，不存储表示方式。Office 应用程序的输出可以直接存储在 SQL Server 中，其中还可以在这些数据中搜索。当然，自定义 XML 数据也可以存储在 SQL Server 中。



不要给关系数据使用 XML 类型。如果搜索一些元素，且为这些数据明确定义了架构，则以关系方式存储这些元素，可以更快访问对应数据。如果数据是层次结构的，且一些元素是可选的，同时随时间变化，存储为 XML 数据就有许多优点。

34.8.1 包含 XML 数据的表

创建包含 XML 数据的表非常简单，只需给列选择 XML 数据类型即可。下面的 SQL 命令 CREATE TABLE 创建 Exams 表，其中的 ID 列也是主键，Number 列和 Info 列的类型是 XML：

```
CREATE TABLE [dbo].[Exams] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Number] [nvarchar] (10) NOT NULL,
    [Info] [xml] NOT NULL,
    CONSTRAINT [PK_Exams] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) ON [PRIMARY]
) ON [PRIMARY]
```

现在进行一个简单的测试，用如下数据填充表：

```
INSERT INTO Exams values('70-502',
    '<Exam Number="70-502">
    <Title> TS: Microsoft .NET Framework 3.5, Windows Presentation Foundation
    Application Development
    </Title>
    <Certification Name="MCTS Windows Presentation Foundation Application Development"
    Status="Core" />
    <Certification Name="MCTS Web Applications" Status="Core" />
    <Certification Name="MCTS Distributed Applications" Status="Core" />
    <Course>6460</Course>
    <Topic>Creating a WPF Application</Topic>
    <Topic>Building User Interfaces</Topic>
    <Topic>Adding and Managing Content</Topic>
    <Topic>Binding to Data Sources</Topic>
```

```

        <Topic>Customizing Appearance</Topic>
        <Topic>Configuring and Deploying WPF Applications</Topic>
    </Exam>')

INSERT INTO Exams values('70-562',
    '<Exam Number="70-562">
        <Title> TS: Microsoft .NET Framework 3.5, ASP.NET Application Development </Title>
        <Certification Name="MCTS ASP.NET Applications" Status="Core" />
        <Course>2310</Course>
        <Course>6463</Course>
        <Topic>Configuring and Deploying Web Applications</Topic>
        <Topic>Consuming and Creating Server Controls</Topic>
        <Topic>Working with Data and Services</Topic>
        <Topic>Troubleshooting and Debugging Web Applications</Topic>
        <Topic>Working with ASP.NET AJAX and Client-Side Scripting</Topic>
        <Topic>Targeting Mobile Devices</Topic>
        <Topic>Programming Web Applications</Topic>
    </Exam>')

INSERT INTO Exams values('70-561',
    '<Exam Number="70-561">
        <Title> TS: Microsoft .NET Framework 3.5, ADO.NET Application Development </Title>
        <Certification Name="MCTS ADO.NET Applications" Status="Core" />
        <Course>2310</Course>
        <Course>6464</Course>
        <Topic>Connecting to Data Sources</Topic>
        <Topic>Selecting and Querying Data</Topic>
        <Topic>Modifying Data </Topic>
        <Topic>Synchronizing Data</Topic>
        <Topic>Working with Disconnected Data</Topic>
        <Topic>Object Relational Mapping by Using the Entity Framework</Topic>
    </Exam>')

```

34.8.2 读取 XML 值

在 ADO.NET 中, 可以用 `SqlDataReader` 对象读取 XML 数据。`SqlDataReader` 对象的 `GetSqlXml()` 方法返回一个 `SqlXml` 对象。`SqlXml` 类有一个 `Value` 属性, 它返回完整的 XML 表示, `SqlXml` 类的 `CreateReader()` 方法返回一个 `XmlReader` 对象。

`XmlReader` 类的 `Read()` 方法在 `while` 循环中重复执行, 逐个节点地读取数据。在输出中, 我们仅对 `Number` 特性的值、`Title` 和 `Course` 元素的值感兴趣。读取器定位的节点与相应的 XML 元素名比较, 把对应的值写入控制台中。

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text;
using System.Xml;

class Program
{
    static void Main()
    {
        string connectionString =

```

```

        @"server=(local);database=ProCSharp;trusted_connection=true";
var connection = new SqlConnection(connectionString);
var command = connection.CreateCommand();
command.CommandText = "SELECT Id, Number, Info FROM Exams";
connection.Open();
var reader = command.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
    SqlXml xml = reader.GetSqlXml(2);

    XmlReader xmlReader = xml.CreateReader();

    StringBuilder courses = new StringBuilder("Course(s): ", 40);
    while (xmlReader.Read())
    {
        if (xmlReader.Name == "Exam" && xmlReader.IsStartElement())
        {
            Console.WriteLine("Exam: {0}", xmlReader.GetAttribute("Number"));
        }
        else if (xmlReader.Name == "Title" && xmlReader.IsStartElement())
        {
            Console.WriteLine("Title: {0}", xmlReader.ReadString());
        }
        else if (xmlReader.Name == "Course" && xmlReader.IsStartElement())
        {
            courses.AppendFormat("{0} ", xmlReader.ReadString());
        }
    }
    xmlReader.Close();
    Console.WriteLine(courses.ToString());
    Console.WriteLine();
}
reader.Close();
}
}

```

运行应用程序, 结果如下:

```

Exam: 70-502
Title: TS: Microsoft .NET Framework 3.5, Windows Presentation Foundation Application Development
Course(s): 6460

Exam: 70-562
Title: TS: Microsoft .NET Framework 3.5, ASP.NET Application Development
Course(s): 2310 6463

Exam: 70-561
Title: TS: Microsoft .NET Framework 3.5, ADO.NET Application Development
Course(s): 2310 6464

```

除了使用 `XmlReader` 类之外, 还可以使用 DOM 模型把完整的 XML 内容读入 `XmlDocument` 类中, 并分析元素。 `SelectSingleNode()` 方法的参数是一个 XPath 表达式, 并返回一个 `XmlNode` 对象。 XPath 表达式 `"//Exam"` 在完整的 XML 树中查找 XML 元素 Exam。 返回的 `XmlNode` 对象可用于读取所表示元素的子元素。 访问 `Number` 属性的值, 把测验数字写入控制台中, 再访问 `Title` 元素, 把 `Title` 元素的内容写入控制台中, 所有 `Course` 元素的内容也都写入控制台中。



可从
wrox.com
下载源代码

```
string connectionString =
    @"server=(local);database=SqlServerSampleDB;trusted_connection=true";
var connection = new SqlConnection(connectionString);
var command = connection.CreateCommand();
command.CommandText = "SELECT Id, Number, Info FROM Exams";
connection.Open();
var reader = command.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
    SqlXml xml = reader.GetSqlXml(2);
    var doc = new XmlDocument();
    doc.LoadXml(xml.Value);

    XmlNode examNode = doc.SelectSingleNode("//Exam");
    Console.WriteLine("Exam: {0}", examNode.Attributes["Number"].Value);
    XmlNode titleNode = examNode.SelectSingleNode("./Title");
    Console.WriteLine("Title: {0}", titleNode.InnerText);
    Console.Write("Course(s): ");

    foreach (XmlNode courseNode in examNode.SelectNodes("./Course"))
    {
        Console.Write("{0} ", courseNode.InnerText);
    }
    Console.WriteLine();
}
reader.Close();
```

代码段 XmlSamples/Program.cs



XmlReader 类和 XmlDocument 类详见第 33 章。

在.NET 4 中, 还有另一个选项可以访问数据库中的 XML 列。可以合并 LINQ to SQL 和 LINQ to XML, 从而使编程代码更短。

从 Data templates 类别中选择 ADO.NET Entity Data Model 模板, 就可以使用 ADO.NET Entity Framework 设计器。把文件命名为 ExamsModel.edmx, 给 SqlServerSampleDB 数据库创建一个映射。选择 Exams 表, 再选择对象名使用单数形式的选项。利用向导创建模型后, 会打开设计界面, 如图 34-4 所示。

设计器创建的数据上下文类是 ExamsEntities, 它定义了一个 Exams 属性, 用于返回所有 exam 行。这里使用一条 foreach 语句迭代所有记录。当然, 如果并非需要所有记录, 就可以定义一个包含 where 表达式的 LINQ 查询。Exam 类根据数据库表中的列定义 Id、Number 和 Info 属性。因为 Info 属性的类型是 string, 所以通过 XElement 类, 可以使用 System.Xml.Linq 名称空间中的 LINQ to XML 类来访问和分析字符串。调用 Element()方法, 传递 XML 元素名 Exam, 返回一个 XElement 对象, 该对象可以用更简单的方式访问 Number 特性的值, 以及 Title 和 Course 元素的值, 这与前面的 XmlDocument 类相同。

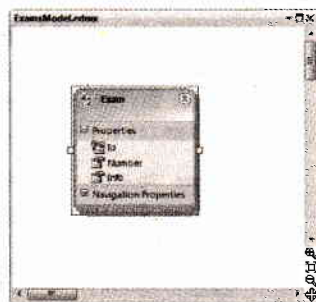


图 34-4

```

using System;
using System.Xml.Linq;

namespace Wrox.ProCSharp.SqlServer
{
    class Program
    {
        static void Main()
        {
            using (ExamsEntities data = new ExamsEntities())
            {
                foreach (Exam item in db.Exams)
                {
                    XElement exam = XElement.Parse(item.Info);
                    Console.WriteLine("Exam: {0}", exam.Attribute("Number").Value);
                    Console.WriteLine("Title: {0}", exam.Element("Title").Value);
                    Console.WriteLine("Course(s): ");
                    foreach (var course in exam.Elements("Course"))
                    {
                        Console.WriteLine("{0} ", course.Value);
                    }
                    Console.WriteLine();
                }
            }
        }
    }
}

```

ADO.NET Entity Framework 和 LINQ to SQL 分别详见第 31 章和第 33 章。

34.8.3 数据的查询

前面还没有涉及 XML 数据类型的主要特性。SQL 语句 SELECT 可以结合 XML Xquery 使用。SELECT 语句结合 XQuery 表达式使用，可以读取 XML 值，如下所示：

```
SELECT [Id], [Number], [Info].query('/Exam/Course') AS Course FROM [Exams]
```

XQuery 表达式/Exam/Course 访问 Exam 元素的 Course 子元素，这个查询的结果返回 id、测验数字和课程。

```

1 70-502<Course>6460</Course>
2 70-562<Course>2310</Course><Course>6463</Course>
3 70-561<Course>2310</Course><Course>6464</Course>

```

通过 XQuery 表达式，可以创建更复杂的语句，查询单元格中 XML 内容的数据。下面的例子把测验信息转换为 XML，XML 列出课程的信息：

```

SELECT [Info].query('
    for $course in /Exam/Course
    return
    <Course>
        <Exam>{data(/Exam[1]/@Number) }</Exam>
        <Number>{data($course) }</Number>
    </Course>')

```

```

AS Course
FROM [Exams]
WHERE Id=2

```

这里用 `SELECT [Info].FROM Exams WHERE Id = 2` 选择一行。通过这个 SQL 查询的结果，使用 XQuery 表达式的 `for` 和 `return` 语句。`for $course in /Exam/Course` 迭代所有 `Course` 元素。`$course` 声明一个变量，它用每个迭代来设置(类似于 C# 的 `foreach` 语句)。在 `return` 语句的后面指定了每一行的查询结果。每个课程元素的结果都放在 `<Course>` 元素中。在 `<Course>` 元素中内嵌了 `<Exam>` 和 `<Number>`。`<Exam>` 元素中的文本用 `data(/Exam[1]/@Number)` 定义。`data()` 是一个 XQuery 函数，返回用参数指定的节点值。`/Exam[1]` 节点用于访问第一个 `<Exam>` 元素，`@Number` 指定 XML 特性 `Number`。`<Number>` 元素中的文本在 `$course` 变量中定义。



在 C# 中，集合中的第一个元素用索引 0 访问，但在 XPath 中，集合中的第一个元素用索引 1 访问。

这个查询的结果如下：

```

<Course>
  <Exam>70-562</Exam>
  <Number>2310</Number>
</Course>
<Course>
  <Exam>70-562</Exam>
  <Number>6463</Number>
</Course>

```

可以修改 XQuery 语句，使之也包含一条 `where` 子句，来筛选 XML 元素。下面的例子仅从 XML 列中返回数值课程编号大于 2542 的课程：

```

SELECT [Info].query('
  for $course in /Exam/Course
  where ($course > 2562)
  return
  <Course>
    <Exam>{ data(/Exam[1]/@Number) }</Exam>
    <Number> { data($course) }</Number>
  </Course> ')
AS Course
FROM [Exams]
WHERE Id=2

```

结果变成只有两个课程编号：

```

<Course>
  <Exam>70-562</Exam>
  <Number>6463</Number>
</Course>

```

SQL Server 中的 XQuery 可以使用几个其他的 XQuery 函数，获得最小值、最大值或总和，处理

字符串、数字, 检查集合中的位置等。

下面的例子使用 `count()` 函数获得 `/Exam/Course` 元素的个数:

```
SELECT [Id], [Number], [Info].query('
    count(/Exam/Course)')
    AS "Course Count"
FROM [Exams]
```

返回的数据显示测验的课程编号:

Id	Number	Course Count
1	70-502	1
2	70-562	2
3	70-561	2

34.8.4 XML 数据修改语言(XML DML)

W3C(<http://www.w3c.org>)在定义 XQuery 时, 只允许查询数据, 当时, Microsoft 在 SQL Server 2005 中实现了 XQuery。由于 XQuery 有这个限制, 因此 Microsoft 定义了 XQuery 的扩展, 称为 XML 数据修改语言(XML DML)。



目前 XQuery Update Facility 1.0 自从 2009 年 6 月 9 日以来成为一个候选建议, 这个建议由 IBM、Oracle 和 Red Hat 制订, 不同于 XML DML。

有了 XML DML, 就可以使用 XQuery 扩展关键字 `insert`、`delete` 和 `replace value of` 修改 XML 数据了。

本节介绍在单元格中插入、删除和修改 XML 内容的一些例子。

可以使用 `insert` 关键字在 XML 列中插入一些 XML 内容, 而无需替换整个 XML 单元格。这里 `<Course>2555</Course>` 插入为第一个 Exam 元素的最后一个子元素:

```
UPDATE [Exams]
SET [Info].modify('
    insert <Course>2555</Course> as last into Exam[1]')
WHERE [Id]=3
```

XML 内容可以用 `delete` 关键字删除。在第一个 Exam 元素中, 删除最后一个 Course 元素。用 `last()` 函数选择最后一个元素:

```
UPDATE [Exams]
SET [Info].modify('
    delete /Exam[1]/Course[last()]')
FROM [Exams] WHERE [Id]=3
```

还可以修改 XML 内容, 这需要使用 `replace value of` 关键字。`/Exam/Course[text()=6463]` 表达式只访问文本内容包含字符串 6463 的子元素 Course。从这些元素中, `text()` 函数只访问要替换的文本内容。如果查询只返回一个元素, 那么还需要指定只替换一个元素。这就是返回的第一个文本元素用 `[1]` 显式指定的原因。2599 指定新的课程编号是 2599:

```
UPDATE [Exams]
SET [Info].modify('
    replace value of (/Exam/Course[text() = 6463]/text())[1] with 2599')
FROM [Exams]
```

34.8.5 XML 索引

如果经常在 XML 数据中搜索某些特定的元素,就可以在 XML 数据类型中指定索引。在 XML 索引中,其类型必须区分为主 XML 索引或次 XML 索引。创建主 XML 索引是为了完整地持久化 XML 值的表示方式。

下面的 SQL 命令 CREATE PRIMARY XML INDEX 在 Info 列上创建了 idx_exams 索引:

```
CREATE PRIMARY XML INDEX idx_exams on Exams (Info)
```

如果查询包含 XPath 表达式,以直接访问 XML 类型的 XML 元素,主索引就没有什么帮助。对于 XPath 和 XQuery 表达式,可以使用 XML 次索引。要创建 XML 次索引,主索引就必须存在。有了次索引,就必须区分索引类型:

- PATH 索引
- VALUE 索引
- PROPERTY 索引

如果使用 exists()或 query()函数,且通过 XPath 表达式访问 XML 元素,就使用 PATH 索引。使用 XPath 表达式/Exam/Course 时,创建 PATH 索引是有帮助的:

```
CREATE XML INDEX idx_examNumbers on [Exams] (Info)
USING XML INDEX idx_exams FOR PATH'
```

如果使用 value()函数从元素中提取属性,就使用 PROPERTY 索引。包含创建索引的 FOR PROPERTY 语句定义一个 PROPERTY 索引:

```
CREATE XML INDEX idx_examNumbers on [Exams] (Info)
USING XML INDEX idx_exams FOR PROPERTY
```

如果使用 XPath 子轴或自轴表达式通过树型结构搜索元素,使用 VALUE 索引就会得到最佳性能。XPath 表达式//Certification 会搜索带有子轴或自轴的所有 Certification 元素。[@Name="MCTS Web Applications"]表达式只返回属性 Name 的值是 MCTS Web Applications 的元素。

```
SELECT [Info].query('/Exam/Title/text()') FROM [Exams]
WHERE [Info].exist('///Certification[@Name="MCTS ASP.NET Applications"]') = 1
```

返回的结果列出了包含所请求证书的测验名:

```
TS: Microsoft .NET Framework 3.5, ASP.NET Application Development
```

VALUE 索引用 FOR VALUE 语句创建:

```
CREATE XML INDEX idx_examNumbers on [Exams] (Info)
USING XML INDEX idx_exams FOR VALUE
```

34.8.6 强类型化的 XML

SQL Server 中的 XML 数据类型也可以用 XML 架构强类型化。通过强类型化的 XML 列，就可以在插入 XML 数据时验证该数据是否遵循该架构。

XML 架构可以用语句 CREATE XML SCHEMA COLLECTION 创建。下面的语句创建一个简单的 XML 架构 CourseSchema，它定义 CourseElt 类型，其中包含一系列 Number 和 Title，它们的类型都是 string，CourseElt 类型还包含一个元素 Any，它可以是任意类型。Number 和 Title 只能出现一次。因为 Any 元素的 minOccurs 属性设置为 0，maxOccurs 属性设置为 unbounded，所以这个元素是可选的。于是可以在未来的版本中给 CourseElt 类型添加额外的信息，而架构仍是有效的。最后，Course 元素名是 CourseElt 类型。

```
CREATE XML SCHEMA COLLECTION CourseSchema AS
'<?xml version="1.0" encoding="UTF-8"?>
<xs:schema id="Courses" targetNamespace="http://thinktecture.com/Courses.xsd"
  elementFormDefault="qualified" xmlns="http://thinktecture.com/Courses.xsd"
  xmlns:mstns="http://thinktecture.com/Courses.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="CourseElt">
    <xs:sequence>
      <xs:element name="Number" type="xs:string" maxOccurs="1"
        minOccurs="1" />
      <xs:element name="Title" type="xs:string" maxOccurs="1"
        minOccurs="1" />
      <xs:element name="Any" type="xs:anyType"
        maxOccurs="unbounded" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Course" type="CourseElt">
  </xs:element>
</xs:schema>'
```

使用这个架构，有效的 XML 如下所示：

```
<Course xmlns="http://thinktecture.com/Courses.xsd">
  <Number>2549</Number>
  <Title>Advanced Distributed Application Development with Visual Studio 2008
</Title>
</Course>
```

在 Visual Studio Database 项目类型中，不支持给数据库添加架构。这个特性不能在 Visual Studio 2010 的 GUI 中使用，但必须手工创建。要使用 Visual Studio 2010 创建 XML 架构，可以使用 Empty Project 模板新建 Visual Studio 项目，给项目添加一个新的 XML 架构，再把架构的 XML 语法复制到 CREATE XML SCHEMA 语句中。

用 XML 数据类型进行设置，就可以把 XML 架构赋予一列：

```
CREATE TABLE [Courses]
(
  [Id] [int] IDENTITY(1,1) NOT NULL,
  [Course] [xml] ([dbo].[CourseSchema]) NOT NULL
)
```

用 Visual Studio 2010 或 SQL Server Management Studio 创建表, 设置属性 XML schema namespace, 就可以把 XML 架构赋予一列。

现在再给 XML 列添加数据时, 会验证其架构。如果 XML 不符合架构定义, 就会抛出一个 SqlException 异常, 显示一个 XML Validation 错误。

34.9 小结

本章讨论了 SQL Server 中与 CLR 功能相关的新特性。因为 SQL Server 包含 CLR, 所以可以用 C# 创建用户定义的类型、聚合函数、存储过程、函数和触发器。

UDT 对 .NET 类有一些严格的要求, 以转换为字符串, 或从字符串转换回来。数据在 SQL Server 中的内部存储方式取决于在类型中定义的格式。用户定义的聚合函数可以使用 .NET 类进行自定义计算。通过存储过程和函数, 可以将 CLR 类用于服务器端代码。

使用 CLR 和 SQL Server, 并不表示 T-SQL 已过时。如果仅进行数据密集型的查询, T-SQL 的优点是代码较少。如果使用加密等 .NET 特性, CLR 类就可以改进数据的处理。

本章还介绍了 SQL Server 的 XML 数据类型, 以把 XQuery 表达式和 T-SQL 语句合并起来。

这是第IV部分的最后一章。第V部分详细介绍了如何定义应用程序的用户界面。在用户界面上可以使用 WPF、Silverlight、Windows 窗体和 ASP.NET。

第 V 部分

显 示

- 第 35 章 核心 WPF
- 第 36 章 用 WPF 编写业务应用程序
- 第 37 章 用 WPF 创建文档
- 第 38 章 Silverlight
- 第 39 章 Windows 窗体
- 第 40 章 核心 ASP.NET
- 第 41 章 ASP.NET 的功能
- 第 42 章 ASP.NET 动态数据和 MVC

第 35 章

核 心 WPF

本章内容:

- 用作基本绘图元素的形状和几何图形
- 利用转换功能实现缩放、旋转和倾斜
- 填充背景的画笔
- WPF 控件及其特性
- 用 WPF 面板定义布局
- 样式、模板和资源
- 触发器和 Visual State Manager
- 动画
- 3D

Windows Presentation Foundation(WPF)是为智能客户端应用程序创建 UI 的一个库。本章介绍 WPF 的重要概念,讨论大量不同的控件及其类别,说明如何用面板安排控件,如何使用样式、资源和模板定制外观,如何用触发器和动画添加动态操作,并介绍 WPF 的 3D 操作。

35.1 概述

WPF 的一个主要特性是设计人员和开发人员的工作很容易分开。设计人员的工作成果可以直接供开发人员使用。为此,必须理解 XAML。XAML 语法可参见第 27 章。

本章的第一个主题是概述 WPF 使用的类层次结构和类别,包括理解 XAML 的原则的额外信息。WPF 由几个包含了上千个类的程序集组成。因此用户可以在这些类中导航,查找需要的类,太致了解 WPF 中的类层次结构和名称空间。

35.1.1 名称空间

Windows 窗体类和 WPF 类很容易混淆。Windows 窗体类位于 `System.Windows.Forms` 名称空间,而 WPF 类位于 `System.Windows` 名称空间及其子名称空间中,但不位于 `System.Windows.Forms` 名称空间。Windows 窗体的 Button 类的全称是 `System.Windows.Forms.Button`,而用于 WPF 的 Button 类的全称是 `System.Windows.Controls.Button`。Windows 窗体参见第 39 章。

WPF 的名称空间及其功能如表 35-1 所述。

表 35-1

名 称 空 间	说 明
System.Windows	这是 WPF 的核心名称空间，其中包含 WPF 的核心类，如 Application 类、用于依赖对象的类、DependencyObject 和 DependencyProperty，所有 WPF 元素的基类 FrameworkElement
System.Windows.Annotations	这个名称空间中的类用于用户在应用程序数据上创建的注释和备注，它们与文档分开存储。System.Windows.Annotations.Storage 名称空间包含存储注释的类
System.Windows.Automation	System.Windows.Automation 名称空间用于自动完成 WPF 应用程序。它有几个子名称空间。System.Windows.Automation.Peers 名称空间提供用于自动化的 WPF 元素，如 ButtonAutomationPeer 和 CheckBoxAutomationPeer。如果创建自定义自动化提供程序，就需要 System.Windows.Automation.Provider 名称空间
System.Windows.Baml2006	这个名称空间是 .NET 4 新增的，包含 Baml2006Reader 类，它用于读取二进制标记语言，生成 XAML
System.Windows.Controls	这个名称空间包含所有 WPF 控件，如 Button、Border、Canvas、ComboBox、Expander、Slider、ToolTip、TreeView 等。System.Windows.Controls.Primitives 名称空间包含在复杂控件中使用的类，如 Popup、ScrollBar、StatusBar、TabPanel 等
System.Windows.Converters	这个名称空间包含用于数据转换的类。但它没有包含所有转换类。核心转换类在 System.Windows 名称空间中定义
System.Windows.Data	这个名称空间由 WPF 数据绑定使用。其中的一个重要类是 Binding 类，它用于定义 WPF 目标元素和 CLR 源之间的绑定。数据绑定参见第 36 章
System.Windows.Documents	在处理文档时，使用这个名称空间中的许多类很有帮助。内容元素 FixedDocument 和 FlowDocument 可以包含这个名称空间中的其他元素。System.Windows.Documents.Serialization 名称空间中的类可以将文档写入磁盘。这个名称空间中的类参见第 37 章
System.Windows.Ink	Windows Tablet PC 和 Ultra Mobile PC 使用得越来越多。在这些 PC 上，喷墨可以用于用户输入。System.Windows.Ink 名称空间包含处理喷墨输入的类
System.Windows.Input	这个名称空间包含的几个类用于命令处理、键盘输入、使用触笔等
System.Windows.Interop	这个名称空间中的类用于集成 Win32 和 WPF
System.Windows.Markup	用于 XAML 标记代码的帮助类位于这个名称空间
System.Windows.Media	要使用图像、音频和视频内容，可以使用这个名称空间中的类
System.Windows.Navigation	这个名称空间包含在窗口之间导航的类
System.Windows.Resources	这个名称空间包含资源的支持类
System.Windows.Shapes	UI 的核心类位于这个名称空间，如 Line、Ellipse、Rectangle 等
System.Windows.Threading	把 WPF 元素绑定到单个线程上。这个名称空间中的类可以处理多个线程，如 Dispatcher 类就属于这个名称空间

(续表)

名 称 空 间	说 明
System.Windows.Xps	XPS (XML Paper Specification, XML 纸张规范)是一个新的文档规范, Microsoft Word 也支持该规范。System.Windows.Xps、System.Windows.Xps.Packaging 和 System.Windows.Xps.Serialization 名称空间包含用于创建和流式传输 XPS 文档的类

35.1.2 类层次结构

WPF 包含上千个类, 有很深的层次结构。为了帮助理解类之间的关系, 图 35-1 列出了一些 WPF 类。表 35-2 描述了一些类及其功能。

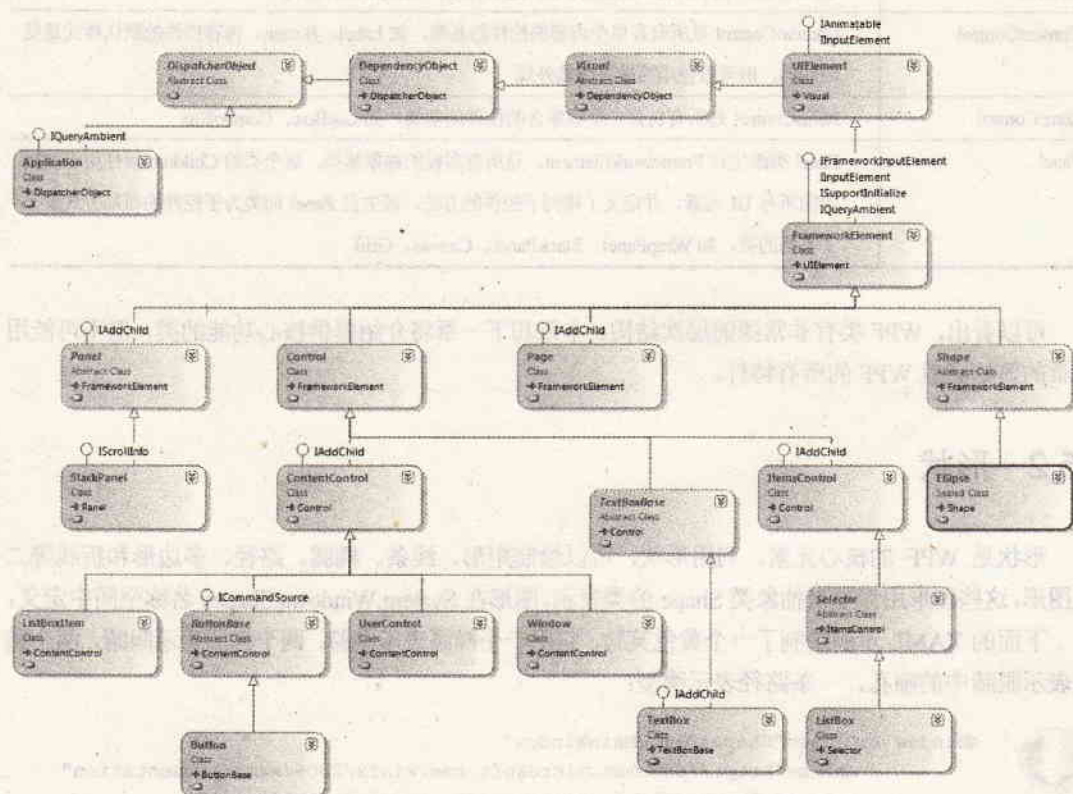


图 35-1

表 35-2

类	说 明
DispatcherObject	DispatcherObject 是一个抽象基类, 用于绑定到一个线程上的类。WPF 控件要求仅从创建线程中调用方法和属性。派生自 DispatcherObject 的类有一个关联的 Dispatcher 对象, 它可以用于切换线程
Application	在 WPF 应用程序中, 会创建 Application 类的一个实例。这个类实现单一模式, 用于访问应用程序的窗口、资源和属性

(续表)

类	说 明
DependencyObject	DependencyObject 是所有支持依赖属性的类的基类。依赖属性参见第 27 章
Visual	所有可见元素的基类是 Visual。这个类包含单击测试和转换等特性
UIElement	所有需要基本显示功能的 WPF 元素的抽象基类是 UIElement。这个类提供鼠标移动、拖放、按键单击的隧道和冒泡事件；提供可以由派生类重写的虚呈现方法；以及布局方法。WPF 不再使用 Window 句柄，这个类就可以用作 Window 句柄
FrameworkElement	FrameworkElement 派生自基类 UIElement，并实现由基类定义的方法的默认行为
Shape	Shape 是所有图形元素的基类，如 Line、Ellipse、Polygon、Rectangle
Control	Control 派生自 FrameworkElement，是所有用户交互元素的基类
ContentControl	ContentControl 是所有有单个内容的控件的基类，如 Label、Button。内容控件的默认样式是受限制的，但可以使用模板改变其外观
ItemsControl	ItemsControl 是所有包含一个项集合的控件的基类，如 ListBox、ComboBox
Panel	Panel 类派生自 FrameworkElement，是所有面板的抽象基类，这个类的 Children 属性用于面板中的所有 UI 元素，并定义了排列子控件的方法。派生自 Panel 的类为子控件的布局方式定义了不同的类，如 WrapPanel、StackPanel、Canvas、Grid

可以看出，WPF 类有非常深的层次结构。本章和下一章将介绍提供核心功能的类，但不可能用两章的篇幅涵盖 WPF 的所有特性。

35.2 形状

形状是 WPF 的核心元素。利用形状，可以绘制矩形、线条、椭圆、路径、多边形和折线等二维图形，这些图形用派生自抽象类 Shape 的类表示。图形在 System.Windows.Shapes 名称空间中定义。

下面的 XAML 示例绘制了一个黄色笑脸，它用一个椭圆表示笑脸，两个椭圆表示眼睛，两个椭圆表示眼睛中的瞳孔，一条路径表示嘴型：



可从
wrox.com
下载源代码

```
<Window x:Class="ShapesDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
  <Canvas>
    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100" Stroke="Blue"
             StrokeThickness="4" Fill="Yellow" />
    <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
          <GradientStop Offset="0.1" Color="DarkGreen" />
          <GradientStop Offset="0.7" Color="Transparent" />
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
```

```

        StrokeThickness="3" Fill="White" />
<Ellipse Canvas.Left="40" Canvas.Top="43" Width="6" Height="5" Fill="Black" />

<Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
        StrokeThickness="3" Fill="White" />
<Ellipse Canvas.Left="75" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
        Data="M 40,74 Q 57,95 80,74 " />
</Canvas>
</Window>

```

代码段 ShapesDemo/MainWindow.xaml

图 35-2 显示了这些 XAML 代码的结果。

无论是按钮还是线条、矩形等图形，所有这些 WPF 元素都可以通过编程来访问。把 Path 元素的 Name 或 x.Name 属性设置为 mouth，就可以用变量名 mouth 以编程方式访问这个元素：

```

<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
        Data="M 40,74 Q 57,95 80,74 " />

```

在 Path 元素的代码隐藏属性 Data 中，把 mouth 设置为一个新的图形。为了设置路径，Path 类支持 PathGeometry 和路径标记语法。字母 M 定义了路径的起点，字母 Q 指定了二次贝塞尔曲线的一个控制点和一个终点。运行应用程序，会看到如图 35-3 所示的窗口。



可从
wrox.com
下载源代码

```

public MainWindow()
{
    InitializeComponent();
    mouth.Data = Geometry.Parse("M 40,92 Q 57,75 80,92");
}

```

代码段 ShapesDemo/MainWindow.xaml.cs



图 35-2



图 35-3

表 35-3 描述了 System.Windows.Shapes 名称空间中可用的图形。

表 35-3

Shape 类	说 明
Line	可以在坐标(X1,Y1)到(X2,Y2)之间绘制一条线
Rectangle	使用 Rectangle 类，通过指定 Width 和 Height 可以绘制一个矩形
Ellipse	使用 Ellipse 类，可以绘制一个椭圆
Path	使用 Path 类可以绘制一系列直线和曲线。Data 属性是 Geometry 类型。还可以使用派生自基类 Geometry 的类绘制图形，或使用路径标记语法来定义图形

(续表)

Shape 类	说 明
Polygon	使用 Polygon 类可以绘制由线段连接而成的封闭图形。多边形由一系列赋予 Points 属性的 Point 对象定义
Polyline	类似于 Polygon 类，使用 Polyline 也可以绘制连接起来的线段。与多边形的区别是折线不一定是封闭图形

35.3 几何图形

其中一种形状Path使用Geometry来绘图。Geometry元素也可用于其他地方，如用于DrawingBrush。Geometry元素非常类似于形状。与Line、Ellipse和Rectangle形状一样，也有绘制这些形状的Geometry元素：LineGeometry、EllipseGeometry和RectangleGeometry。形状与几何图形有显著的区别。Shape是一个FrameworkElement，可以用于把UIElement用作其子元素的任意类。FrameworkElement派生自UIElement。形状会参与系统的布局，并呈现自身。而Geometry类不呈现自身，特性和系统开销也比Shape类少。Geometry类派生自Freezable基类，可以在多个线程中共享。

Path类使用Geometry来绘图。几何图形可以用Path的Data属性设置。可以设置的简单的几何图形元素有绘制椭圆的EllipseGeometry、绘制线条的LineGeometry和绘制矩形的RectangleGeometry。如下面的示例所示，使用CombineGeometry可以合并多个几何图形。

CombineGeometry有Geometry1和Geometry2属性，使用GeometryCombineMode可以合并它们，构成Union、Intersect、Xor和Exclude。Union会合并两个几何图形，Intersect只取两个几何图形都覆盖的区域，Xor与Intersect相反，显示一个几何图形覆盖的区域，但不显示两个几何图形都覆盖的区域。Exclude显示第一个几何图形减去第二个几何图形的区域。

下面的示例合并了一个EllipseGeometry和一个RectangleGeometry，生成并集，如图35-4所示。



可从
wrox.com
下载源代码

```
<Path Canvas.Top="0" Canvas.Left="250" Fill="Blue" Stroke="Black">
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="30,60 105 50" />
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

代码段 GeometryDemo/MainWindow.xaml

也可以使用段来创建几何图形。几何图形类PathGeometry使用段来绘图。下面的代码段使用BezierSegment和LineSegment元素来绘制一个红色的图形和一个绿色的图形，如图35-5所示。第一个BezierSegment在图形的起点(70,40)、终点(150,63)、控制点(90,37)和(130,46)之间绘制了一条贝



图 35-4

塞尔曲线。下面的 `LineSegment` 使用贝塞尔曲线的终点和(120,110)绘制了一条线段:

```
<Path Canvas.Left="0" Canvas.Top="0" Fill="Red" Stroke="Blue"
      StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="70,40" IsClosed="True">
            <PathFigure.Segments>
              <BezierSegment Point1="90,37" Point2="130,46"
                             Point3="150,63" />
              <LineSegment Point="120,110" />
              <BezierSegment Point1="100,95" Point2="70,90"
                             Point3="45,91" />
              <LineSegment Point="70,40" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>

<Path Canvas.Left="0" Canvas.Top="0" Fill="Green" Stroke="Blue"
      StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="160,70">
            <PathFigure.Segments>
              <BezierSegment Point1="175,85" Point2="200,99"
                             Point3="215,100" />
              <LineSegment Point="195,148" />
              <BezierSegment Point1="174,150" Point2="142,140"
                             Point3="129,115" />
              <LineSegment Point="160,70" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>
```

除了 `BezierSegment` 和 `LineSegment` 元素之外, 还可以使用 `ArcSegment` 元素在两点之间绘制椭圆弧。使用 `PolyLineSegment` 可以绘制一组线段, `PolyBezierSegment` 由多条贝塞尔曲线组成, `QuadraticBezierSegment` 创建一条二次贝塞尔曲线, `PolyQuadraticBezierSegment` 由多条二次贝塞尔曲线组成。

使用 `StreamGeometry` 可以进行高效的绘图。通过编程, 可以利用 `StreamGeometryContext` 类的成员创建线段、贝塞尔曲线和圆弧, 以定义图形。通过 XAML 可以使



图 35-5

用路径标记语法。路径标记语法可以与 Path 类的 Data 属性一起使用,来定义 StreamGeometry。特殊字符定义点的连接方式。在下面的示例中,M 标记起点,L 是到指定点的线条命令,Z 是闭合图形的闭合命令。图 35-6 显示了这个绘图操作的结果。路径标记语法允许使用更多的命令,如水平线(H)、垂直线(V)、三次贝塞尔曲线(C)、二次贝塞尔曲线(Q)、光滑的三次贝塞尔曲线(S)、光滑的二次贝塞尔曲线(T),以及椭圆弧(A):

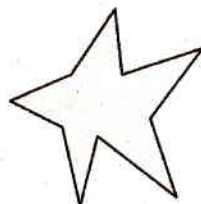


图 35-6

```
<Path Canvas.Left="0" Canvas.Top="200" Fill="Yellow" Stroke="Blue"
      StrokeThickness="2.5"
      Data="M 120,5 L 128,80 L 220,50 L 160,130 L 190,220 L 100,150 L 80,230 L
          60,140 L0,110 L70,80 Z" StrokeLineJoin="Round">
</Path>
```

35.4 变换

因为 WPF 基于矢量,所以可以重置每个元素的大小。基于矢量的图形现在可以缩放、旋转和倾斜。即使不需要手工计算位置,也可以进行单击测试(如移动鼠标和鼠标单击)。

给 Canvas 元素的 LayoutTransform 属性添加 ScaleTransform 元素,如下所示,把整个画布的内容在 X 和 Y 方向上放大两倍。



可从
wrox.com
下载源代码

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
</Canvas.LayoutTransform>
```

代码段 TransformationDemo/MainWindow.xaml

旋转与缩放的执行方式相同。使用 RotateTransform 元素,可以定义旋转的角度:

```
<Canvas.LayoutTransform>
  <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

对于倾斜,可以使用 SkewTransform 元素。此时可以指定 X 和 Y 方向的倾斜角度:

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

为了同时执行旋转和倾斜操作,可以定义一个 TransformGroup,它同时包含 RotateTransform 和 SkewTransform。也可以定义一个 MatrixTransform,其中 Matrix 元素指定了用于拉伸的 M11 和 M22 属性,以及用于倾斜的 M12 和 M21 属性。

```
<Canvas.LayoutTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix M11="0.8" M22="1.6" M12="1.3" M21="0.4" />
    </MatrixTransform.Matrix>
  </MatrixTransform>
</Canvas.LayoutTransform>
```

图 35-7 显示了这些变换的结果。这些图放在一个 `StackPanel` 中。从左到右, 第 1 个图重置了大小, 第 2 个图旋转了, 第 3 个图倾斜了, 第 4 个图使用一个矩阵进行变换。为了更容易看出它们的区别, 可以把 `Canvas` 元素的 `Background` 属性设置为不同的颜色。

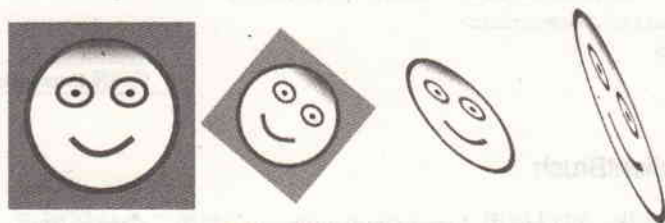


图 35-7

35.5 画笔

本节介绍如何使用 WPF 提供的画笔绘制背景和前景。本节将参考图 35-8, 它显示了在 `Button` 元素的一条 `Path` 和 `Background` 属性上使用各种画笔的效果。

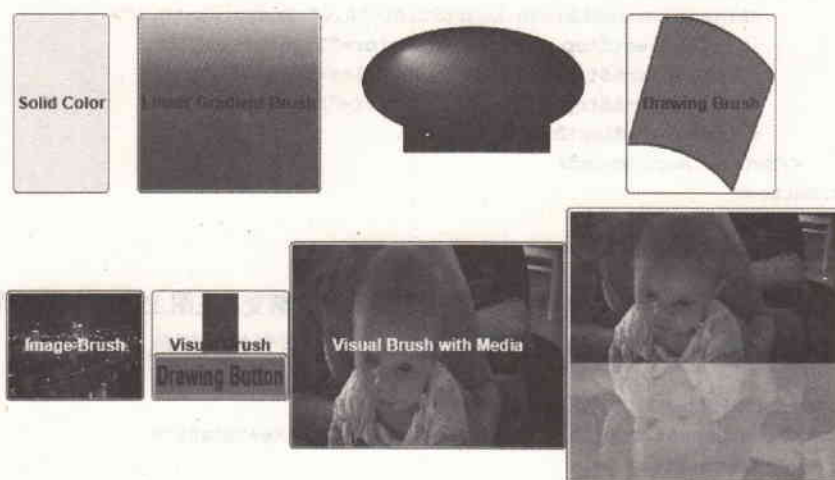


图 35-8

35.5.1 SolidColorBrush

图 35-8 中的第一个按钮使用了 `SolidColorBrush`, 顾名思义, 这支画笔使用纯色。全部区域用同一种颜色绘制。

把 `Background` 属性设置为定义纯色的字符串, 就可以定义纯色。使用 `BrushValueSerializer` 把该字符串转换为一个 `SolidColorBrush` 元素。

```
<Button Height="30" Background="PapayaWhip"> Solid Color</Button>
```

当然, 设置 **Background** 子元素, 把 **SolidColorBrush** 元素添加为它的内容, 也可以得到同样效果。应用程序中的第一个按钮把 **PapayaWhip** 用作纯背景色:



可从
wrox.com
下载源代码

```
<Button Content="Solid Color" Margin="10">
    <Button.Background>
        <SolidColorBrush Color="PapayaWhip" />
    </Button.Background>
</Button>
```

代码段 **BrushesDemo/MainWindow.xaml**

35.5.2 LinearGradientBrush

对于平滑的颜色变化, 可以使用 **LinearGradientBrush**, 如第二个按钮所示。这个画笔定义了 **StartPoint** 和 **EndPoint** 属性。使用这些属性可以为线性渐变指定二维坐标。默认的渐变方向是从(0,0)到(1,1)的对角线。定义其他值可以给渐变指定不同的方向。例如, **StartPoint** 指定为(0,0), **EndPoint** 指定为(0,1), 就得到了一个垂直渐变。**StartPoint** 不变, **EndPoint** 值指定为(1,0), 就得到了一个水平渐变。

通过该画笔的内容, 可以用 **GradientStop** 元素定义指定偏移位置的颜色值。在各个偏移位置之间, 颜色是平滑过渡的。

```
<Button Content="Linear Gradient Brush" Margin="10">
    <Button.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="LightGreen" />
            <GradientStop Offset="0.4" Color="Green" />
            <GradientStop Offset="1" Color="DarkGreen" />
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

35.5.3 RadialGradientBrush

使用 **RadialGradientBrush** 可以以放射方式产生平滑的颜色渐变。在图 35-8 中, 第 3 个元素 **Path** 使用了 **RadialGradientBrush**。该画笔定义了从 **GradientOrigin** 点开始的颜色。

```
<Canvas Width="200" Height="150">
    <Path Canvas.Top="0" Canvas.Left="20" Stroke="Black">
        <Path.Fill>
            <RadialGradientBrush GradientOrigin="0.2,0.2">
                <GradientStop Offset="0" Color="LightBlue" />
                <GradientStop Offset="0.6" Color="Blue" />
                <GradientStop Offset="1.0" Color="DarkBlue" />
            </RadialGradientBrush>
        </Path.Fill>
    <Path.Data>
        <CombinedGeometry GeometryCombineMode="Union">
            <CombinedGeometry.Geometry1>
                <EllipseGeometry Center="80,60" RadiusX="80"
                    RadiusY="40" />
            </CombinedGeometry.Geometry1>
            <CombinedGeometry.Geometry2>
```

```

        <RectangleGeometry Rect="30,60 105 50" />
    </CombinedGeometry.Geometry2>
</CombinedGeometry>
</Path.Data>
</Path>
</Canvas>

```

35.5.4 DrawingBrush

DrawingBrush 可以定义用画笔绘制的图形。用画笔绘制的图形在 **GeometryDrawing** 元素中定义。**Geometry** 属性中的 **GeometryGroup** 元素包含本章前面讨论的 **Geometry** 元素。

```

<Button Content="Drawing Brush" Margin="10" Padding="10">
    <Button.Background>
        <DrawingBrush>
            <DrawingBrush.Drawing>
                <GeometryDrawing Brush="Red">
                    <GeometryDrawing.Pen>
                        <Pen>
                            <Pen.Brush>
                                <SolidColorBrush Blue />
                            </Pen.Brush>
                        </Pen>
                    </GeometryDrawing.Pen>
                    <GeometryDrawing.Geometry>
                        <PathGeometry>
                            <PathGeometry.Figures>
                                <PathFigure StartPoint="70,40">
                                    <PathFigure.Segments>
                                        <BezierSegment Point1="90,37"
                                            Point2="130,46" Point3="150,63" />
                                        <LineSegment Point="120,110" />
                                        <BezierSegment Point1="100,95"
                                            Point2="70,90" Point3="45,91" />
                                        <LineSegment Point="70,40" />
                                    </PathFigure.Segments>
                                </PathFigure>
                            </PathGeometry.Figures>
                        </PathGeometry>
                    </GeometryDrawing.Geometry>
                </GeometryDrawing>
            </DrawingBrush.Drawing>
        </DrawingBrush>
    </Button.Background>
</Button>

```

35.5.5 ImageBrush

要把图像加载到画笔中，可以使用 **ImageBrush** 元素。通过这个元素，显示 **ImageSource** 属性定义的图像。图像可以从文件系统中访问，或者在程序集的资源中访问。在本例中，图像添加为程序集的一个资源，并通过程序集和资源名来引用：

```

<Button Content="Image Brush" Width="100" Height="80" Margin="5"
    Foreground="White">

```

```

<Button.Background>
    <ImageBrush ImageSource="/BrushesDemo;component/Budapest.jpg" />
</Button.Background>
</Button>

```

35.5.6 VisualBrush

VisualBrush 可以在画笔中使用其他 WPF 元素。下面给 **Visual** 属性添加一个 WPF 元素。图 35-8 中的第 6 个元素包含一个矩形和一个按钮。

```

<Button Content="Visual Brush" Width="100" Height="80">
    <Button.Background>
        <VisualBrush>
            <VisualBrush.Visual>
                <StackPanel Background="White">
                    <Rectangle Width="25" Height="25" Fill="Blue" />
                    <Button Content="Drawing Button" Background="Red" />
                </StackPanel>
            </VisualBrush.Visual>
        </VisualBrush>
    </Button.Background>
</Button>

```

可以给 **VisualBrush** 添加任意 **UIElement**。一个例子是可以使用 **MediaElement** 播放视频：

```

<Button Content="Visual Brush with Media" Width="200" Height="150"
    Foreground="White">
    <Button.Background>
        <VisualBrush>
            <VisualBrush.Visual>
                <MediaElement Source="./Stephanie.wmv" />
            </VisualBrush.Visual>
        </VisualBrush>
    </Button.Background>
</Button>

```

在 **VisualBrush** 中，还可以创建反射等有趣的效果。这里显示的按钮包含一个 **StackPanel**，它包含一个播放视频的 **MediaElement** 和一个 **Border**。**Border** 边框包含一个用 **VisualBrush** 填充的矩形。这支画笔定义了一个不透明值和一个变换。把 **Visual** 属性绑定到 **Border** 元素上。变换通过设置 **VisualBrush** 的 **RelativeTransform** 属性来完成。这个变换使用了相对坐标。把 **ScaleY** 设置为 -1，完成 Y 方向上的反射。**TranslateTransform** 在 Y 方向上移动变换，从而使反射效果位于原始对象的下面。图 35-8 中的第 8 个元素显示了其效果。



这里使用的数据绑定和 Binding 元素详见第 36 章。

```

<Button Width="200" Height="200" Foreground="White">
    <StackPanel>
        <MediaElement x:Name="reflected" Source="./Stephanie.wmv" />
        <Border Height="100">
            <Rectangle>

```

```

        <Rectangle.Fill>
            <VisualBrush Opacity="0.35" Stretch="None"
                Visual="{Binding ElementName=reflected}">
                <VisualBrush.RelativeTransform>
                    <TransformGroup>
                        <ScaleTransform ScaleX="1" ScaleY="-1" />
                        <TranslateTransform Y="1" />

                    </TransformGroup>
                </VisualBrush.RelativeTransform>
            </VisualBrush>
        </Rectangle.Fill>
    </Rectangle>
</Border>
</StackPanel>
</Button>

```

35.6 控件

可以对 WPF 使用上百个控件。为了更好地理解它们，我们把控件分为如下几组：

- 简单控件
- 内容控件
- 带标题的内容控件
- 项控件
- 带标题的项控件
- 修饰控件

35.6.1 简单控件

简单控件是没有 Content 属性的控件。例如，Button 类可以包含任意形状或任意元素，这对于简单控件没有问题。表 35-4 列出了简单控件及其功能。

表 35-4

简 单 控 件	说 明
PasswordBox	PasswordBox 控件用于输入密码。这个控件有用于输入密码的专用属性，例如，PasswordChar 属性定义了用户在输入密码时显示的字符，Password 属性可以访问输入的密码。PasswordChanged 事件在修改密码时立即调用
ScrollBar	ScrollBar 控件包含一个 Thumb，用户可以从 Thumb 中选择一个值。例如，如果文档在屏幕中放不下，就可以使用滚动条。一些控件包含滚动条，如果内容过多，就显示滚动条
ProgressBar	使用 ProgressBar 控件，可以指示时间较长的操作的进度
Slider	使用 Slider 控件，用户可以移动 Thumb，选择一个范围的值。ScrollBar、ProgressBar 和 Slider 派生自同一个基类 RangeBase
Textbox	Textbox 控件用于显示简单的无格式文本

(续表)

简单控件	说 明
RichTextbox	RichTextbox 控件通过 FlowDocument 类支持带格式的文本。RichTextBox 和 TextBox 派生自同一个基类 TextBoxBase
Calendar	Calendar 控件是.NET 4 新增的, 它可以显示年份、月份或 10 年。用户可以选择一个日期或日期范围
DatePicker	DatePicker 控件会打开 Calendar 屏幕, 供用户选择日期



尽管简单控件没有 Content 属性, 但通过定义模板, 完全可以定制这些控件的外观。模板详见本章后面的内容。

35.6.2 内容控件

ContentControl 有 Content 属性, 利用 Content 属性, 可以给控件添加任意内容。因为 Button 类派生自基类 ContentControl, 所以可以在这个控件中添加任意内容。在上面的例子中, 在 Button 类中有一个 Canvas 控件。表 35-5 列出了内容控件。

表 35-5

ContentControl 控件	说 明
Button RepeatButton ToggleButton CheckBox RadioButton	Button、RepeatButton、ToggleButton 和 GridViewColumnHeader 类派生自同一个基类 ButtonBase。所有这些按钮都响应 Click 事件。RepeatButton 类会重复引发 Click 事件, 直到释放按钮为止。ToggleButton 是 CheckBox 和 RadioButton 的基类。这些按钮有开关状态。CheckBox 可以由用户选择和取消选择, RadioButton 可以由用户选择。清除 RadioButton 的选择必须通过编程方式实现
Label	Label 类表示控件的文本标签。这个类也支持访问键, 如菜单命令
Frame	Frame 控件支持导航。使用 Navigate()方法可以导航到一个页面内容上。如果该内容是一个网页, 就使用浏览器控件来显示
ListBoxItem	ListBoxItem 是 ListBox 控件中的一项
StatusBarItem	StatusBarItem 是 StatusBar 控件中的一项
ScrollViewer	ScrollViewer 控件是一个包含滚动条的内容控件, 可以把任意内容放入这个控件中, 滚动条会在需要时显示
ToolTip	ToolTip 创建一个弹出窗口, 以显示控件的附加信息
UserControl	将 UserControl 类用作基类, 可以为创建自定义控件提供一种简单方式。但是, 基类 UserControl 不支持模板
Window	Window 类可以创建窗口和对话框。使用这个类, 会获得一个带有最小化/最大化/关闭按钮和系统菜单的框架。在显示对话框时, 可以使用 ShowDialog()方法, Show()方法会打开一个窗口
NavigationWindow	类 NavigationWindow 派生自 Window 类, 支持内容导航

只有一个 Frame 控件包含在下面 XAML 代码的因为把 Window 中。因为把 Source 属性设置为 <http://www.thinktecture.com>，所以 Frame 控件导航到这个网站上，如图 35-9 所示。

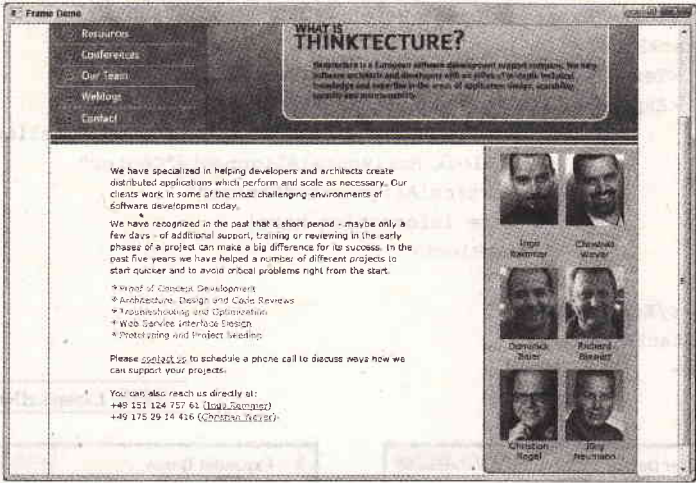


图 35-9



```
<Window x:Class="FrameDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Frame Demo" Height="240" Width="500">
    <Frame Source="http://www.thinktecture.com" />
</Window>
```

代码段 FrameDemo/MainWindow.xaml

35.6.3 带标题的内容控件

带标题的内容控件派生自 HeaderContentControl 基类。HeaderContentControl 类又派生自基类 ContentControl。HeaderContentControl 类的 Header 属性定义了标题的内容，HeaderTemplate 属性可以对标题进行完全的定制。派生自基类 HeaderContentControl 的控件如表 35-6 所示。

表 35-6

HeaderContentControl	说 明
Expander	使用 Expander 控件，可以创建一个带对话框的“高级”模式，它在默认情况下不显示所有的信息，只有用户展开它，才会显示更多的信息。在未展开模式下，只显示标题信息；在展开模式下，显示内容
GroupBox	GroupBox 控件提供了边框和标题来组合控件
TabItem	TabItem 控件是 TabControl 类中的项。TabItem 的 Header 属性定义了标题的内容，这些内容用 TabControl 的标签显示

Expander 控件的简单用法如下面的例子所示。把 Expander 控件的 Header 属性设置为 Click for more。这个文本用于显示扩展。这个控件的内容只有在控件展开时才显示。图 35-10 中的应用程序包含折叠的 Expander 控件，图 35-11 中的同一个应用程序包含展开的 Expander 控件。



```
<Window x:Class="ExpanderDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Expander Demo" Height="240" Width="500">
    <StackPanel>
        <TextBlock> Short information</TextBlock>
        <Expander Header="Additional Information">
            <Border Height="200" Width="200" Background="Yellow">
                <TextBlock HorizontalAlignment="Center"
                    VerticalAlignment="Center">
                    More information here!
                </TextBlock>
            </Border>
        </Expander>
    </StackPanel>
</Window>
```

代码段 ExpanderDemo/MainWindow.xaml

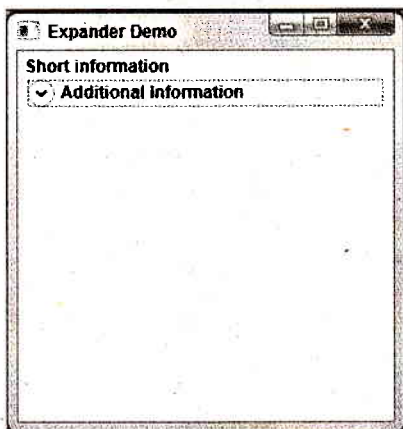


图 35-10

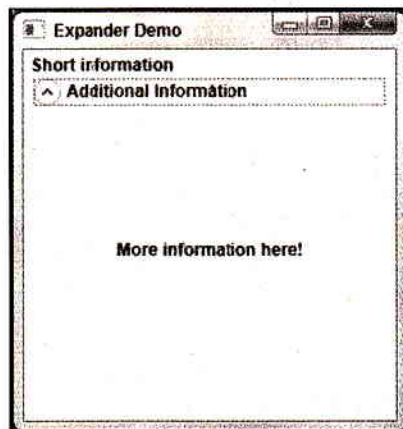


图 35-11



在展开 Expander 控件时, 如果要修改该控件的标题文本, 就可以创建一个触发器。触发器详见本章后面的内容。

35.6.4 项控件

ItemsControl 类包含一个可以用 Items 属性访问的数据项列表。派生自 ItemsControl 的类如表 35-7 所示。

表 35-7

ItemsControl	说 明
Menu	Menu 类和 ContextMenu 类派生自抽象基类 MenuBase。把 MenuItem 元素放在数据项列表和相关的命令中, 就可以给用户提
ContextMenu	提供菜单

(续表)

ItemsControl	说 明
StatusBar	StatusBar 控件通常显示在应用程序的底部，为用户提供状态信息。可以把 StatusBarItem 元素放在 StatusBar 列表中
TreeView	要分层显示数据项，可以使用 TreeView 控件
ListBox	ListBox、ComboBox 和 TabControl 都有相同的抽象基类 Selector。这个基类可以从列表中选择数据项。ListBox 显示列表中的数据项，ComboBox 有一个附带的 Button 控件，只有单击该按钮，才会显示数据项。在 TabControl 中，内容可以排列为表格形式
ComboBox	
TabControl	
DataGrid	DataGrid 控件是显示数据的可定制网格，这个控件是 .NET 4 新增的，详见下一章

35.6.5 带标题的项控件

HeaderItemsControl 是不仅包含数据项而且包含标题的控件的基类。HeaderItemsControl 类派生自 ItemsControl。

派生自 HeaderItemsControl 的类如表 35-8 所示。

表 35-8

HeaderedItemsControl	说 明
MenuItem	菜单类 Menu 和 ContextMenu 包含 MenuItem 类型的数据项。菜单项可以连接到命令上，因为 MenuItem 类实现了 ICommandSource 接口
TreeViewItem	TreeViewItem 类可以包含 TreeViewItem 类型的数据项
ToolBar	ToolBar 控件是一组控件(通常是 Button 和 Separator 元素)的容器。可以将 ToolBar 放在 ToolBarTray 中，它会重新排列 ToolBar 控件

35.6.6 修饰

给单个元素添加修饰可以使用 Decorator 类完成。Decorator 是一个基类，派生自它的类有 Border、Viewbox 和 BulletDecorator。主题元素如 ButtonChrome 和 ListBoxChrome 也是修饰器。

下面的例子说明了 Border、Viewbox 和 BulletDecorator 类，如图 35-12 所示。



图 35-12

Border 类给子元素四周添加边框，以修饰子元素。可以给子元素定义画笔和边框的宽度、背景、圆角半径和填充图案：



可从
wrox.com
下载源代码

```
<Border BorderBrush="Violet" BorderThickness="5.5">
  <Label>Label with a border</Label>
</Border>
```

代码段 DecorationsDemo/MainWindow.xaml

Viewbox 将其子元素拉伸并缩放到可用的空间中。**StretchDirection** 和 **Stretch** 属性专用于 **Viewbox** 的功能，它们允许设置子元素是否双向拉伸，以及是否保持宽高比：

```
<Viewbox StretchDirection="Both" Stretch="Uniform">
  <Label>Label with a viewbox</Label>
</Viewbox>
```

BulletDecorator 类用一个项目符号修饰其子元素。子元素可以是任意元素(在本例中是一个组合框)。同样，项目符号也可以是任意元素，本例使用一个 **Image** 元素，也可以使用任意 **UIElement**：

```
<BulletDecorator>
  <BulletDecorator.Bullet>
    <Image Width="25" Height="25" Margin="5" HorizontalAlignment="Center"
      VerticalAlignment="Center"
      Source="/DecorationsDemo;component/images/apple1.jpg" />
  </BulletDecorator.Bullet>
  <BulletDecorator.Child>
    <ComboBox Margin="25,5,5,5" Width="120" HorizontalAlignment="Left">
      <ComboBoxItem>Granny Smith</ComboBoxItem>
      <ComboBoxItem>Gravenstein</ComboBoxItem>
      <ComboBoxItem>Golden Delicious</ComboBoxItem>
      <ComboBoxItem>Braeburn</ComboBoxItem>
    </ComboBox>
  </BulletDecorator.Child>
</BulletDecorator>
```

35.7 布局

为了定义应用程序的布局，可以使用派生自 **Panel** 基类的类。这里讨论几个布局容器。布局容器要完成两个主要任务：测量和排列。在测量时，容器要求其子控件有合适的大小。因为控件的整体大小不一定合适，所以容器需要确定和排列其子控件的大小和位置。

35.7.1 StackPanel

Window 可以只包含一个元素，作为其内容。如果要在其中包含多个元素，就可以将 **StackPanel** 用作 **Window** 的一个子元素，并在 **StackPanel** 的内容中添加元素。**StackPanel** 是一个简单的容器控件，只能逐个地显示元素。**StackPanel** 的方向可以是水平或垂直。**ToolBarPanel** 类派生自 **StackPanel**。



可从
wrox.com
下载源代码

```
<Window x:Class="LayoutDemo.StackPanelWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="StackPanelWindow" Height="300" Width="300">
  <StackPanel Orientation="Vertical">
    <Label>Label</Label>
    <TextBox>TextBox</TextBox>
```

```

<CheckBox>CheckBox</CheckBox>
<CheckBox>CheckBox</CheckBox>
<ListBox>
    <ListBoxItem>ListBoxItem One</ListBoxItem>
    <ListBoxItem>ListBoxItem Two</ListBoxItem>
</ListBox>
<Button>Button</Button>
</StackPanel>
</Window>

```

代码段 LayoutDemo/StackPanelWindow.xaml

在图 35-13 中, 可以看到 StackPanel 垂直显示的子控件。

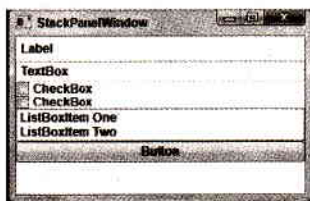


图 35-13

35.7.2 WrapPanel

WrapPanel 将子元素自左向右逐个地排列, 若一个水平行中放不下, 就排在下一行。面板的方向可以是水平或垂直。



可从
wrox.com
下载源代码

```

<Window x:Class="LayoutDemo.WrapPanelWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WrapPanelWindow" Height="300" Width="300">
    <WrapPanel>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
        <Button Width="100" Margin="5">Button</Button>
    </WrapPanel>
</Window>

```

代码段 LayoutDemo/WrapPanelWindow.xaml

图 35-14 显示了面板的排列结果。如果重新设置应用程序的大小, 按钮就会重新排列, 以便填满一行。

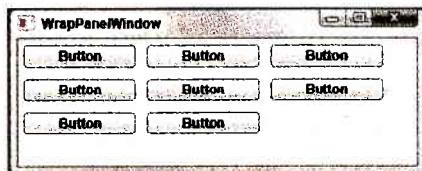


图 35-14

35.7.3 Canvas

Canvas 是一个允许显式指定控件位置的面板。它定义了相关的 Left、Right、Top 和 Bottom 属性, 这些属性可以由子元素在面板中定位时使用。



可从
wrox.com
下载源代码

```
<Window x:Class="LayoutDemo.CanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CanvasWindow" Height="300" Width="300">
    <Canvas Background="LightBlue">
        <Label Canvas.Top="30" Canvas.Left="20"> Enter here:</Label>
        <TextBox Canvas.Top="30" Canvas.Left="120" Width="100"/>
        <Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />
    </Canvas>
</Window>
```

代码段 LayoutDemo/CanvasWindow.xaml

图 35-15 显示了 Canvas 面板的结果, 其中定位了子元素 Label、TextBox 和 Button。

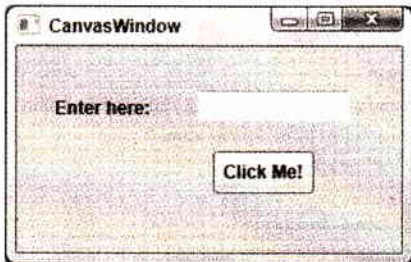


图 35-15

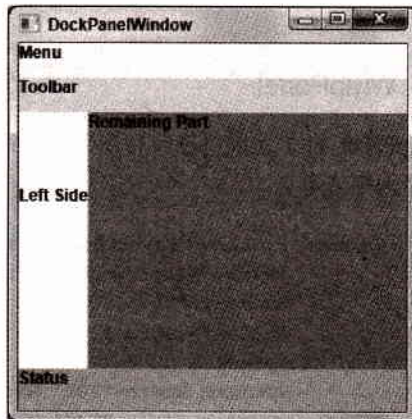


图 35-16

35.7.4 DockPanel

DockPanel 非常类似于 Windows 窗体的停靠功能。DockPanel 可以指定排列子控件的区域。DockPanel 定义了相关的 Dock 属性, 可以在控件的子控件中将它设置为 Left、Right、Top 和 Bottom。图 35-16 显示了排列在 DockPanel 中的带边框的文本块。为了便于区别, 为不同的区域指定了不同的颜色:



可从
wrox.com
下载源代码

```
<Window x:Class="LayoutDemo.DockPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DockPanelWindow" Height="300" Width="300">
    <DockPanel>
        <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
            <TextBlock>Menu</TextBlock>
        </Border>
        <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
```

```

        <TextBlock>Toolbar</TextBlock>
    </Border>
    <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
        <TextBlock>Status</TextBlock>
    </Border>
    <Border Height="80" Background="Azure" DockPanel.Dock="Left">
        <TextBlock>Left Side</TextBlock>
    </Border>
    <Border Background="HotPink">
        <TextBlock>Remaining Part</TextBlock>
    </Border>
</DockPanel>
</Window>

```

代码段 LayoutDemo/DockPanelWindow.xaml

35.7.5 Grid

使用 Grid, 可以在行和列中排列控件。对于每一列, 可以指定一个 ColumnDefinition; 对于每一行, 可以指定一个 RowDefinition。下面的示例代码显示两列和三行。在每一列和每一行中, 都可以指定宽度或高度。ColumnDefinition 有一个 Width 依赖属性, RowDefinition 有一个 Height 依赖属性。可以以像素、厘米、英寸或点为单位定义高度和宽度, 或者把它们设置为 Auto, 根据内容来确定其大小。Grid 还允许根据具体情况指定大小, 即根据可用的空间以及与其他行和列的相对位置, 计算行和列的空间。在为列提供可用空间时, 可以将 Width 属性设置为 “*”。要使某一列的空间是另一列的两倍, 应指定 “2*”。下面的示例代码定义了两列和三行, 但没有定义列定义和行定义的其他设置, 默认使用根据具体情况指定大小的设置。

这个 Grid 包含几个 Label 和 TextBox 控件。因为这些控件的父控件是 Grid, 所以可以设置相关的 Column、ColumnSpan、Row 和 RowSpan 属性。



可从
wrox.com
下载源代码

```

<Window x:Class="LayoutDemo.GridWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="GridWindow" Height="300" Width="300">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
            VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title" />
        <Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center"
            Content="Firstname:" Margin="10" />
        <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />
        <Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center"
            Content="Lastname:" Margin="10" />
        <TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />
    </Grid>
</Window>

```

```
</Grid>
</Window>
```

代码段 LayoutDemo/GridWindow.xaml

在 Grid 中排列控件的结果如图 35-17 所示。为了便于看到列和行，把 ShowGridLines 属性设置为 true。

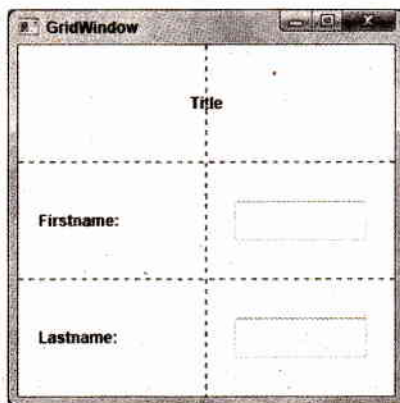


图 35-17



要使 Grid 的每个单元格有相同的尺寸，可以使用 UniformGrid 类。

35.8 样式和资源

设置 Button 元素的 FontSize 和 Background 属性，就可以定义 WPF 元素的外观，如 Button 元素所示：



可从
wrox.com
下载源代码

```
<Button Width="150" FontSize="12" Background="AliceBlue" Content="Click Me!" />
```

代码段 StylesAndResources/MainWindow.xaml

除了定义每个元素的外观之外，还可以定义用资源存储的样式。为了完全定制控件的外观，可以使用模板，再把它们存储到资源中。

35.8.1 样式

控件的 Style 属性可以赋予包含 Setter 相关联的 Style 元素。Setter 元素定义 Property 和 Value 属性，并给指定的属性设置一个值。这里设置 Background、FontSize 和 FontWeight 属性。把 Style 设置为 TargetType Button，以便可以直接访问 Button 的属性。如果没有设置样式的 TargetType，就可以通过 Button.Background、Button.FontSize 访问属性。如果需要设置不同元素类型的属性，这就很重要。



可从
wrox.com
下载源代码

```
<Button Width="150" Content="Click Me!">
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
</Button>
```

代码段 StylesAndResources/MainWindow.xaml

直接通过 Button 元素设置 Style 对样式的共享没有什么帮助。样式可以放在资源中。在资源中，可以把样式赋予指定的元素，把一个样式赋予某一类型的所有元素，或者为该样式使用一个键。要把样式赋予某一类型的所有元素，可使用 Style 的 TargetType 属性，指定 x:Type 标记扩展{x:Type Button}，从而将样式赋予一个按钮。要定义需要引用的样式，必须设置 x:Key:

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    <Setter Property="Background" Value="LemonChiffon" />
    <Setter Property="FontSize" Value="18" />
  </Style>
  <Style x:Key="ButtonStyle">
    <Setter Property="Button.Background" Value="Red" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.FontSize" Value="18" />
  </Style>
</Window.Resources>
```

在下面的 XAML 代码中第一个按钮没有用元素属性定义样式，而是使用为 Button 类型定义的样式。通过下一个按钮，把 Style 属性用 StaticResource 标记扩展设置为{StaticResource ButtonStyle}，其中因为 ButtonStyle 指定了前面定义的样式资源的键值，所以该按钮的背景为红色，前景是白色。

```
<Button Width="200" Content="Uses named style"
  Style="{StaticResource ButtonStyle}" Margin="3" />
```

除了把按钮的 Background 设置为单个值之外，还可以将 Background 属性设置为定义了渐变色的 LinearGradientBrush，如下所示：

```
<Style x:Key="FancyButtonStyle">
  <Setter Property="Button.FontSize" Value="22" />
  <Setter Property="Button.Foreground" Value="White" />
  <Setter Property="Button.Background">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0.0" Color="LightCyan" />
        <GradientStop Offset="0.14" Color="Cyan" />
        <GradientStop Offset="0.7" Color="DarkCyan" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

本例中下一个按钮的样式采用青色的线性渐变效果：

```
<Button Width="200" Content="Fancy button style"
        Style="{StaticResource FancyButtonStyle}" Margin="3" />
```

样式提供了一种继承方式。一个样式可以基于另一个样式。下面的 `AnotherButtonStyle` 样式基于 `FancyButtonStyle` 样式。它使用该样式定义的所有设置，且通过 `BasedOn` 属性引用，但 `Foreground` 属性除外，它设置为 `LinearGradientBrush`：

```
<Style x:Key="AnotherButtonStyle" BasedOn="{StaticResource FancyButtonStyle}"
        TargetType="Button">
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0.2" Color="White" />
                <GradientStop Offset="0.5" Color="LightYellow" />
                <GradientStop Offset="0.9" Color="Orange" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

最后一个按钮应用了 `AnotherButtonStyle`：

```
<Button Width="200" Content="Style inheritance"
        Style="{StaticResource AnotherButtonStyle}" Margin="3" />
```

图 35-18 显示了所有这些按钮的效果。



图 35-18

35.8.2 资源

从样式示例可以看出，样式通常存储在资源中。可以在资源中定义任意可冻结的元素。例如，前面为按钮的背景样式创建了画笔，它本身就可以定义为一个资源，这样就可以在需要画笔的地方使用它。

下面的示例在 `StackPanel` 资源中定义一个 `LinearGradientBrush`，它的键名是 `MyGradientBrush`。`Button1` 使用 `StaticResource` 标记扩展将 `Background` 属性赋予 `MyGradientBrush` 资源。图 35-19 显示了这段 XAML 代码的结果。



可从
wrox.com
下载源代码

```
<StackPanel x:Name="myContainer">
    <StackPanel.Resources>
        <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0,0"
            EndPoint="0.3,1">
            <GradientStop Offset="0.0" Color="LightCyan" />
            <GradientStop Offset="0.14" Color="Cyan" />
            <GradientStop Offset="0.7" Color="DarkCyan" />
        </LinearGradientBrush>
    </StackPanel.Resources>
</StackPanel>
```

```

</StackPanel.Resources>
<Button Width="200" Height="50" Foreground="White" Margin="5"
        Background="{StaticResource MyGradientBrush}" Content="Click Me!" />
</StackPanel>

```

代码段 StylesAndResources/MainWindow.xaml

这里,资源用 StackPanel 定义。在上面的例子中,资源用 Window 元素定义。基类 FrameworkElement 定义 ResourceDictionary 类型的 Resources 属性。这就是资源可以用派生自 FrameworkElement 的所有类(任意 WPF 元素)来定义的原因。



图 35-19

资源按层次结构来搜索。如果用 Window 元素定义资源,它就会应用于 Window 的所有子元素。如果 Window 包含一个 Grid,且该 Grid 包含一个 StackPanel,同时如果资源是用 StackPanel 定义的,该资源就会应用于 StackPanel 中的所有控件。如果 StackPanel 包含一个按钮,但只用该按钮定义资源,这个样式就只对该按钮有效。



对于层次结构,需要注意是否为样式使用了没有 Key 的 TargetType。如果用 Canvas 元素定义一个资源,并把样式的 TargetType 设置为应用于 TextBox 元素,该样式就会应用于 Canvas 中的所有 TextBox 元素。如果 Canvas 中有一个 ListBox,该样式甚至会应用于 ListBox 包含的 TextBox 元素。

如果需要将同一个样式应用于多个窗口,就可以用应用程序定义样式。在一个 Visual Studio WPF 项目中,创建 App.xaml 文件,以定义应用程序的全局资源。应用程序样式对其中的每个窗口都有效;每个元素都可以访问用应用程序定义的资源。如果通过父窗口找不到资源,就可以通过 Application 继续搜索资源。



可从
wrox.com
下载源代码

```

<Application x:Class="StylesAndResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <Application.Resources>

  </Application.Resources>
</Application>

```

代码段 StylesAndResources/App.xaml

35.8.3 系统资源

还有许多系统范围内的颜色和字体资源,它们可用于所有应用程序。这些资源用 SystemColors、SystemFonts 和 SystemParameters 类定义。

- 使用 SystemColors 类可以获得边框、控件、桌面和窗口的颜色设置,例如,ActiveBorderColor、ControlBrush、DesktopColor、WindowColor 和 WindowBrush 等。
- SystemFonts 类返回菜单、状态栏、消息框等的字体设置,例如,CaptionFont、DialogFont、MenuFont、MessageBoxFont 和 StatusFont 等。

- `SystemParameters` 类设置菜单按钮、光标、图标、边框、标题、时间信息、键盘设置的大小。例如, `BorderWidth`、`CaptionHeight`、`CaptionWidth`、`MenuButtonWidth`、`MenuPopupAnimation`、`MenuShowDelay`、`SmallIconHeight` 和 `SmallIconWidth` 等。

35.8.4 从代码中访问资源

要从代码隐藏中访问资源, 因为基类 `FrameworkElement` 实现 `FindResource()` 方法, 所以可以通过每个 WPF 对象调用 `FindResource()` 方法。

为此, `button1` 没有指定背景, 但将 `Click` 事件赋予 `button1_Click()` 方法。



可从
wrox.com
下载源代码

```
<Button Name="button1" Width="220" Height="50" Margin="5" Click="button1_Click">
    Apply Resource Programmatically
</Button>
```

代码段 `StylesAndResources/ResourceDemo.xaml`

通过 `button1_Click()` 方法的实现方式, 给单击的按钮调用 `FindResource()` 方法。然后按照层次结构搜索 `MyGradientBrush` 资源, 将该画笔应用于控件的 `Background` 属性。前面在 `StackPanel` 的资源中创建了资源 `MyGradientBrush`:



可从
wrox.com
下载源代码

```
public void button1_Click(object sender, RoutedEventArgs e)
{
    Control ctrl = sender as Control;
    ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

代码段 `StylesAndResources/ResourceDemo.xaml.cs`



如果 `FindResource()` 方法没有找到资源键, 就会抛出一个异常。如果不知道资源是否可用, 就可以使用 `TryFindResource()` 方法来替代。如果找不到资源, `TryFindResource()` 方法就返回 `null`。

35.8.5 动态资源

通过 `StaticResource` 标记扩展, 在加载期间搜索资源。如果在运行程序的过程中改变了资源, 就应使用 `DynamicResource` 标记扩展。

下面的例子使用与前面定义的相同资源。前面的示例使用 `StaticResource`, 而这个按钮通过 `DynamicResource` 标记扩展使用 `DynamicResource`。这个按钮的事件处理程序以编程方式改变资源。把处理程序方法 `button2_Click` 赋予 `Click` 事件处理程序。



可从
wrox.com
下载源代码

```
<Button Name="button2" Width="200" Height="50" Foreground="White" Margin="5"
    Background="{DynamicResource MyGradientBrush}" Content="Change Resource"
    Click="button2_Click" />
```

代码段 `StylesAndResources/ResourceDemo.xaml`

`button2_Click()` 方法的实现代码清除了 `StackPanel` 的资源, 并用相同的 `MyGradientBrush` 名称添加了一个新资源。这个新资源非常类似于在 XAML 代码中定义的资源, 它只定义了不同的颜色。



可从
wrox.com
下载源代码

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    myContainer.Resources.Clear();
    var brush = new LinearGradientBrush
    {
        StartPoint = new Point(0, 0),
        EndPoint = new Point(0, 1)
    };

    brush.GradientStops = new GradientStopCollection()
    {
        new GradientStop(Colors.White, 0.0),
        new GradientStop(Colors.Yellow, 0.14),
        new GradientStop(Colors.YellowGreen, 0.7)
    };
    myContainer.Resources.Add("MyGradientBrush", brush);
}
```

代码段 StylesAndResources/ResourceDemo.xaml.cs

运行应用程序时,单击 **Change Resource** 按钮,可以动态地更改资源。使用通过动态资源定义的按钮会获得动态创建的资源,而用静态资源定义的按钮看起来与以前一样。



DynamicResource 需要的性能资源比 **StaticResource** 更多,因为其资源总是在需要时加载。只有需要在运行期间进行修改,才给资源使用 **DynamicResource**。

35.8.6 资源字典

如果相同的资源可用于不同的应用程序,把资源放在一个资源字典中就比较有效。使用资源字典,可以在多个应用程序之间共享文件,也可以把资源字典放在一个程序集中,供应用程序共享。

要共享程序集中的资源字典,应创建一个库。可以把资源字典文件(这里是 **Dictionary1.xaml**)添加到程序集中。这个文件的构建动作必须设置为 **Resource**,从而把它作为资源添加到程序集中。

Dictionary1.xaml 定义了两个资源:一个包含 **CyanGradientBrush** 键的 **LinearGradientBrush**;另一个是用于按钮的样式,它可以通过 **PinkButtonStyle** 键来引用:



可从
wrox.com
下载源代码

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="CyanGradientBrush" StartPoint="0,0" EndPoint="0.3,1">
        <GradientStop Offset="0.0" Color="LightCyan" />
        <GradientStop Offset="0.14" Color="Cyan" />
        <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>

    <Style x:Key="PinkButtonStyle" TargetType="Button">
        <Setter Property="FontSize" Value="22" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="Background">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Offset="0.0" Color="Pink" />
                    <GradientStop Offset="0.3" Color="DeepPink" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</ResourceDictionary>
```

```

        <GradientStop Offset="0.9" Color="DarkOrchid" />
    </LinearGradientBrush>
</Setter.Value>
</Setter>
</Style>
</ResourceDictionary>

```

代码段 ResourcesLib/Dictionary1.xaml

对于目标项目，需要引用这个库，并把资源字典添加到这个字典中。通过 `ResourceDictionary` 的 `MergedDictionaries` 属性，可以使用添加进来的多个资源字典文件。可以把一个资源字典列表添加到合并的字典中。使用 `ResourceDictionary` 的 `Source` 属性，可以引用字典。在引用时，使用包 URI 语法。包 URI 语法可以指定为绝对的，其中 URI 以 `pack://` 开头，也可以指定为相对的，如本例所示。使用相对语法，包含字典的引用程序集 `ResourceLib` 跟在 `/` 的后面，其后是 `component`。Component 表示，该字典作为程序集中的一个资源包含在内。之后添加字典文件的名称 `Dictionary1.xaml`。如果把字典添加到子文件夹中，则还必须声明子文件夹名。



可从
wrox.com
下载源代码

```

<Application x:Class="StylesAndResources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="/ResourceLib;component/Dictionary1.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>

```

代码段 StylesAndResources/App.xaml

现在可以像本地资源那样使用引用程序集中的资源了：



可从
wrox.com
下载源代码

```

<Button Width="300" Height="50" Style="{StaticResource PinkButtonStyle}"
    Content="Referenced Resource" />

```

代码段 StylesAndResources/ResourceDemo.xaml

35.9 触发器

使用触发器，可以动态地更改控件的外观，因为一些事件或属性值改变了。例如，用户在按钮上移动鼠标，按钮就会改变其外观。通常，这必须在 C# 代码中实现。使用 WPF，也可以用 XAML 实现，而这只会影响 UI。

XAML 有几个触发器。属性触发器在属性值改变时激活。多触发器基于多个属性值。事件触发器在事件发生时激活。数据触发器在绑定的数据改变时激活。本节讨论属性触发器、多触发器和数据触发器。事件触发器在后面与动画一起论述。

35.9.1 属性触发器

Style 类有一个 Triggers 属性,通过它可以指定属性触发器。下面的示例将一个 Button 元素放在一个 Grid 面板中。利用 Window 资源定义 Button 元素的默认样式。这个样式指定,将 Background 属性设置为 LightBlue,将 FontSize 属性设置为 17。这是应用程序启动时 Button 元素的样式。使用触发器可以改变控件的样式。触发器在 Style.Triggers 元素中用 Trigger 元素定义。将一个触发器赋予 IsMouseOver 属性,另一个触发器赋予 IsPressed 属性。这两个属性通过应用了样式的 Button 类定义。如果 IsMouseOver 属性的值是 true,就会激活触发器,将 Foreground 属性设置为 Red,将 FontSize 属性设置为 22。如果按下该按钮,IsPressed 属性就是 true,激活第二个触发器,并将 TextBox 的 Foreground 属性设置为 Yellow。



如果把 IsPressed 属性设置为 true, IsMouseOver 属性也就是 true。按下该按钮也需要把鼠标放在按钮上。按下该按钮会激活 IsMouseOver 属性触发器,并改变属性。这里触发器的激活顺序很重要。如果 IsPressed 属性触发器在 IsMouseOver 属性触发器之前激活,IsMouseOver 属性触发器就会覆盖 IsPressed 属性触发器设置的值。



可从
wrox.com
下载源代码

```
<Window x:Class="TriggerDemo.PropertyTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PropertyTriggerWindow" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Background" Value="LightBlue" />
            <Setter Property="FontSize" Value="17" />
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="Red" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
                <Trigger Property="IsPressed" Value="True">
                    <Setter Property="Foreground" Value="Yellow" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <Grid>
        <Button Width="200" Height="30" Content="Click me!" />
    </Grid>
</Window>
```

代码段 TriggerDemo/PropertyTriggerWindow.xaml

当激活触发器的原因不再有效时,就不必将属性值重置为原始值。例如,不必定义 IsMouseOver=true 和 IsMouseOver=false 的触发器。只要激活触发器的原因不再有效,触发器操作进行的修改就会自动重置为原始值。

Click me!

图 35-20

图 35-20 显示了触发器示例应用程序，其中，鼠标指向按钮时，按钮的前景和字体大小就会不同于其原始值。



使用属性触发器，很容易改变控件的外观、字体、颜色、不透明度等。在鼠标滑过控件时，键盘设置焦点时——都不需要编写任何代码。

Trigger 类定义了表 35-9 中的属性，以指定触发器操作。

表 35-9

Trigger 属性	说 明
Property	使用属性触发器, Property 和 Value 属性用于指定触发器的激活时间, 例如, Property = "IsMouseOver",
Value	Value = "True"
Setters	一旦激活触发器, 就可以使用 Setters 定义一个 Setter 元素集合, 来改变属性值. Setter 类定义 Property、TargetName 和 Value 属性, 以修改对象属性
EnterActions	除了定义 Setters 之外, 还可以定义 EnterActions 和 ExitActions. 使用这两个属性, 可以定义一个
ExitActions	TriggerAction 元素集合. EnterActions 在启动触发器时激活(此时通过属性触发器应用 Property/Value 组合). ExitActions 在触发器结束之前激活(此时不再应用 Property/Value 组合). 用这些操作指定的触发器操作派生自基类 TriggerAction, 如 SoundPlayerAction 和 BeginStoryboard. 使用 SoundPlayerAction 基类可以开始播放声音, BeginStoryboard 基类用于动画, 详见本章后面的内容

35.9.2 多触发器

属性的值变化时，就会激活属性触发器，如果因为两个或多个属性有特定的值，而需要设置触发器，就可以使用 MultiTrigger。

MultiTrigger 有一个 Conditions 属性，可以在其中设置属性的有效值。它还有一个 Setters 属性，可以在其中指定需要设置的属性。在下面的示例中，给 TextBox 元素定义了一个样式，如果 IsEnabled 属性是 True, Text 属性的值是 Test, 就应用触发器。如果应用这两个触发器，就把 TextBox 的 Foreground 属性设置为 Red:



可从
wrox.com
下载源代码

```
<Window x:Class="TriggerDemo.MultiTriggerWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MultiTriggerWindow" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="TextBox">
      <Style.Triggers>
        <MultiTrigger>
          <MultiTrigger.Conditions>
            <Condition Property="IsEnabled" Value="True" />
            <Condition Property="Text" Value="Test" />
          </MultiTrigger.Conditions>
          <MultiTrigger.Setters>
            <Setter Property="Foreground" Value="Red" />
          </MultiTrigger.Setters>
        </MultiTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
</Window>
```

```

        </MultiTrigger>
    </Style.Triggers>
</Style>
</Window.Resources>
<Grid>
    <TextBox />
</Grid>
</Window>

```

代码段 TriggerDemo/MultiTriggerWindow.xaml

35.9.3 数据触发器

如果绑定到控件上的数据满足指定的条件，就激活数据触发器。下面的例子使用 `Book` 类，它根据图书的出版社显示不同的内容。

`Book` 类定义 `Title` 和 `Publisher` 属性，还重载 `ToString()` 方法：



可从
wrox.com
下载源代码

```

public class Book
{
    public string Title { get; set; }
    public string Publisher { get; set; }

    public override string ToString()
    {
        return Title;
    }
}

```

代码段 TriggerDemo/Book.cs

在 XAML 代码中，给 `ListBoxItem` 元素指定了一个样式。该样式包含 `DataTrigger` 元素，它绑定到用于列表项的类的 `Publisher` 属性上。如果 `Publisher` 属性的值是 `Wrox Press`，`Background` 就设置为 `Red`。对于 `Dummies` 和 `Sybox` 出版社，把 `Background` 分别设置为 `Yellow` 和 `LightBlue`：



可从
wrox.com
下载源代码

```

<Window x:Class="TriggerDemo.DataTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Data Trigger Window" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="ListBoxItem">
            <Style.Triggers>
                <DataTrigger Binding="{Binding Path=Publisher}" Value="Wrox Press">
                    <Setter Property="Background" Value="Red" />
                </DataTrigger>
                <DataTrigger Binding="{Binding Path=Publisher}" Value="Dummies">
                    <Setter Property="Background" Value="Yellow" />
                </DataTrigger>
                <DataTrigger Binding="{Binding Path=Publisher}" Value="Sybox">
                    <Setter Property="Background" Value="LightBlue" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <Grid>
        <ListBox x:Name="list1" />
    </Grid>

```

```
</Grid>
</Window>
```

代码段 TriggerDemo/DataTriggerWindow.xaml

在代码隐藏中，把列表 list1 初始化为包含几个 Book 对象：



可从
wrox.com
下载源代码

```
public DataTriggerWindow()
{
    InitializeComponent();
    list1.Items.Add(new Book
    {
        Title = "Professional C# 4.0",
        Publisher = "Wrox Press"
    });
    list1.Items.Add(new Book
    {
        Title = "C# 2008 for Dummies",
        Publisher = "Dummies"
    });
    list1.Items.Add(new Book
    {
        Title = "Mastering Integrated HTML and CSS",
        Publisher = "Sybex"
    });
}
```

代码段 TriggerDemo/DataTriggerWindow.xaml.cs

运行应用程序，ListBoxItem 元素就会根据 Publisher 的值进行格式化，如图 35-21 所示。

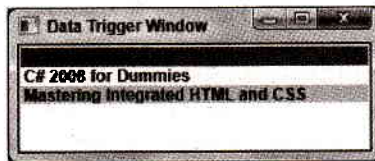


图 35-21

使用 DataTrigger 时，必须给 MultiDataTrigger 设置多个属性(类似于 Trigger 和 MultiTrigger)。

35.10 模板

本章前面介绍过，Button 控件可以包含任何内容，如简单的文本，还可以给按钮添加 Canvas 元素，Canvas 元素可以包含形状。也可以给按钮添加 Grid 或视频。然而，按钮还可以完成更多的操作。

控件的外观及其功能在 WPF 中是完全分离的。虽然按钮有默认的外观，但可以用模板完全定制其外观。

如图 35-10 所示，WPF 提供了几个模板类型，它们派生自基类 FrameworkTemplate。

表 35-10

模 板 类 型	说 明
ControlTemplate	使用 ControlTemplate 可以指定控件的可视化结构, 重新设计其外观
ItemsPanelTemplate	对于 ItemsControl, 可以赋予一个 ItemsPanelTemplate, 以指定其项的布局。每个 ItemsControl 都有一个默认的 ItemsPanelTemplate。MenuItem 使用 WrapPanel, StatusBar 使用 DockPanel, ListBox 使用 VirtualizingStackPanel
DataTemplate	DataTemplates 非常适用于对象的图形表示。给列表框指定样式, 默认情况下, 列表框中的项根据 ToString() 方法的输出来显示。应用 DataTemplate, 可以重写其操作, 定义项的自定义表示
HierarchicalDataTemplate	HierarchicalDataTemplate 用于排列对象的树形结构。这个控件支持 HeaderedItemsControls, 如 TreeViewItem 和 MenuItem

35.10.1 控件模板

本章前面介绍了如何给控件的属性定义样式。如果设置控件的简单属性得不到需要的外观, 就可以修改 Template 属性。使用 Template 属性可以定制控件的整体外观。

下面的例子说明了定制按钮的过程, 后面逐步地说明了列表框的定制, 以便显示出改变的中间结果。

Button 类型的定制在一个单独的资源字典文件 Styles.xaml 中进行。这里定义了键名为 RoundedGelButton 的样式。

GelButton 样式设置 Background、Height、Foreground、Margin 和 Template 属性。Template 属性是这个样式中最有趣的部分, 它指定一个仅包含一行一列的网格。

在这个单元格中, 有一个名为 GelBackground 的椭圆。这个椭圆给笔触设置了一个线性渐变画笔。包围矩形的笔触非常细, 因为把 StrokeThickness 设置为 0.5。

因为第二个椭圆 GelShine 比较小, 其尺寸由 Margin 属性定义, 所以在第一个椭圆内部是可见的。因为其笔触是透明的, 所以该椭圆没有边框。这个椭圆使用一个线性渐变填充画笔, 从部分透明的浅色变为完全透明, 这使椭圆具有“亦真亦幻”的效果。



可从
wrox.com
下载源代码

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Style x:Key="RoundedGelButton" TargetType="Button">
        <Setter Property="Width" Value="100" />
        <Setter Property="Height" Value="100" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
                            <Ellipse.Stroke>
                                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                    <GradientStop Offset="0" Color="#ff7e7e7e" />
                                    <GradientStop Offset="1" Color="Black" />
                                </LinearGradientBrush>
                            </Ellipse.Stroke>
                        </Ellipse>
                    </Grid>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

```

        <Ellipse Margin="15,5,15,50">
            <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Offset="0" Color="#aaffffff" />
                    <GradientStop Offset="1" Color="Transparent" />
                </LinearGradientBrush>
            </Ellipse.Fill>
        </Ellipse>
    </Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</ResourceDictionary>

```

代码段 TemplateDemo/Styles.xaml

从 app.xaml 文件中, 引用资源字典, 如下所示:



```

<Application x:Class="TemplateDemo.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary Source="Styles.xaml" />
    </Application.Resources>
</Application>

```

代码段 TemplateDemo/App.xaml

现在可以把 Button 控件关联到样式上。按钮的新外观如图 35-22 所示。

```
<Button Style="{StaticResource RoundedGelButton}" Content="Click Me!" />
```

按钮现在的外观完全不同, 但按钮的内容未在图 35-22 中显示出来。必须扩展前面创建的模板, 以把按钮的内容显示在新外观上。为此需要添加一个 ContentPresenter。ContentPresenter 是控件内容的占位符, 并定义了放置这些内容的位置。这里把内容放在网格的第一行上, 即 Ellipse 元素所在的位置。ContentPresenter 的 Content 属性定义了内容的外观。把内容设置为 TemplateBinding 标记表达式。



图 35-22

TemplateBinding 绑定父模板, 这里是 Button 元素。{TemplateBinding Content} 指定, Button 控件的 Content 属性值应作为内容放在占位符内。图 35-23 显示了带内容的按钮:



```

<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
            <Grid>
                <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
                    <Ellipse.Stroke>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                            <GradientStop Offset="0" Color="#ff7e7e" />
                            <GradientStop Offset="1" Color="Black" />
                        </LinearGradientBrush>
                    </Ellipse.Stroke>
                </Ellipse>
            </Grid>
        </ControlTemplate>
    </Setter.Value>
</Setter>

```

```

</Ellipse>
<Ellipse Margin="15,5,15,50">
<Ellipse.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="#aaffffff" />
        <GradientStop Offset="1" Color="Transparent" />
    </LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
<ContentPresenter Name="GelButtonContent"
    VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Content="{TemplateBinding Content}" />
</Grid>
</ControlTemplate>
</Setter.Value>

```

代码段 TemplateDemo/StylizedButtonWindow.xaml

现在这样一个样式化的按钮在屏幕上看起来很漂亮。但仍有一个问题：如果用鼠标单击该按钮，或使鼠标滑过该按钮，则它不会有任何动作。这不是用户操作按钮时的一般情况。解决方法如下：对于模板样式的按钮，必须给它指定触发器，使按钮在响应鼠标移动和鼠标单击时有不同的外观。

使用属性触发器(参见前面的内容)，就很容易解决这个问题。只需把该触发器添加到 `ControlTemplate` 的 `Triggers` 集合中，如下所示。这里定义了两个触发器，一个属性触发器在按钮的 `IsMouseOver` 属性为 `true` 时激活，接着把椭圆 `GelBackground` 的 `Fill` 属性改为 `RadialGradientBrush`，其值从 `Lime` 改为 `DarkGreen`。使用 `IsPressed` 属性给 `RadialGradientBrush` 指定其他颜色：



图 35-23

```

<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="Ellipse.Fill" TargetName="GelBackground">
            <Setter.Value>
                <RadialGradientBrush>
                    <GradientStop Offset="0" Color="Lime" />
                    <GradientStop Offset="1" Color="DarkGreen" />
                </RadialGradientBrush>
            </Setter.Value>
        </Setter>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
        <Setter Property="Ellipse.Fill" TargetName="GelBackground">
            <Setter.Value>
                <RadialGradientBrush>
                    <GradientStop Offset="0" Color="#ffcc34" />
                    <GradientStop Offset="1" Color="#cc9900" />
                </RadialGradientBrush>
            </Setter.Value>
        </Setter>
    </Trigger>
</ControlTemplate.Triggers>

```

现在就可以运行应用程序，只要把鼠标悬停在按钮上，或者单击鼠标，就会看到按钮的可见反馈。

35.10.2 数据模板

ContentControl 元素的内容可以是任意内容——不仅可以是 WPF 元素，还可以是 .NET 对象。例如，可以把 **Country** 类型的对象赋予 **Button** 类的内容。创建 **Country** 类，以表示国家名称和国旗(用一幅图像的路径表示)。这个类定义 **Name** 和 **ImagePath** 属性，并重写 **ToString()** 方法，用于默认的字符串表示：



可从
wrox.com
下载源代码

```
public class Country
{
    public string Name { get; set; }
    public string ImagePath { get; set; }

    public override string ToString()
    {
        return Name;
    }
}
```

代码段 TemplateDemo/Country.cs

这些内容在按钮或任何其他 **ContentControl** 中会如何显示？默认情况下会调用 **ToString()** 方法，显示对象的字符串表示。要获得自定义外观，还可以为 **Country** 类型创建一个 **DataTemplate**。

这里在 **Window** 的资源中创建一个 **DataTemplate**。这个 **DataTemplate** 没有指定键，因此默认是 **Country.src** 类型——它也是引用 .NET 程序集的 XML 名称空间和 .NET 名称空间的别名。在 **DataTemplate** 内部，主元素是一个文本框，其 **Text** 属性绑定到 **Country** 的 **Name** 属性上，**Source** 属性的 **Image** 绑定到 **Country** 的 **ImagePath** 属性上。**Grid**、**Border** 和 **Rectangle** 元素定义了布局和可见外观：



可从
wrox.com
下载源代码

```
<Window.Resources>
    <DataTemplate DataType="{x:Type src:Country}">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="60" />
            </Grid.RowDefinitions>
            <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                Text="{Binding Name}" FontWeight="Bold" Grid.Column="0" />
            <Border Margin="4,0" Grid.Column="1" BorderThickness="2"
                CornerRadius="4">
                <Border.BorderBrush>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                        <GradientStop Offset="0" Color="#aaa" />
                        <GradientStop Offset="1" Color="#222" />
                    </LinearGradientBrush>
                </Border.BorderBrush>
            </Border>
            <Rectangle>
                <Rectangle.Fill>
```

```

        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#444" />
            <GradientStop Offset="1" Color="#fff" />
        </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>
    <Image Width="48" Margin="2,2,2,1"
        Source="{Binding ImagePath}" />
</Grid>
</Border>
</Grid>
</DataTemplate>
</Window.Resources>

```

代码段 TemplateDemo/StyledButtonWindow.xaml

在 XAML 代码中, 定义一个简单的 Button 元素 button1:

```
<Button Grid.Row="1" x:Name="button1" Margin="10" />
```

在代码隐藏中, 实例化一个新的 Country 对象, 并把它赋予 button1 的 Content 属性:



可从
wrox.com
下载源代码

```

public StyledButtonWindow()
{
    InitializeComponent();
    button1.Content = new Country
    {
        Name = "Austria",
        ImagePath = "images/Austria.bmp"
    };
}

```

代码段 TemplateDemo/StyledButtonWindow.xaml.cs

运行这个应用程序, 可以看出, DataTemplate 应用于 Button, 因为 Country 数据类型有默认的模板, 如图 35-24 所示。



图 35-24

当然, 也可以创建一个控件模板, 并从中使用数据模板。

35.10.3 样式化列表框

更改按钮或标签的样式是一个简单的任务。如何改变包含一个元素列表的父元素的样式, 例如, 如何更改列表框? 列表框也有操作方式和外观。列表框可以显示一个元素列表, 用户可以从列表选择一个或多个元素。至于操作方式, ListBox 类定义了方法、属性和事件。ListBox 的外观与其操作是分开的。ListBox 元素有一个默认的外观, 但可以通过创建模板, 改变这个外观。

对于列表框, ControlTemplate 定义了整个控件的外观, ItemTemplate 定义了列表项的外观,

DataTemplate 定义了项中的类型。

为了给列表框填充一些项，静态类 **Countries** 返回几个要显示出来的国家对应的一个列表：



```
public class Countries
{
    public static IEnumerable<Country>GetCountries()
    {
        return new List<Country>
        {
            new Country { Name = "Austria", ImagePath = "Images/Austria.bmp" },
            new Country { Name = "Germany", ImagePath = "Images/Germany.bmp" },
            new Country { Name = "Norway", ImagePath = "Images/Norway.bmp" },
            new Country { Name = "USA", ImagePath = "Images/USA.bmp" }
        };
    }
}
```

代码段 TemplateDemo/Countries.cs

在代码隐藏文件中，在 **StyledListBoxWindow1** 类的构造函数中，将 **StyledListBoxWindow1** 实例的 **DataContext** 属性设置为要从 **Countries.GetCountries()** 方法中返回的国家列表(**DataContext** 属性是数据绑定的一个功能，详见下一章)。



```
public partial class StyledListBoxWindow1 : Window
{
    public StyledListBoxWindow1()
    {
        InitializeComponent();
        this.DataContext = Countries.GetCountries();
    }
}
```

代码段 TemplateDemo/StyledListBoxWindow1.xaml.cs

在 XAML 代码中，定义了 **countryList1** 列表框。**countryList1** 没有不同的样式，它使用 **ListBox** 元素的默认外观。把 **ItemsSource** 属性设置为 **Binding** 标记扩展，它由数据绑定使用。从代码隐藏中，可以看到数据绑定用于一个 **Country** 对象数组。图 35-25 显示了 **ListBox** 的默认外观。在默认情况下，只在一个简单的列表中显示 **ToString()** 方法返回的国家的名称。



```
<Window x:Class="TemplateDemo.StyledListBoxWindow1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:src="clr-namespace:TemplateDemo"
        Title="StyledListBoxWindow1" Height="300" Width="300">
    <Grid>
        <ListBox ItemsSource="{Binding}" Margin="10" />
    </Grid>
</Window>
```

代码段 TemplateDemo/StyledListBoxWindow1.xaml

35.10.4 ItemTemplate

Country 对象有名称和国旗。当然，还可以在列表框中显示这两个值。为此，必须定义一个模板。

ListBox 元素包含 ListBoxItem 元素。使用 ItemTemplate 可以定义列表项的内容。listBoxStyle1 样式定义一个 ItemTemplate，其值为 DataTemplate。DataTemplate 用于将数据绑定到元素上。Binding 标记扩展可以用于 DataTemplate 元素。



图 35-25

DataTemplate 包含一个三列的栅格。第一列包含字符串“Country:”，第二列包含国家的名称，第三列包含该国家的国旗。因为国家名称的长度不同，但看起来每个国家名称应有相同的大小，所以给第二列定义设置 SharedSizeGroup 属性。只有这一列使用共享大小的信息，因为也设置了 Grid.IsSharedSizeScope 属性。

行和列定义之后，就可以看到两个 TextBlock 元素。第一个 TextBlock 元素包含文本“Country:”，第二个 TextBlock 元素绑定到 Country 类定义的名称属性上。

第三列的内容是一个包含栅格的 Border 元素。栅格包含一个带线性渐变画笔的矩形和一个绑定到 Country 类的 ImagePath 属性上的 Image 元素。图 35-26 显示了列表框中的国家，其效果与前面完全不同。

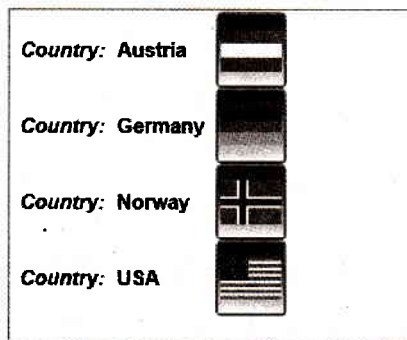


图 35-26



```
<Style x:Key="ListBoxStyle1" TargetType="{x:Type ListBox}">
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width=" * " SharedSizeGroup="MiddleColumn" />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
          </Grid.RowDefinitions>
          <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
            FontStyle="Italic" Grid.Column="0" Text="Country:" />
          <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
            Text="{Binding Name}" FontWeight="Bold"
            Grid.Column="1" />
          <Border Margin="4,0" Grid.Column="2" BorderThickness="2"
            CornerRadius="4">
            <Border.BorderBrush>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#aaa" />
                <GradientStop Offset="1" Color="#222" />
              </LinearGradientBrush>
            </Border.BorderBrush>
          </Border>
          <Grid>
            <Rectangle>
              <Rectangle.Fill>
                <LinearGradientBrush StartPoint="0,0"
```

```

        EndPoint="0,1">
        <GradientStop Offset="0" Color="#444" />
        <GradientStop Offset="1" Color="#fff" />
    </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
<Image Width="48" Margin="2,2,2,1"
    Source="{Binding ImagePath}" />
</Grid>
</Border>
</Grid>
</DataTemplate>
</Setter.Value>
</Setter>
<Setter Property="Grid.IsSharedSizeScope" Value="True" />
</Style>

```

代码段 TemplateDemo/Styles.xaml

35.10.5 列表框元素的控件模板

列表框中的项不一定垂直排列,还可以给同一个功能提供另一种视图。下一个样式 `ListBoxStyle2` 定义了一个模板,使列表项水平显示,且带一个滚动条。

在前面的示例中,只创建了一个 `LitemTemplate`,来定义列表项在默认列表框中的外观。现在创建一个模板,以定义另一个列表框。该模板包含一个 `ControlTemplate` 元素,它定义了列表框的元素。`ControlTemplate` 元素现在是一个包含 `StackPanel` 的 `ScrollViewer`——带滚动条的视图。因为列表项现在应水平排列,所以把 `StackPanel` 的 `Orientation` 设置为 `Horizontal`。`StackPanel` 还包含用 `ItemsTemplate` 定义的列表项,于是把 `StackPanel` 元素的 `IsItemsHost` 属性设置为 `true`。`IsItemsHost` 属性可以用于包含一个列表项对应列表的所有 `Panel` 元素。

`ItemsTemplate` 定义了 `StackPanel` 中数据项的外观。`ItemsTemplate` 使用 `ListBoxStyle1` 样式, `ListBoxStyle2` 基于 `ListBoxStyle1` 样式。

图 35-27 显示了使用 `listBoxStyle2` 样式的列表框,当视图太小,不能显示列表中的所有项时,会自动显示滚动条。



可从
wrox.com
下载源代码

```

<Style x:Key="ListBoxStyle2" TargetType="{x:Type ListBox}"
    BasedOn="{StaticResource ListBoxStyle1}">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type ListBox}">
                <ScrollViewer HorizontalScrollBarVisibility="Auto">
                    <StackPanel Name="StackPanel1" IsItemsHost="True"
                        Orientation="Horizontal" />
                </ScrollViewer>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
    <Setter Property="VerticalAlignment" Value="Center" />
</Style>

```

代码段 TemplateDemo/Styles.xaml

显然,将控件的外观与其行为分开很有好处。也许用户有许多方式,能很好地显示列表中的项,

使之满足应用程序的要求。也许用户只想在窗口中显示尽量多的项，先水平排列，一行放不下时，就继续在下一行竖直排列。此时就可以使用 `.WrapPanel`。当然，可以给列表框的模板定义一个 `WrapPanel`，如 `listBoxStyle3` 所示。图 35-28 显示了使用 `WrapPanel` 的结果：

```
<Style x:Key="ListBoxStyle3" TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer VerticalScrollBarVisibility="Auto"
          HorizontalScrollBarVisibility="Disabled">
          <WrapPanel IsItemsHost="True" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
            Source="{Binding ImagePath}" />
          <TextBlock Grid.Row="1" FontSize="14"
            HorizontalAlignment="Center" Margin="5"
            Text="{Binding Name}" />
        </Grid>
      </DataTemplate>
    </Setter.Value>
  </Setter>
</Style>
```



图 35-27



图 35-28

35.11 动画

在动画中，可以使用移动的元素、颜色变化、变换等制作平滑的变换效果。WPF 使动画的制作非常简单。还可以连续改变任意依赖属性的值。不同的动画类可以根据其类型，连续改变不同属性的值。

动画的主要元素如下：

- 时间轴——定义了值随时间的变化方式。有不同类型的时间轴，可用于改变不同类型的值。所有时间轴的基类都是 `Timeline`。为了连续改变 `double` 值，可以使用 `DoubleAnimation` 类。`Int32Animation` 类是 `int` 值的动画类。`PointAnimation` 类用于连续改变点，`ColorAnimation` 类用于连续改变颜色。
- 故事板——用于合并动画。`Storyboard` 类派生自基类 `TimelineGroup`，`TimelineGroup` 又派生自基类 `Timeline`。使用 `DoubleAnimation` 类，可以连续改变 `double`，使用 `Storyboard` 类可以合并所有应连接在一起的动画。
- 触发器——通过触发器可以启动和停止动画。前面介绍了属性触发器。当属性值变化时，属性触发器就会激活。还可以创建事件触发器，当事件发生时，事件触发器就会激活。



动画类的名称空间是 `System.Windows.Media.Animation`。

35.11.1 时间轴

时间轴定义了值随时间变化的方式。第一个示例连续改变椭圆的大小。其中 `DoubleAnimation` 时间轴改变为 `double` 值。把 `Ellipse` 类的 `Triggers` 属性设置为 `EventTrigger`。椭圆加载时，就激活用 `EventTrigger` 的 `RoutedEvent` 属性定义的事件触发器。`BeginStoryboard` 是启动故事板的触发器动作。在故事板中，`DoubleAnimation` 元素用于连续改变 `Ellipse` 类的 `Width` 属性。动画在 3 秒内把椭圆的宽度从 100 改为 300，在之后的 3 秒内再恢复原状。`ColorAnimation` 连续改变 `ellipseBrush` 中用于填充椭圆的颜色：



可从
wrox.com
下载源代码

```
<Ellipse Height="50" Width="100">
  <Ellipse.Fill>
    <SolidColorBrush x:Name="ellipseBrush" Color="SteelBlue" />
  </Ellipse.Fill>
  <Ellipse.Triggers>
    <EventTrigger RoutedEvent="Ellipse.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
            <DoubleAnimation
              Storyboard.TargetProperty="(Ellipse.Width)"
              Duration="0:0:3" AutoReverse="True"
              FillBehavior="Stop"
              RepeatBehavior="Forever"
              AccelerationRatio="0.9" DecelerationRatio="0.1"
              From="100" To="300" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Ellipse.Triggers>
</Ellipse>
```

```

        <ColorAnimation Storyboard.TargetName="ellipseBrush"
            Storyboard.TargetProperty="(SolidColorBrush.Color)"
            Duration="0:0:3" AutoReverse="True"
            FillBehavior="Stop" RepeatBehavior="Forever"
            From="Yellow" To="Red" />

    </Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Ellipse.Triggers>
</Ellipse>

```

代码段 AnimationDemo/EllipseWindow.xaml

图 35-29 和 35-30 显示了具有动画效果的椭圆的两个状态。



图 35-29



图 35-30

动画并不仅仅是一直和立刻显示在屏幕上的一般窗口动画。还可以给业务应用程序添加动画，使用户界面的响应性更好。

下面的例子演示了一个相当好的动画，还说明了如何在样式中定义动画。在 Window 资源中，有一个用于按钮的 `AnimatedButtonStyle` 样式。在模板中定义了一个矩形 `outline`。这个模板使用很细的笔触，把其宽度设置为 0.4。



图 35-31



图 35-32

该模板为 `IsMouseOver` 属性定义了一个属性触发器。当鼠标移过按钮时，就应用这个触发器的 `EnterActions` 属性。启动动作是 `BeginStoryboard`，它是一个触发器动作，可以包含并启动 `Storyboard` 元素。`Storyboard` 元素定义了一个 `DoubleAnimation`，可以为 `double` 值制作动画。在这个动画中改变的属性值是 `Rectangle` 元素 `outline` 的 `StrokeThickness` 属性。该值平滑地增加 1.2，因为 `By` 属性指定，该属性变化的时间长度是 `Duration` 属性设置的 0.3 秒。在动画结束时，笔触的宽度重置为其初始值，因为 `AutoReverse="True"`。总之，只要鼠标移过按钮，`outline` 的边框就在 0.3 秒内增加 1.2。图 35-31 显示了没有改变的按钮，图 35-32 显示了鼠标移过按钮 0.3 秒后的按钮。在纸质媒介上，不可能显示平滑的动画和按钮外观的中间状态。



可从
wrox.com
下载源代码

```

<Window x:Class="AnimationDemo.ButtonAnimationWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ButtonAnimationWindow" Height="300" Width="300">
    <Window.Resources>
        <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Grid>
                            <Rectangle Name="outline" RadiusX="9" RadiusY="9"
                                Stroke="Black" Fill="{TemplateBinding Background}"
                                StrokeThickness="1.6">

```

```

        <ContentPresenter VerticalAlignment="Center"
            HorizontalAlignment="Center" />
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Trigger.EnterActions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Duration="0:0:0.3"
                            AutoReverse="True"
                            Storyboard.TargetProperty=
                                "(Rectangle.StrokeThickness)"
                            Storyboard.TargetName="outline"
                            By="1.2" />
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.EnterActions>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<Grid>
    <Button Style="{StaticResource AnimatedButtonStyle}" Width="200"
        Height="100" Content="Click Me!" />
</Grid>
</Window>

```

代码段 AnimationDemo/ButtonAnimationWindow.xaml

Timeline 可以完成的任务如表 35-10 所示。

表 35-10

Timeline 属性	说 明
AutoReverse	使用 AutoReverse 属性，可以指定连续改变的值在动画结束后是否返回初始值
SpeedRatio	使用 SpeedRatio，可以改变动画的移动速度。在这个属性中，可以定义父子元素的相对关系。默认值为 1；将速率设置为较小的值，会使动画移动较慢；将速率设置为高于 1 的值，会使动画移动较快
BeginTime	使用 BeginTime，可以指定从触发器事件开始到动画开始移动之间的时间长度。其单位可以是天、小时、分钟、秒和几分之一秒。根据 SpeedRatio，这可以不是真实的时间。例如，如果把 SpeedRatio 设置为 2，把开始时间设置为 6 秒，动画就在 3 秒后开始
AccelerationRatio DecelerationRatio	在动画中，值不一定是线性变化。可以指定 AccelerationRatio 和 DecelerationRatio，定义加速度和减速度。这两个值的总和不能超过 1
Duration	使用 Duration 属性，可以指定动画重复一次的时间长度
RepeatBehavior	给 RepeatBehavior 属性指定一个 RepeatBehavior 结构，可以定义动画的重复次数或重复时间

(续表)

Timeline 属性	说 明
FillBehavior	如果父元素的时间轴有不同的持续时间, FillBehavior 属性就很重要。例如, 如果把父元素的时间轴比实际动画的持续时间短, 将 FillBehavior 设置为 Stop 就表示实际动画停止。如果父元素的时间轴比实际动画的持续时间长, HoldEnd 就会一直执行动画, 直到把它重置为初始值为止(假定把 AutoReverse 设置为 true)

根据 Timeline 类的类型, 还可以使用其他一些属性。例如, 使用 DoubleAnimation, 可以为动画的开始和结束设置 From 和 To 属性。还可以设置 By 属性, 用 Bound 属性的当前值启动动画, 该属性值会递增由 By 属性指定的值。

35.11.2 非线性动画

定义非线性动画的一种方式设置 AccelerationRatio 和 DecelerationRatio, 指定动画在开始和结束时的速度。NET 4 有比这更灵活的方式。

几个动画类有 EasingFunction 属性。这个属性接受一个实现了 IEasingFunction 接口的对象。通过这个接口, 缓动函数对象可以定义值随着时间如何变化动画效果。有几个缓动函数可用于创建非线性动画, 如 ExponentialEase, 它给动画使用指数公式; QuadraticEase、CubicEase、QuarticEase 和 QuinticEase 的指数分别是 2、3、4、5, PowerEase 的指数是可以配置的。特别有趣的是 SineEase, 它使用正弦曲线, BounceEase 创建弹跳效果, ElasticEase 用弹簧的来回震荡来模拟动画值。

要在 XAML 中指定这种缓动效果时, 具体方法是把该缓动效果添加到动画的 EasingFunction 属性中, 如下所示。添加不同的缓动函数, 就会看到动画的有趣效果:



```
<DoubleAnimation
    Storyboard.TargetProperty="(Ellipse.Width)"
    Duration="0:0:3" AutoReverse="True"
    FillBehavior="Stop"
    RepeatBehavior="Forever"
    From="100" To="300">
    <DoubleAnimation.EasingFunction>
        <BounceEase EasingMode="EaseInOut" />
    </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

代码段 AnimationDemo/EllipseWindow.xaml

35.11.3 事件触发器

除了使用属性触发器之外, 还可以定义一个事件触发器, 来启动动画。属性触发器在属性改变其值时激活, 事件触发器在事件发生时激活。这种事件的例子包括控件的 Load 事件、按钮的 Click 事件, 以及 MouseMove 事件等。

下一个例子将为前面通过形状创建的笑脸创建一个动画。创建动画后, 一旦激活按钮的 Click 事件, 眼睛就会移动。

在 Window 元素中, 定义了一个 DockPanel 元素, 来排列笑脸和控制动画的按钮。包含 3 个按钮的 StackPanel 停靠在顶部。包含笑脸的 Canvas 元素占用了 DockPanel 的其余部分。

第1个按钮用于启动眼睛的动画，第2个按钮用于停止动画，第3个按钮用于启动另一个重置笑脸大小的动画。

动画在 `DockPanel.Triggers` 部分中定义。这里没有使用属性触发器，而使用了事件触发器。`RoutedEvent` 和 `SourceName` 属性定义了 `buttonBeginMoveEyes` 按钮，一旦该按钮的 `Click` 事件发生，就激活第1个事件触发器。触发器动作由 `BeginStoryboard` 元素定义，它启动所包含的 `Storyboard`。`BeginStoryboard` 定义了一个名称，它用于控制故事板的暂停、继续和停止动作。`Storyboard` 元素包含4个动画。前两幅动画连续改变左眼，后两幅动画连续改变右眼。第1幅和第3幅动画改变眼睛的 `Canvas.Left` 位置，第2幅和第4幅动画改变 `Canvas.Top`。动画在 `x` 和 `y` 方向上有不同的时间值，使用指定的重复行为使眼睛的运动更有趣。

一旦 `buttonStopMoveEyes` 按钮的 `Click` 事件发生，就激活第2个事件触发器。在这里，故事板用 `StopStoryboard` 元素停止，该元素引用了起始故事板 `beginMoveEye`。

第3个事件触发器通过单击 `buttonResize` 按钮来激活。在这幅动画中，改变了 `Canvas` 元素的变换。因为这幅动画不会无限制地运行下去，所以它不需要停止。这个故事板也使用前面介绍的 `EaseFunction`：



可从
wrox.com
下载源代码

```
<Window x:Class="AnimationDemo.EventTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EventTriggerWindow" Height="300" Width="300">
    <DockPanel>
        <DockPanel.Triggers>
            <EventTrigger RoutedEvent="Button.Click" SourceName="buttonBeginMoveEyes">
                <BeginStoryboard x:Name="beginMoveEyes">
                    <Storyboard>
                        <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                                         AutoReverse="True" By="6" Duration="0:0:1"
                                         Storyboard.TargetName="eyeLeft"
                                         Storyboard.TargetProperty="(Canvas.Left)" />
                        <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                                         By="6" Duration="0:0:5"
                                         Storyboard.TargetName="eyeLeft"
                                         Storyboard.TargetProperty="(Canvas.Top)" />
                        <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                                         AutoReverse="True" By="-6" Duration="0:0:3"
                                         Storyboard.TargetName="eyeRight"
                                         Storyboard.TargetProperty="(Canvas.Left)" />
                        <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                                         By="6" Duration="0:0:6"
                                         Storyboard.TargetName="eyeRight"
                                         Storyboard.TargetProperty="(Canvas.Top)" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
            <EventTrigger RoutedEvent="Button.Click" SourceName="buttonStopMoveEyes">
                <StopStoryboard BeginStoryboardName="beginMoveEyes" />
            </EventTrigger>
            <EventTrigger RoutedEvent="Button.Click" SourceName="buttonResize">
                <BeginStoryboard>
                    <Storyboard>
```

```

        <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
            From="0.1" To="3" Duration="0:0:5">
            <DoubleAnimation.EasingFunction>
                <ElasticEase />
            </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
        <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="(ScaleTransform.ScaleY)"
            From="0.1" To="3" Duration="0:0:5">
            <DoubleAnimation.EasingFunction>
                <BounceEase />
            </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</DockPanel.Triggers>
<StackPanel Orientation="Vertical" DockPanel.Dock="Top">
    <Button x:Name="buttonBeginMoveEyes" Content="Start Move Eyes" Margin="5" />
    <Button x:Name="buttonStopMoveEyes" Content="Stop Move Eyes" Margin="5" />
    <Button x:Name="buttonResize" Content="Resize" Margin="5" />
</StackPanel>
<Canvas>
    <Canvas.LayoutTransform>
        <ScaleTransform x:Name="scale1" ScaleX="1" ScaleY="1" />
    </Canvas.LayoutTransform>

    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
        Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
    <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
        <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
                <GradientStop Offset="0.1" Color="DarkGreen" />
                <GradientStop Offset="0.7" Color="Transparent" />
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
        Stroke="Blue" StrokeThickness="3" Fill="White" />
    <Ellipse x:Name="eyeLeft" Canvas.Left="40" Canvas.Top="43" Width="6"
        Height="5" Fill="Black" />

    <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
        Stroke="Blue" StrokeThickness="3" Fill="White" />
    <Ellipse x:Name="eyeRight" Canvas.Left="75" Canvas.Top="43" Width="6"
        Height="5" Fill="Black" />
    <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
        Data="M 40,74 Q 57,95 80,74 " />
</Canvas>

</DockPanel>
</Window>

```

代码段 AnimationDemo/EventTriggerWindow.xaml

图 35-33 显示了运行应用程序的结果。

除了直接从 XAML 的事件触发器中直接启动和停止动画之外，还可以从代码隐藏中控制动画。只需给故事板指定一个名称，并调用 `Begin()`、`Stop()`、`Pause()` 和 `Resume()` 方法。

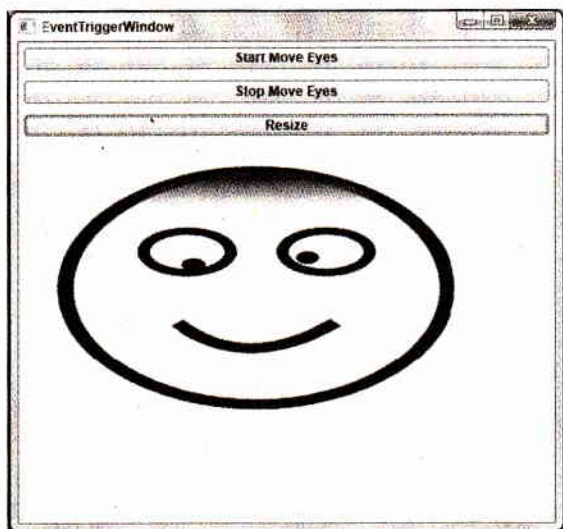


图 35-33

35.11.4 关键帧动画

如前所述，使用加速比、减速比和缓动函数，就可以用非线性的方式制作动画。如果需要为动画指定几个值，就可以使用关键帧动画。与正常的动画一样，关键帧动画也有不同的动画类型，它们可以改变不同类型的属性。

`DoubleAnimationUsingKeyFrames` 是双精度类型的关键帧动画。其他关键帧动画类型有 `Int32AnimationUsingKeyFrames`、`PointAnimationUsingKeyFrames`、`ColorAnimationUsingKeyFrames`、`SizeAnimationUsingKeyFrames` 和 `ObjectAnimationUsingKeyFrames`。

示例 XAML 代码连续地改变 `TranslateTransform` 元素的 X 值和 Y 值，从而改变椭圆的位置。把 `EventTrigger` 定义为 `RoutedEvent` 的 `Ellipse.Loaded`，动画就会在加载椭圆时启动。事件触发器用 `BeginStoryboard` 元素启动一个 `Storyboard`。该 `Storyboard` 包含两个 `DoubleAnimationUsingKeyFrame` 类型的关键帧动画。关键帧动画由帧元素组成。第一幅关键帧动画使用一个 `LinearKeyFrame`、一个 `DiscreteDoubleKeyFrame` 和一个 `SplineDoubleKeyFrame`；第二幅关键帧动画是一个 `EasingDoubleKeyFrame`。`LinearDoubleKeyFrame` 使对应值线性变化。`KeyTime` 属性定义了动画应何时达到 `Value` 属性的值。这里 `LinearDoubleKeyFrame` 用 3 秒的时间使 X 属性到达值 30。`DiscreteDoubleKeyFrame` 在 4 秒后立即改变为新值。`SplineDoubleKeyFrame` 使用贝塞尔曲线，其中的两个控制点由 `KeySpline` 属性指定。`EasingDoubleKeyFrame` 是 .NET 4 中新增的一个帧类，它支持设置缓动函数(如 `BounceEase`)来控制动画值：



可从
wrox.com
下载源代码

```
<Canvas>
  <Ellipse Fill="Red" Canvas.Left="20" Canvas.Top="20" Width="25" Height="25">
    <Ellipse.RenderTransform>
      <TranslateTransform X="50" Y="50" x:Name="ellipseMove" />
    </Ellipse.RenderTransform>
  </Ellipse>
</Canvas>
```

```

<Ellipse.Triggers>
  <EventTrigger RoutedEvent="Ellipse.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="X"
          Storyboard.TargetName="ellipseMove">
          <LinearDoubleKeyFrame KeyTime="0:0:2" Value="30" />
          <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="80" />
          <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
            KeyTime="0:0:10" Value="300" />
          <LinearDoubleKeyFrame KeyTime="0:0:20" Value="150" />
        </DoubleAnimationUsingKeyFrames>
        <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Y"
          Storyboard.TargetName="ellipseMove">
          <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
            KeyTime="0:0:2" Value="50" />
          <EasingDoubleKeyFrame KeyTime="0:0:20" Value="300">
            <EasingDoubleKeyFrame.EasingFunction>
              <BounceEase />
            </EasingDoubleKeyFrame.EasingFunction>
          </EasingDoubleKeyFrame>
        </DoubleAnimationUsingKeyFrames>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Ellipse.Triggers>
</Ellipse>
</Canvas>

```

代码段 AnimationDemo/KeyFrameWindow.xaml

35.12 可见状态管理器

自从.NET 4 以来, Visual State Manager 提供了控制动画的另一种方式。控件可以有特定的状态。状态定义了到达该状态时应用于控件的外观。状态切换定义了一种状态变成另一种状态时会发生什么。

在数据网格中, 可以使用 Read、Selected 和 Edit 状态, 根据用户的选择给行定义不同的外观。MouseOver 和 IsPressed 可以是状态, 也可以用来替代前面讨论的触发器。

下面的示例代码显示一个婴儿的不同状态, 并定义 Sleeping、Playing 和 Crying 状态。状态在 VisualStateManager.VisualStateGroups 元素中定义。本例中的状态在一个 Button 元素中定义, 因为它们仅需要在该元素中使用。对于内容, Button 定义 Border 和 Image 元素, 它们分别命名为 border1 和 image1。通过 VisualStateManager 定义一个 VisualStateGroup, 命名为 CommonStates。这个状态组包含 VisualState 元素。在可见状态中定义一个 Storyboard。通过所有这些状态, 故事板包含一个 ColorAnimation, 它改变边框画笔的颜色和 ObjectAnimationUsingKeyFrames。这个关键帧动画仅对 Image 的 Source 属性进行一次性修改。UriSource 属性需要一个已创建的 BitmapImage 和一个在项目内部存储的文件名。

VisualStateGroup 也包含 VisualTransition 元素, 它定义在状态改变之前应发生什么。通过 VisualTransition, 可以指定 From 和 To 值。通过这些值可以在状态从 Sleeping 变成 Crying 时、从 Playing

变成 Crying 时创建不同的切换效果。本例仅使用 To 属性, 因此无论前一个状态是什么, 都应用这个切换效果。在切换过程中, 会改变 Button 的 Width 属性和 Border 元素的 Thickness:



```
<Window x:Class="VisualStateDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="240" Width="500">
    <DockPanel>
        <StackPanel Orientation="Vertical" DockPanel.Dock="Left">
            <Button Margin="5" Padding="3" Content="Sleeping" Click="OnSleeping" />
            <Button Margin="5" Padding="3" Content="Playing" Click="OnPlaying" />
            <Button Margin="5" Padding="5" Content="Crying" Click="OnCrying" />
        </StackPanel>
    </DockPanel>
    <Grid>
        <Button Name="button1" Width="300" Height="200">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="CommonStates">
                    <VisualState Name="Sleeping">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames Duration="0:0:0"
                                Storyboard.TargetProperty="(Image.Source)"
                                Storyboard.TargetName="image1">
                                <DiscreteObjectKeyFrame KeyTime="0:0:0">
                                    <DiscreteObjectKeyFrame.Value>
                                        <BitmapImage
                                            UriSource="Images/Sleeping.jpg" />
                                    </DiscreteObjectKeyFrame.Value>
                                </DiscreteObjectKeyFrame>
                            </ObjectAnimationUsingKeyFrames>
                            <ColorAnimation To="LightBlue"
                                Storyboard.TargetProperty="(SolidColorBrush.Color)"
                                Storyboard.TargetName="brush1" Duration="0:0:5" />
                        </Storyboard>
                    </VisualState>
                    <VisualState Name="Playing">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames Duration="0:0:0"
                                Storyboard.TargetProperty="(Image.Source)"
                                Storyboard.TargetName="image1">
                                <DiscreteObjectKeyFrame KeyTime="0:0:0">
                                    <DiscreteObjectKeyFrame.Value>
                                        <BitmapImage
                                            UriSource="Images/Playing.jpg" />
                                    </DiscreteObjectKeyFrame.Value>
                                </DiscreteObjectKeyFrame>
                            </ObjectAnimationUsingKeyFrames>
                            <ColorAnimation To="Red"
                                Storyboard.TargetProperty="(SolidColorBrush.Color)"
                                Storyboard.TargetName="brush1" Duration="0:0:5" />
                        </Storyboard>
                    </VisualState>
                    <VisualState Name="Crying">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames Duration="0"
                                Storyboard.TargetProperty="(Image.Source)"
                                Storyboard.TargetName="image1">
                                <DiscreteObjectKeyFrame KeyTime="0:0:0">
                                    <DiscreteObjectKeyFrame.Value>
                                        <BitmapImage
                                            UriSource="Images/Crying.jpg" />
                                    </DiscreteObjectKeyFrame.Value>
                                </DiscreteObjectKeyFrame>
                            </ObjectAnimationUsingKeyFrames>
                            <ColorAnimation To="Red"
                                Storyboard.TargetProperty="(SolidColorBrush.Color)"
                                Storyboard.TargetName="brush1" Duration="0:0:5" />
                        </Storyboard>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Button>
    </Grid>
</Window>
```

```

Storyboard.TargetProperty="(Image.Source)"
Storyboard.TargetName="image1">
<DiscreteObjectKeyFrame KeyTime="0">
    <DiscreteObjectKeyFrame.Value>
        <BitmapImage UriSource="Images/Crying.jpg" />
    </DiscreteObjectKeyFrame.Value>
</DiscreteObjectKeyFrame>
</ObjectAnimationUsingKeyFrames>
<ColorAnimation To="LightBlue"
Storyboard.TargetProperty="(SolidColorBrush.Color)"
Storyboard.TargetName="brush1" Duration="0:0:5" />
</Storyboard>
</VisualState>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup.Transitions>
        <VisualTransition To="Sleeping">
            <Storyboard>
                <DoubleAnimation By="50" AutoReverse="True"
Storyboard.TargetProperty="Width"
Storyboard.TargetName="button1"
Duration="0:0:1.2">
                    <DoubleAnimation.EasingFunction>
                        <BounceEase />
                    </DoubleAnimation.EasingFunction>
                </DoubleAnimation>
            </Storyboard>
        </VisualTransition>
        <VisualTransition To="Crying">
            <Storyboard>
                <ThicknessAnimation Duration="0:0:2" By="100,100"
AutoReverse="True"
Storyboard.TargetProperty=
    "(Border.BorderThickness)"
Storyboard.TargetName="border1" />
            </Storyboard>
        </VisualTransition>
        <VisualTransition To="Playing">
            <Storyboard>
                <DoubleAnimation By="50" AutoReverse="True"
Storyboard.TargetProperty="Width"
Storyboard.TargetName="button1"
Duration="0:0:1.2">
                    <DoubleAnimation.EasingFunction>
                        <ElasticEase />
                    </DoubleAnimation.EasingFunction>
                </DoubleAnimation>
            </Storyboard>
        </VisualTransition>
    </VisualStateGroup.Transitions>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<Border x:Name="border1" BorderThickness="12">
    <Border.BorderBrush>
        <SolidColorBrush x:Name="brush1" Color="White" />
    </Border.BorderBrush>

```

```

        <Image x:Name="image1" />
    </Border>
</Button>
</Grid>
</DockPanel>
</Window>

```

代码段 VisualStateDemo/MainWindow.xaml

通过 Click 事件, 在代码隐藏的 StackPanel 处理程序中调用 Button 元素。在代码隐藏中, 调用 VisualStateManager 的 GoToElementState() 方法, 把 button1 对象改为新状态。这个方法第 3 个参数指定在变成这个状态时是否应调用切换效果:



```

private void OnSleeping(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToElementState(button1, "Sleeping", true);
}

private void OnPlaying(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToElementState(button1, "Playing", true);
}

private void OnCrying(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToElementState(button1, "Crying", true);
}

```

代码段 VisualStateDemo/MainWindow.xaml.cs

运行应用程序, 就会看到状态的改变, 如图 35-34 所示。



图 35-34

35.13 3-D

本章的最后一节介绍 WPF 的 3D 功能, 其中包含了开始使用该功能的信息。



WPF 的 3D 特性在 System.Windows.Media.Media3D 名称空间中。

为了理解 WPF 的 3D 功能,一定要知道坐标系统之间的区别。图 35-35 显示了 WPF 中的 3D 坐标系统。原点位于中心。 x 轴的正值在右边,负值在左边。 y 轴是垂直的,正值在上边,负值在下边。 z 轴在指向观察者的方向上定义正值。

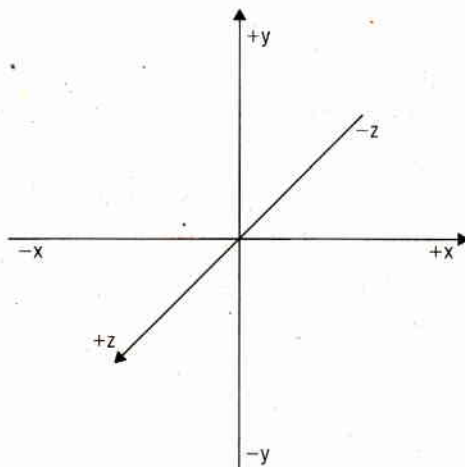


图 35-35

为了理解 WPF 的 3D 特性,最重要的概念是模型、照相机和光线。模型定义了使用三角形显示的内容,照相机定义了查看模型的位置和方式,而没有光线,模型就是黑暗的。光线定义了完整的场景如何照明。本节介绍如何使用 WPF 定义模型、照相机和光线,以及有哪些不同的选项。另外,还讨论如何为场景制作动画。

35.13.1 模型

本节将创建一个具有 3D 效果的图书模型。因为 3D 模型由三角形组成,所以最简单的模型是仅一个三角形。更复杂的模型由多个三角形组成。矩形由两个三角形组成,球由多个三角形组成,所使用的三角形越多,球就越光滑。

对于图书模型,每一面都是矩形,矩形仅由两个三角形组成。但是,因为书的封面有 3 种不同的材质,所以要使用 6 个三角形。

三角形用 `MeshGeometry3D` 的 `Positions` 属性定义。这里仅是书的封面的一部分。`MeshGeometry3D` 定义两个三角形。可以只计算 5 个点的坐标,因为第一个三角形的第三个点就是第二个三角形的第一个点。还可以进行优化,以减小模型的尺寸。所有这些点都使用相同的 z 坐标 0,而 (x,y) 坐标分别是 $(0,0)$, $(10,0)$, $(0,10)$, $(10,10)$ 和 $(10,0)$ 。`TriangleIndices` 属性指定这些点的顺序。第一个三角形按顺时针定义,第二个三角形按逆时针定义。使用这个属性可以指定三角形的哪一边是可见的。三角形的一边显示 `GeometryModel3D` 类的 `Material` 属性定义的颜色,另一边显示 `BackMaterial` 属性。

3D 的渲染表面是包围模型的 `ModelVisual3D`, 如下所示:



可从
wrox.com
下载源代码

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <Model3DGroup>

      <!-- front -->
      <GeometryModel3D>
```

```

<GeometryModel3D.Geometry>
  <MeshGeometry3D
    Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
    TriangleIndices="0, 1, 2, 2, 4, 3"
  />
</GeometryModel3D.Geometry>

```

代码段 3DDemo/MainWindow.xaml

GeometryModel3D 类的 **Material** 属性指定模型使用什么材质。根据视点的不同, **Material** 或 **BackMaterial** 属性很重要。

WPF 提供不同的材质类型: **DiffuseMaterial**、**EmissiveMaterial** 和 **SpecularMaterial**。材质和用于给场景照明的光线会影响模型的外观。计算 **EmissiveMaterial** 和应用用于材质画笔的颜色可定义光线, 以显示模型。**SpecularMaterial** 在发生高光反射时会添加眩目的高光反射效果。示例代码使用 **DiffuseMaterial**, 并引用 **mainCover** 资源中的画笔:

```

<GeometryModel3D.Material>
  <DiffuseMaterial Brush="{StaticResource mainCover}" />
</GeometryModel3D.Material>
</GeometryModel3D>

```

用于主封面的画笔是 **VisualBrush**。**VisualBrush** 有一个 **Border** 元素, 它是由两个 **Label** 元素组成的 **Grid**。一个 **Label** 元素定义了文本 **Professional C# 4**, 并写在封面上:

```

<VisualBrush x:Key="mainCover">
  <VisualBrush.Visual>
    <Border Background="Red">
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="30" />
          <RowDefinition Height=" * " />
        </Grid.RowDefinitions>
        <Label Grid.Row="0" HorizontalAlignment="Center">
          Professional C# 4 </Label>
        <Label Grid.Row="1"> </Label>
      </Grid>
    </Border>
  </VisualBrush.Visual>
</VisualBrush>

```

因为画笔用 2D 坐标系统定义, 而模型在 3D 坐标系统中定义, 所以需要对他们进行转换。这个转换使用 **MeshGeometry3D** 的 **TextureCoordinates** 属性来完成。**TextureCoordinates** 属性指定了三角形的每个点, 指定这些点如何映射到 2D 上。第一个点(0,0,0)映射到(0,1)上, 第二个点(10,0,0)映射到(1,1)上, 以此类推。注意 **y** 在 3D 和 2D 坐标系统中有不同的方向。图 35-36 显示了 2D 坐标系统。

```

<MeshGeometry3D
  Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
  TriangleIndices="0, 1, 2, 2, 4, 3"
  TextureCoordinates="0 1, 1 1, 0 0, 1 0, 1 1"
/>

```

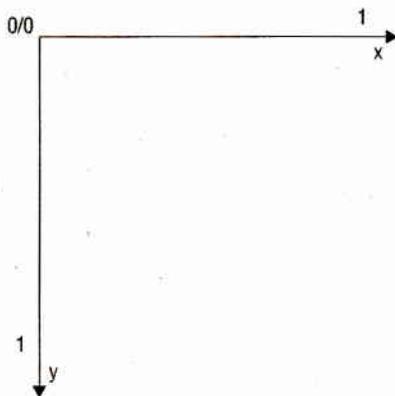


图 35-36

35.13.2 照相机

需要对 3D 模型使用照相机，以查看模型。本例使用 `PerspectiveCamera`，它需要指定位置和方向。把照相机的位置向左移动，会使模型向右移动，反之亦然。改变照相机的 `y` 坐标，模型会显得更大或更小。使照相机与模型相距越远，模型就越小：



可从
wrox.com
下载源代码

```
<Viewport3D.Camera>
  <PerspectiveCamera Position="0,0,25" LookDirection="15,6,-50" />
</Viewport3D.Camera>
```

代码段 3DDemo/MainWindow.xaml

WPF 还有一个 `OrtographicCamera`，因为它在场景中没有地平线，所以如果它移动得远一些，元素的大小不会改变。使用 `MatrixCamera`，可以精确地指定照相机的行为。

35.13.3 光线

没有光线，就是一片黑暗。3D 场景需要光线使模型可见。可以使用不同的光线。`AmbientLight` 会均匀地照亮场景。`DirectionalLight` 只照亮一个方向，类似于阳光。`PointLight` 在空间中有一个位置，它会照亮所有方向。`SpotLight` 也有一个位置，但其照亮的区域仅限于一个圆锥。

示例代码使用 `SpotLight`，指定了位置、方向和锥角：

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <SpotLight Color="White" InnerConeAngle="20" OuterConeAngle="60"
      Direction="15,6,-50" Position="0,0,25" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

35.13.4 旋转

为了得到模型的 3D 外观，模型还应能旋转。要进行旋转，可以使用 `RotateTransform3D` 元素，来定义旋转的中心和旋转角度：

```
<Model3DGroup.Transform>
  <RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">
    <RotateTransform3D.Rotation>
```

```

        <AxisAngleRotation3D x:Name="angle" Axis="-1,-1,-1"
                               Angle="70" />
    </RotateTransform3D.Rotation>
</RotateTransform3D>
</Model3DGroup.Transform>

```

要旋转已完成的模型，可以通过一个事件触发器启动一幅动画。该动画连续地改变 AxisAngleRotation3D 元素的 Angle 属性：

```

<Window.Triggers>
    <EventTrigger RoutedEvent=f"Window.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation From="0" To="360" Duration="00:00:10"
                                   Storyboard.TargetName="angle"
                                   Storyboard.TargetProperty="Angle"
                                   RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Window.Triggers>

```

运行应用程序，结果如图 35-37 所示。

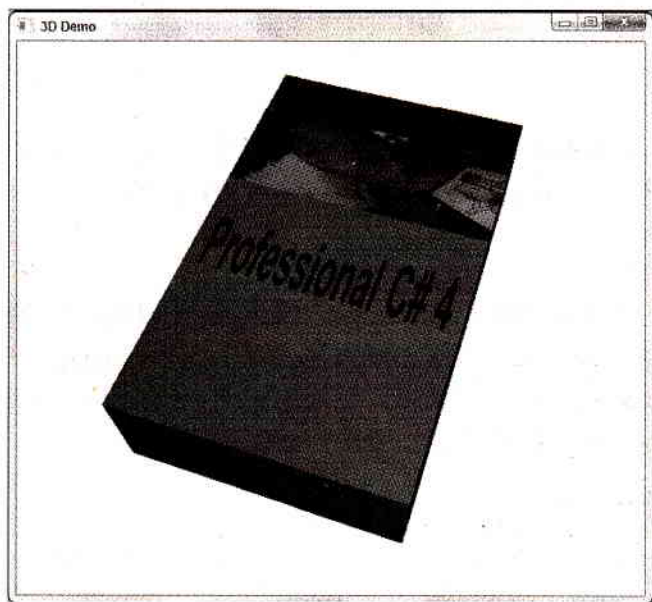


图 35-37

35.14 小结

本章介绍了 WPF 的许多功能。WPF 便于分开开发人员和设计人员的工作。所有 UI 功能都可以使用 XAML 创建，其功能用代码隐藏创建。

我们还探讨了基于矢量图形的许多控件和容器。因为 WPF 元素是矢量图形，所以可以缩放、

倾斜和旋转。因为内容控件的内容非常灵活，所以事件处理机制基于冒泡和隧道事件。

可以使用不同类型的画笔绘制背景和前景元素，不仅可以使用纯色画笔、线性渐变或放射性渐变画笔，而且可视化画笔完成反射功能或显示视频。

样式和模板可以定制控件的外观。触发器可以动态地改变 WPF 控件的属性。连续改变 WPF 控件的属性值，就可以轻松地制作出动画。

下一章继续介绍 WPF，主要探讨数据绑定、命令、导航和其他几个功能。

第36章

用 WPF 编写业务应用程序

本章内容:

- 绑定到元素、对象、列表和 XML 的数据
- 值的转换和验证
- Commanding
- 使用 TreeView 显示层次数据
- 使用 DataGrid 显示和组合数据

上一章介绍了 WPF 的一些核心功能,本章继续用 WPF 编程,介绍创建完整应用程序的一些重要方面,如数据绑定和命令处理,以及新的 DataGrid 控件。数据绑定是把.NET 类中的数据提供给用户界面的一个重要概念,还允许用户修改数据。Commanding 可以把 UI 的事件映射到代码上。与事件模型相反,它更好地分隔开了 XAML 和代码。TreeView 和 DataGrid 控件是显示绑定数据的 UI 控件。

36.1 数据绑定

与以前的技术相比, WPF 数据绑定向前迈了一大步。数据绑定把数据从.NET 对象传递给 UI, 或从 UI 传递给.NET 对象。简单对象可以绑定到 UI 元素、对象列表和 XAML 元素上。在 WPF 数据绑定中,目标可以是 WPF 元素的任意依赖属性,CLR 对象的每个属性都可以是绑定源。因为 WPF 元素作为.NET 类实现,所以每个 WPF 元素也可以用作绑定源。图 36-1 显示了绑定源和绑定目标之间的连接。Binding 对象定义了该连接。

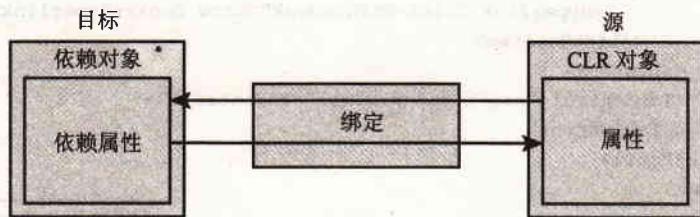


图 36-1

Binding 对象支持源与目标之间的几种绑定模式。绑定可以是单向的,即从源信息指向目标,但

如果用户在用户界面上修改了该信息，则源不会更新。要更新源，需要双向绑定。
表 36-1 列出了绑定模式及其要求。

表 36-1

绑定模式	说明
一次性	绑定从源指向目标，且仅在应用程序启动时，或数据环境改变时绑定一次。通过这种模式可以获得数据的快照
单向	绑定从源指向目标。这对于只读数据很有用，因为它不能从用户界面中修改数据。要更新用户界面，源必须实现 INotifyPropertyChanged 接口
双向	在双向绑定中，用户可以从 UI 中修改数据。绑定是双向的——从源指向目标，从目标指向源。源对象需要实现读写属性，才能把改动的内容从 UI 更新到源对象上
指向源的单向	采用这种绑定模式，如果目标属性改变，源对象也会更新

除了绑定模式之外，WPF 数据绑定还涉及许多方面。本节详细介绍与 XAML 元素、简单的.NET 对象和列表的绑定。通过更改通知，可以使用绑定对象中的更改更新 UI。本节还会讨论从对象数据提供程序中获取数据和直接从代码中获取数据。多绑定和优先绑定说明了与默认绑定不同的绑定可能性，本节也将论述如何动态地选择数据模板，以及绑定值的验证。

下面从 BooksDemo 示例应用程序开始。

36.1.1 BooksDemo 应用程序

本节将新建一个 WPF 应用程序 BooksDemo，本章的数据绑定和 commanding 将使用这个应用程序。

修改 XAML 文件 MainWindow.xaml，并分别添加一个 DockPanel、ListBox、Hyperlink 和 TabControl。



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
Insert IconMargin [FILENAME]
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
Title="Main Window" Height="400" Width="600">
  <DockPanel>
    <ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
      <ListBoxItem>
        <Hyperlink Click=OnShowBook>Show Book</Hyperlink>
      </ListBoxItem>
    </ListBox>
    <TabControl Margin="5" x:Name="tabControl1">
      </TabControl>
    </DockPanel>
  </Window>
```

代码段 BooksDemo/MainWindow.xaml

现在添加一个 WPF 用户控件 BookUC。这个用户控件包含一个 DockPanel、一个几行几列的 Grid、一个 Label 和多个 TextBox 控件：



可从
wrox.com
下载源代码

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <DockPanel>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
                HorizontalAlignment="Left" VerticalAlignment="Center" />
            <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
                Margin="10,0,5,0" HorizontalAlignment="Left"
                VerticalAlignment="Center" />
            <Label Content="Isbn" Grid.Row="2" Grid.Column="0"
                Margin="10,0,5,0" HorizontalAlignment="Left"
                VerticalAlignment="Center" />
            <TextBox Grid.Row="0" Grid.Column="1" Margin="5" />
            <TextBox Grid.Row="1" Grid.Column="1" Margin="5" />
            <TextBox Grid.Row="2" Grid.Column="1" Margin="5" />
            <StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
                <Button Content="Show Book" Margin="5" Click="OnShowBook" />
            </StackPanel>
        </Grid>
    </DockPanel>
</UserControl>
```

代码段 BooksDemo/BookUC.xaml

在 MainWindow.xaml.cs 的 OnShowBook() 处理程序中, 新建 BookUC 用户控件的一个实例, 给 TabControl 添加一个新的 TabItem。接着修改 TabControl 的 SelectedIndex 属性, 以打开新的选项卡:



可从
wrox.com
下载源代码

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
    var bookUI = new BookUC();
    this.tabControl1.SelectedIndex =
        this.tabControl1.Items.Add(
            new TabItem { Header = "Book", Content = bookUI });
}
```

代码段 BooksDemo/MainWindow.xaml.cs

构建项目后, 就可以启动应用程序, 单击超链接, 以打开 TabControl 中的用户控件。

36.1.2 用 XAML 绑定

WPF 元素不仅是数据绑定的目标，它还可以是绑定的源。可以把一个 WPF 元素的源属性绑定到另一个 WPF 元素的目标属性上。

在下面的代码示例中，数据绑定用于通过一个滑块重置用户控件中控件的大小。给用户控件 BookUC 添加一个 StackPanel 控件，该 StackPanel 控件包含一个标签和一个滑块控件。滑块控件定义了 Minimum 和 Maximum 值，以指定缩放比例，把其初始值 1 赋予 Value 属性：



可从
wrox.com
下载源代码

```
<DockPanel>
  <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
    HorizontalAlignment="Right">
    <Label Content="Resize" />
    <Slider x:Name="slider1" Value="1" Minimum="0.4" Maximum="3"
      Width="150" HorizontalAlignment="Right" />
  </StackPanel>
</DockPanel>
```

代码段 BooksDemo/BookUC.xaml

设置 Grid 控件的 LayoutTransform 属性，并添加一个 ScaleTransform 元素。通过 ScaleTransform 元素，对 ScaleX 和 ScaleY 属性进行数据绑定。这两个属性都用 Binding 标记扩展来设置。在 Binding 标记扩展中，把 ElementName 设置为 slider1，以引用前面创建的滑块控件。把 Path 设置为 Value，从 Value 属性中获取滑块的值。

```
<Grid>
  <Grid.LayoutTransform>
    <ScaleTransform x:Name="scale1"
      ScaleX="{Binding Path=Value, ElementName=slider1}"
      ScaleY="{Binding Path=Value, ElementName=slider1}" />
  </Grid.LayoutTransform>
</Grid>
```

运行应用程序时，可以移动滑块，从而重置 Grid 中的控件，如图 36-2 和图 36-3 所示。

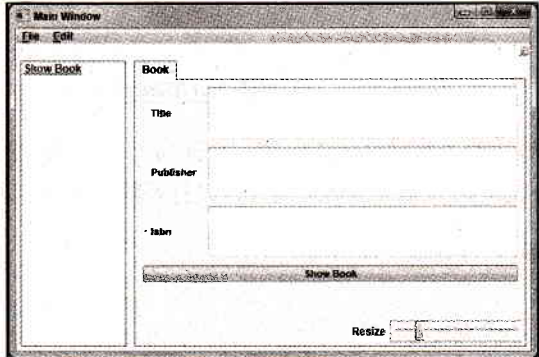


图 36-2

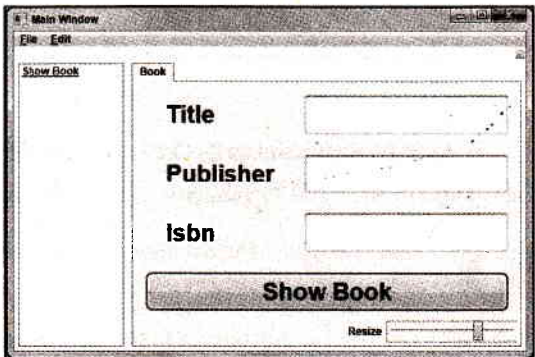


图 36-3

除了用 XAML 代码定义绑定信息之外，如上述代码使用 Binding 元数据扩展来定义，还可以使用代码隐藏。在代码隐藏中，必须新建一个 Binding 对象，并设置 Path 和 Source 属性。必须把 Source 属性设置为源对象，这里是 WPF 对象 slider1。把 Path 属性设置为一个 PropertyPath 实例，它用源对象的 Value 属性名初始化。对于派生自 FrameworkElement 的控件，可以调用 SetBinding() 方法来定义绑定。但是，ScaleTransform 不派生自 FrameworkElement，而派生自 Freezable 基类。使用辅助类

BindingOperations 可以绑定这类控件。BindingOperations 类的 SetBinding() 方法需要一个 DependencyObject, 在本例中是 ScaleTransform 实例。SetBinding() 方法还需要绑定目标的 dependency 属性和 Binding 对象。

```
var binding = new Binding
{
    Path = new PropertyPath("Value"),
    Source = slider1
};
BindingOperations.SetBinding(scale1,
    ScaleTransform.ScaleXProperty, binding);
BindingOperations.SetBinding(scale1,
    ScaleTransform.ScaleYProperty, binding);
```



派生自 DependencyObject 的所有类都可以有依赖属性。依赖属性参见第 27 章。

使用 Binding 类, 可以配置许多绑定选项, 如表 36-2 所示。

表 36-2

Binding 类的成员	说 明
Source	使用 Source 属性, 可以定义数据绑定的源对象
RelativeSource	使用 RelativeSource 属性, 可以指定与目标对象相关的源对象。当错误来源于同一个控件时, 它对于显示错误消息很有用
ElementName	如果源对象是一个 WPF 元素, 就可以用 ElementName 属性指定源对象
Path	使用 Path 属性, 可以指定到源对象的路径。它可以是源对象的属性, 但也支持子元素的索引器和属性
XPath	使用 XML 数据源时, 可以定义一个 XPath 查询表达式, 来获得要绑定的数据
Mode	模式定义了绑定的方向。Mode 属性是 BindingMode 类型。BindingMode 是一个枚举, 其值如下: Default、OneTime、OneWay、TwoWay、OneWayToSource。默认模式依赖于目标: 对于文本框, 默认是双向绑定; 对于只读的标签, 默认为单向。OneTime 表示数据仅从源中加载一次; OneWay 将对源对象的修改更新到目标对象中。TwoWay 绑定表示, 对 WPF 元素的修改可以写回源对象中。OneWayToSource 表示, 从不读取数据, 但总是从目标对象写入源对象中
Converter	使用 Converter 属性, 可以指定一个转换器类, 该转换器类来回转换 UI 的数据。转换器类必须实现 IValueConverter 接口, 它定义了 Convert() 和 ConvertBack() 方法。使用 ConverterParameter 属性可以给转换方法传递参数。转换器区分文化, 文化可以用 ConverterCultue 属性设置
FallbackValue	使用 FallbackValue 属性, 可以定义一个在绑定没有返回值时使用的默认值
ValidationRules	使用 ValidationRules 属性, 可以定义一个 ValidationRule 对象集合, 在从 WPF 目标元素更新源对象之前检查该集合。ExceptionValidationRule 类派生自 ValidationRule 类, 负责检查异常

36.1.3 简单对象的绑定

要绑定 CLR 对象，只需使用.NET 类定义属性，如下面的例子就使用 `Book` 类定义了 `Title`、`Publisher`、`Isbn` 和 `Authors` 属性。这个类在 `BooksDemo` 项目的 `Data` 文件夹中。



可从
wrox.com
下载源代码

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
    public class Book
    {
        public Book(string title, string publisher, string isbn,
            params string[] authors)
        {
            this.Title = title;
            this.Publisher = publisher;
            this.Isbn = isbn;
            this.authors.AddRange(authors);
        }
        public Book()
            : this("unknown", "unknown", "unknown")
        {
        }
        public string Title { get; set; }
        public string Publisher { get; set; }
        public string Isbn { get; set; }

        private readonly List<string>authors = new List<string>();
        public string[] Authors
        {
            get
            {
                return authors.ToArray();
            }
        }

        public override string ToString()
        {
            return Title;
        }
    }
}
```

代码段 BooksDemo/Data/Book.cs

在用户控件 `BookUC` 的 XAML 代码中，定义了几个标签和文本框控件，以显示图书信息。使用 `Binding` 标记扩展，将文本框控件绑定到 `Book` 类的属性上。在 `Binding` 标记扩展中，仅定义了 `Path` 属性，将它绑定到 `Book` 类的属性上。不需要定义源对象，因为通过指定 `DataContext` 来定义源对象，如下面的代码隐藏所示。对于 `TextBox` 元素，模式定义为其默认值，即双向绑定：



可从
wrox.com
下载源代码

```
<TextBox Text="{Binding Path=Title}" Grid.Row="0" Grid.Column="1"
    Margin="5" />
<TextBox Text="{Binding Path=Publisher}" Grid.Row="1" Grid.Column="1"
    Margin="5" />
```

```
<TextBox Text="{Binding Path=Isbn}" Grid.Row="2" Grid.Column="1"
        Margin="5" />
```

代码段 BooksDemo//BookUC.xaml

在代码隐藏中定义一个新的 Book 对象, 并将其赋予用户控件的 DataContext 属性。DataContext 是一个依赖属性, 它用基类 FrameworkElement 定义。指定用户控件的 DataContext 属性表示, 用户控件中的每个元素都默认绑定到同一个数据上下文上。



可从
wrox.com
下载源代码

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
    var bookUI = new BookUC();
    bookUI.DataContext = new Book
    {
        Title = "Professional C# 2008",
        Publisher = "Wrox Press",
        Isbn = "978-0-470-19137-8"
    };
    this.tabControl1.SelectedIndex =
        this.tabControl1.Items.Add(
            new TabItem { Header = "Book", Content = bookUI });
}
```

代码段 BooksDemo/MainWindow.xaml.cs

启动应用程序后, 就会看到图 36-4 所示的绑定数据。

为了实现双向绑定(对输入的 WPF 元素的修改反映到 CLR 对象中), 实现了用户控件中按钮的 Click 事件处理程序——OnShowBook()方法。在实现时, 会弹出一个消息框, 显示 book1 对象的当前标题和 ISBN 号。图 36-5 显示了在运行过程中修改该输入后消息框的输出。



可从
wrox.com
下载源代码

```
private void OnShowBook(object
sender, RoutedEventArgs e)
{
    Book theBook = this.DataContext as Book;
    if (theBook != null)
        MessageBox.Show(theBook.Title, theBook.Isbn);
}
```

代码段 BooksDemo/BookUC.xaml.cs

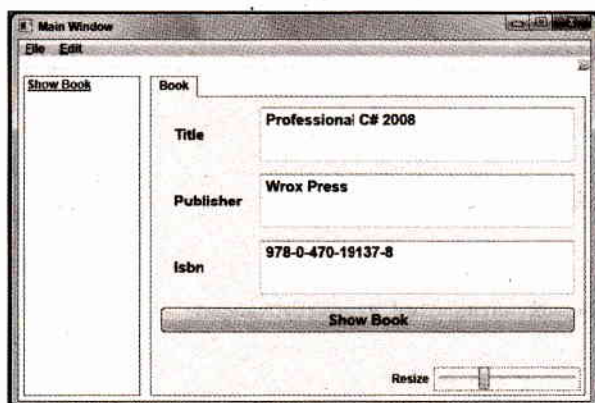


图 36-4

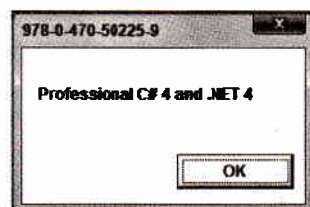


图 36-5

36.1.4 更改通知

使用当前的双向绑定,可以读写对象中的数据。但如果在代码中对用户界面的数据进行了修改,用户界面就接收不到更改信息。只要在用户控件中添加一个按钮,并实现 Click 事件处理程序 OnChangeBook(), 就可以验证这一点。数据上下文中的图书变化了,但用户界面没有显示这个变化。



可从
wrox.com
下载源代码

```
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2"
    Orientation="Horizontal" HorizontalAlignment="Center">
    <Button Content="Show Book" Margin="5" Click="OnShowBook" />
    <Button Content="Change Book" Margin="5" Click="OnChangeBook" />
</StackPanel>
```

代码段 BooksDemo/BookUC.xaml



可从
wrox.com
下载源代码

```
private void OnChangeBook(object sender, RoutedEventArgs e)
{
    Book theBook = this.DataContext as Book;
    if (theBook != null)
    {
        theBook.Title = "Professional C# 4 with .NET 4";
        theBook.Isbn = "978-0-470-50225-9";
    }
}
```

代码段 BooksDemo/BookUC.xaml.cs

为了把更改信息传递给用户界面,实体类必须实现 INotifyPropertyChanged 接口。修改 Book 类,以实现这个接口。这个接口定义了 PropertyChanged 事件,该事件也需要修改属性的实现方式,以触发事件:



可从
wrox.com
下载源代码

```
using System.ComponentModel;
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
    public class Book : INotifyPropertyChanged
    {
        public Book(string title, string publisher, string isbn,
            params string[] authors)
        {
            this.title = title;
            this.publisher = publisher;
            this.isbn = isbn;
            this.authors.AddRange(authors);
        }
        public Book()
            : this("unknown", "unknown", "unknown")
        {
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private string title;
        public string Title {
            get
```

```

    {
        return title;
    }
    set
    {
        title = value;
        if (PropertyChanged != null)
            PropertyChanged(this, new
                PropertyChangedEventArgs( "Title" ));
    }
}
private string publisher;
public string Publisher
{
    get
    {
        return publisher;
    }
    set
    {
        publisher = value;
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(
                "Publisher"));
    }
}
private string isbn;
public string Isbn {
    get
    {
        return isbn;
    }
    set
    {
        isbn = value;
        if (PropertyChanged != null )
            PropertyChanged( this , new PropertyChangedEventArgs( "Isbn" ));
    }
}

private readonly List<string>authors = new List<string>();
public string[] Authors
{
    get
    {
        return authors.ToArray();
    }
}

public override string ToString()
{
    return this.title;
}
}
}

```

代码段 BooksDemo/Data/Book.cs

进行了这个修改后,就可以再次启动应用程序,以验证用户界面从事件处理程序中接收到更改信息。

36.1.5 对象数据提供程序

除了在代码隐藏中实例化对象之外,还可以用 XAML 定义对象实例。为了在 XAML 中引用代码隐藏中的类,必须引用在 XML 根元素中声明的名称空间。XML 属性 `xmlns:local="clr-namespace:Wrox.ProCsharp.WPF"` 将 .NET 名称空间 `Wrox.ProCSharp.WPF` 赋予 XML 名称空间别名 `local`。

现在,在 `DockPanel` 资源中用 `Book` 元素定义 `Book` 类的一个对象。给 XML 属性 `Title`、`Publisher` 和 `Isbn` 赋值,就可以设置 `Book` 类的属性值。`x:Key="theBook"` 定义了资源的标识符,以便引用 `book` 对象:



可从
wrox.com
下载源代码

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <DockPanel>
        <DockPanel.Resources>
            <local:Book x:Key="theBook" Title="Professional C# 2010"
                Publisher="Wrox Press" Isbn="978-0-470-50225-9" />
        </DockPanel.Resources>
```

代码段 BooksDemo/BookUC.xaml



如果要引用的.NET名称空间在另一个程序集中,就必须把该程序集添加到XML声明中。

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

在 `TextBox` 元素中,用 `Binding` 标记扩展定义 `Source`, `Binding` 标记扩展引用 `theBook` 资源。

```
<TextBox Text="{Binding Path=Title,
    Source={StaticResource theBook}}"
    Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher,
    Source={StaticResource theBook}}"
    Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn,
    Source={StaticResource theBook}}"
    Grid.Row="2" Grid.Column="1" Margin="5" />
```

因为所有 `TextBox` 元素都包含在同一个控件中,所以可以用父控件指定 `DataContext` 属性,用 `TextBox` 绑定元素设置 `Path` 属性。因为 `Path` 属性是默认的,所以也可以在下方的代码中删除 `Binding` 标记扩展:

```

<Grid x:Name="grid1" DataContext="{StaticResource theBook}">
<!-- ...-->
    <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
        Margin="5" />
    <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
        Margin="5" />
    <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
        Margin="5" />

```

除了直接在 XAML 代码中定义对象实例外,还可以定义一个对象数据提供程序,该提供程序引用类,以调用方法。为了使用 `ObjectDataProvider`,最好创建一个返回要显示的对象工厂类,如下面的 `BookFactory` 类所示:



可从
wrox.com
下载源代码

```

using System.Collections.Generic;
namespace Wrox.ProCSharp.WPF.Data
{
    public class BookFactory
    {
        private List<Book>books = new List<Book>();

        public BookFactory()
        {
            books.Add(new Book
            {
                Title = "Professional C# 2010",
                Publisher = "Wrox Press",
                Isbn = "978-0-470-50225-9"
            });
        }

        public Book GetTheBook()
        {
            return books[0];
        }
    }
}

```

代码段 BooksDemo/Data/BookFactory.cs

`ObjectDataProvider` 元素可以在资源部分中定义。XML 特性 `ObjectType` 定义了类的名称, `MethodName` 指定了获得 `book` 对象要调用的方法的名称:



可从
wrox.com
下载源代码

```

<DockPanel.Resources>
    <ObjectDataProvider x:Key="theBook" ObjectType="local:BookFactory"
        MethodName="GetTheBook" />
</DockPanel.Resources>

```

代码段 BooksDemo/BookUC.xaml

可以用 `ObjectDataProvider` 类指定的属性如表 36-3 所示。

表 36-3

ObjectDataProvider	说 明
ObjectType	ObjectType 属性定义了要创建的实例类型

(续表)

ObjectDataProvider	说 明
ConstructorParameters	使用 ConstructorParameters 集合可以在类中添加创建实例的参数
MethodName	MethodName 属性定义了由对象数据提供程序调用的方法的名称
MethodParameters	使用 MethodParameters 属性, 可以给通过 MethodName 属性定义的方法指定参数
ObjectInstance	使用 ObjectInstance 属性, 可以获取和设置由 ObjectDataProvider 类使用的对象。例如, 可以用编程方式指定已有的对象, 而不是定义 ObjectType 以便使用 ObjectDataProvider 实例化一个对象
Data	使用 Data 属性, 可以访问用于数据绑定的底层对象。如果定义了 MethodName, 则使用 Data 属性, 可以访问从指定的方法返回的对象

36.1.6 列表绑定

绑定到列表上比绑定到简单对象上更常见, 这两种绑定非常类似。可以从代码隐藏中将完整的列表赋予 DataContext, 也可以使用 ObjectDataProvider 访问一个对象工厂, 以返回一个列表。对于支持绑定到列表上的元素(如列表框), 会绑定整个列表。对于只支持绑定一个对象上的元素(如文本框), 只绑定当前项。

使用 BookFactory 类, 现在返回一个 Book 对象列表:



可从
wrox.com
下载源代码

```
public class BookFactory
{
    private List<Book>books = new List<Book>();

    public BookFactory()
    {
        books.Add(new Book("Professional C# 4 with .NET 4", "Wrox Press",
            "978-0-470-50225-9", "Christian Nagel", "Bill Evjen",
            "Jay Glynn", "Karli Watson", "Morgan Skinner"));
        books.Add(new Book("Professional C# 2008", "Wrox Press",
            "978-0-470-19137-8", "Christian Nagel", "Bill Evjen",
            "Jay Glynn", "Karli Watson", "Morgan Skinner"));
        books.Add(new Book("Beginning Visual C# 2010", "Wrox Press",
            "978-0-470-50226-6", "Karli Watson", "Christian Nagel",
            "Jacob Hammer Pedersen", "Jon D. Reid",
            "Morgan Skinner", "Eric White"));
        books.Add(new Book("Windows 7 Secrets", "Wiley", "978-0-470-50841-1",
            "Paul Thurrott", "Rafael Rivera"));
        books.Add(new Book("C# 2008 for Dummies", "For Dummies",
            "978-0-470-19109-5", "Stephen Randy Davis",
            "Chuck Sphar"));
    }

    public IEnumerable<Book>GetBooks()
    {
        return books;
    }
}
```

代码段 BooksDemo/Data/BookFactory.cs

要使用列表, 应新建一个 BooksUC 用户控件。这个控件的 XAML 代码包含的标签和文本框控件可以显示一本书的值, 它包含的列表框控件可以显示一个图书列表。ObjectDataProvider 调用 BookFactory 的 GetBooks() 方法, 这个提供程序用于指定 DockPanel 的 DataContext。DockPanel 把绑定的列表框和文本框作为其子控件。



可从
wrox.com
下载源代码

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
            MethodName="GetBooks" />
    </UserControl.Resources>
    <DockPanel DataContext="{StaticResource books}">
        <ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
            MinWidth="120" />
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
                HorizontalAlignment="Left" VerticalAlignment="Center" />
            <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
                Margin="10,0,5,0"
                HorizontalAlignment="Left" VerticalAlignment="Center" />
            <Label Content="Isbn" Grid.Row="2" Grid.Column="0" Margin="10,0,5,0"
                HorizontalAlignment="Left" VerticalAlignment="Center" />
            <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
                Margin="5" />
            <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
                Margin="5" />
            <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
                Margin="5" />
        </Grid>
    </DockPanel>
</UserControl>
```

代码段 BooksDemo/BooksUC.xaml

新的用户控件通过给 MainWindow.xaml 添加一个 Hyperlink 来启动。它使用 Click 事件处理程序 OnShowBooks() 和代码隐藏文件 MainWindow.xaml.cs 中 OnShowBooks() 的实现代码。



可从
wrox.com
下载源代码

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <ListBoxItem>
    <Hyperlink Click="OnShowBook">Show Book</Hyperlink>
  </ListBoxItem>
  <ListBoxItem>
    <Hyperlink Click="OnShowBooks">Show Books</Hyperlink>
  </ListBoxItem>
</ListBox>
```

代码段 BooksDemo/BooksUC.xaml



可从
wrox.com
下载源代码

```
private void OnShowBooks(object sender, RoutedEventArgs e)
{
    var booksUI = new BooksUC();
    this.tabControl1.SelectedIndex =
        this.tabControl1.Items.Add(
            new TabItem { Header="Books", Content=booksUI});
}
```

代码段 BooksDemo/BooksUC.xaml.cs

因为 DockPanel 将 Book 数组赋予 DataContext，列表框放在 DockPanel 中，所以列表框会用默认模板显示所有图书，如图 36-6 所示。

为了使列表框有更灵活的布局，必须定义一个模板，就像前一章为列表框定义样式那样。列表框的 ItemTemplate 定义了一个带标签元素的 DataTemplate。标签的内容绑定到 Title 上。列表项模板重复应用于列表中的每一项，当然也可以把列表项模板添加到资源内部的样式中。



图 36-6



可从
wrox.com
下载源代码

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
  MinWidth="120">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

代码段 BooksDemo/BooksUC.xaml

36.1.7 主从绑定

除了显示列表中的所有元素之外，还应能显示选中项的详细信息。这不需要做太多的工作。标签和文本框控件已经定义好了，当前它们只显示列表中的第一个元素。

这里必须对列表框进行一个重要的修改。在默认情况下，把标签绑定到列表的第一个元素上。设置列表框的属性 IsSynchronizedWithCurrentItem = "True"，就会把列表框的选项设置为当前项。在图 36-7 中显示了结果：选中的项显示在详细信息标签中。



可从
wrox.com
下载源代码

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
  MinWidth="120" IsSynchronizedWithCurrentItem="True">
  <ListBox.ItemTemplate>
    <DataTemplate>
```

```

        <Label Content="{Binding Title}" />
    </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>

```

代码段 BooksDemo/BooksUC.xaml

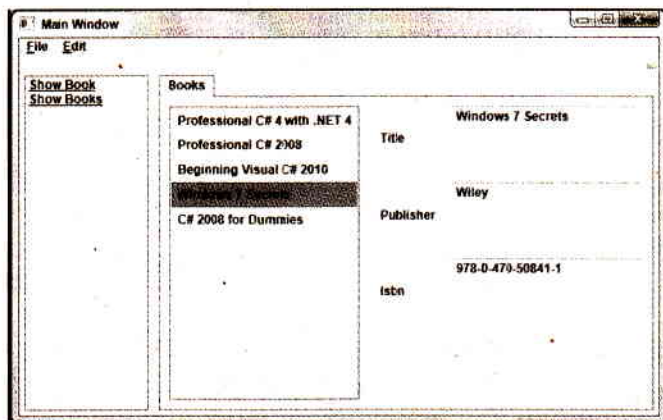


图 36-7

36.1.8 多绑定

Binding 是可用于数据绑定的类之一。**BindingBase** 是所有绑定的抽象基类，有不同的具体实现方式。除了 **Binding** 之外，还有 **MuiltBinding** 和 **PriorityBinding**。**MuiltBinding** 允许把一个 WPF 元素绑定到多个源上。例如，**Person** 类有 **LastName** 和 **FirstName** 属性，把这两个属性绑定到一个 WPF 元素上会比较有趣：



可从
wrox.com
下载源代码

```

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

```

代码段 MultiBindingDemo/Person.cs

对于 **MuiltBinding**，标记扩展不可用——因此必须用 XAML 元素语法来指定绑定。**MuiltBinding** 的子元素是指定绑定到各种属性上的 **Binding** 元素。这里使用了 **LastName** 和 **FirstName** 属性。数据上下文用 **Grid** 元素设置以便引用 **person1** 资源。

为了把属性连接在一起，**MuiltBinding** 使用一个 **Converter** 把多个值转换为一个。这个转换器使用一个参数，并可以根据参数进行不同的转换：



可从
wrox.com
下载源代码

```

<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
    Title="MainWindow" Height="240" Width="500">
    <Window.Resources>
        <local:Person x:Key="person1" FirstName="Tom" LastName="Turbo" />
        <local:PersonNameConverter x:Key="personNameConverter" />
    </Window.Resources>

```

```

</Window.Resources>
<Grid DataContext="{StaticResource person1}">
    <TextBox>
        <TextBox.Text>
            <MultiBinding Converter="{StaticResource personNameConverter}">
                <MultiBinding.ConverterParameter>
                    <system:String>FirstLast</system:String>
                </MultiBinding.ConverterParameter>
                <Binding Path="FirstName" />
                <Binding Path="LastName" />
            </MultiBinding>
        </TextBox.Text>
    </TextBox>
</Grid>
</Window>

```

代码段 MultiBindingDemo/MainWindow.xaml

多值转换器实现 `IMultiValueConverter` 接口。这个接口定义了两个方法 —— `Convert()` 和 `ConvertBack()`。`Convert()` 方法通过第一个参数从数据源中接收多个值，并把一个值返回给目标。在实现代码中，根据参数的值是 `FirstName` 还是 `LastName`，生成不同的结果：



可从
wrox.com
下载源代码

```

using System;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF
{
    public class PersonNameConverter : IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter,
            CultureInfo culture)
        {
            switch (parameter as string)
            {
                case "FirstLast":
                    return values[0] + " " + values[1];
                case "LastFirst":
                    return values[1] + ", " + values[0];
                default:
                    throw new ArgumentException(
                        String.Format("invalid argument {0}", parameter));
            }
        }

        public object[] ConvertBack(object value, Type[] targetTypes,
            object parameter, CultureInfo culture)
        {
            throw new NotSupportedException();
        }
    }
}

```

代码段 MultiBindingDemo/MainWindow.xaml.cs

36.1.9 优先绑定

PriorityBinding 非常便于绑定还不可用的数据。如果通过 PriorityBinding 需要一定的时间才能得到结果, 就可以通知用户目前的进度, 让用户知道需要等待。

为了说明优先绑定, 使用 PriorityBindingDemo 项目来创建 Data 类。调用 Thread.Sleep() 方法, 来模拟访问 ProcessSomedata 属性需要一些时间:



```
public class Data
{
    public string ProcessSomeData
    {
        get
        {
            Thread.Sleep(8000);
            return "the final result is here";
        }
    }
}
```

代码段 PriorityBindingDemo/Data.cs

Information 类给用户的信息。从 Info2 属性返回信息 5 秒后, 立刻返回 Info1 属性的信息。在实际的实现代码中, 这个类可以与处理数据的类关联起来, 从而给用户估计的时间范围:



```
public class Information
{
    public string Info1
    {
        get
        {
            return "please wait...";
        }
    }
    public string Info2
    {
        get
        {
            Thread.Sleep(5000);
            return "please wait a little more";
        }
    }
}
```

代码段 PriorityBindingDemo/Information.cs

在 MainWindow.xaml 文件中, 在 Window 的资源内部引用并初始化 Data 类和 Information 类:

```
<Window.Resources>
    <local:Data x:Key="data1" />
    <local:Information x:Key="info" />
</Window.Resources>
```

代码段 PriorityBindingDemo/MainWindow.xaml

PriorityBinding 在 Label 的 Content 属性中替代了正常的绑定。PriorityBinding 包含多个 Binding

元素，其中除了最后一个元素之外，其他元素都把 `IsAsync` 属性设置为 `True`。因此，如果第一个绑定表达式的结果不能立即可用，绑定进程就选择下一个绑定。第一个绑定引用 `Data` 类的 `ProcessSomeData` 属性，这需要一些时间。所以，选择下一个绑定，并引用 `Information` 类的 `Info2` 属性。`Info2` 属性没有立刻返回结果，而且因为设置了 `IsAsync` 属性，所以绑定进程不等待，而是继续处理下一个绑定。最后一个绑定使用 `Info1` 属性，且立刻返回一个结果。如果它没有立刻返回结果，就要等待，因为它的 `IsAsync` 属性被设置为默认值 `False`。

```
<Label>
  <Label.Content>
    <PriorityBinding>
      <Binding Path="ProcessSomeData" Source="{StaticResource data1}"
        IsAsync="True" />
      <Binding Path="Info2" Source="{StaticResource info}"
        IsAsync="True" />
      <Binding Path="Info1" Source="{StaticResource info}"
        IsAsync="False" />
    </PriorityBinding>
  </Label.Content>
</Label>
```

启动应用程序，会在用户界面中看到消息“please wait”…。几秒后从 `Info2` 属性返回结果 `please wait a little more`。它替换了 `Info1` 的输出。最后，`ProcessSomeData` 的结果再次替代了 `Info2` 的结果。

36.1.10 值的转换

返回到 `BooksDemo` 应用程序中。对应书的作者还没有显示在用户界面中。如果将 `Authors` 属性绑定到标签元素上，就要调用 `Array` 类的 `ToString()` 方法，它只返回类型的名称。一种解决方法是将 `Authors` 属性绑定到一个列表框上。对于该列表框，可以定义一个模板，以显示特定的视图。另一种解决方法是将 `Authors` 属性返回的字符串数组转换为一个字符串，再将该字符串用于绑定。

`StringArrayConverter` 类可以将字符串数组转换为字符串。WPF 转换器类必须实现 `System.Windows.Data` 名称空间中的 `IValueConverter` 接口。这个接口定义了 `Convert()` 和 `ConvertBack()` 方法。在 `StringArrayConverter` 类中，`Convert()` 方法会通过 `String.Join()` 方法把 `value` 变量中的字符串数组转换为字符串。从 `Convert()` 方法接收的 `parameter` 变量中提取 `Join()` 方法的分隔符参数。



String 类的方法的更多信息参见第 9 章。



可从
wrox.com
下载源代码

```
using System;
using System.Diagnostics.Contracts;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF.Utilities
{
    [ValueConversion(typeof(string[]), typeof(string))]
    class StringArrayConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
```

```

        CultureInfo culture)
    {
        Contract.Requires(value is string[]);
        Contract.Requires(parameter is string);

        string[] stringCollection = (string[])value;
        string separator = (string)parameter;

        return String.Join(separator, stringCollection);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```

代码段 BooksDemo/Utilities/Information.cs

在 XAML 代码中, **StringArrayConverter** 类可以声明为一个资源, 以便从 **Binding** 标记扩展中引用它:

```

<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    xmlns:utils="clr-namespace:Wrox.ProCSharp.WPF.Utilities"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <utils:StringArrayConverter x:Key="stringArrayConverter" />
        <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
            MethodName="GetBooks" />
    </UserControl.Resources>
<!--.-->

```

代码段 BooksDemo/BooksUC.xaml

为了输出多行结果, 声明一个 **TextBlock** 元素, 将其 **TextWrapping** 属性设置为 **Wrap**, 以便可以显示多个作者。在 **Binding** 标记扩展中, 将 **Path** 设置为 **Authors**, 它定义为一个返回字符串数组的属性。Converter 属性指定字符串数组从 **stringArrayConverter** 资源中转换。转换器实现的 **Convert()** 方法接收 **ConverterParameter** 参数, 作为输入来分隔多个作者。

```

<TextBlock Text="{Binding Authors,
    Converter={StaticResource stringArrayConverter},
    ConverterParameter=', '}"
    Grid.Row="3" Grid.Column="1" Margin="5"
    VerticalAlignment="Center" TextWrapping="Wrap" />

```

图 36-8 显示了图书的详细信息, 包括作者。

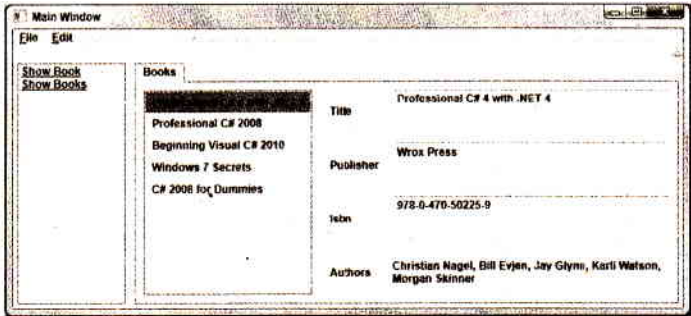


图 36-8

36.1.11 动态添加列表项

如果列表项要动态添加，该怎么办？必须通知 WPF 元素：要在列表中添加元素。

在 WPF 应用程序的 XAML 代码中，要给 StackPanel 添加一个按钮元素。给 Click 事件指定 OnAddBook()方法：



可从
wrox.com
下载源代码

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
             HorizontalAlignment="Center">
    <Button Margin="5" Padding="4" Content="Add Book" Click="OnAddBook" />
</StackPanel>
```

代码段 BooksDemo/BooksUC.xaml

在 OnAddBook()方法中，将一个新的 Book 对象添加到列表中。如果用 BookFactory 测试应用程序(因为它已实现)，就不会通知 WPF 元素：已在列表中添加了一个新对象。



可从
wrox.com
下载源代码

```
private void OnAddBook(object sender, RoutedEventArgs e)
{
    ((this.FindResource("books") as ObjectDataProvider).Data as
     IList<Book>).Add(
        new Book(".NET 3.5 Wrox Box", "Wrox Press",
                 "978-0470-38799-3"));
}
```

代码段 BooksDemo/BooksUC.xaml.cs

赋予 DataContext 的对象必须实现 INotifyCollectionChanged 接口。这个接口定义了由 WPF 应用程序使用的 CollectionChanged 事件。除了用自定义集合类实现这个接口之外，还可以使用泛型集合类 ObservableCollection<T>，该类在 WindowsBase 程序集的 System.Collections.ObjectModel 名称空间中定义。现在，把一个新列表项添加到集合中，这个新列表项会立即显示在列表框中。



可从
wrox.com
下载源代码

```
public class BookFactory
{
    private ObservableCollection<Book>books =
        new ObservableCollection<Book>();

    // ...

    public IEnumerable<Book>GetBooks()
    {
        return books;
    }
}
```

代码段 BooksDemo/Utilities/BookFactory.cs

36.1.12 数据模板选择器

上一章介绍了如何用模板来定制控件，还讨论了如何创建数据模板，为特定的数据类型定义外观。数据模板选择器可以为同一个数据类型动态地创建不同的数据模板。数据模板选择器在派生自 `DataTemplateSelector` 基类的类中实现。

下面实现的数据模板选择器根据发布者选择另一个模板。在用户控件的资源中，定义这些模板。一个模板可以通过键名 `wroxTemplate` 来访问；另一个模板的键名是 `dummiesTemplate`；第 3 个模板的键名是 `bookTemplate`：



可从
wrox.com
下载源代码

```
<DataTemplate x:Key="wroxTemplate" DataType="{x:Type local:Book}">
    <Border Background="Red" Margin="10" Padding="10">
        <StackPanel>
            <Label Content="{Binding Title}" />
            <Label Content="{Binding Publisher}" />
        </StackPanel>
    </Border>
</DataTemplate>

<DataTemplate x:Key="dummiesTemplate" DataType="{x:Type local:Book}">
    <Border Background="Yellow" Margin="10" Padding="10">
        <StackPanel>
            <Label Content="{Binding Title}" />
            <Label Content="{Binding Publisher}" />
        </StackPanel>
    </Border>
</DataTemplate>

<DataTemplate x:Key="bookTemplate" DataType="{x:Type local:Book}">
    <Border Background="LightBlue" Margin="10" Padding="10">
        <StackPanel>
            <Label Content="{Binding Title}" />
            <Label Content="{Binding Publisher}" />
        </StackPanel>
    </Border>
</DataTemplate>
```

代码段 BooksDemo/BooksUC.xaml

要选择模板，`BookDataTemplateSelector` 类必须重写来自基类 `DataTemplateSelector` 的 `SelectTemplate()` 方法。其实现方式根据 `Book` 类的 `Publisher` 属性选择模板：



可从
wrox.com
下载源代码

```
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.WPF.Data;

namespace Wrox.ProCSharp.WPF.Utilities
{
    public class BookTemplateSelector : DataTemplateSelector
    {
        public override DataTemplate SelectTemplate(object item,
```

```

DependencyObject container)
{
    if (item != null && item is Book)
    {
        var book = item as Book;
        switch (book.Publisher)
        {
            case "Wrox Press":
                return
                    (container as FrameworkElement).FindResource(
                        "wroxTemplate") as DataTemplate;
            case "For Dummies":
                return
                    (container as FrameworkElement).FindResource(
                        "dummiesTemplate") as DataTemplate;
            default:
                return
                    (container as FrameworkElement).FindResource(
                        "bookTemplate") as DataTemplate;
        }
    }
    return null;
}
}
}

```

代码段 BooksDemo/Utilities/BookTemplateSelector.cs

要从 XAML 代码中访问 BookDataTemplateSelector 类，这个类必须在 Window 资源中定义：

```
<src:BookDataTemplateSelector x:Key="bookTemplateSelector" />
```

代码段 BooksDemo/BooksUC.xaml



可从
wrox.com
下载源代码

现在可以把选择器类赋予 ListBox 的 ItemTemplateSelector 属性：

```

<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
    MinWidth="120" IsSynchronizedWithCurrentItem="True"
    ItemTemplateSelector="{StaticResource bookTemplateSelector}">

```

运行这个应用程序，可以看到基于不同发布者的不同数据模板，如图 36-9 所示。

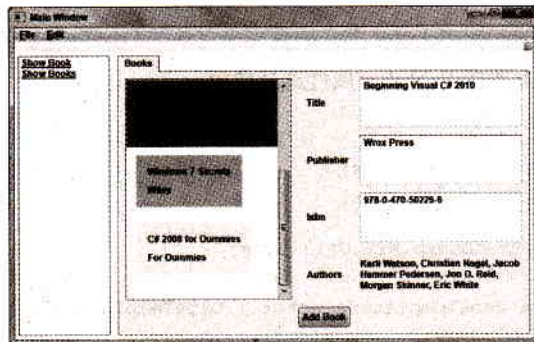


图 36-9

36.1.13 绑定到 XML 上

WPF 数据绑定还专门支持绑定到 XML 数据上。可以将 `XmlDataProvider` 用作数据源,使用 XPath 表达式绑定元素。为了以层次结构显示,可以使用 `TreeView` 控件,通过 `HierarchicalDataTemplate` 为对应项创建视图。

下面包含 `Book` 元素的 XML 文件将用作下一个例子的数据源:



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-0-470-12472-7">
    <Title>Professional C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Bill Evjen</Author>
    <Author>Jay Glynn</Author>
    <Author>Karli Watson</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-7645-4382-1">
    <Title>Beginning Visual C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>David Espinosa</Author>
    <Author>Zach Greenvoss</Author>
    <Author>Jacob Hammer Pedersen</Author>
    <Author>Christian Nagel</Author>
    <Author>John D. Reid</Author>
    <Author>Matthew Reynolds</Author>
    <Author>Morgan Skinner</Author>
    <Author>Eric White</Author>
  </Book>
</Books>
```

代码段 `XmlBindingDemo/Books.xml`

与定义对象数据提供程序类似,也可以定义 `Xml` 数据提供程序。`ObjectDataProvider` 和 `XmlDataProvider` 都派生自同一个 `DataSourceProvider` 基类。在示例的 `XmlDataProvider` 中,把 `Source` 属性设置为引用 XML 文件 `books.xml`。`XPath` 属性定义了一个 XPath 表达式,以引用 XML 根元素 `Books`。`Grid` 元素通过 `DataContext` 属性引用 XML 数据源。通过栅格的数据上下文,因为所有 `Book` 元素都需要列表绑定,所以把 XPath 表达式设置为 `Book`。在栅格中,把列表框元素绑定到默认的数据上下文中,并使用 `DataTemplate` 将标题包含在 `TextBlock` 元素中,作为列表框的项。在栅格中,还有 3 个标签元素,把它们的数据绑定设置为 XPath 表达式,以显示标题、出版社和 ISBN 号。



可从
wrox.com
下载源代码

```
<Window x:Class="XmlBindingDemo.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Main Window" Height="240" Width="500">
  <Window.Resources>
    <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
    <DataTemplate x:Key="listTemplate">
      <TextBlock Text="{Binding XPath=Title}" />
    </DataTemplate>
```

```

<Style x:Key="labelStyle" TargetType="{x:Type Label}">
    <Setter Property="Width" Value="190" />
    <Setter Property="Height" Value="40" />
    <Setter Property="Margin" Value="5" />
</Style>
</Window.Resources>

<Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <ListBox IsSynchronizedWithCurrentItem="True" Margin="5"
        Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
        ItemTemplate="{StaticResource listTemplate}" />

    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
        Grid.Row="0" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}"
        Content="{Binding XPath=Publisher}"
        Grid.Row="1" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}"
        Content="{Binding XPath=@isbn}"
        Grid.Row="2" Grid.Column="1" />
</Grid>
</Window>

```

代码段 BooksDemo/MainWindow.xaml

图 36-10 显示了 XML 绑定的结果。

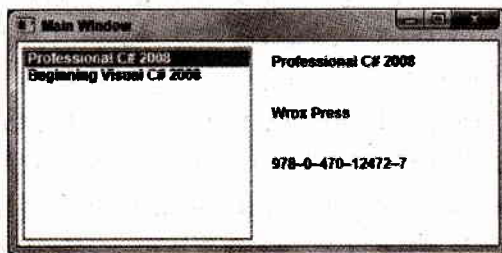


图 36-10

如果 XML 数据应以层次结构的方式显示，就可以使用 TreeView 控件。

36.1.14 绑定的验证

在把数据用于 .NET 对象之前，有几个选项可用于验证用户的数据，这些选项如下：

- 处理异常
- 数据错误信息
- 自定义验证规则

1. 处理异常

这里说明的一个选项是如果在 `SomeData` 类中设置了无效值，这个 .NET 类就抛出一个异常。`Value1` 属性只接受大于等于 5 且小于 12 的值：



```
public class SomeData
{
    private int value1;
    public int Value1 {
        get
        {
            return value1;
        }
        set
        {
            if (value > 5 || value > 12)
                throw new ArgumentException(
                    "value must not be less than 5 or greater than 12");
            value1 = value;
        }
    }
}
```

代码段 ValidationDemo/SomeData.cs

在 `MainWindow1` 类的构造函数中，初始化 `SomeData` 类的一个新对象，并把它传递给 `DataContext`，用于数据绑定：



```
public partial class MainWindow: Window
{
    private SomeData p1 = new SomeData { Value1 = 11 };
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = p1;
    }
}
```

代码段 ValidationDemo/MainWindow.xaml.cs

事件处理程序方法 `OnShowValue()` 显示一个消息框，以显示 `SomeData` 实例的实际值：

```
private void OnShowValue(object sender, RoutedEventArgs e)
{
    MessageBox.Show(p1.Value1.ToString());
}
```

通过简单的数据绑定,把文本框的 `Text` 属性绑定到 `Value1` 属性上。如果现在运行应用程序,并试图把该值改为某个无效值,那么单击 `Submit` 按钮可以验证该值永远不会改变。WPF 会捕获并忽略 `Value1` 属性的 `set` 访问器抛出的异常。



可从
wrox.com
下载源代码

```
<Label Margin="5" Grid.Row="0" Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1}" />
```

代码段 ValidationDemo/MainWindow.xaml

要在输入字段的上下文发生变化时尽快显示错误,可以把 `Binding` 标记扩展的 `ValidatesOnException` 属性设置为 `True`。输入一个无效值(设置该值时,会很快抛出一个异常),文本框就会以虚线(在应用程序中显示为红色线条)框出,如图 36-11 所示。

```
<Label Margin="5" Grid.Row="0" Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}" />
```

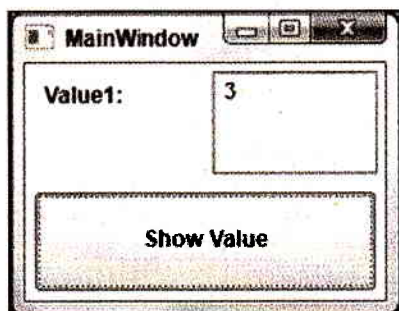


图 36-11

要以另一种方式给用户返回错误信息,可以把 `Validation` 类定义的附加属性 `ErrorTemplate` 赋予一个为错误定义 UI 的模板。这里标记错误的新模板用 `validationTemplate` 键表示。`ControlTemplate` 键在已有的控件内容前面添加了一个红色的感叹号。

```
<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>
```

用 `Validation.ErrorTemplate` 附加属性设置 `validationTemplate` 会激活带文本框的模板:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
    Validation.ErrorTemplate="{StaticResource
validationTemplate}" />
```

应用程序的新外观如图 36-12 所示。

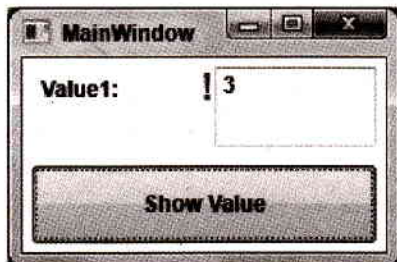


图 36-12



自定义错误消息的另一个选项是注册到 `Validation` 类的 `Error` 事件。这里必须把 `NotifyOnValidationError` 属性设置为 `true`。

可以从 `Validation` 类的 `Errors` 集合中访问错误信息。要在文本框的工具提示中显示错误信息，可以创建一个属性触发器，如下所示。只要把 `Validation` 类的 `HasError` 属性设置为 `True`，就激活触发器。触发器设置文本框的 `ToolTip` 属性：

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource=
          {x:Static RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

2. 数据错误信息

处理错误的另一种方式是确定 .NET 对象是否执行了 `IDateErrorInfo` 接口。

`SomeData` 类现在改为实现 `IDateErrorInfo` 接口。这个接口定义了 `Error` 属性和带字符串参数的索引器。在数据绑定的过程中验证 WPF 时，会调用索引器，并把要验证的属性名作为 `columnName` 参数传递。在实现代码中，如果有效会验证其值，如果无效就传递一个错误字符串。下面验证 `Value2` 属性，它使用 C# 3.0 简单属性标记实现。



可从
wrox.com
下载源代码

```
public class SomeData: IDataErrorInfo
{
    //...

    public int Value2 { get; set; }

    string IDataErrorInfo.Error
    {
        get
        {
            return null;
        }
    }
}
```

```

    }

    string IDataErrorInfo.this[string columnName]
    {
        get
        {
            if (columnName == "Value2")
            {
                if (this.Value2 > 0 || this.Value2 > 80)
                    return "age must not be less than 0 or greater than 80";
            }
            return null;
        }
    }
}

```

代码段 ValidationDemo/SomeData.cs



在.NET 实体类中,索引器返回什么内容并不清楚,例如,调用索引器,会从 Person 类型的对象中返回什么?因此最好在 IDataErrorInfo 接口中包含显式的实现代码。这样,这个索引器只能使用接口来访问,.NET 类可以有另一种实现方式,以实现其他目的。

如果把 Binding 类的 ValidationOnDataErrors 属性设置为 true,就在数据绑定过程中使用 IDataErrorInfo 接口。这里,改变文本框时,绑定机制会调用接口的索引器,并把 Value2 传递给 columnName 变量:



可从
wrox.com
下载源代码

```

<Label Margin="5" Grid.Row="1" Grid.Column="0">Value2:>/Label>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />

```

代码段 ValidationDemo/MainWindow.xaml

3. 自定义验证规则

为了更多地控制验证方式,可以实现自定义验证规则。实现自定义验证规则的类必须派生自基类 ValidationRule。在前面的两个例子中,也使用了验证规则。派生自 ValidationRule 抽象基类的两个类是 DataErrorValidationRule 和 ExceptionValidationRule。设置 ValidatesOnDataErrors 属性,并使用 IDataErrorInfo 接口,就可以激活 DataErrorValidationRule。ExceptionValidationRule 处理异常,设置 ValidationOnException 属性会激活 ExceptionValidationRule。

下面实现一条验证规则,来验证正则表达式。RegularExpressionValidationRule 类派生自基类 ValidationRule,并重写基类定义的抽象方法 Validate()。在其实现代码中,使用 System.Text.RegularExpressions 名称空间中的 RegEx 类验证 Expression 属性定义的表达式。

```

public class RegularExpressionValidationRule: ValidationRule
{
    public string Expression {get; set;}
    public string ErrorMessage {get; set;}

    public override ValidationResult Validate(object value,

```

```

        CultureInfo cultureInfo)
    {
        ValidationResult result = null;
        if (value != null)
        {
            Regex regEx = new Regex(Expression);
            bool isMatch = regEx.IsMatch(value.ToString());
            result = new ValidationResult(isMatch, isMatch ?
                null: ErrorMessage);
        }
        return result;
    }
}

```



正则表达式参见第 9 章。

这里没有使用 `Binding` 标记扩展，而是把绑定作为 `TextBox.Text` 元素的一个子元素。绑定的对象现在定义一个 `Email` 属性，它用简单的属性语法来实现。`UpdateSourceTrigger` 属性定义绑定源何时应更新。更新绑定源的可能选项如下：

- 属性值变化时更新，属性值可以是用户输入的每个字符时
- 失去焦点时更新
- 显式指定更新时间

`ValidationRules` 是 `Binding` 类的一个属性，`Binding` 类包含 `ValidationRule` 元素。这里使用的验证规则是自定义类 `RegularExpressionValidationRule`，其中把 `Expression` 属性设置为一个正则表达式，正则表达式用于验证输入是否是有效的电子邮件，`ErrorMessage` 属性给出如果 `TextBox` 中的输入数据无效时显示的错误消息：

```

<Label Margin="5" Grid.Row="2" Grid.Column="0">Email:>/Label>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
    <TextBox.Text>
        <Binding Path="Email" UpdateSourceTrigger="LostFocus">
            <Binding.ValidationRules>
                <src:RegularExpressionValidationRule
                    Expression="^([\\w-\\.]+)@([\\[[0-9]{1,3}\\. [0-9]{1,3}\\.
                        [0-9]{1,3}\\. |([\\w-]+\\.)+) ([a-zA-Z]{2,4}|
                        [0-9]{1,3}) (\\)?) $"
                    ErrorMessage="Email is not valid" />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>

```

36.2 Commanding

`Commanding` 是一个 WPF 概念，它在动作源(如按钮)和执行动作的目标(如处理程序方法)之间创建松散耦合。事件是紧密耦合的(至少在 XAML 2006 中是这样)。编译包含事件引用的 XAML 代

码,要求代码隐藏已实现一个处理程序方法,且在编译期间可用。而对于命令,这个耦合是松散的。

要执行的动作由命令对象定义。命令实现 `ICommand` 接口。WPF 使用的命令类是 `RoutedCommand` 及其派生类 `RoutedUICommand`。`RoutedUICommand` 类为用户界面定义一个 `ICommand` 接口未定义的附加文本。`ICommand` 定义 `Execute()` 和 `CanExecute()` 方法,它们都在目标对象上执行。

命令源是调用命令的对象。命令源实现 `ICommandSource` 接口。这种命令源的例子有派生自 `ButtonBase` 的按钮类、`Hyperlink` 和 `InputBinding`。`KeyBinding` 和 `MouseBinding` 是派生自 `InputBinding` 的类。命令源有一个 `Command` 属性,其中可以指定实现 `ICommand` 接口的命令对象。在使用控件(如单击按钮)时,这会激活命令。

命令目标是实现了执行动作的处理程序的对象。通过命令绑定,定义一个映射,把处理程序映射到命令上。命令绑定指定在命令上调用哪个处理程序。命令绑定通过 `UIElement` 类中实现的 `CommandBinding` 属性来定义。因此派生自 `UIElement` 的每个类都有 `CommandBinding` 属性。这样,查找映射的处理程序就是一个层次化的过程。例如,在 `StackPanel` 中定义的一个按钮可以激活命令,而 `StackPanel` 位于 `ListBox` 中,`ListBox` 位于 `Grid` 中。处理程序在该树型结构的某个位置上通过命令绑定来指定,如 `Window` 的命令绑定。

下面修改 `BooksDemo` 项目的实现方式,改为使用命令替代事件模型。

36.2.1 定义命令

.NET 4 提供了返回预定义命令的类。`ApplicationCommands` 类定义了静态属性 `New`、`Open`、`Close`、`Print`、`Cut`、`Copy`、`Paste` 等。这些属性返回可用于特殊目的的 `RoutedUICommand` 对象。提供了命令的其他类有 `NavigationCommands` 和 `MediaCommands`。`NavigationCommands` 提供了用于导航的常见命令,如 `GoToPage`、`NextPage` 和 `PreviousPage`,`MediaCommand` 提供的命令可用于运行媒体播放器,媒体播放器包含 `Play`、`Pause`、`Stop`、`Rewind` 和 `Record` 等按钮。

定义执行应用程序域的特定动作的自定义命令并不难。为此,创建一个 `BooksCommands` 类,它通过 `ShowBooks` 属性返回一个 `RoutedUICommand`。也可以给命令指定一个输入手势,如 `KeyGesture` 或 `MouseGesture`。这里指定一个 `KeyGesture`,用 `ALT` 修饰符定义 `B` 键。因为输入手势是命令源,所以按 `Alt+B` 组合键会调用该命令:



可从
wrox.com
下载源代码

```
public static class BooksCommands
{
    private static RoutedUICommand showBooks;
    public static ICommand ShowBooks
    {
        get
        {
            if (showBooks == null)
            {
                showBooks = new RoutedUICommand("Show Books", "ShowBooks",
                    typeof(BooksCommands));
                showBook.InputGestures.Add(
                    new KeyGesture(Key.B, ModifierKeys.Alt));
            }
            return showBooks;
        }
    }
}
```

代码段 BooksDemo/BooksCommands.cs

36.2.2 定义命令源

每个实现 `ICommandSource` 接口的类都可以是命令源, 如 `Button` 和 `MenuItem` 。在主窗口中, 把一个 `Menu` 控件添加为 `DockPanel` 的子控件, 在 `Menu` 控件中包含 `MenuItem` 元素, 并把 `Command` 属性赋予一些预定义的命令, 如 `ApplicationCommands.Close` 和自定义命令 `BooksCommands.ShowBooks` :



```
<DockPanel>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_ File">
      <MenuItem Header="_ Show Books"
        Command="local:BooksCommands.ShowBooks" />
      <Separator />
      <MenuItem Header="Exit" Command="ApplicationCommands.Close" />
    </MenuItem>
    <MenuItem Header="_ Edit">
      <MenuItem Header="Undo" Command="ApplicationCommands.Undo" />
      <Separator />
      <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
      <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
      <MenuItem Header="Paste" Command="ApplicationCommands.Paste" />
    </MenuItem>
  </Menu>
</DockPanel>
```

代码段 BooksDemo/MainWindow.xaml

36.2.3 命令绑定

必须添加命令绑定, 才能把它们连接到处理程序方法上。这里在 `Window` 元素中定义命令绑定, 这样这些绑定就可用于窗口中的所有元素。执行 `ApplicationCommands.Close` 命令时, 会调用 `OnClose()` 方法。执行 `BooksCommands.ShowBooks` 命令时, 会调用 `OnShowBooks()` 方法:



```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Close" Executed="OnClose" />
  <CommandBinding Command="local:BooksCommands.ShowBooks"
    Executed="OnShowBooks" />
</Window.CommandBindings>
```

代码段 BooksDemo/MainWindow.xaml

通过命令绑定, 还可以指定 `CanExecute` 属性, 在该属性中, 会调用一个方法来验证命令是否可用。例如, 如果文件没有变化, `ApplicationCommands.Save` 命令就是不可用的。

需要用两个参数定义处理程序, 它们分别是 `sender` 的 `object` 和可以从其中访问命令信息的 `ExecutedRoutedEventArgs` :



```
private void OnClose(object sender, ExecutedRoutedEventArgs e)
{
    Application.Current.Shutdown();
}
```

代码段 BooksDemo/MainWindow.xaml.cs



还可以通过命令传递参数。为此，可以通过命令源(如 MenuItem)指定 CommandParameter 属性。使用 ExecutedRoutedEventArgs 的 Parameter 属性可以访问该参数。

命令绑定也可以通过控件来定义。TextBox 控件给 ApplicationCommands.Cut、ApplicationCommands.Copy、ApplicationCommands.Paste 和 ApplicationCommands.Undo 定义了绑定。这样，就只需指定命令源，并使用 TextBox 控件中的已有功能。

36.3 TreeView

TreeView 控件可以显示层次数据。绑定到 TreeView 非常类似于前面的绑定到 ListBox，其区别是绑定 TreeView 会显示分层数据——可以使用 HierarchicalDataTemplate。

下面的示例使用分层显示方式和新的 DataGrid 控件。Formula1 样本数据库通过 ADO.NET Entity Framework 来访问。所使用的映射如图 36-13 所示。Race 类包含竞赛日期的信息，且关联到 Circuit 类上。Circuit 类包含 Country 和竞赛环形跑道的信息。Race 类还与 RaceResult 类关联起来。RaceResult 类包含 Racer 和 Team 的信息。

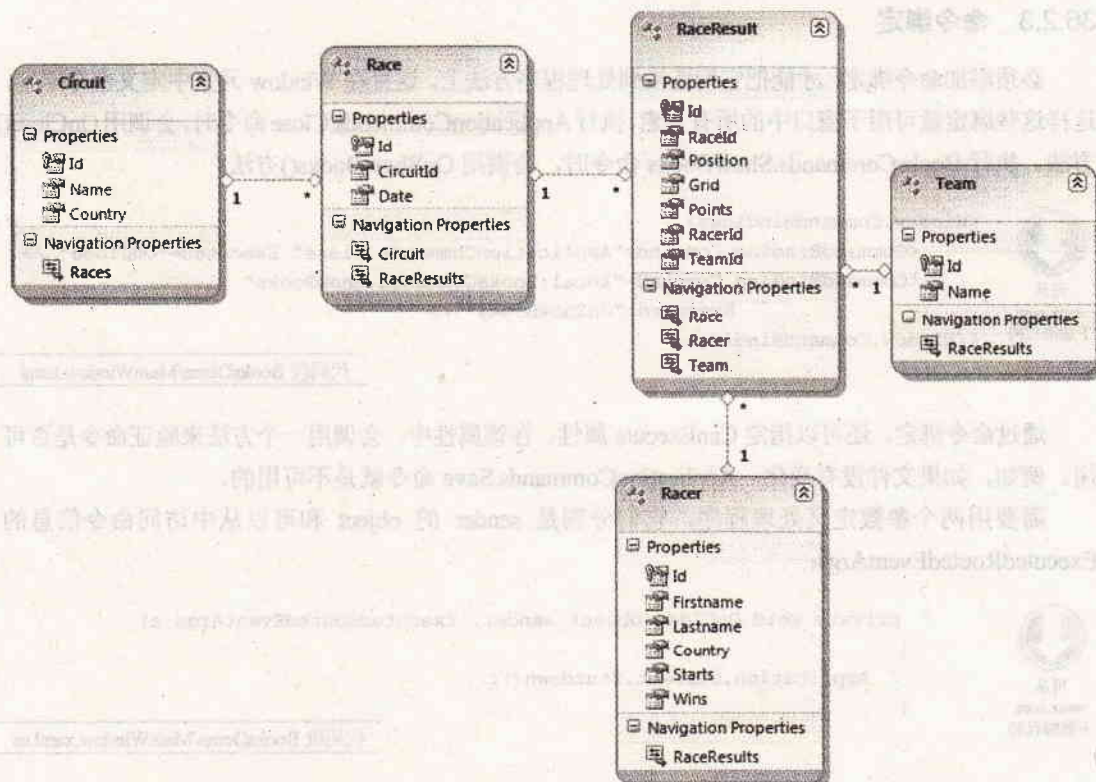


图 36-13



ADO.NET Entity Framework 参见第 31 章。

使用 XAML 代码声明一个 `TreeView`。`TreeView` 派生自基类 `ItemsControl`，其中，与列表的绑定可以通过 `ItemsSource` 属性来完成。把 `ItemsSource` 属性绑定到数据上下文上。数据上下文在代码隐藏中指定，如下所示。当然，这也可以通过 `ObjectDataProvider` 来实现。为了定义分层数据的自定义显示方式，定义了 `HierarchicalDataTemplate` 元素。这里的数据模板是用 `DataType` 属性为特定的数据类型定义的。第一个 `HierarchicalDataTemplate` 是 `Championship` 类的模板，它把这个类的 `Year` 属性绑定到 `TextBlock` 的 `Text` 属性上。`ItemsSource` 属性定义了该数据模板本身的绑定，以指定数据层次结构中的下一层。如果 `Championship` 类的 `Races` 属性返回一个集合，就直接把 `ItemsSource` 属性绑定到 `Races` 上。但是，因为这个属性返回一个 `Lazy<T>` 对象，所以绑定到 `Races.Value` 上。`Lazy<T>` 类的优点在本章后面讨论。

第二个 `HierarchicalDataTemplate` 元素定义 `F1Race` 类的模板，并绑定这个类的 `Country` 和 `Date` 属性。利用 `Date` 属性，通过绑定定义一个 `StringFormat`。把 `ItemsSource` 属性绑定到 `Races.Value` 上，来定义层次结构中的下一层。

因为 `F1RaceResult` 类没有子集合，所以层次结构到此为止。对于这个数据类型，定义一个正常的 `DataTemplate`，来绑定 `Position`、`Racer` 和 `Car` 属性：



可从
wrox.com
下载源代码

```
<UserControl x:Class="Formula1Demo.TreeUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Formula1Demo"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">

    <Grid>
        <TreeView ItemsSource="{Binding}">
            <TreeView.Resources>
                <HierarchicalDataTemplate DataType="{x:Type local:Championship}"
                    ItemsSource="{Binding Races.Value}">
                    <TextBlock Text="{Binding Year}" />
                </HierarchicalDataTemplate>

                <HierarchicalDataTemplate DataType="{x:Type local:F1Race}"
                    ItemsSource="{Binding Results.Value}">
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Country}" Margin="5,0,5,0" />
                        <TextBlock Text="{Binding Path=Date, StringFormat=d}"
                            Margin="5,0,5,0" />
                    </StackPanel>
                </HierarchicalDataTemplate>

                <DataTemplate DataType="{x:Type local:F1RaceResult}">
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Position}" Margin="5,0,5,0" />
                        <TextBlock Text="{Binding Racer}" Margin="5,0,0,0" />
                        <TextBlock Text=", " />
                    </StackPanel>
                </DataTemplate>
            </TreeView.Resources>
        </TreeView>
    </Grid>
```

```

        <TextBlock Text="{Binding Car}" />
    </StackPanel>
</DataTemplate>
</TreeView.Resources>
</TreeView>
</Grid>
</UserControl>

```

代码段 Formula1Demo/TreeUC.xaml

下面是填充分层控件的代码。在 XAML 代码的代码隐藏文件中，把 DataContext 赋予 Years 属性。Years 属性使用一个 LINQ 查询，而不是 ADO.NET Entity Framework 数据上下文，来获取数据库中所有一级方程式比赛的年份，并为每个年份新建一个 Championship 对象。通过 Championship 类的实例设置 Year 属性。这个类也有一个 Races 属性，可返回该年份的比赛信息，但这些信息还没有填充。



LINQ 参见第 11 章和第 31 章。



可从
wrox.com
下载源代码

```

using System.Collections.Generic;
using System.Linq;
using System.Windows.Controls;

namespace Formula1Demo
{
    public partial class TreeUC : UserControl
    {
        private Formula1Entities data = new Formula1Entities();

        public TreeUC()
        {
            InitializeComponent();
            this.DataContext = Years;
        }

        public IEnumerable<Championship>Years
        {
            get
            {
                FlDataContext.Data = data;
                return (from r in data.Races
                        orderby r.Date ascending
                        select
                            new Championship
                            {
                                Year = r.Date.Year,
                            }).Distinct();
            }
        }
    }
}

```

代码段 Formula1Demo/TreeUC.xaml.cs

Championship 类有一个用于返回年份的简单的自动属性。Races 属性的类型是 `Lazy<IEnumerable<F1Race>>`。Lazy<T>类是.NET 4 新增的,用于懒惰初始化。对于 TreeView 控件,这个类非常方便。如果表达式树中的数据非常多,且不希望提前加载整个表达式树,但仅在用户做出选择时加载,就可以使用懒惰加载方式。在 Lazy<T>类的构造函数中使用 `Func<IEnumerable<F1Race>>` 委托。在这个委托中,需要返回 `IEnumerable<F1Race>`。赋予该委托的 Lambda 表达式的实现方式使用一个 LINQ 查询,来创建一个 F1Race 对象列表,并指定它们的 Date 和 Country 属性:



可从
wrox.com
下载源代码

```
public class Championship
{
    public int Year { get; set; }
    public Lazy<IEnumerable<F1Race>> Races
    {
        get
        {
            return new Lazy<IEnumerable<F1Race>>(() =>
            {
                return from r in F1DataContext.Data.Races
                    where r.Date.Year == Year
                    orderby r.Date
                    select new F1Race
                    {
                        Date = r.Date,
                        Country = r.Circuit.Country
                    };
            });
        }
    }
}
```

代码段 Formula1Demo/Championship.cs

F1Race 类也定义了 Results 属性,该属性使用 Lazy<T>类型返回一个 F1RaceResult 对象列表:



可从
wrox.com
下载源代码

```
public class F1Race
{
    public string Country { get; set; }
    public DateTime Date { get; set; }
    public Lazy<IEnumerable<F1RaceResult>> Results
    {
        get
        {
            return new Lazy<IEnumerable<F1RaceResult>>(() =>
            {
                return from rr in F1DataContext.Data.RaceResults
                    where rr.Race.Date == this.Date
                    select new F1RaceResult
                    {
                        Position = rr.Position,
                        Racer = rr.Racer.Firstname + " " +
                            rr.Racer.Lastname,
                        Car = rr.Team.Name
                    };
            });
        }
    }
}
```

代码段 Formula1Demo/F1Race.cs

层次结构中的最后一个类是 F1RaceResult, 它是 Position、Racer 和 Car 的简单数据存储器:



可从
wrox.com
下载源代码

```
public class F1RaceResult
{
    public int Position { get; set; }
    public string Racer { get; set; }
    public string Car { get; set; }
}
```

代码段 Formula1Demo/Championship.cs

运行应用程序, 首先会在树形视图中看到所有年份的冠军。因为使用了绑定, 所以也访问了下一层——每个 Championship 对象已经关联到 F1Race 对象。用户不需要等待, 就可以看到年份下面的第一级, 也不需要默认显示的小三角形来打开某个年份的信息。图 36-14 打开了 1984 年的信息。只要用户单击某个年份, 就会看到第二级绑定, 第三级也绑定了, 并检索出比赛结果。

当然也可以定制 TreeView 控件, 并为整个模板或视图中的项定义不同的样式。

36.4 DataGrid

在 .NET 4 之前, WPF 中没有 DataGrid 控件, 现在有了! 通过 DataGrid 控件, 可以把信息显示在行和列中, 还可以编辑它们。

DataGrid 控件是一个 ItemsControl, 定义了绑定到集合上的 ItemsSource 属性。这个用户界面的 XAML 代码也定义了两个 RepeatButton 控件, 用于实现分页功能。这里不是一次加载所有比赛信息, 而是使用分页功能, 这样用户就可以翻看各个页面。在简单的场景中, 只需要指定 DataGrid 的 ItemsSource 属性。默认情况下, DataGrid 会根据绑定数据的属性来创建列:



可从
wrox.com
下载源代码

```
<UserControl x:Class="Formula1Demo.GridUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Grid.RowDefinitions>
            <RepeatButton Margin="5" Click="OnPrevious">Previous</RepeatButton>
            <RepeatButton Margin="5" Click="OnNext">Next</RepeatButton>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal" Grid.Row="0">
            <Button Click="OnPrevious">Previous</Button>
            <Button Click="OnNext">Next</Button>
        </StackPanel>
        <DataGrid Grid.Row="1" ItemsSource="{Binding}" />
    </Grid>
</UserControl>
```

代码段 Formula1Demo/GridUC.xaml

代码隐藏使用与前面 `TreeView` 示例相同的 `Formula1` 数据库。把 `UserControl` 的 `DataContext` 设置为 `Races` 属性。这个属性返回 `IEnumerable<object>`。这里不指定强类型化的枚举, 而使用一个 `object`, 以通过 LINQ 查询创建一个匿名类。该 LINQ 查询使用 `Year`、`Country`、`Position`、`Racer` 和 `Car` 属性创建匿名类, 并使用复合语句访问 `Races` 和 `RaceResults` 属性。它还访问 `Races` 的其他关联属性, 以获取国籍、赛手和团队信息。使用 `Skip()` 和 `Take()` 方法实现分页功能。页面的大小固定为 50 项, 当前页面使用 `OnNext()` 和 `OnPrevious()` 处理程序来改变:



可从
wrox.com
下载源代码

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

namespace Formula1Demo
{
    public partial class GridUC : UserControl
    {
        private int currentPage = 0;
        private int pageSize = 50;
        private Formula1Entities data = new Formula1Entities();
        public GridUC()
        {
            InitializeComponent();
            this.DataContext = Races;
        }

        public IEnumerable<object> Races
        {
            get
            {
                return (from r in data.Races
                        from rr in r.RaceResults
                        orderby r.Date ascending
                        select new
                        {
                            Year = r.Date.Year,
                            Country = r.Circuit.Country,
                            Position = rr.Position,
                            Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                            Car = rr.Team.Name
                        }).Skip(currentPage * pageSize).Take(pageSize);
            }
        }

        private void OnPrevious(object sender, RoutedEventArgs e)
        {
            if (currentPage > 0)
            {
                currentPage--;
                this.DataContext = Races;
            }
        }

        private void OnNext(object sender, RoutedEventArgs e)
        {

```

```
currentPage++;
this.DataContext = Races;
```

代码段 Formula1Demo/GridUC.xaml.cs

图 36-15 显示了正在运行的应用程序，其中使用了默认的网格样式和标题。

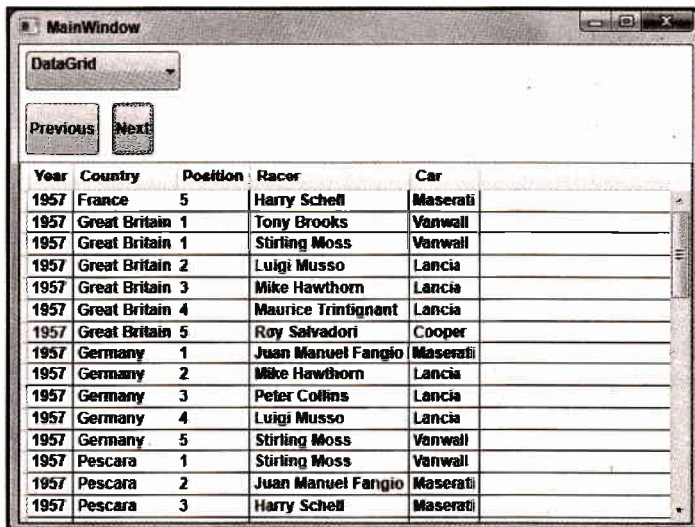


图 36-15

在下一个 DataGrid 示例中，用自定义列和组合来定制网格。

36.4.1 自定义列

把 DataGrid 的 `AutoGenerateColumns` 属性设置为 `False`，就不会生成默认的列。使用 `Columns` 属性可以创建自定义列。还可以指定派生自 `DataGridColumn` 的元素，也可以使用预定义的类型。`DataGridTextColumn` 可以用于读取和编辑文本。`DataGridHyperlinkColumn` 可显示超链接。`DataGridCheckBoxColumn` 可给布尔数据显示复选框。如果某列有一个项列表，就可以使用 `DataGridComboBoxColumn`。将来会有更多的 `DataGridColumn` 类型，但如果现在就需要其他表示方式，可以使用 `DataGridTemplateColumn` 定义并绑定任意需要的元素。

下面的示例代码使用 `DataGridTextColumn` 来绑定到 `Position` 和 `Racer` 属性。把 `Header` 属性设置为要显示的字符串，当然也可以使用模板给列定义完全自定义的标题：



可从
wrox.com
下载源代码

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Position, Mode=OneWay}"
      Header="Position" />
    <DataGridTextColumn Binding="{Binding Racer, Mode=OneWay}"
      Header="Racer" />
  </DataGrid.Columns>
```

代码段 Formula1Demo/GridUC.xaml.cs

36.4.2 行的细节

选择一行时, `DataGrid` 可以显示该行的其他信息。为此, 需要指定 `DataGrid` 的 `RowDetailsTemplate`。把一个 `DataTemplate` 赋予这个 `RowDetailsTemplate`, 其中包含几个显示汽车和赛点的 `TextBlock` 元素:



```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <StackPanel Orientation="Horizontal">
      <TextBlock Text="Car:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Car}" Margin="5,0,0,0" />
      <TextBlock Text="Points:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Points}" />
    </StackPanel>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

代码段 Formula1Demo/GridUC.xaml.cs

36.4.3 用 `DataGrid` 进行分组

一级方程式比赛有几行包含相同的信息, 如年份和国籍。对于这类数据, 可以使用分组功能, 给用户组织信息。

对于分组功能, 可以在 XAML 代码中使用 `CollectionViewSource`, 来支持分组、排序和筛选功能。在代码隐藏中, 也可以使用 `ListCollectionView` 类, 它仅由 `CollectionViewSource` 使用。

`CollectionViewSource` 在 `Resources` 集合中定义。`CollectionViewSource` 的源是 `ObjectDataProvider` 的结果。`ObjectDataProvider` 调用 `F1Races` 类型的 `GetRaces()` 方法。这个方法有两个 `int` 参数, 它们从 `MethodParameters` 集合中指定。`CollectionViewSource` 给分组使用了两个描述, 分别用于 `Year` 属性和 `Country` 属性:



```
<Grid.Resources>
  <ObjectDataProvider x:Key="races" ObjectType="{x:Type local:F1Races}"
    MethodName="GetRaces">
    <ObjectDataProvider.MethodParameters>
      <sys:Int32>0</sys:Int32>
      <sys:Int32>20</sys:Int32>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  <CollectionViewSource x:Key="viewSource"
    Source="{StaticResource races}">
    <CollectionViewSource.GroupDescriptions>
      <PropertyGroupDescription PropertyName="Year" />
      <PropertyGroupDescription PropertyName="Country" />
    </CollectionViewSource.GroupDescriptions>
  </CollectionViewSource>
</Grid.Resources>
```

代码段 Formula1Demo/GridGroupingUC.xaml

这里显示的组使用 `DataGrid` 的 `GroupStyle` 属性定义。对于 `GroupStyle` 元素, 需要自定义 `ContainsStyle`、`HeaderTemplate` 和整个面板。为了动态选择 `GroupStyle` 和 `HeaderStyle`, 还可以编写一个容器样式选择器和一个标题模板选择器。它们的功能非常类似于前面的数据模板选择器。

示例中的 `GroupStyle` 设置了 `GroupStyle` 的 `ContainsStyle` 属性。在这个样式中, 用模板定制

GroupItem。使用分组功能时，GroupItem 显示为组的根元素。在组中使用 Names 属性显示名字，使用 ItemCount 属性显示项数。Grid 的第 3 列使用 ItemPresenter 包含所有正常的项。如果行按国籍分组，Name 属性的标签就会有不同的宽度，这看起来不太好。因此，使用 Grid 的第二列设置 SharedSizeGroup 属性，使所有的项有相同的大小。还需要设置共享的尺寸范围，使所有的元素有相同的大小，为此在 DataGrid 中设置 Grid.IsSharedSizeScope="True"。

```
<DataGrid.GroupStyle>
    <GroupStyle>
        <GroupStyle.ContainerStyle>
            <Style TargetType="{x:Type GroupItem}">
                <Setter Property="Template">
                    <Setter.Value>
                        <ControlTemplate>
                            <StackPanel Orientation="Horizontal">
                                <Grid>
                                    <Grid.ColumnDefinitions>
                                        <ColumnDefinition
                                            SharedSizeGroup="LeftColumn"
                                        />
                                        <ColumnDefinition />
                                        <ColumnDefinition />
                                    </Grid.ColumnDefinitions>
                                    <Label Grid.Column="0"
                                        Background="Yellow"
                                        Content="{Binding Name}" />
                                    <Label Grid.Column="1"
                                        Content="{Binding ItemCount}"
                                    />
                                    <Grid Grid.Column="2"
                                        HorizontalAlignment="Center"
                                        VerticalAlignment="Center">
                                        <ItemsPresenter/>
                                    </Grid>
                                </Grid>
                            </StackPanel>
                        </ControlTemplate>
                    </Setter.Value>
                </Setter>
            </Style>
        </GroupStyle.ContainerStyle>
    </GroupStyle>
</DataGrid.GroupStyle>
```

ObjectDataProvider 使用了类 F1Races，F1Races 使用 LINQ 访问 Formula1 数据库，并返回一个匿名类型列表，以及 Year、Country、Position、Racer、Car 和 Points 属性。这里再次使用 Skip() 和 Take() 方法访问部分数据：



可从
wrox.com
下载源代码

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

namespace Formula1Demo
{
    public class F1Races
```

```

    private int lastpageSearched = -1;
    private IEnumerable<object> cache = null;
    private Formula1Entities data = new Formula1Entities();

    public IEnumerable<object> GetRaces(int page, int pageSize)
    {
        if (lastpageSearched == page)
            return cache;
        lastpageSearched = page;

        var q = (from r in data.Races
                 from rr in r.RaceResults
                 orderby r.Date ascending
                 select new
                 {
                     Year = r.Date.Year,
                     Country = r.Circuit.Country,
                     Position = rr.Position,
                     Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                     Car = rr.Team.Name,
                     Points = rr.Points
                 }).Skip(page * pageSize).Take(pageSize);
        cache = q;
        return cache;
    }
}

```

代码段 Formula1Demo/F1Races.cs

现在只需为用户设置页码, 修改 `ObjectDataProvider` 的参数。在用户界面中, 定义一个文本框和一个按钮:



可从
wrox.com
下载源代码

```

<StackPanel Orientation="Horizontal" Grid.Row="0">
    <TextBlock Margin="5" Padding="4"
               VerticalAlignment="Center">Page:</TextBlock>
    <TextBox Margin="5" Padding="4" VerticalAlignment="Center"
             x:Name="textPageNumber" Text="0" />
    <Button Click="OnGetPage">Get Page</Button>
</StackPanel>

```

代码段 Formula1Demo/GridGroupingUC.xaml

在代码隐藏中, 按钮的 `OnGetPage()` 处理程序访问 `ObjectDataProvider`, 并修改方法的第一个参数。接着调用 `Refresh()` 方法, 以便 `ObjectDataProvider` 请求新页面:



可从
wrox.com
下载源代码

```

private void OnGetPage(object sender, RoutedEventArgs e)
{
    int page = int.Parse(textPageNumber.Text);
    var odp = (sender as FrameworkElement).FindResource("races")
              as ObjectDataProvider;
    odp.MethodParameters[0] = page;
    odp.Refresh();
}

```

代码段 Formula1Demo/GridGroupingUC.xaml.cs

运行应用程序，就会看到分组和行的细节信息，如图 36-16 所示。

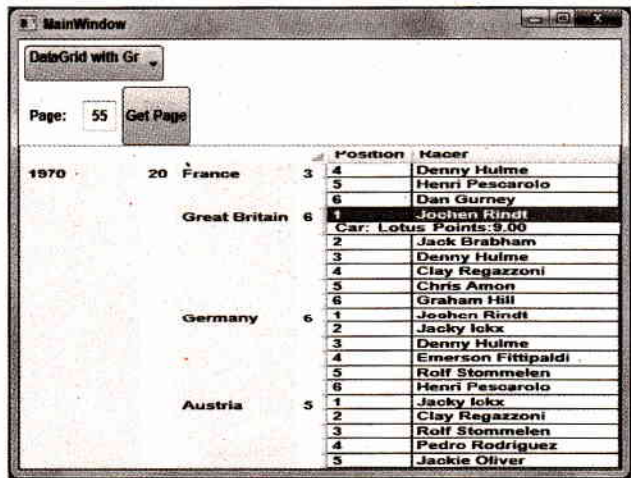


图 36-16

36.5 小结

本章介绍了 WPF 中对于业务应用程序非常重要的一些功能。

WPF 的数据绑定功能比 Windows 窗体前进了一大步。可以把.NET 类的任意属性绑定到 WPF 元素的属性上。绑定模式定义了绑定的方向。可以绑定.NET 对象和列表，定义数据模板，从而通过数据模板为.NET 类创建默认的外观。

命令绑定可以把处理程序的代码映射到菜单和工具栏上。还可以用 WPF 进行复制和粘贴，因为这个技术的命令处理程序已经包含在 TextBox 控件中。

下一章讨论 WPF 的另一个方面：用 WPF 创建文档。

第 37 章

用 WPF 创建文档

本章内容:

- 创建流文档
- 创建固定的文档
- 创建 XPS 文档
- 打印文档

创建文档是 WPF 的一个主要部分。`System.Windows.Documents` 名称空间支持创建流文档和固定文档。这个名称空间包含的元素可以利用类似于 Word 的方式创建流文档,也可以创建 WYSIWYG(所见即所得)固定文档。

流文档面向屏幕读取;文档的内容根据窗口的大小来排列,如果窗口重置了大小,文档的流就会改变。固定文档主要用于打印和面向页面的内容,其内容总是按照相同的方式排列。

本章讨论如何创建、打印流文档和固定文档,并涵盖 `System.Windows.Documents`、`System.Windows.Xps` 和 `System.IO.Packaging` 名称空间。

37.1 文本元素

要构建文档的内容,需要文档元素。这些元素的基类是 `TextElement`。这个类定义了字体设置、前景和背景,以及文本效果的常见属性。`TextElement` 是 `Block` 类和 `Inline` 类的基类,这两个类的功能在后面的几节中介绍。

37.1.1 字体

文本的一个重要方面是文本的外观,即字体。通过 `TextElement`,可以用 `FontWeight`、`FontStyle`、`FontStretch`、`FontSize` 和 `FontFamily` 属性指定字体。

- 预定义的 `FontWeight` 值由 `FontWeights` 类定义,这个类提供的值包括 `UltraLight`、`Light`、`Medium`、`Normal`、`Bold`、`UltraBold` 和 `Heavy`。
- `FontStyle` 的值由 `FontStyles` 类定义,可以是 `Normal`、`Italic` 和 `Oblique`。
- 利用 `FontStretch` 可以指定字体相对于正常宽高比的拉伸程度。`FontStretch` 指定了预定义的拉伸率从 50%(`UltraCondensed`)到 200%(`UltraExpanded`)。在这个范围之间的预定义值是 `ExtraCondensed`(62.5%)、`Condensed`(75%)、`SemiCondensed`(87.5%)、`Normal`(100%)、`SemiExpanded`(112.5%)、`Expanded`(125%)、`ExtraExpanded`(150%)。

- **FontSize** 是 **double** 类型，可以用于指定字体的大小，其单位与设备无关，如英寸、厘米和点。
- 利用 **FontFamily** 可以定义首选字体系列的名称，如 **Arial** 或 **Times New Roman**。使用这个属性可以指定一个字体系列名列表，这样，如果某个字体不可用，就使用列表中的下一个字体。(如果所选字体和备用字体都不可用，流文档就使用默认的 **MessageFontFamily**)。还可以从资源中引用字体系列，或者使用 **URI** 引用服务器上的字体。对于固定的文档，不会出现字体不可用的情况，因为字体是通过文档提供的。

为了了解不同字体的外观，下面的示例 WPF 应用程序包含一个列表框。该列表框为列表中的每一项定义了一个 **ItemTemplate**。这个模板使用 4 个 **TextBlock** 元素，这些元素的 **FontFamily** 绑定到 **FontFamily** 对象的 **Source** 属性上。给不同的 **TextBlock** 元素设置 **FontWeight** 和 **FontStyle**：



可从
wrox.com
下载源代码

```
<ListBox ItemsSource="{Binding}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Margin="3, 0, 3, 0"
          FontFamily="{Binding Path=Source}"
          FontSize="18" Text="{Binding Path=Source}" />
        <TextBlock Margin="3, 0, 3, 0"
          FontFamily="{Binding Path=Source}"
          FontSize="18" FontStyle="Italic" Text="Italic" />
        <TextBlock Margin="3, 0, 3, 0"
          FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraBold"
          Text="UltraBold" />
        <TextBlock Margin="3, 0, 3, 0"
          FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraLight"
          Text="UltraLight" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

代码段 ShowFonts/ShowFontsWindow.xaml

在代码隐藏中，数据上下文设置为 **System.Windows.Media.Font** 类的 **SystemFontFamilies** 属性值：



可从
wrox.com
下载源代码

```
public partial class ShowFontsWindow : Window
{
    public ShowFontsWindow()
    {
        InitializeComponent();

        this.DataContext = Fonts.SystemFontFamilies;
    }
}
```

代码段 ShowFonts/ShowFontsWindow.xaml.cs

运行应用程序，会显示一个很长的列表，其中包含系统字体系列的斜体、黑体、**UltraBold** 和 **UltraLight** 样式，如图 37-1 所示。



图 37-1

37.1.2 TextEffect

下面看看 `TextEffect`, 因为它也是所有文档元素共有的。`TextEffect` 在名称空间 `System.Windows.Media` 中定义, 派生自基类 `Animatable`, 允许生成文本的动画效果。

`TextEffect` 可以为裁剪区域、前景画笔和变换创建动画效果。利用 `PositionStart` 和 `PositionCount` 属性可以指定在文本中应用动画的位置。

要应用文本效果, 应设置 `Run` 元素的 `TextEffects` 属性。该属性内部指定的 `TextEffect` 元素定义了前景和变换效果。对于前景, 使用名为 `brush1` 的 `SolidColorBrush` 画笔, 通过 `ColorAnimation` 元素生成动画效果。转换使用名为 `scale1` 的 `ScaleTransformation`, 从两个 `DoubleAnimation` 元素中制作动画效果。



```
<TextBlock>
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation AutoReverse="True" RepeatBehavior="Forever"
            From="Red" To="Yellow" Duration="0:0:3"
            Storyboard.TargetName="brush1"
            Storyboard.TargetProperty="Color" />
          <DoubleAnimation AutoReverse="True"
            RepeatBehavior="Forever"
            From="0.2" To="12" Duration="0:0:6"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="ScaleX" />
          <DoubleAnimation AutoReverse="True"
            RepeatBehavior="Forever"
            From="0.2" To="12" Duration="0:0:6"
            Storyboard.TargetName="scale1"
            Storyboard.TargetProperty="ScaleY" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </TextBlock.Triggers>
  <Run FontFamily="Mangal">
    cn|elements
  </Run>
  <TextEffects>
    <TextEffect PositionStart="0" PositionCount="30">
      <TextEffect.Foreground>
        <SolidColorBrush x:Name="brush1" Color="Blue" />
      </TextEffect.Foreground>
    </TextEffect>
  </TextEffects>
</TextBlock>
```

```

        <TextEffect.Transform>
            <ScaleTransform x:Name="scale1" ScaleX="3" ScaleY="3" />
        </TextEffect.Transform>
    </TextEffect>
</Run.TextEffects>
</Run>
</TextBlock>

```

代码段 TextEffectsDemo/MainWindow.xaml

运行应用程序，会看到大小和颜色的变化，如图 37-2 和图 37-3 所示。



图 37-2

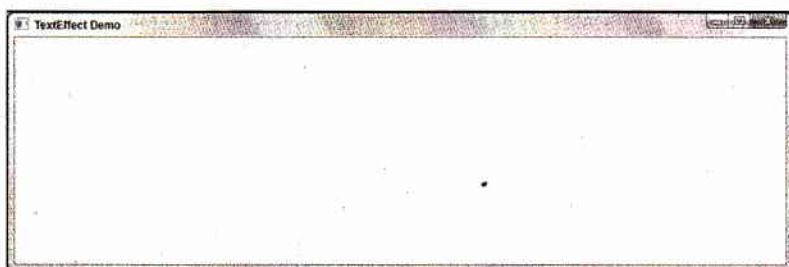


图 37-3



在纯 XAML 示例中，也可以使用 XAMLPad.exe 输入代码，而不是创建可执行程序。这个工具是 Windows SDK 的一部分。

37.1.3 内联

所有内联流内容元素的基类都是 **Inline**。可以在流文档的段落中使用 **Inline** 元素。因为在段落中，**Inline** 元素可以一个跟着一个，所以 **Inline** 类提供了 **PreviousInline** 和 **NextInline** 属性，从一个元素导航到另一个元素。也可以使用 **SiblingNextInlines** 获取所有同级内联元素的集合。

前面用于输出一些文本的 **Run** 元素是一个 **Inline** 元素，它可输出格式化或非格式化的文本，还有许多其他的 **Inline** 元素。**Run** 元素后的换行可以用 **LineBreak** 元素来获得。

Span 元素派生自 **Inline** 类，它允许组合 **Inline** 元素。在 **Span** 的内容中只能包含 **Inline** 元素。含义明确的 **Bold**、**Hyperlink**、**Italic** 和 **Underline** 类都派生自 **Span**，因此允许 **Inline** 元素和其中的内容具有相同的功能，但对这些元素的操作不同。下面的 XAML 代码说明了 **Bold**、**Italic**、**Underline** 和 **LineBreak** 的用法，如图 37-4 所示。



可从
wrox.com
下载源代码

```
<Paragraph FontWeight="Normal">
  <Span>
    <Span>Normal</Span>
    <Bold>Bold</Bold>
    <Italic>Italic</Italic>
    <LineBreak />
    <Underline>Underline</Underline>
  </Span>
</Paragraph>
```

代码段 FlowDocumentsDemo/FlowDocument1.xaml

Normal **Bold** *Italic*
Underline

图 37-4

AnchoredBlock 是一个派生自 **Inline** 的抽象类，用于把 **Block** 元素锚定到流内容上。**Figure** 和 **Floater** 是派生自 **AnchoredBlock** 的具体类。因为这两个内联元素在涉及块时比较有趣，所以本章后面讨论它们。

另一个映射到 UI 元素上的 **Inline** 元素是 **InlineUIContainer**，它在前面的章节使用过。**InlineUIContainer** 允许给文档添加所有的 **UIElement** 对象(如按钮)。下面的代码段给文档添加了一个 **InlineUIContainer**，其中包含组合框、单选按钮和文本框元素，结果如图 37-5 所示。



图 37-5



当然，也可以给 UI 元素添加样式，参见第 35 章。



可从
wrox.com
下载源代码

```
<Paragraph TextAlignment="Center">
  <Span FontSize="36">
    <Italic> cn|elements </Italic>
  </Span>
  <LineBreak />
  <LineBreak />
  <InlineUIContainer>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
    </Grid>
  </InlineUIContainer>
```

```

</Grid.ColumnDefinitions>
<ComboBox Width="140" Margin="3" Grid.Row="0">
  <ComboBoxItem>Filet Mignon</ComboBoxItem>
  <ComboBoxItem>Rib Eye</ComboBoxItem>
  <ComboBoxItem>Sirloin</ComboBoxItem>
</ComboBox>

<StackPanel Grid.Row="0" Grid.RowSpan="2" Grid.Column="1">
  <RadioButton>Raw</RadioButton>
  <RadioButton>Medium</RadioButton>
  <RadioButton>Well done</RadioButton>
</StackPanel>
<TextBox Grid.Row="1" Grid.Column="0" Width="140"></TextBox>
</Grid>
</InlineUIContainer>
</Paragraph>

```

代码段 FlowDocumentsDemo/FlowDocument2.xaml

37.1.4 块

Block 是块级元素的抽象基类。块可以把包含其中的元素组合到特定的视图上。所有块都通用的属性有 PreviousBlock、NextBlock 和 SiblingBlocks，它们允许从一个块导航到下一个块。在块开始之前设置 BreakPageBefore 换页符和 BreakColumnBefore 换行符。块还使用 BorderBrush 和 BorderThickness 属性定义边框。

派生自 Block 的类有 Paragraph、Section、List、Table 和 BlockUIContainer。BlockUIContainer 类似于 InlineUIContainer，其中也可以添加派生自 UIElement 的元素。

Paragraph 和 Section 是简单的块，其中 Paragraph 包含内联元素；Section 用于组合其他 Block 元素。使用 Paragraph 块可以确定在段落内部或段落之间是否允许添加换页符或换行符。KeepTogether 可用于禁止在段落内部换行，KeepWithNext 尝试把一个段落与下一个段落合并起来。如果段落用换页符或换行符隔开，那么 MinWindowLines 会定义分隔符之后的最小行数，MinOrphanLines 定义分隔符之前的最小行数。

Paragraph 块也允许在段落内部用 TextDecoration 元素装饰文本。预定义的文本装饰由 TextDecoration 定义：Baseline、Overline、Strikethrough 和 Underline。

下面的 XAML 代码显示了多个 Paragraph 元素。一个 Paragraph 元素包含标题，其后的另一个 Paragraph 元素包含属于上述标题的内容。这两个段落通过特性 KeepWithNext 连接起来。把 KeepTogether 设置为 True，也确保包含内容的段落不被隔开。结果如图 37-6 所示。



可从
wrox.com
下载源代码

```

<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  ColumnWidth="300" FontSize="16" FontFamily="Georgia">
  <Paragraph FontSize="36">
    <Run>Lyrics</Run>
  </Paragraph>

  <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
    <Bold>
      <Run>Mary had a little lamb</Run>
    </Bold>
  </Paragraph>

  <Paragraph KeepTogether="True">

```

```

<Run>
    Mary had a little lamb,
</Run>
<LineBreak />
<Run>
    little lamb, little lamb,
</Run>
<LineBreak />
<Run>
    Mary had a little lamb,
</Run>
<LineBreak />
<Run>
    whose fleece was white as snow.
</Run>
<LineBreak />
<Run>
    And everywhere that Mary went,
</Run>
<LineBreak />
<Run>
    Mary went, Mary went,
</Run>
<LineBreak />
<Run>
    and everywhere that Mary went,
</Run>
<LineBreak />
<Run>
    the lamb was sure to go.
</Run>
</Paragraph>
<Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
    <Bold>
        <Run>Humpty Dumpty</Run>
    </Bold>
</Paragraph>
<Paragraph KeepTogether="True">
    <Run>Humpty dumpty sat on a wall</Run>
    <LineBreak />
    <Run>
        Humpty dumpty had a great fall</Run>
    <LineBreak />
    <Run>
        All the King's horses</Run>
    <LineBreak />
    <Run>
        And all the King's men</Run>
    <LineBreak />
    <Run>
        Couldn't put Humpty together again
    </Run>
</Paragraph>
</FlowDocument>

```

代码段 FlowDocumentsDemo/ParagraphDemo.xaml

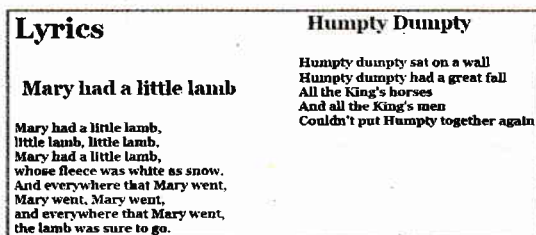


图 37-6

37.1.5 列表

List 类用于创建无序或有序的文本列表。List 通过设置 MarkerStyle 属性，定义了其列表项的项目符号样式。MarkerStyle 的类型是 TextMarkerStyle，它可以是数字(Decimal)、字母(LowerLatin 和 UpperLatin)、罗马数字(LowerRoman 和 UpperRoman)或图片(Disc、Circle、Square、Box)。List 只能包含 ListItem 元素，ListItem 只能包含 Block 元素。

用 XAML 定义如下列表，结果如图 37-7 所示。



图 37-7



可从
wrox.com
下载源代码

```
<List MarkerStyle="Square">
  <ListItem>
    <Paragraph>Monday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Tuesday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Wednesday</Paragraph>
  </ListItem>
</List>
```

代码段 FlowDocumentsDemo/ListDemo.xaml

37.1.6 表

Table 类非常类似于第 35 章讨论的 Grid 类，它也定义行和列。下面的例子说明了如何使用 Table 创建 FlowDocument。要创建表，可以给 Columns 属性添加 TableColumn 对象。而利用 TableColumn 可以指定宽度和背景。

Table 还包含 TableRowGroup 对象。TableRowGroup 有一个 Rows 属性，可以在 Rows 属性中添加 TableRow 对象。TableRow 类定义了一个 Cells 属性，在 Cells 属性中可以添加 TableCell 对象。TableCell 对象可以包含任意 Block 元素。这里使用了一个 Paragraph 元素，其中包含 Inline 元素 Run：



可从
wrox.com
下载源代码

```
var doc = new FlowDocument();
var t1 = new Table();
t1.Columns.Add(new TableColumn
{ Width = new GridLength(50, GridUnitType.Pixel) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });
```

```

var titleRow = new TableRow();
titleRow.Background = new SolidColorBrush(Colors.LightBlue);
var titleCell = new TableCell
    { ColumnSpan = 3, TextAlignment = TextAlignment.Center };
titleCell.Blocks.Add(
    new Paragraph(new Run("Formula 1 Championship 2009")
        { FontSize=24, FontWeight = FontWeights.Bold }));
titleRow.Cells.Add(titleCell);

var headerRow = new TableRow
    { Background = new SolidColorBrush(Colors.LightGoldenrodYellow) };
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Pos"))
    { FontSize = 14, FontWeight=FontWeights.Bold}));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Name"))
    { FontSize = 14, FontWeight = FontWeights.Bold }));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Points"))
    { FontSize = 14, FontWeight = FontWeights.Bold }));

var row1 = new TableRow();
row1.Cells.Add(new TableCell(new Paragraph(new Run("1."))));
row1.Cells.Add(new TableCell(new Paragraph(
    new Run("Jenson Button"))));
row1.Cells.Add(new TableCell(new Paragraph(new Run("95"))));

var row2 = new TableRow { Background =
    new SolidColorBrush(Colors.LightGray));
row2.Cells.Add(new TableCell(new Paragraph(new Run("2."))));
row2.Cells.Add(new TableCell(new Paragraph(new Run("Sebastian Vettel"))));
row2.Cells.Add(new TableCell(new Paragraph(new Run("84"))));

var row3 = new TableRow();
row3.Cells.Add(new TableCell(new Paragraph(new Run("3."))));
row3.Cells.Add(new TableCell(new Paragraph(
    new Run("Rubens Barrichello"))));
row3.Cells.Add(new TableCell(new Paragraph(new Run("77"))));

var rowGroup = new TableRowGroup();
rowGroup.Rows.Add(titleRow);
rowGroup.Rows.Add(headerRow);
rowGroup.Rows.Add(row1);
rowGroup.Rows.Add(row2);
rowGroup.Rows.Add(row3);
t1.RowGroups.Add(rowGroup);

doc.Blocks.Add(t1);

reader.Document = doc;

```

代码段 TableDemo/MainWindow.xaml

运行应用程序，会显示一个格式化好的表，如图 37-8 所示。

Formula 1 Championship 2009		
Pos	Name	Points
1.	Jenson Button	95
2.	Sebastian Vettel	84
3.	Rubens Barrichello	77

图 37-8

37.1.7 块的锚定

既然学习了 Inline 和 Block 元素, 就可以使用 AnchoredBlock 类型的 Inline 元素合并它们。AnchoredBlock 是一个抽象基类, 它有两个具体的实现方式 Figure 和 Floater。

Floater 使用属性 HorizontalAlignment 和 Width 同时显示其内容和主要内容。

从上面的例子开始, 添加一个包含 Floater 的新段落。这个 Floater 采用左对齐方式, 宽度为 160。如图 37-9 所示, 下一个段落将环绕它。

```
<Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
  <Bold>
    <Run>Mary had a little lamb</Run>
  </Bold>
</Paragraph>
<Paragraph>
  <Floater HorizontalAlignment="Left" Width="160">
    <Paragraph Background="LightGray">
      <Run>Sarah Josepha Hale</Run>
    </Paragraph>
  </Floater>
</Paragraph>
<Paragraph KeepTogether="True">
  <Run>
    Mary had a little lamb
  </Run>
  <LineBreak />
  <!-- ... -->
</Paragraph>
```

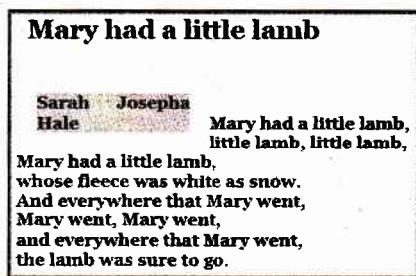


图 37-9

Figure 采用水平和垂直对齐方式, 可以锚定到页面、内容、列或段落上。下面代码中的 Figure 锚定到页面中心处, 但水平和垂直方向有偏移。设置 WrapDirection, 使左列和右列环绕着图片, 图 37-10 显示了环绕的结果。

```
<Paragraph>
  <Figure HorizontalAnchor="PageCenter" HorizontalOffset="20"
    VerticalAnchor="PageCenter" VerticalOffset="20" WrapDirection="Both">
    <Paragraph Background="LightGray" FontSize="24">
      <Run> Lyrics Samples </Run>
    </Paragraph>
  </Figure>
</Paragraph>
```

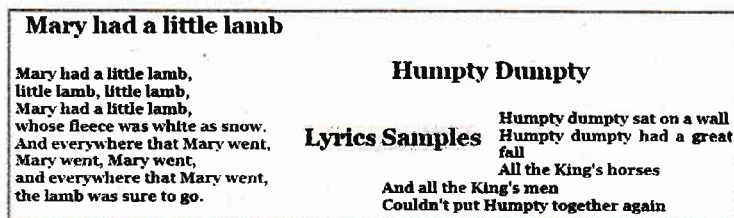


图 37-10

Figure 和 Floater 都用于添加不在主流中的内容, 尽管这两个功能看起来类似, 但它们的特征大不相同。表 37-1 列出了 Figure 和 Floater 的区别。

表 37-1

特 征	Floater	Figure
位置	Floater 不能定位, 在空间可用时显示它	Figure 可以用水平和垂直锚点来定位, 它可以停靠在页面、内容、列或段落上
宽度	Floater 只能放在一列中。如果它设置的宽度大于列宽, 就忽略它	Figure 可以跨越多列。Figure 的宽度可以设置为半页或两列
分页	如果 Floater 高于列高, 就分解 Floater, 分页到下一列或下一页上	如果 Figure 高于列高, 就只显示列中的部分, 其他内容会丢失

37.2 流文档

前面介绍了所有 Inline 和 Block 元素, 现在我们知道应把什么内容放在流文档中。FlowDocument 类可以包含 Block 元素, Block 元素可以包含 Block 或 Inline 元素, 这取决于 Block 的类型。

FlowDocument 类的一个主要功能是把流分解为多个页面。这是通过 FlowDocument 实现的 IDocumentPaginatorSource 接口实现。

FlowDocument 的其他选项包括建立默认字体、前景画笔和背景画笔, 以及配置页面和列的大小。下面 FlowDocument 的 XAML 代码定义了默认字体、字体大小、列宽和列之间的标尺:

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ColumnWidth="300" FontSize="16" FontFamily="Georgia"
    ColumnRuleWidth="3" ColumnRuleBrush="Violet">
```

现在需要一种方式来查看文档。以下列表描述了几个查看器:

- RichTextBox——一个简单的查看器, 还允许编辑(只要 IsReadOnly 属性没有设置为 true)。RichTextBox 不在多列中显示文档, 而以滚动模式显示文档。这类似于 Microsoft Word 中的 Web 布局。把 HorizontalScrollbarVisibility 设置为 ScrollbarVisibility.Auto, 就可以启用滚动条。
- FlowDocumentScrollViewer——一个读取器, 只能读取文档, 不能编辑文档, 这个读取器允许放大文档, 工具栏中的滑块可以使用 IsToolBarEnabled 属性, 来启用其缩放功能。设置 CanIncreaseZoom、CanDecreaseZoom、MinZoom 和 MaxZoom 都允许设置缩放功能。
- FlowDocumentPageViewer——给文档分页的查看器。使用这个查看器, 不仅可以通过其工具栏放大文档, 还可以在页面之间切换。

- **FlowDocumentReader**——这个查看器合并了 **FlowDocumentScrollViewer** 和 **FlowDocumentPageViewer** 的功能，它支持不同的查看模式，这些模式可以在工具栏中设置，或者使用 **FlowDocumentReaderViewingMode** 类型的 **ViewingMode** 属性来设置。这个枚举的值可以是 **Page**、**TwoPage** 和 **Scroll**。也可以根据需要禁用查看模式。

图 37-11 在 **TwoPage** 中用 **FlowDocumentReader** 模式显示了前面创建的流文档。

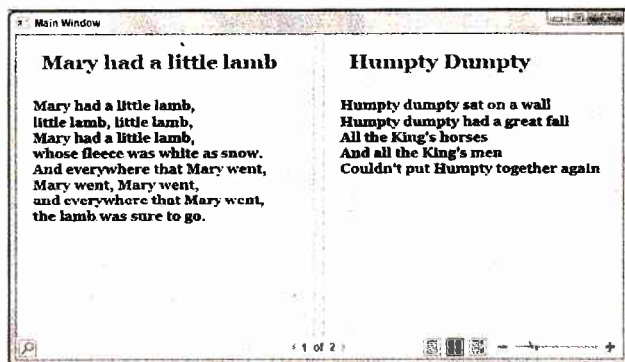


图 37-11

37.3 固定文档

无论固定文档在哪里复制或使用，它总是定义相同的外观、相同的分页方式，并使用相同的字体。WPF 定义了用于创建固定文档的 **FixedDocument** 类，和用于查看固定文档的 **DocumentViewer** 类。

本章使用一个示例应用程序，通过编程方式创建一个固定文档，该程序要求用户输入一个用于创建固定文档的菜单规划。图 37-12 显示了这个应用程序的主用户界面，用户可以在其中用 **DatePicker** 类选择某一天，在 **DataGrid** 中输入一周的菜单，再单击 **Create Doc** 按钮，新建一个 **FixedDocument**。这个应用程序使用 **Page** 对象在 **NavigationWindow** 中导航。单击 **Create Doc** 按钮会导航到一个包含固定文档的新页面上。

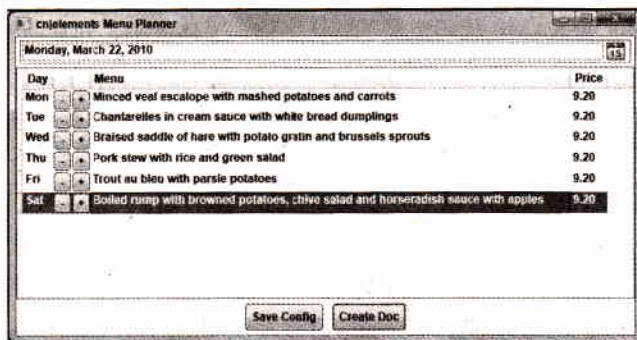


图 37-12

Create Doc 按钮的事件处理程序 **OnCreateDoc()** 导航到一个新页面上。为此，处理程序实例化新页面 **DocumentPage**。这个页面包含一个 **NavigationService_LoadCompleted()** 处理程序，把它赋予 **NavigationService** 的 **LoadCompleted** 事件。在这个处理程序中，新页面可以访问传送给页面的内容。接着调用 **Navigate()** 方法导航到 **page2**。新页面接收对象 **menus**，该对象包含了构建固定页面所需的

所有菜单信息。menus 变量的类型是 `ObservableCollection<MenuItem>`。



可从
wrox.com
下载源代码

```
private void OnCreateDoc(object sender, RoutedEventArgs e)
{
    if (menus.Count == 0)
    {
        MessageBox.Show("Select a date first", "Menu Planner",
            MessageBoxButton.OK);
        return;
    }
    var page2 = new DocumentPage();
    NavigationService.LoadCompleted +=
        page2.NavigationService_LoadCompleted;
    NavigationService.Navigate(page2, menus);
}
```

代码段 CreateXps/MenuPlannerPage.xaml.cs

在 `DocumentPage` 中, 使用 `DocumentViewer` 获取对固定文档的读取访问权限。固定文档在 `NavigationService_LoadCompleted()` 方法中创建。在这个事件处理程序中, 从第一个页面传递的数据通过 `NavigationEventArgs` 的 `ExtraData` 属性接收。把接收到的 `ObservableCollection<MenuItem>` 赋予 `menus` 变量, 该变量用于构建固定页面:



可从
wrox.com
下载源代码

```
internal void NavigationService_LoadCompleted(object sender,
        NavigationEventArgs e)
{
    menus = e.ExtraData as ObservableCollection<MenuItem>;

    fixedDocument = new FixedDocument();
    var pageContent1 = new PageContent();
    fixedDocument.Pages.Add(pageContent1);
    var page1 = new FixedPage();
    pageContent1.Child = page1;
    page1.Children.Add(GetHeaderContent());
    page1.Children.Add(GetLogoContent());
    page1.Children.Add(GetDateContent());
    page1.Children.Add(GetMenuContent());

    viewer.Document = fixedDocument;

    NavigationService.LoadCompleted -= NavigationService_LoadCompleted;
}
```

代码段 CreateXps/DocumentPage.xaml.cs

固定文档用 `FixedDocument` 类创建。`FixedDocument` 元素只包含可通过 `Pages` 属性访问的 `PageContent` 元素。`PageContent` 元素必须按它们显示在页面上的顺序添加到文档中。`PageContent` 定义了单个页面的内容。

`PageContent` 有一个 `Child` 属性, 因此可以把 `PageContent` 关联到 `FixedPage` 上。在 `FixedPage` 上可以把 `UIElement` 类型的元素添加到 `Children` 集合中。在这个集合中可以添加前两章介绍的所有元素, 包括 `TextBlock`, 它本身可以包含 `Inline` 和 `Block` 元素。

在示例代码中, `FixedPage` 的子元素用辅助方法 `GetHeaderContent()`、`GetLogoContent()`、`GetDateContent()` 和 `GetMenuContent()` 创建。

`GetHeaderContent()`方法创建一个 `TextBlock`, 并返回它。给 `TextBlock` 添加 `Inline` 元素 `Bold`, 又给 `Bold` 添加 `Run` 元素。`Run` 元素包含文档的标题文本。利用 `FixedPage.SetLeft()`和 `FixedPage.SetTop()`, 定义 `TextBox` 在固定页面中的位置。

```
private UIElement GetHeaderContent()
{
    var text1 = new TextBlock();
    text1.FontFamily = new FontFamily("Mangal");
    text1.FontSize = 34;
    text1.HorizontalAlignment = HorizontalAlignment.Center;
    text1.Inlines.Add(new Bold(new Run("cn|elements")));
    FixedPage.SetLeft(text1, 170);
    FixedPage.SetTop(text1, 40);
    return text1;
}
```

`GetLogoContent()`方法在固定文档中使用 `RadialGradientBrush` 添加一个 `Ellipse` 形状的徽标:

```
private UIElement GetLogoContent()
{
    var ellipse = new Ellipse
    {
        Width = 90,
        Height = 40,
        Fill = new RadialGradientBrush(Colors.Yellow, Colors.DarkRed)
    };

    FixedPage.SetLeft(ellipse, 500);
    FixedPage.SetTop(ellipse, 50);
    return ellipse;
}
```

`GetDateContent()`方法访问 `menus` 集合, 把一个日期范围添加到文档中:

```
private UIElement GetDateContent()
{
    string dateString = String.Format("{0:d} to {1:d}",
        menus[0].Day, menus[menus.Count - 1].Day);
    var text1 = new TextBlock
    {
        FontSize = 24,
        HorizontalAlignment = HorizontalAlignment.Center
    };
    text1.Inlines.Add(new Bold(new Run(dateString)));
    FixedPage.SetLeft(text1, 130);
    FixedPage.SetTop(text1, 90);
    return text1;
}
```

最后, `GetMenuContent()`方法创建并返回一个 `Grid` 控件, 这个网格中的列和行包含日期、菜单和价格信息:

```
private UIElement GetMenuContent()
{
    var grid1 = new Grid { ShowGridLines = true };
}
```

```

grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width= new GridLength(50)});
grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width = new GridLength(300)});
grid1.ColumnDefinitions.Add(new ColumnDefinition
    { Width = new GridLength(70) });

for (int i = 0; i < menus.Count; i++)
{
    grid1.RowDefinitions.Add(new RowDefinition
        { Height = new GridLength(40) });

    var t1 = new TextBlock(
        new Run(String.Format("{0:ddd}", menus[i].Day)));
    t1.VerticalAlignment = VerticalAlignment.Center;
    t1.Margin = new Thickness(5, 2, 5, 2);
    Grid.SetColumn(t1, 0);
    Grid.SetRow(t1, i);
    grid1.Children.Add(t1);

    var t2 = new TextBlock(new Run(menus[i].Menu));
    t2.VerticalAlignment = VerticalAlignment.Center;
    t2.Margin = new Thickness(5, 2, 5, 2);
    Grid.SetColumn(t2, 1);
    Grid.SetRow(t2, i);
    grid1.Children.Add(t2);

    var t3 = new TextBlock(new Run(menus[i].Price.ToString()));
    t3.VerticalAlignment = VerticalAlignment.Center;
    t3.Margin = new Thickness(5, 2, 5, 2);
    Grid.SetColumn(t3, 2);
    Grid.SetRow(t3, i);
    grid1.Children.Add(t3);
}

FixedPage.SetLeft(grid1, 100);
FixedPage.SetTop(grid1, 140);
return grid1;
}

```

运行应用程序，所创建的固定文档如图 37-13 所示。

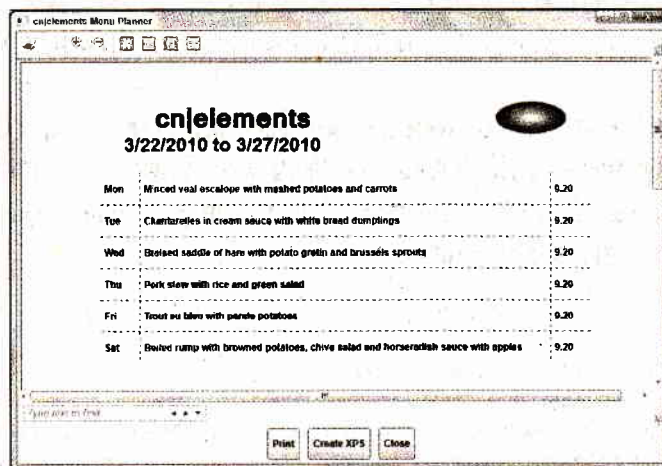


图 37-13

37.4 XPS 文档

使用 Microsoft Word, 可以把文档另存为 PDF 或 XPS 文件。自从 Word 2007 SP2 版本发布以来, 这个功能就包含在 Microsoft Word 中。如果有 Word 2007, 但没有安装 SP2, 就可以免费下载这个插件。XPS 是 XML 纸张规范(XML Paper Specification), 是 WPF 的一个子集。Windows 包含一个 XPS 读取器。

.NET 在 System.Windows.Xps、System.Windows.Xps.Packaging 和 System.IO.Packaging 名称空间中包含读写 XPS 文档的类和接口。

因为 XPS 以 ZIP 文件格式打包, 所以很容易把扩展名为 .xps 文件重命名为 .zip, 打开该归档文件, 来分析 XPS 文档。

XPS 文件需要在 .zip 文档中有 XML 纸张规范(可从 <http://www.microsoft.com/whdc/xps/xpsspec.mspx> 上下载)定义的特定结构。这个结构基于 OPC(Open Packaging Convention, 开放打包约定), Word 文档(OOXML 或 Office Open XML)也基于 OPC。在这个文件中, 可以包含用于元数据、资源(如字体和图片)和文档本身的不同文件夹。在 XPS 文档的文档文件夹中, 可以找到表示 XAML 的 XPS 子集的 XAML 代码。

要创建 XPS 文档, 可使用 System.Windows.Xps.Packaging 名称空间中的 XpsDocument 类。要使用这个类, 也需要引用程序集 ReachFramework。通过这个类可以给文档添加缩略图(AddThumbnail())和固定文档序列(AddFixedDocumentSequence()), 还可以给文档加上数字签名。固定文档序列使用 IXpsFixedDocumentSequenceWriter 接口写入, 该接口使用 IXpsFixedDocumentWriter 在序列中写入文档。

如果 FixedDocument 已经存在, 写入 XPS 文档就有一个更简单的方法。不需要添加每个资源和每个文档页, 而可以使用 System.Windows.Xps 名称空间中的 XpsDocumentWriter 类。要使用这个类, 必须引用 System.Printing 程序集。

下面的代码段包含一个创建 XPS 文档的处理程序。首先创建用于菜单规划的文件名, 它使用星期几和名称 menunplan。星期几用 GregorianCalendar 类来计算。接着打开 SaveFileDialog, 让用户覆盖已创建的文件名, 并选择在其中存储文件的目录。SaveFileDialog 类在名称空间 Microsoft.Win32 中定义, 它封装本地文件对话框。接着新建一个 XpsDocument, 其中将文件名传送给构造函数。因为 XPS 文件使用 ZIP 格式压缩内容, 所以使用 CompressionOption 可以指定该压缩是在时间上还是空间上进行优化。

之后使用静态方法 XpsDocument.CreateXpsDocumentWriter() 创建一个 XpsDocumentWriter。重载 XpsDocumentWriter 的 Write() 方法, 从而接受不同的内容或内容部分写入文档中。Write() 可方法接受的选项有 FixedDocumentSequence、FixedDocument、FixedPage、string 和 DocumentPaginator。在示例代码中, 仅传送了前面创建的 fixedDocument:



可从
wrox.com
下载源代码

```
private void OnCreateXPS(object sender, RoutedEventArgs e)
{
    var c = new GregorianCalendar();
    int weekNumber = c.GetWeekOfYear(menus[0].Day,
        CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
    string fileName = String.Format("menuplan{0}", weekNumber);

    var dlg = new SaveFileDialog
```

```

{
    FileName = fileName,
    DefaultExt = ".xps",
    Filter = "XPS Documents|*.xps|All Files|*.*",
    AddExtension = true
};

if (dlg.ShowDialog() == true)
{
    XpsDocument doc = new XpsDocument(dlg.FileName, FileAccess.Write,
                                     CompressionOption.Fast);
    XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(doc);
    writer.Write(fixedDocument);
    doc.Close();
}
}

```

代码段 CreateXps/DocumentPage.xaml.cs

运行应用程序，以存储 XPS 文档，就可以用 XPS 查看器查看文档，如图 37-14 所示。

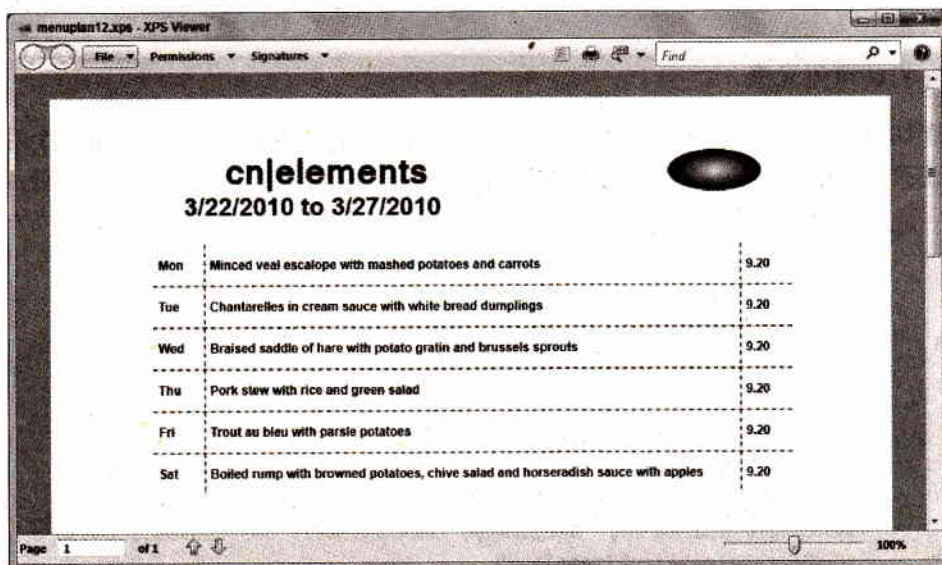


图 37-14

还可以给 `XpsDocumentWriter` 的一个重载版本的 `Write()` 方法传递 `Visual`，`Visual` 是 `UIElement` 的基类，因此可以给写入器传递任意 `UIElement`，从而方便地创建 XPS 文档。这个功能在下面的打印示例中使用。

37.5 打印

用 `DocumentViewer` 打印显示在屏幕上的 `FixedDocument`，最简单的方法是使用关联到该文档上的 `DocumentViewer` 的 `Print()` 方法。对于菜单规划应用程序，这都在 `OnPrint()` 处理程序中完成。`DocumentViewer` 的 `Print()` 方法会打开 `PrintDialog`，把关联的 `FixedDocument` 发送给选中的打印机：



可从
wrox.com
下载源代码

```
private void OnPrint(object sender, RoutedEventArgs e)
{
    viewer.Print();
}
```

代码段 CreateXps/DocumentPage.xaml.cs

37.5.1 用 PrintDialog 打印

如果希望更多地控制打印过程,就可以实例化 `PrintDialog`,并用 `PrintDocument()`方法打印文档。`PrintDocument()` 方法需要把 `DocumentPaginator` 作为第一个参数。`FixedDocument` 通过 `DocumentPaginator` 属性返回一个 `DocumentPaginator` 对象。第二个参数定义了当前打印机在“打印机”对话框中为打印作业显示的字符串:



可从
wrox.com
下载源代码

```
var dlg = new PrintDialog();
if (dlg.ShowDialog() == true)
{
    dlg.PrintDocument(fixedDocument.DocumentPaginator, "Menu Plan");
}
```

代码段 CreateXps/DocumentPage.xaml.cs

37.5.2 打印可见元素

创建 `UIElement` 对象也很简单。下面的 XAML 代码定义了一个椭圆、一个矩形和一个用两个椭圆元素表示的按钮。利用该按钮的 `Click` 处理程序 `OnPrint()`, 会启动可见元素的打印作业:



可从
wrox.com
下载源代码

```
<Canvas x:Name="canvas1">
    <Ellipse Canvas.Left="10" Canvas.Top="20" Width="180" Height="60"
        Stroke="Red" StrokeThickness="3">
        <Ellipse.Fill>
            <RadialGradientBrush>
                <GradientStop Offset="0" Color="LightBlue" />
                <GradientStop Offset="1" Color="DarkBlue" />
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle Width="180" Height="90" Canvas.Left="50" Canvas.Top="50">
        <Rectangle.LayoutTransform>
            <RotateTransform Angle="30" />
        </Rectangle.LayoutTransform>
        <Rectangle.Fill>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="Aquamarine" />
                <GradientStop Offset="1" Color="ForestGreen" />
            </LinearGradientBrush>
        </Rectangle.Fill>
        <Rectangle.Stroke>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="LawnGreen" />
                <GradientStop Offset="1" Color="SeaGreen" />
            </LinearGradientBrush>
        </Rectangle.Stroke>
    </Rectangle>
</Canvas>
```

```

<Button Canvas.Left="90" Canvas.Top="190" Content="Print" Click="OnPrint">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
                <Ellipse Grid.Row="0" Grid.RowSpan="2" Width="60"
                    Height="40" Fill="Yellow" />
                <Ellipse Grid.Row="0" Width="52" Height="20"
                    HorizontalAlignment="Center">
                    <Ellipse.Fill>
                        <LinearGradientBrush StartPoint="0.5,0"
                            EndPoint="0.5,1">
                            <GradientStop Color="White" Offset="0" />
                            <GradientStop Color="Transparent"
                                Offset="0.9" />
                        </LinearGradientBrush>
                    </Ellipse.Fill>
                </Ellipse>
                <ContentPresenter Grid.Row="0" Grid.RowSpan="2"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center" />
            </Grid>
        </ControlTemplate>
    </Button.Template>
</Button>
</Canvas>

```

代码段 PrintingDemo/MainWindow.xaml

在 OnPrint() 处理程序中, 调用 PrintDialog 的 PrintVisual() 方法可启动打印作业。PrintVisual() 接受派生自 Visual 基类的任意对象:



可从
wrox.com
下载源代码

```

private void OnPrint(object sender, RoutedEventArgs e)
{
    PrintDialog dlg = new PrintDialog();
    if (dlg.ShowDialog() == true)
    {
        dlg.PrintVisual(canvas1, "Print Demo");
    }
}

```

代码段 PrintingDemo/MainWindow.xaml.cs

为了通过编程方式来打印, 而无需用户的干涉, System.Printing 名称空间中的 PrintDialog 类可用于创建一个打印作业, 并调整打印设置。LocalPrintServer 类提供了打印队列的信息, 并用 DefaultPrintQueue 属性返回默认的 PrintQueue。使用 PrintTicket 可以配置打印作业。PrintQueue.DefaultPrintTicket 返回与队列关联的默认 PrintTicket。PrintQueue 的 GetPrintCapabilities() 方法返回打印机的功能, 根据该功能可以配置 PrintTicket, 如下面的代码段所示。配置完 PrintTicket 后, 静态方法 PrintQueue.CreateXpsDocumentWriter() 返回一个 XpsDocumentWriter 对象。XpsDocumentWriter 类以前用于创建 XPS 文档, 也可以使用它启动打印作业。XpsDocumentWriter 的 Write() 方法不仅接受 Visual 或 FixedDocument 作为第一个参数, 还接

受 `PrintTicket` 作为第二个参数。如果把 `PrintTicket` 和第二个参数一起传递，写入器的目标就是与对应标记关联的打印机，因此写入器把打印作业发送给打印机。

```
LocalPrintServer printServer = new LocalPrintServer();

PrintQueue queue = printServer.DefaultPrintQueue;

PrintTicket ticket = queue.DefaultPrintTicket;
PrintCapabilities capabilities = queue.GetPrintCapabilities(ticket);
if (capabilities.DuplexingCapability.Contains(
    Duplexing.TwoSidedLongEdge))
    ticket.Duplexing = Duplexing.TwoSidedLongEdge;
if (capabilities.InputBinCapability.Contains(InputBin.AutoSelect))
    ticket.InputBin = InputBin.AutoSelect;
if (capabilities.MaxCopyCount > 3)
    ticket.CopyCount = 3;
if (capabilities.PageOrientationCapability.Contains(
    PageOrientation.Landscape))
    ticket.PageOrientation = PageOrientation.Landscape;
if (capabilities.PagesPerSheetCapability.Contains(2))
    ticket.PagesPerSheet = 2;
if (capabilities.StaplingCapability.Contains(
    Stapling.StapleBottomLeft))
    ticket.Stapling = Stapling.StapleBottomLeft;

XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(queue);
writer.Write(canvas1, ticket);
```

37.6 小结

本章学习了如何把 WPF 功能用于文档，如何创建根据屏幕大小自动调整的流文档，以及如何创建外观总是不变的固定文档。我们还讨论了如何打印文档，如何把可见元素发送给打印机。

下一章继续讨论 WPF 技术，并揭示 Silverlight 的功能。

第 38 章

Silverlight

本章内容:

- WPF 与 Silverlight 的区别
- 创建 Silverlight 项目
- 在页面之间导航
- 使用 System.Net、WCF 和数据服务联网
- 与浏览器集成

前几章创建了 WPF 应用程序。Silverlight 是 WPF 的一个子集，它提供了运行在 Web 浏览器中的应用程序。从 3.0 版本开始，应用程序也可以独立运行。Silverlight 的优点是不必在 Windows 操作系统上运行它，它也可以在其他平台上运行：只需在浏览器上安装一个包含 Silverlight 运行库的插件即可。Silverlight 运行库包含 .NET Framework 的一个子集和 WPF 的一个子集。

在开始本章的内容之前，应先了解 XAML 和 WPF，如第 27 章和第 35 章所述。

38.1 WPF 和 Silverlight 的比较

Silverlight 和 WPF 在许多方面都是类似的，但也有重要的区别。Silverlight 由核心显示架构(WPF 的一个子集)、.NET Framework for Silverlight(.NET Framework 的一个子集)、安装程序和更新程序组成。WPF 应用程序运行在 Windows 系统上，至少需要 .NET Client Profile。Silverlight 使用一个插件模型，并驻留在 Web 浏览器中。

Silverlight 可用于许多浏览器和操作系统。除了 Internet Explorer 之外，Silverlight 也可以用于 Firefox、Safari、Opera 和 Google Chrome。并不是所有浏览器都支持 Silverlight。<http://go.microsoft.com/fwlink/?LinkId=128526> 包含一个支持 Silverlight 的操作系统和浏览器列表。



在 Linux 系统上，可以使用 Moonlight 运行 Silverlight 应用程序。Moonlight 可以从 <http://www.gomono.com/moonlight> 上下载。在这个页面上查看 Moonlight 支持的功能的当前状态。

编译 WPF 应用程序时，会得到一个可执行的程序集，其中包含二进制格式的 XAML 代码(称为 BAML)，作为资源。而对于 Silverlight 应用程序，编译器会创建一个 XAP 文件，这是一个 ZIP 包，

其中包含程序集和配置。

.NET Framework for Silverlight 中的类与完整的 .NET Framework 相同,但并不包含 .NET Framework 的所有类。一些类被删除了。另外,类中还添加了几个可用于 Silverlight 的方法和属性。

对于 Silverlight 3 完全不可用而 WPF 支持的功能是流文档和固定文档(参见第 37 章)和 3-D(参见第 35 章)。在 Silverlight 中可以用 2-D 模拟 3-D,但这当然完全不同于 WPF 的 3-D 功能。与 Windows 窗体的交互操作功能也不可用。无论如何,最好用 XAML 编写新的 UI,而不是集成 Windows 窗体控件。

如果同时为 WPF 和 Silverlight 编写应用程序,则还有一些小区别要注意。

- **鼠标单击和事件:** 在 Silverlight 中,鼠标右击总是由浏览器捕获,不会涉及 Silverlight 控件。所以 Silverlight 没有鼠标右键的按下和释放事件。也没有双击事件。Silverlight 没有用于鼠标滚轮的事件。在读取 `RoutedEventArgs` 的属性时, Silverlight 只能使用 `Handled` 和 `OriginalSource` 属性。这在 Silverlight 4 中有了改变,该版本支持鼠标右击和鼠标滚轮。
- **画笔:** Silverlight 没有 `DrawingBrush`、`VisualBrush` 和 `TileBrush` 画笔。但可使用 `VideoBrush` 画笔,给画笔添加视频。
- **字体:** 并不是所有字体都可以在所有平台上使用,所以 Silverlight 只能使用它自己提供的字体。
- **控件:** Silverlight 没有 `Menu`、`ToolBar`、`Window` 和 `WebBrowser` 控件。但 Silverlight 提供的一些控件也不能用于 WPF。随着时间的推移,这些区别会消失。`Calendar`、`DatePicker`、`DataGrid` 控件最初出现在 Silverlight 中,现在也可用于 WPF。WPF 目前没有 `AutoCompleteBox` 和 `NumericUpDown` 控件。
- **联网:** 为了避免 UI 线程的阻塞,只有异步调用可用。在 Silverlight 3 中,只能通过 WCF 使用 `BasicHttpBinding` 和 `PollingDuplexHttpBinding`。但还可以使用二进制编码。
- **文件系统访问:** Silverlight 3 应用程序不允许读写客户端系统。可以读写独立的存储器。`IsolatedStorageFile` 和 `IsolatedStorageFileStream` 是可用的。独立的存储器参见第 29 章。在运行的 Silverlight 控件上,可以单击鼠标右键,打开 Silverlight 菜单,其中提供了几个包含信息和配置选项的选项卡。`Application Storage` 选项卡提供了通过独立存储器使用数据的应用程序的相关信息,还可以删除它。可以把这些信息与 Web 应用程序的 cookie 比较,cookie 和独立存储器之间的配额显著不同。在 Silverlight 4 中,可以使用文件系统上的更多控件。
- **浏览器集成:** 因为 Silverlight 控件一般运行在 Web 浏览器中,所以与浏览器的集成很重要。可以定义能从 JavaScript 中调用的 .NET 类,使用 `System.Windows.Browser` 名称空间中的类还可以在 Silverlight .NET 代码中调用 HTML 和 JavaScript。
- **媒体:** 这是 Web 应用程序中得到 Silverlight 特别支持的一个重要方面。通过流,可以使用渐进下载和平滑流等功能,还可以给流添加时间轴标记。利用 `DeepZoom` 技术和 `MultiScaleImage` 控件,可以创建令人印象深刻的用户体验,允许用户放大从几个图像文件中构建的大型图像。要创建 `DeepZoom` 图像,可以使用 `DeepZoom` 合成器,它会创建 Silverlight 代码,以使用 `MultiScaleImage` 控件。

38.2 创建 Silverlight 项目

用 Visual Studio 新建 Silverlight 项目时,系统会询问用户是否同时创建一个 Web 项目,如图 38-1

所示。该 Web 项目包含可测试 Silverlight 应用程序的测试页面，以及 Silverlight 项目的二进制文件。

对于 Web 项目类型，可以选择 ASP.NET Web Application Project 和 ASP.NET MVC Web Project。ASP.NET MVC 参见第 42 章。

本章创建的项目是 SilverlightDemos，对应的 Web 项目是 SilverlightDemos.Web。在 Visual Studio 2010 中，Web Project 项目设置是一个名为 SilverlightApplication 的选项卡，它包含对解决方案中对 Silverlight 项目的引用，以及复制 Silverlight 项目的二进制代码(XAP 文件)的目标路径，通常是目录 ClientBin。

HTML 测试页面包含一个 object 标记，它通过 source 参数引用 Silverlight 控件的二进制代码。参数 minRuntimeVersion 定义在客户端系统上需要的 Silverlight 运行库的版本。如果客户端系统没有安装 Silverlight 运行库，就用默认图像指定一个指向 Microsoft 站点的超链接，如图 38-2 所示，通知用户 Microsoft Silverlight 的安装信息。这个测试页面启动时，Silverlight 控件就会加载到 Web 浏览器中。

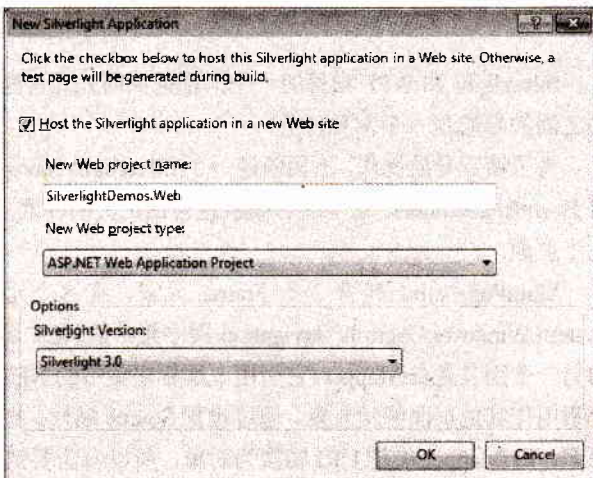


图 38-1



图 38-2



可从
wrox.com
下载源代码

```
<object data="data:application/x-silverlight-2,"
        type="application/x-silverlight-2" width="100%" height="100%">
  <param name="source" value="ClientBin/SilverlightDemos.xap"/>
  <param name="onError" value="onSilverlightError" />
  <param name="background" value="white" />
  <param name="minRuntimeVersion" value="3.0.40818.0" />
  <param name="autoUpgrade" value="true" />
  <a href="http://go.microsoft.com/fwlink/?LinkId=149156 & v=3.0.40818.0"
    style="text-decoration:none">
    
  </a>
</object>
```

代码段 SilverlightDemos.Web/SilverlightDemosTestPage.html

38.3 导航

Silverlight 和 WPF 项目的一个区别是如何处理页面之间的导航。当然，在 Web 应用程序中，页面之间的导航是一个常见任务。

为了演示导航操作，下面新建一个 Silverlight Navigation Application 类型的 Visual Studio 2010 项目 EventRegistration。这个项目模板包含预定义的样式，一个主页，以及 Views 文件夹中要导航到的几个页面。

MainPage.xaml 包含一个 Frame 元素，在这个元素的前面加上别名 navigation，该别名引用 System.Windows.Controls.Navigation 程序集中的 .NET 名称空间 System.Windows.Controls。这里使用的另一个别名是 uriMapper，它引用上述程序集中的 .NET 名称空间 System.Windows.Navigation。Frame 元素用作其他页面的父元素。通过设置 Source 属性，把加载的第一个页面从默认页面改为 /Welcome 页面。UriMapper 类把 URI 转换为对象。所以不需要把 /Views/Welcome.xaml 指定为导航源页面，使用 /Welcome 就足够了，它会转换为 Views 目录下的 Welcome.xaml 文件。

这里引用的所有样式都定义为 Style.xaml 文件中的资源。在文件 app.xaml 中，把这个文件添加到应用程序资源文件中。



可从
wrox.com
下载源代码

```
<Border x:Name="ContentBorder" Style="{StaticResource ContentBorderStyle}">
    <navigation:Frame x:Name="ContentFrame"
        Style="{StaticResource ContentFrameStyle}"
        Source="/Welcome" Navigated="ContentFrame_Navigated"
        NavigationFailed="ContentFrame_NavigationFailed">
        <navigation:Frame.UriMapper>
            <uriMapper:UriMapper>
                <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml"/>
                <uriMapper:UriMapping Uri="{pageName}"
                    MappedUri="/Views/{pageName}.xaml"/>
            </uriMapper:UriMapper>
        </navigation:Frame.UriMapper>
    </navigation:Frame>
</Border>
```

代码段 WCFRegistration/MainPage.xaml

MainPage.xaml 包含一个徽标、应用程序名称段和 HyperlinkButton 控件，该控件允许用户在应用程序的主页之间导航。徽标用资源 LogoIcon 指定，LogoIcon 用文件 Style.xaml 中的 Path 元素定义。HyperlinkButton 元素的样式也在样式文件中定义。把 NavigateUri 属性设置为 Uri，它通过前面的 UriMapper 定义来映射。

```
<Grid x:Name="NavigationGrid" Style="{StaticResource NavigationGridStyle}">
    <Border x:Name="BrandingBorder"
        Style="{StaticResource BrandingBorderStyle}">
        <StackPanel x:Name="BrandingStackPanel"
            Style="{StaticResource BrandingStackPanelStyle}">
            <ContentControl Style="{StaticResource LogoIcon}" />
            <TextBlock x:Name="ApplicationNameTextBlock"
                Style="{StaticResource ApplicationNameStyle}"
                Text="CN innovation"/>
        </StackPanel>
    </Border>
</Grid>
```

```

        </StackPanel>
    </Border>
    <Border x:Name="LinksBorder" Style="{StaticResource LinksBorderStyle}">
        <StackPanel x:Name="LinksStackPanel"
            Style="{StaticResource LinksStackPanelStyle}">
            <HyperlinkButton x:Name="Link1"
                Style="{StaticResource LinkStyle}"
                NavigateUri="/Home" TargetName="ContentFrame"
                Content="home"/>
            <Rectangle x:Name="Divider1"
                Style="{StaticResource DividerStyle}"/>
            <HyperlinkButton x:Name="Link2"
                Style="{StaticResource LinkStyle}"
                NavigateUri="/About" TargetName="ContentFrame"
                Content="about"/>
        </StackPanel>
    </Border>
</Grid>

```

显示 Welcome 页面时, 会加载 Silverlight 控件, 因为它直接通过 Frame 元素的 Source 属性来引用。这个 Page 仅包含一个 TextBlock 元素和应用程序的信息, 应用程序的信息包含在 Border 元素内部, 以定义一些视觉效果, 这些视觉效果包含在一个 Canvas 元素中, 以进行定位。



可从
wrox.com
下载源代码

```

<navigation:Page x:Class="Wrox.ProCSharp.Silverlight.Views.Welcome"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:navigation="clr-namespace:System.Windows.Controls;
        assembly=System.Windows.Controls.Navigation"
    Title="Welcome Page">
    <Canvas x:Name="LayoutRoot">
        <Canvas.Resources>
            <!-- Storyboard Resources-->
        </Canvas.Resources>

        <Border x:Name="border1" Canvas.Left="0" Canvas.Top="0" Width="240"
            Height="80" CornerRadius="9">
            <Border.Effect>
                <DropShadowEffect ShadowDepth="7" BlurRadius="3"
                    Color="DarkGreen" />
            </Border.Effect>
            <Border.Background>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <GradientStop x:Name="gradient1" Offset="0"
                        Color="DarkGreen" />
                    <GradientStop x:Name="gradient2" Offset="0.4" Color="Green" />
                    <GradientStop x:Name="gradient3" Offset="0.7"
                        Color="MediumSeaGreen" />
                    <GradientStop x:Name="gradient4" Offset="0.9"
                        Color="LightGreen" />
                </LinearGradientBrush>
            </Border.Background>

            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"

```

```

Text="Event Registration" FontFamily="Comic Sans MS"
FontSize="24" Foreground="Wheat" />

</Border>
</Canvas>
</navigation:Page>

```

代码段 WCFRegistration/Views/Welcome.xaml

在 Canvas 元素的 Resource 部分，定义了两个 Storyboard 元素，以执行动画。这非常类似于第 35 章用于 WPF 的动画。这两种技术在动画方面有微小的差别，例如，Silverlight 动画没有定义 AccelerationRatio 和 DecelerationRatio 属性，而 WPF 定义了这两个属性。但是使用关键帧动画或缓动函数也可以获得相同的功能。缓动函数在可用于 WPF 前，仅包含在 Silverlight 中，在 .NET 4 中扩展 WPF 以包含这个功能。

```

<Canvas.Resources>
  <Storyboard x:Name="startStoryboard">
    <DoubleAnimation From="0" To="400" Duration="00:00:4"
      Storyboard.TargetName="border1"
      Storyboard.TargetProperty="(Canvas.Left)">
      <DoubleAnimation.EasingFunction>
        <CircleEase EasingMode="EaseIn" />
      </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
    <DoubleAnimation From="0" To="300" Duration="00:00:4"
      Storyboard.TargetName="border1"
      Storyboard.TargetProperty="(Canvas.Top)">
      <DoubleAnimation.EasingFunction>
        <BounceEase Bounces="4" EasingMode="EaseIn" />
      </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
  </Storyboard>
  <Storyboard x:Name="endStoryboard">
    <DoubleAnimation To="50" Duration="00:00:4"
      Storyboard.TargetName="border1"
      Storyboard.TargetProperty="(Canvas.Left)">
      <DoubleAnimation.EasingFunction>
        <CircleEase EasingMode="EaseOut" />
      </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
    <DoubleAnimation To="50" Duration="00:00:4"
      Storyboard.TargetName="border1"
      Storyboard.TargetProperty="(Canvas.Top)">
      <DoubleAnimation.EasingFunction>
        <ElasticEase EasingMode="EaseOut" Springiness="4"
          Oscillations="3" />
      </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
    <ColorAnimation Duration="00:00:4"
      Storyboard.TargetName="gradient1"
      Storyboard.TargetProperty="Color" To="DarkBlue" />
    <ColorAnimation Duration="00:00:4"
      Storyboard.TargetName="gradient2"
      Storyboard.TargetProperty="Color" To="Blue" />
    <ColorAnimation Duration="00:00:4"

```

```

Storyboard.TargetName="gradient3"
Storyboard.TargetProperty="Color"
To="MediumBlue" />
<ColorAnimation Duration="00:00:4"
Storyboard.TargetName="gradient4"
Storyboard.TargetProperty="Color" To="LightBlue" />
<DoubleAnimation Duration="00:00:4" BeginTime="00:00:2"
Storyboard.TargetName="border1"
Storyboard.TargetProperty="Width" To="500" />
<DoubleAnimation Duration="00:00:4" BeginTime="00:00:2"
Storyboard.TargetName="border1"
Storyboard.TargetProperty="Height" To="300" />
</Storyboard>
</Canvas.Resources>

```



动画和缓动函数参见第 35 章。

第一个故事板动画 `startStoryboard` 在 `OnNavigatedTo()` 事件处理程序中通过调用 `Begin()` 方法来启动。把第一个故事板动画的 `Completed` 事件设置为 `Lambda` 表达式，在第一个故事板结束时启动第二个故事板 `endStoryboard`。第二个故事板完成时，使用 `NavigationService.Navigate()` 导航到主页上。



可从
wrox.com
下载源代码

```

using System;
using System.Windows.Controls;
using System.Windows.Navigation;

namespace Wrox.ProCSharp.Silverlight.Views
{
    public partial class Welcome : Page
    {
        public Welcome()
        {
            InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            startStoryboard.Completed += (sender1, e1) =>
            {
                endStoryboard.Completed += (sender2, e2) =>
                NavigationService.Navigate(
                    new Uri("/Home", UriKind.Relative));
                endStoryboard.Begin();
            };
            startStoryboard.Begin();
        }
    }
}

```

代码段 WCFRegistration/Views/Welcome.xaml.cs

运行应用程序，会看到动画和超链接按钮，如图 38-3 所示。

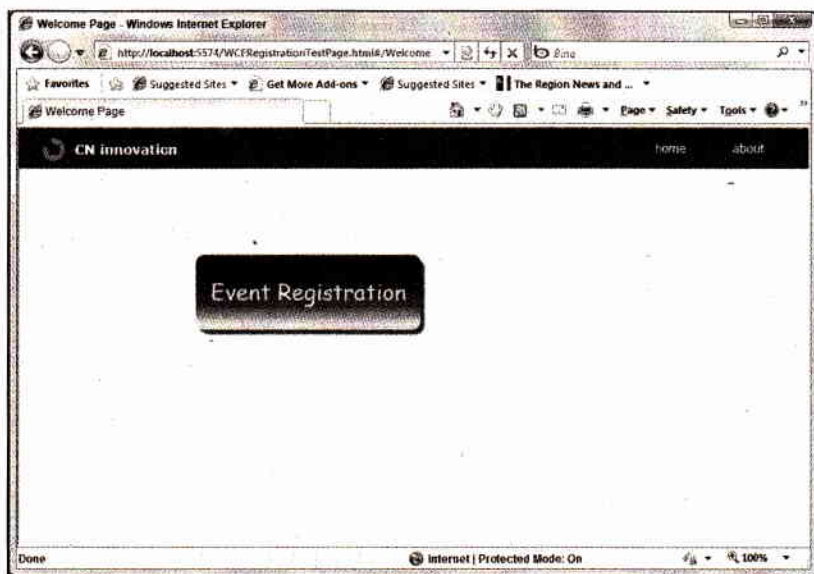


图 38-3

38.4 网络

在 Silverlight 中,网络在某些方面与 .NET Framework 中的栈相同,但也在一些重要的方面有区别。可以使用第 24 章介绍的套接字类 `HttpRequest` 和 `WebClient`,WCF(第 43 章)和 WCF 数据服务(第 32 章)也是可用的。但只能使用异步方法在网络上收发请求。UI 线程不应阻塞,因为这可能给用户带来致命错误。

在 Silverlight 3.0 以前,唯一可用于 Silverlight 的网络栈是浏览器提供的,自从 Silverlight 3.0 开始,情况有了变化,还可以使用另一个 HTTP 栈。

在客户端的浏览器内部运行的应用程序有一个重要的限制:客户端只能从控件来自的同一个服务器中访问网络服务。自从有了 Adobe 的 Flash 技术,这种情况就改变了,但目标服务器必须支持这种技术。Silverlight 网络栈会检查服务器上是否存在 Silverlight Policy 文件 `clientaccesspolicy.xml`。如果这个文件不存在,则 Silverlight 也接受 `crossdomain.xml` 文件,Adobe Flash 也使用这个文件。如果第一个文件存在,就不检查第二个文件,Silverlight 不使用 `crossdomain.xml` 文件中的所有项。如果使用这个文件,则要求整个域标记为公共。

下面是一个示例 Silverlight Policy 文件 `clientaccesspolicy.xml`。对于客户域 `www.cninnovation.com`,把访问权限授予所有子路径。可以限制能访问的资源,也可以给域 URI 使用*,允许客户端来自任何地方。

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="http://www.cninnovation.com"/>
      </allow-from>
    </policy>
  </cross-domain-access>
</access-policy>
```

```

    <grant-to>
      <resource path="/" include-subpaths="true"/>
    </grant-to>
  </policy>
</cross-domain-access>
</access-policy>

```

下面的例子使用 Silverlight 的网络栈。为此，先创建一个 ADO.NET Entity Data Model，在网络上读写数据库。

38.4.1 创建 ADO.NET Entity Data Model

示例 Silverlight 控件用于把出席者注册信息写入数据库中，首先用户必须从一个事件列表中选择，之后才能注册事件。数据库 EventRegistration 定义了 3 个表：Events、Attendees 和 RegistrationCodes，它们分别映射到 ADO.NET Entity Data Model 实体类 Event、Attendee 和 RegistrationCode，如图 38-4 所示。

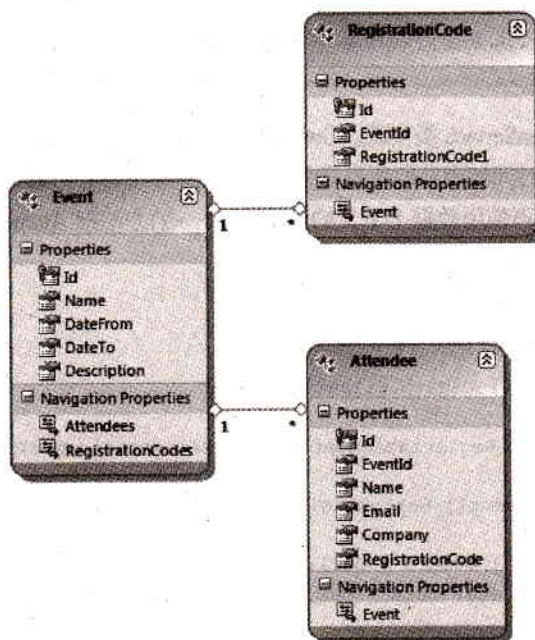


图 38-4



创建和使用 ADO.NET Entity Data Model 的内容参见第 31 章。

38.4.2 为 Silverlight 客户端创建 WCF 服务

为了访问生成的实体类，创建一个 WCF 服务。该服务的合同由 IRegistrationService 接口定义。这个服务合同定义了 GetEvents() 和 RegisterAttendee() 操作。GetEvents() 接受一个用于选择事件的日期范围，并返回由 Entity Data Model 定义的一个 Event 对象数组。RegisterAttendee() 操作需要一个

Attendee 对象作为参数，并根据注册成功与否，返回 true 或 false。



可从
wrox.com
下载源代码

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.Silverlight.Web
{
    [ServiceContract]
    public interface IRegistrationService
    {
        [OperationContract]
        Event[] GetEvents(DateTime fromTime, DateTime toTime);

        [OperationContract]
        bool RegisterAttendee(Attendee attendee);
    }
}
```

代码段 SilverlightDemos.Web/IRegistrationService.cs



WCF 和定义服务合同的内容参见第 43 章。

服务协定用 `RegistrationService` 类实现。`GetEvents()` 方法使用一个 LINQ 查询访问 Entity Framework 的数据上下文，从指定的时间帧中获取所有事件，并返回一个 `Event` 对象数组。`RegisterAttendee()` 方法先验证注册码是有效的，再使用数据上下文的 `SaveChanged()` 方法把 `Attendee` 对象添加到数据库中。



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.ServiceModel.Activation;

namespace Wrox.ProCSharp.Silverlight.Web
{
    public class RegistrationService : IRegistrationService
    {
        public Event[] GetEvents(DateTime fromTime, DateTime toTime)
        {
            Event[] events = null;
            using (var data = new EventRegistrationEntities())
            {
                events = (from e in data.Events
                           where e.DateFrom >= fromTime && e.DateFrom <= toTime
                           select e).ToArray();
                foreach (var ev in events)
                {
                    data.Detach(ev);
                }
            }
            return events;
        }

        public bool RegisterAttendee(Attendee attendee)
        {
            using (EventRegistrationEntities data =
                new EventRegistrationEntities())
            {
            }
        }
    }
}
```

```

    {
        if ((from rc in data.RegistrationCodes
            where rc.RegistrationCode1 == attendee.RegistrationCode
            select rc).Count() < 1)
        {
            return false;
        }
        else
        {
            data.Attendees.AddObject(attendee);
            if (data.SaveChanges() == 1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
}
}
}

```

代码段 SilverlightDemos.Web/RegistrationService.svc.cs

ASP.NET Web 应用程序中的 WCF 宿主使用默认的 `BasicHttpBinding`，它也可以由 Silverlight 客户端使用，因此不需要任何配置。在 Visual Studio 中添加服务引用时，唯一需要的配置是允许使用 HTTP GET 协议发布元数据，如下所示：



可从
wrox.com
下载源代码

```

<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="">
        <serviceMetadata httpGetEnabled="true" />
        <serviceDebug includeExceptionDetailInFaults="false" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>

```

代码段 SilverlightDemos.Web/Web.config



从 Silverlight 3 开始，可以使用 HTTP 传输配置二进制编码。Silverlight 3 实现了这种编码，得到了更好的性能。如果添加一个支持 Silverlight 的 WCF 服务，就默认使用这个自定义绑定。

38.4.3 调用 WCF 服务

在调用服务之前，需要创建 UI。在与前面相同的 Silverlight 项目中，其中实现了动画，把 Home.xaml 页面改为包含组合框和文本框控件，以允许输入事件的信息并注册事件。该应用程序的设计视图如图 38-5 所示。

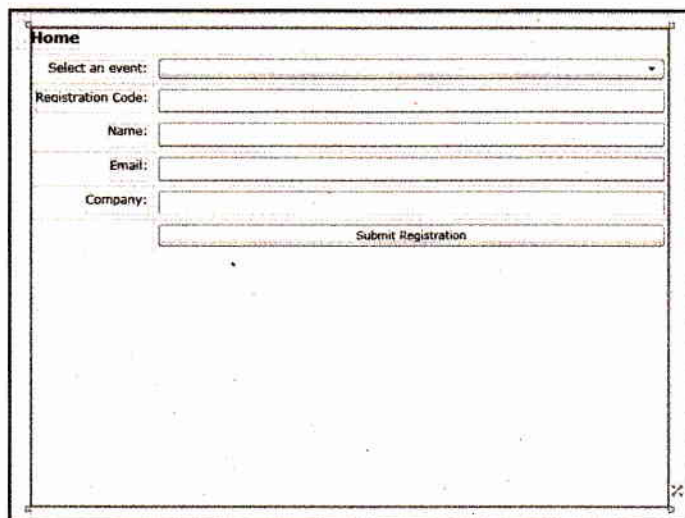


图 38-5

组合框绑定到数据上下文的 `EventList` 属性上，它使用带两个 `TextBlock` 元素的 `DataTemplate`，这两个 `TextBlock` 绑定到项的 `Name` 和 `DateFrom` 属性上。



可从
wrox.com
下载源代码

```
<ComboBox x:Name="comboEvents"
    ItemsSource="{Binding Path=EventList, Mode=OneWay}"
    Grid.Row="0" Grid.Column="1" Margin="5"
    VerticalAlignment="Center">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Vertical">
                <TextBlock Text="{Binding Path=Name}" />
                <TextBlock Text="{Binding Path=DateFrom}" />
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

代码段 `WCRegistration/Views/Home.xaml`

对于数据输入，控件使用几个文本框元素，它们绑定到 `CurrentAttendee` 属性返回的 `Attendee` 对象的属性上。对于 WPF，默认的绑定模式取决于在何处使用控件；对于 Silverlight，默认的绑定模式是 `OneWay`，因此需要改为双向绑定。按钮元素的处理程序方法 `OnRegistration()` 关联到其 `Click` 事件上。

```
<TextBox Grid.Row="1" Grid.Column="1" Margin="5"
    Text="{Binding Path=CurrentAttendee.RegistrationCode,
    Mode=TwoWay}" />
<TextBox Grid.Row="2" Grid.Column="1" Margin="5"
    Text="{Binding Path=CurrentAttendee.Name,
    Mode=TwoWay}" />
<TextBox Grid.Row="3" Grid.Column="1" Margin="5"
    Text="{Binding Path=CurrentAttendee.Email,
    Mode=TwoWay}" />
<TextBox Grid.Row="4" Grid.Column="1" Margin="5"
    Text="{Binding Path=CurrentAttendee.Company,
    Mode=TwoWay}" />
```

```
<Button Grid.Row="5" Grid.Column="1"
        Content="Submit Registration" Margin="5"
        Click="OnRegistration" />
```

在代码隐藏中,把 Silverlight 页面的 `DataContext` 设置为 `this`,以映射 `EventList` 和 `CurrentAttendee` 属性。



可从
wrox.com
下载源代码

```
using System.Collections.ObjectModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Wrox.ProCSharp.Silverlight.RegistrationService;

namespace Wrox.ProCSharp.Silverlight
{
    public partial class Home : Page
    {
        public Home()
        {
            CurrentAttendee = new Attendee();
            EventList = new ObservableCollection<Event>();
            InitializeComponent();

            this.DataContext = this;
        }

        public ObservableCollection<Event>EventList { get; private set; }
        public Attendee CurrentAttendee { get; private set; }
    }
}
```

代码段 WCFRegistration/Views/Home.xaml.cs

通过给 `RegistrationService` 添加一个服务引用,以使用代理类创建异步方法。这个代理类使用异步组件模式。



异步模式和异步组件模式参见第 20 章。

在导航页面时,调用重写的方法 `OnNavigatedTo()`。这里,调用 `GetEventsAsync()`方法,来调用 WCF 服务的 `GetEvents` 操作。完成了异步操作后,就触发 `GetEventsCompleted` 事件。把一个 Lambda 表达式赋予这个事件,以将返回的事件添加到 `eventList` 集合中。因为 `eventList` 是一个 `ObservableCollection<T>`,所以用户界面会在这个列表改变时更新。

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    var client = new RegistrationServiceClient();
    client.GetEventsCompleted += (sender, e1) =>
    {
        if (e1.Error != null)
        {
        }
        else
        {
        }
    }
}
```

```

        {
            foreach (var ev in e1.Result)
            {
                EventList.Add(ev);
            }
        }
    };
    client.GetEventsAsync(DateTime.Today, DateTime.Today.AddMonths(2));
}

```

触发 Submit Registration 按钮的 Click 事件时，会调用 OnRegistration()方法。这里又调用一个异步方法 RegisterAttendeeAsync()来注册出席者。调用完这个方法后，使用 NavigationService.Navigate()把用户导航到另一个页面上。

```

private void OnRegistration(object sender, RoutedEventArgs e)
{
    var client = new RegistrationServiceClient();
    CurrentAttendee.EventId = (comboEvents.SelectedItem as Event).Id;
    client.RegisterAttendeeCompleted += (sender1, e1) =>
    {
        if (e1.Error == null)
        {
            if (e1.Result)
            {
                NavigationService.Navigate(
                    new Uri("/Success", UriKind.Relative));
            }
            else
            {
                NavigationService.Navigate(
                    new Uri("/ErrorPage", UriKind.Relative));
            }
        }
        else
        {
            // display error
        }
    };
    client.RegisterAttendeeAsync(CurrentAttendee);
}

```

现在可以启动 Silverlight 应用程序，以调用 WCF Web 服务。

38.4.4 使用 WCF 数据服务

Silverlight 还允许访问 WCF 数据服务的客户端部分。有了这个功能，就可以使用 HTTP 请求检索和更新 Silverlight 控件中的数据，而无需手工定义服务方法。数据的检索、更新和添加可以使用简单的 HTTP 请求来完成。



WCF 数据服务参见第 32 章。

为了使用 WCF 数据服务给 Silverlight 应用程序提供服务, 在 Web 项目中添加了一个新的 WCF 数据服务项 `EventRegistrationDataService`。其实现代码派生自 `DataService<EventRegistrationEntities>`。基类 `DataService<T>` 的泛型参数是实体数据上下文, 它由 ADO.NET Entity Framework 的模板创建。使用 `config.SetEntitySetAccessRule()`, 把访问规则定义为: 允许读取 `Events` 实体集, 允许访问 `Attendee` 实体集。



可从
wrox.com
下载源代码

```
using System.Collections.Generic;
using System.Data.Services;
using System.Data.Services.Common;
using System.Linq;
using System.ServiceModel.Web;

namespace Wrox.ProCSharp.Silverlight.Web
{
    public class EventRegistrationDataService :
        DataService<EventRegistrationEntities>
    {
        public static void InitializeService(DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("Events", EntitySetRights.AllRead);
            config.SetEntitySetAccessRule("Attendees", EntitySetRights.All);
            config.SetServiceOperationAccessRule("AddAttendee",
                ServiceOperationRights.All);
            config.DataServiceBehavior.MaxProtocolVersion =
                DataServiceProtocolVersion.V2;
        }
    }
}
```

代码段 `SilverlightDemos.Web/EventRegistrationDataService.svc.cs`

为了能执行使用 WCF 数据服务注册出席者的操作(这个操作比添加新记录复杂), 在 `EventRegistrationDataService` 类中添加了 `AddAttendee()` 方法。WebGet 属性指定, 该方法可以使用 HTTP GET 请求来访问。对于 POST、PUT 或 DELETE 请求, 可以使用 `WebInvoke` 属性。在 `InitializeService()` 方法中, 使用 `SetServiceOperationAccessRule()` 方法给这个操作授予访问权限。`AddAttendee()` 方法通过各个参数接收出席者的信息, 第一个参数验证 `registrationCode` 是否存在, 再把出席者写入 `Attendees` 表中, 该表从 Entity Data Model 中映射。使用这个方法的各个参数, 来替代 `Attendee` 对象时, 需要用 HTTP GET 请求调用这个方法。

```
[WebGet]
public bool AddAttendee(string name, string email, string company,
    string registrationCode, int eventId)
{
    if ((from rc in this.CurrentDataSource.RegistrationCodes
        where rc.RegistrationCode1 == registrationCode
        select rc).Count() < 1)
    {
        return false;
    }
    else
    {
        var attendee = new Attendee {
            Name = name,
            Email = email,
```

```

        Company = company,
        RegistrationCode = registrationCode,
        EventId = eventId
    };
    this.CurrentDataSource.Attendees.AddObject(attendee);
    return !(this.CurrentDataSource.SaveChanges() < 1);
}
}
}
}
}

```

给 Silverlight 控件项目 `EventRegistration` 添加一个服务引用, 就可以使用 WCF 数据服务客户端类来访问该服务。`OnNavigatedTo()`方法现在改为使用从数据服务客户端上生成的代理, 而不是前面使用的纯 WCF 代理。

`EventRegistrationEntities` 是一个在添加服务引用时生成的类, 它派生自名称空间 `System.Data.Service.Client` 中的 `DataServiceContext`。这个类为客户端定义了一个上下文, 可用于跟踪和修改信息, 并从 LINQ 查询中创建 HTTP 请求。LINQ 查询定义为从指定的日期范围中获取所有事件。与 WCF 数据服务相比, 这里的区别是不能使用同步方法, 而这正是我们期望的。调用该查询(因为它由变量 `q` 定义)会导致一个 `NotSupportedException` 异常。而必须调用 `DataServiceQuery<T>` 类的 `BeginExecute()` 方法。这个方法使用异步模式, 接受一个 `AsyncCallback` 委托作为第一个参数。这里把一个 `Lambda` 表达式传递给这个委托参数, 只要从服务中返回数据, 就调用这个委托。调用 `EndExecute()` 检索返回的数据, 并把它写入 `ObservableCollection<T>` 中。



可从
wrox.com
下载源代码

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Uri serviceRoot =
        new Uri("EventRegistrationDataService.svc", UriKind.Relative);
    var data = new EventRegistrationEntities(serviceRoot);
    var q = from ev in data.Events
            where ev.DateFrom >= DateTime.Today &&
                  ev.DateFrom <= DateTime.Today.AddMonths(2)
            select ev;
    DataServiceQuery<Event> query = (DataServiceQuery<Event>)q;
    query.BeginExecute(ar =>
    {
        var query1 = ar.AsyncState as DataServiceQuery<Event>;
        var events = query1.EndExecute(ar);
        foreach (var ev in events)
        {
            EventList.Add(ev);
        }
    }, query);
}
}

```

代码段 EventRegistration/Home.xaml.cs

调用 `AddAttendee()` 方法通过 `System.Net` 名称空间中的 `WebClient` 类完成, 如下所述。

38.4.5 使用 System.Net 访问服务

因为服务提供的 `AddAttendee()` 方法可以直接通过 HTTP GET 请求来访问, 所以这个请求很容易

通过 System.Net 名称空间中的类发出, 如 WebClient 或 HttpWebRequest。当然, 对于 Silverlight, 只能使用异步方法。WebClient 类实现异步组件模式; HttpWebRequest 实现异步模式。



System.Net 名称空间参见第 24 章。

现在, OnRegistration() 事件处理方法已修改的实现代码使用 WebClient 类。调用 DownloadStringAsync() 方法会开始请求把 URL 字符串传递给方法。在 GET 请求中, 所有参数都使用 URL 字符串定义。只要从服务中返回数据, 就用 WebClient 类触发 DownloadStringCompleted 事件。这里创建了一个 Lambda 表达式, 以解析 XML 消息中的布尔返回值。根据解析结果, 导航到 Success 或 ErrorPage。



可从
wrox.com
下载源代码

```
private void OnRegistration(object sender, RoutedEventArgs e)
{
    CurrentAttendee.EventId = (comboEvents.SelectedItem as Event).Id;

    var client = new WebClient();
    client.DownloadStringCompleted += (sender1, e1) =>
    {
        if (e1.Error != null)
        {
            NavigationService.Navigate(
                new Uri("/ErrorPage", UriKind.Relative));
        }

        bool result = bool.Parse(XElement.Parse(e1.Result).Value);
        if (result)
            NavigationService.Navigate(
                new Uri("/Success", UriKind.Relative));
        else
            NavigationService.Navigate(
                new Uri("/ErrorPage", UriKind.Relative));
    };

    Uri requestUri = new Uri(String.Format(
        "../../../EventRegistrationDataService.svc/AddAttendee?name='{0}'" +
        "& email='{1}' & company='{2}' & registrationCode='{3}' & eventId={4}",
        CurrentAttendee.Name, CurrentAttendee.Email,
        CurrentAttendee.Company, CurrentAttendee.RegistrationCode,
        CurrentAttendee.EventId), UriKind.Relative);
    client.DownloadStringAsync(requestUri);
}
```

代码段 EventRegistration/Home.xaml.cs

运行应用程序, 就可以注册事件了。

自从 Silverlight 3 以来, Silverlight 应用程序就不必使用浏览器中的网络栈了。浏览器中的网络栈仅限于它能执行的 HTTP 请求的方法, 且只返回关于响应状态码的有限信息。但是, 现在可以使用另一种网络栈, 它可以用 System.Net.Browser 名称空间中的 WebRequestCreator 类选择。下面的代

码段把网络栈改为 ClientHttp。BrowseHttp 是 WebRequestCreator 中的第二个选项。这个修改会把网络栈切换为使用我们见过的所有网络选项。

```
bool result = WebRequest.RegisterPrefix("http://", WebRequestCreator.ClientHttp);
```

38.5 浏览器集成

与 Web 浏览器的集成常常是使用 Silverlight 的一个重要部分。Silverlight 1.0 只能使用 JavaScript 编程，Silverlight 2.0 改变了这一状况，但仍有许多情形需要在 Silverlight 控件中控制 HTML 和 JavaScript 代码，或者从 JavaScript 中调用 .NET 方法。这两种情况可以使用 System.Windows.Browser 名称空间来处理。

为了使用现有的 Web 解决方案来说明这种集成，下面的例子新建一个 Silverlight 项目 JavaScriptInterop。这个控件用于说明在 Silverlight 控件中调用 HTML 代码，以及从 JavaScript 中调用 .NET 方法。

38.5.1 调用 JavaScript

为了从 Silverlight 控件中调用 HTML 页面，HTML 测试页面 JavaScriptInteropTestPage.html 包含一个 HTML 按钮控件 button1。这个按钮要从 Silverlight 控件中修改。



可从
wrox.com
下载源代码

```
<form id="form1" runat="server" style="height: 100%">
  <input id="button1" type="button" value="Click me!" />
```

代码段 SilverlightDemos.Web/JavaScriptInteropTestPage.html

Silverlight 控件的 XAML 代码有一个文本框控件和一个按钮控件。文本框允许输入要显示在 HTML 按钮上的文本。



可从
wrox.com
下载源代码

```
<TextBox Grid.Row="0" Grid.Column="0" x:Name="text1" Margin="5" />
<Button Grid.Row="0" Grid.Column="1" Content="Set value for HTML button"
  Click="OnChangeHtml" Margin="5" />
<TextBlock Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="5"
  Text="" x:Name="text2" />
```

代码段 JavaScriptInterop/MainPage.xaml

在按钮的 Click 事件的处理程序方法 OnChangeHtml() 中，HTML 元素 button1 通过调用 HtmlDocument 的 GetElementById() 方法来访问。如果习惯使用 HTML 的 DOM 模型，这里的代码看起来就非常类似。使用 HtmlElement 就可以修改 HTML 元素中的特性，例如，value 特性可以改为 text1.Text，把它设置为用户输入的文本。



可从
wrox.com
下载源代码

```
private void OnChangeHtml(object sender, RoutedEventArgs e)
{
    HtmlDocument doc = HtmlPage.Document;
    HtmlElement element = doc.GetElementById("button1");
    element.SetAttribute("value", text1.Text);
}
```

代码段 JavaScriptInterop/MainPage.xaml.cs

还可以在 `HtmlElement` 上调用 `AttachEvent()` 方法, 给 HTML 元素添加事件处理程序, 如下所示。这里把 `OnChangeHtmlButtonClick()` 方法赋予 HTML 按钮 `button1` 的 `onclick` 事件。

```
public MainPage()
{
    InitializeComponent();
    HtmlDocument doc = HtmlPage.Document;
    HtmlElement element = doc.GetElementById("button1");
    element.AttachEvent("onclick", OnHtmlButtonClick);
}

private void OnHtmlButtonClick(object sender, HtmlEventArgs e)
{
    text2.Text = "HTML button onclick fired";
}
```

使用 `System.Windows.Browser` 名称空间中的类, 可以访问 HTML 中的所有信息。`HtmlPage`、`HtmlDocument`、`HtmlWindow` 和 `HtmlObject` 是其中最重要的类, 分别用于访问浏览器信息、cookie、页面中的元素和样式表, 还可以读取和修改内容。还可以采用另一种方式: 从 JavaScript 中调用 Silverlight, 如下所述。

38.5.2 JavaScript 调用 Silverlight

为了使 .NET 方法可用于 JavaScript, 必须应用 `ScriptableMember` 特性。如果要使类的所有成员可用于 JavaScript, 就可以应用 `ScriptableType` 特性。这里把 `ScriptableMember` 特性应用于 `ToUpper()` 方法, 该方法把输入字符串改为大写:



可从
wrox.com
下载源代码

```
[ScriptableMember]
public string ToUpper(string s)
{
    return s.ToUpper();
}
```

代码段 `JavaScriptInterop/MainPage.xaml.cs`

如果类型包含可编写脚本的成员, 则必须在 HTML 页面上注册, 以便脚本引擎能找到该类型。为此, 需要调用 `HtmlPage` 类中的 `RegisterScriptableObject()` 方法。这个方法的一个参数是 JavaScript 用于查找对象的键名, 第二个参数定义了用于访问可编写脚本的成员的实例:

```
public MainPage()
{
    InitializeComponent();
    //...

    HtmlPage.RegisterScriptableObject("ScriptKey", this);
}
```

现在可以从 JavaScript 中访问 .NET 代码了。为了从 JavaScript 中找到可编写脚本的对象, 必须访问 Silverlight 控件。为了便于访问 Silverlight 控件, 把 `id` 特性添加到引用 Silverlight 包的 `object` 标记上。另外, 把文本和按钮元素添加到 HTML 代码中。按钮把 JavaScript 函数 `callDotnet()` 赋予 `onclick` 事件, 当调用托管代码时, 就会触发该事件。



可从
wrox.com
下载源代码

```
<input id="text1" type="text" />
<input id="button2" type="button" value="Convert with a .NET method"
      onclick="callDotnet()" />
<div id="silverlightControlHost">
  <object id="plugin" data="data:application/x-silverlight-2,"
        type="application/x-silverlight-2"
        width="30%" height="30%">
    <param name="source" value="ClientBin/JavaScriptInterop.xap" />
    <param name="onError" value="onSilverlightError" />
    <param name="background" value="white" />
    <param name="minRuntimeVersion" value="3.0.40818.0" />
    <param name="autoUpgrade" value="true" />
    <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=3.0.40818.0"
      style="text-decoration: none">
      
    </a>
  </object>
```

代码段 SilverlightDemos.Web/JavaScriptInteropTestPage.html

JavaScript 函数 `callDotnet()` 使用插件标识符, 通过 `getElementById()` 查找 Silverlight 控件。注册时使用前面定义的 `ScriptKey`, 就可以访问可编写脚本的类型。使用这个可编写脚本的类型, 就可以把 `ToUpper()` 方法用于将输入改为大写。

```
function callDotnet() {
    var plugin = document.getElementById("plugin");
    var dotnet = plugin.content.ScriptKey;
    var input = document.getElementById("text1").value;
    var output = dotnet.ToUpper(input);
    document.getElementById("text1").value = output;
}
```

如果不打算只使用 Silverlight 建立网站, 而希望混合 ASP.NET 或 HTML 功能, 再用 Silverlight 增强它, 浏览器集成功能就很方便。

38.6 在浏览器外运行的 Silverlight 应用程序

从 Silverlight 3 版本开始, Silverlight 应用程序也可以在浏览器之外运行。该应用程序可以由没有管理员权限的用户安装。但应用程序仍仅限于安全沙箱, 在客户端系统上不能被完全信任。与 WPF 相比, 在浏览器外运行的 Silverlight 应用程序有什么优点? 对于 Silverlight, 无需安装 .NET Framework, 只需安装 Silverlight 运行库, 有必要将该运行库与浏览器插件一起安装。因为这个插件可用于不同的平台, 所以与 WPF 应用程序相比, 在浏览器外运行的应用程序可以运行在更多平台上。

在 Visual Studio 2010 的 Silverlight 项目设置中, 可以单击对应的复选框, 以允许应用程序在浏览器外运行, 并修改在浏览器外面的设置, 如图 38-6 所示。通过这些设置, 可以定义应用程序图标、标题、描述, 以及是否应使用 GPU 加速设置, 该设置可为特定的图形显著提高性能。选中 `Show install menu` 复选框, 就会给用户提供一个安装选项, 允许在 Silverlight 控件上单击鼠标右键。

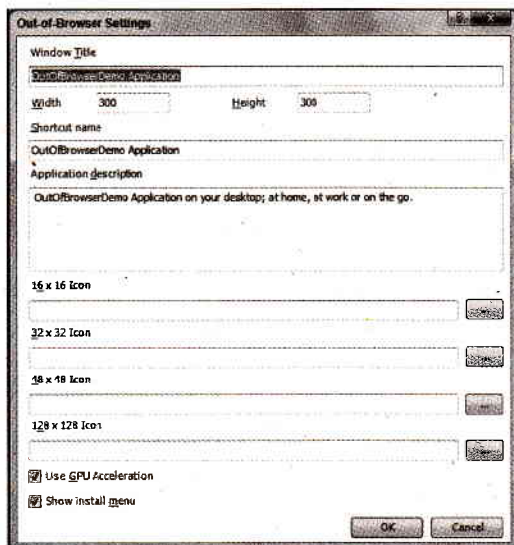


图 38-6

可以控制 Silverlight 应用程序的安装和更新，如下面的项目 OutOfBrowserDemo 所示。该项目的用户界面包含两个按钮和一个 TextBlock 控件。第一个按钮 installButton 关联到一个事件处理程序方法 OnInstall(), 这样用户可以在客户端系统上安装应用程序。按钮 updateButton 关联到事件处理程序 OnUpdate(), 以检查可用的更新。



可从
wrox.com
下载源代码

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Button x:Name="installButton" Grid.Row="0"
        Content="Install Out-of-Browser" Click="OnInstall" Margin="5" />
    <TextBlock x:Name="text1" Grid.Row="1" FontSize="24"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Button x:Name="updateButton" Grid.Row="2" Content="Check for Updates"
        Click="OnUpdate" Margin="5" />
</Grid>
```

代码段 OutOfBrowserDemo/MainPage.xaml

构造函数 MainPage() 检查应用程序是在浏览器之外运行，还是在浏览器内部运行，并据此设置 TextBlock 控件的文本。为了检查应用程序是否在浏览器之外运行，可以使用 Application 类的 IsRunningOutOfBrowser 属性。这个类还通过 InstallState 属性提供了应用程序的安装状态，该属性的值可以是 InstallState 枚举 NotInstalled、Installing、Installed 和 InstallFailed。根据这个信息显示或隐藏 installButton 和 updateButton 控件。



可从
wrox.com
下载源代码

```
public MainPage()
{
    InitializeComponent();
    if (App.Current.IsRunningOutOfBrowser)
    {
```

```

        text1.Text = "running out of browser";
    }
    else
    {
        text1.Text = "running in the browser";
        updateButton.Visibility = Visibility.Collapsed;
    }
    if (App.Current.InstallState == InstallState.Installed)
    {
        installButton.Visibility = Visibility.Collapsed;
    }
}

```

代码段 OutOfBrowserDemo/MainPage.xaml.cs

单击 `installButton` 按钮，就会调用事件处理程序 `OnInstall()`。它调用 `Application` 类的 `Install()` 方法，启动安装过程。

```

private void OnInstall(object sender, RoutedEventArgs e)
{
    if (App.Current.InstallState == InstallState.NotInstalled)
    {
        bool result = App.Current.Install();
        if (result)
            text1.Text = "installation successful";
    }
}

```

在服务器上检查更新也可以通过编程方式用 `Application` 类实现。这里，更新在事件处理程序方法 `OnUpdate()` 中完成。`CheckAndDownloadUpdateAsync()` 检查是否有可用的更新包，并安装该更新包。这是一个使用异步组件模式的异步操作，在更新完成时触发事件 `CheckAndDownloadUpdateCompleted`。在这个事件的处理程序中写入成功或错误信息。成功更新后，下一次启动应用程序时，就可以使用新版本。

```

private void OnUpdate(object sender, RoutedEventArgs e)
{
    App.Current.CheckAndDownloadUpdateCompleted += (sender1, e1) =>
    {
        if (e1.Error != null)
        {
            text1.Text = e1.Error.Message;
        }
        else
        {
            if (e1.UpdateAvailable)
                text1.Text =
                    "Update successful and will be used with the next start";
        }
    };
    App.Current.CheckAndDownloadUpdateAsync();
}

```

第一次在浏览器中运行应用程序时，可以单击 `Install` 按钮，打开如图 38-7 所示的对话框。用户可以选择在“开始”菜单和桌面中包含应用程序的快捷方式。安装后，应用程序就可以在浏览器之

外启动了。



图 38-7

修改了 Silverlight 项目后，就可以单击 Update 按钮，检查更新了。要删除该应用程序，可以单击鼠标右键，打开上下文菜单，从中删除应用程序。

38.7 小结

本章介绍了功能丰富的 Silverlight 架构，它包含 WPF 应用程序的一个功能子集，并根据运行在 Web 上的应用程序所需的功能，包含一些增强功能。

本章讨论了如何创建 Silverlight 项目，它们与 WPF 的区别，如何与浏览器集成，如何在浏览器外运行 Silverlight 应用程序，如何使用联网功能访问服务器上的资源。

下一章介绍用户界面的一个传统技术：Windows 窗体。

第 39 章

Windows 窗体

本章内容:

- Form 类
- Windows 窗体的类层次结构
- System.Windows.Forms 名称空间中的控件和组件
- 菜单和工具栏
- 创建用户控件

基于 Web 的应用程序在过去几年非常流行,正在迅速变成标准。从管理员的角度来看,能够把所有应用程序逻辑驻留在一个中央服务器上非常吸引人。基于 Web 的应用程序的缺点是它们不能提供丰富的用户体验。NET Framework 允许开发人员创建智能的富客户端应用程序,而且不再有部署问题和以前的“DLL Hell”。无论选择 Windows 窗体还是 WPF(参见第 35 章),客户端应用程序都不再难以开发或部署。

Visual Basic 开发人员对 Windows 窗体应比较熟悉。新建窗体(也称为窗口或对话框)也采用把控件从工具箱拖放到窗体设计器上的方式。但是,如果您在创建消息泵和监视消息时使用的是 C 的传统 Windows 编程方式,或者您是一位 MFC 程序员,就会发现现在可以获得需要的低级内部功能了。现在可以重写窗口过程,捕获这些消息,但常常并不是真需要它们。

39.1 创建 Windows 窗体应用程序

在下面的示例中创建一个非常简单的 Windows 窗体应用程序(本章后面的内容使用一个简单的示例应用程序,它在下载的代码 zip 文件的 03 MainExample 文件夹中)。第一个示例没有使用 Visual Studio .NET,而是在文本编辑器中输入代码,使用命令行编译器进行编译。



```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
    public class MyForm: System.Windows.Forms.Form
    {
        public MyForm()
        {
        }
    }
    [STAThread]
```

```
static void Main()
{
    Application.Run(new MyForm());
}
```

代码段 01 MyForm/NotepadForms.cs

在编译和运行这个示例时，会得到一个没有标题的小空白窗体。该应用程序没有什么实际功能，但它列出了最少的实现代码。

代码中有两个地方需要注意。第一个是使用继承功能来创建 MyForm 类。下面的代码声明 MyForm 派生于 System.Windows.Forms.Form 类。

```
public class MyForm: System.Windows.Forms.Form
```

Form 类是 System.Windows.Forms 名称空间的一个主要类。代码的其他部分如下：

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

Main()是 C#客户端应用程序的默认入口。一般在大型应用程序中，Main()方法不位于窗体中，而是位于一个单独的类中。在本例中，我们在 Project Properties 对话框中设置启动类的名称。注意属性[STAThread]，它把 COM 线程模型设置为单线程单元(Single-Threaded Apartment, STA)。COM 交互操作需要 STA 线程模型，默认为添加到 Windows 窗体项目中。

Application.Run()方法负责启动标准的应用程序消息循环。它有 3 个重载版本：第 1 个重载版本不带参数，第 2 个重载版本把 ApplicationContext 对象作为其参数，本例中的第 3 个重载版本把窗体对象作为其参数。在这个示例中，MyForm 对象是应用程序的主窗体，这表示在关闭这个窗体时，应用程序就结束了。使用 ApplicationContext 类，可以对主消息循环何时结束和应用程序何时退出有更多的控制权。

Application 类包含一些非常有用的功能。它提供了一些静态方法和属性，用于控制应用程序的启动和停止过程，访问由应用程序处理的 Windows 消息。表 39-1 列出了其中一些比较有用的方法和属性。

表 39-1

方法/属性	说 明
CommonAppDataPath	对应用程序的所有用户都通用的数据的路径。一般是 BasePath\Company Name\Product Name\Version，其中 BasePath 是 C:\Documents and Settings\ username\ApplicationData。如果该路径不存在，就创建一条路径
ExecutablePath	这是启动应用程序的可执行文件的路径和文件名
LocalUserAppDataPath	类似于 CommonAppDataPath，但这个属性支持漫游
MessageLoop	如果在当前线程上存在消息循环，就返回 True；否则返回 false

(续表)

方法/属性	说 明
StartupPath	类似于 ExecutablePath, 但不返回文件名
AddMessageFilter	用于预处理消息。在基于 IMessageFilter 的对象上实现, 消息可以从消息循环中筛选出来, 或者在消息发送到循环中之前进行特殊的处理
DoEvents	类似于 Visual Basic 的 DoEvents 语句, 允许处理队列中的消息
EnableVisualStyles	允许对应用程序的各种可视化元素使用 XP 可视化样式。它有两个重载版本, 接收清单信息。一个重载版本的参数是清单流, 另一个重载版本的参数是清单所在位置的完整名称和路径
Exit 和 ExitThread	Exit 结束所有当前运行的消息循环, 并退出应用程序。ExitThread 只结束消息循环, 关闭当前线程上的所有窗口

在 Visual Studio 中生成这个示例应用程序时, 它会是什么样子? 首先要注意, 创建了两个文件, 其原因是 Visual Studio 利用了 Framework 的部分类特性, 把设计器生成的所有代码放在一个独立的文件中。使用默认名称 Form1, 这两个文件就是 Form1.cs 和 Form1.Designer.cs。除非选择 Project 菜单中的 Show All Files 选项, 否则在 Solution Explorer 窗口中看不到 Form1.Designer.cs。下面是 Visual Studio 为两个文件生成的代码。第一个文件是 Form1.cs:



可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace VisualStudioForm
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

代码段 Form1.cs

这很简单, 其中包含一些 using 语句和一个简单的构造函数。接着是 Form1.Designer.cs 的代码:



可从
wrox.com
下载源代码

```
namespace VisualStudioForm
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
```

```

    /// </summary>
    private System.ComponentModel.IContainer components = null;
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing"> true if managed resources
    /// should be disposed; otherwise, false. </param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support-do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.components =
            new System.ComponentModel.Container();
        this.AutoScaleMode =
            System.Windows.Forms.AutoScaleMode.Font;
        this.Text = "Form1";
    }
    #endregion
}

```

代码段 Form1.Designer.cs

窗体的设计器文件一般不应直接编辑。唯一的例外是要在 `Dispose()` 方法中进行特殊的处理。`InitializeComponent()` 方法详见本章后面的内容。

首先，从整体上看，这个示例应用程序的代码比简单的命令行示例长。在类的开头有好几条 `using` 语句，在本例中大多数是不必要的。但保留它们并无大碍。`Form1` 类派生自 `System.Windows.Forms.Form`，与前面的 `Notepad` 示例一样，但代码从一开始就不同。`Form1.Designer` 文件的第一行代码如下：

```
private System.ComponentModel.IContainer components = null;
```

在这个示例中，这行代码并没有做什么工作。当给窗体添加组件时，也就把该组件添加到 `Components` 对象中，该组件对象是一个容器。添加到这个容器中的原因与窗体的释放有关。窗体类支持 `IDisposable` 接口，因为它在 `Component` 类中实现。把组件添加到 `Components` 对象中时，容器将确保组件被正确地跟踪，并在释放窗体时释放它。如果查看代码中的 `Dispose()` 方法，就可以看到它：

```

protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
}

```

```
base.Dispose(disposing);
```

在此可以看到，在调用 `Dispose()` 方法时，也会调用 `Components` 对象的 `Dispose()` 方法，因为 `Components` 对象包含其他组件，所以它们也会被释放。

`Form1` 类的构造函数在 `Form1.cs` 中，如下所示：

```
public Form1()
{
    InitializeComponent();
}
```

注意对 `InitializeComponent()` 的调用。`InitializeComponent()` 在 `Form1.Designer.cs` 中，顾名思义，`InitializeComponent()` 初始化已添加到窗体中的所有控件，它还初始化窗体的属性。在本示例中，`InitializeComponent()` 如下所示：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
```

这是很基本的初始化代码。该方法与 Visual Studio 的设计器相关联。使用设计器修改窗体时，这些改动会在 `InitializeComponent()` 中反映出来。如果在 `InitializeComponent()` 中修改了任意类型的代码，下次在设计器中进行修改时，这些改动就会丢失。每次在设计器中进行修改后，`InitializeComponent()` 都会重新生成。如果需要为窗体或窗体上的控件和组件添加其他初始化代码，就应在调用 `InitializeComponent()` 后添加。`InitializeComponent()` 还负责实例化控件，这样在 `InitializeComponent()` 之前所有引用控件的调用都会失败，并抛出一个空引用异常。

要在窗体中添加控件或组件，可以按 `Ctrl+Alt+X` 组合键或者在 Visual Studio 的 `View` 菜单中选择 `Toolbox`。此时 `Form1` 应处于设计模式。在 `Solution Explorer` 窗口中右击 `Form1.cs`，从上下文菜单中选择 `View Designer` 命令。选择 `Button` 控件，把它拖放到设计器的窗体上。也可以双击该控件，把它添加到窗体上。对 `TextBox` 控件进行相同的操作。

既然在窗体中添加了 `TextBox` 控件和 `Button` 控件，`InitializeComponent()` 就会扩展，以包含如下代码：

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(77, 137);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
```

```

        this.button1.UseVisualStyleBackColor = true;
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(67, 75);
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(100, 20);
        this.textBox1.TabIndex = 1;
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(284, 264);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
        this.PerformLayout();
    }

```

如果查看该方法中的前3行代码, 就会看到 **TextBox** 控件和 **Button** 控件被实例化了。注意给控件指定的名称 **textBox1** 和 **button1**。默认情况下, 设计器会使用控件的名称, 并在该名称的最后添加一个整数值。在添加另一个按钮时, 设计器会使用名称 **button2**, 依次类推。通常应改变这些名称, 以满足应用程序的要求。下一行代码是 **SuspendLayout()** 和 **ResumeLayout()** 方法对的部分。**SuspendLayout()** 临时挂起控件第一次初始化时发生的布局事件。在该方法的最后, 将调用 **ResumeLayout()** 方法, 把事件重置为正常状态。这个过程可以用于优化窗体的初始化, 避免在设置窗体时重绘屏幕。在包含许多控件的复杂窗体中, **InitializeComponent()** 方法会非常长。

要修改控件的属性值, 可以按 **F4** 键, 或从 **View** 菜单中选择 **Properties Window** 命令。该窗口允许修改控件或组件的大多数属性。在 **Properties** 窗口中进行了修改后, 就会重写 **InitializeComponent()** 方法, 以反映新属性值。例如, 如果在 **Properties** 窗口中把 **Text** 属性改为 **My Button**, **InitializeComponent()** 就会包含如下代码:

```

//
// button1
//
this.button1.Location = new System.Drawing.Point(77, 137);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button";
* this.button1.UseVisualStyleBackColor = true;

```



如果使用的不是 **Visual Studio .NET** 的编辑器, 就需要在设计中包含 **InitializeComponent()** 类型函数。把所有这些初始化代码放在一个地方, 有助于使构造函数更简洁, 如果有多个构造函数, 则还需要确保能从每个构造函数中调用初始化代码。

类层次结构

在设计和构建自定义控件时,理解层次结构非常重要。如果自定义控件派生自当前控件,例如,对于带有额外属性和方法的文本框,就应使自定义控件继承自文本框控件,再重写、添加属性和方法,以满足要求。但是,如果创建的控件与 .NET Framework 中的已有控件不匹配,就必须从 3 个基类中的一个继承:如果需要自动滚动功能,就从 `Control` 或 `ScrollableControl` 中派生;如果控件应是其他控件的容器,就应从 `ContainerControl` 类中派生。

本章的剩余内容将介绍其中的许多类,它们如何协同工作,以及如何使用它们构建具有专业外观的客户端应用程序。

39.2 Control 类

`System.Windows.Forms` 名称空间中有一个特殊的类,它几乎是每个控件和窗体的基类,这个类就是 `System.Windows.Forms.Control`。`Control` 类实现核心功能,以创建用户所见的界面。`Control` 类派生自 `System.ComponentModel.Component` 类。`Component` 类为 `Control` 类提供了必要的结构,在把控件拖放到设计界面上以及包含在另一个对象时需要它。`Control` 类为派生自它的类提供了一个很长的功能列表。因为这个列表太长,不能在这里全部列出,所以我们仅介绍 `Control` 类提供的比较重要的项。本章后面在介绍基于 `Control` 类的特定控件时,将在一些示例代码中论述属性和方法。下面几小节将按照功能组合方法和属性,把相关的功能放在一起讨论。

39.2.1 大小和位置

控件的大小和位置由属性 `Height`、`Width`、`Top`、`Bottom`、`Left`、`Right` 以及辅助属性 `Size` 和 `Location` 确定。区别是 `Height`、`Width`、`Top`、`Bottom`、`Left`、`Right` 属性值都是一个整数,而 `Size` 的值使用一个 `Size` 结构来表示, `Location` 的值使用一个 `Point` 结构来表示。`Size` 结构和 `Point` 结构都包含 X、Y 坐标。`Point` 结构一般相对于一个位置,而 `Size` 结构是对象的高和宽。`Size` 和 `Point` 都位于 `System.Drawing` 名称空间。它们非常类似,因为它们都提供了 X、Y 坐标对,还拥有用于简单的比较和转换的重写运算符。例如,可以对两个 `Size` 结构执行相加操作。对于 `Point` 结构,已重写了加法运算符,以便把 `Size` 结构添加到 `Point` 结构中,并返回一个新的 `Point`。其结果是给某个位置添加某个距离值,并得到一个新位置。如果动态地创建窗体或控件,这非常方便。

`Bounds` 属性返回一个 `Rectangle` 对象,它表示一个控件的区域。这个区域包含滚动条和标题栏。`Rectangle` 也位于 `System.Drawing` 名称空间。`ClientSize` 属性是一个 `Size` 结构,它表示控件的工作区,不包含滚动条和标题栏。

`PointToClient()` 和 `PointToScreen()` 方法是方便的转换方法,它们接受一个 `Point` 结构,并返回一个 `Point` 结构。`PointToClient()` 接受的 `Point` 参数表示屏幕坐标,该方法把屏幕坐标转换为基于当前客户对象的坐标。这非常便于进行拖放操作。`PointToScreen` 正好与之相反,它接受客户对象的坐标,把它们转换为屏幕坐标。还有 `RectangleToScreen()` 和 `ScreenToRectangle()` 方法,它们具有相同的功能,只是用 `Rectangle` 结构代替 `Point` 结构。

`Dock` 属性确定子控件停放在父控件的哪条边上。`DockStyle` 枚举值用作其属性值。这个值可以是 `Top`、`Bottom`、`Right`、`Left`、`Fill` 或 `None`。`Fill` 会使控件的大小正好匹配父控件的工作区。

Anchor 属性把子控件的一条边锚定到父控件的一条边上,这与停靠不同,因为它不设置距离父控件的一条边,而是把从该条边开始的当前距离设置为常量。例如,如果把子控件的右边界锚定到父控件的右边界上,并重新设置父控件的大小,则子控件的右边界到父控件的右边界的距离将保持不变。Anchor 属性接受 AnchorStyles 枚举的值,其值是 Top、Bottom、Right、Left 和 None。通过设置该属性值,可以在重新设置父控件的大小时,动态地设置子控件的大小。这样,当用户重新设置窗体的大小时,按钮和文本框就不会被剪切或隐藏。

Dock 和 Anchor 属性与 Flow 和 Table 布局控件(详见本章后面的内容)一起使用时,可以创建非常复杂的用户窗口。对于包含许多控件的复杂窗体,窗口大小的重新设置比较困难。这些工具有助于完成这个任务。

39.2.2 外观

与控件外观相关的属性有 BackColor 和 ForeColor,它们把 System.Drawing.Color 对象作为其值。BackColorImage 属性把基于 Image 的对象作为其值。System.Drawing.Image 是一个抽象类,它用作 Bitmap 和 Metafile 类的基类。BackColorImageLayout 属性使用 ImageLayout 枚举设置图像在控件上的显示方式,其有效值是 Center、Tile、Stretch、Zoom 和 None。

Font 和 Text 属性处理已写入文字的显示方式。要修改 Font 属性,需要创建一个 Font 对象。在创建 Font 对象时,要指定字体名称、字号和样式。

39.2.3 用户交互操作

用户交互操作最好描述为控件创建和响应的各种事件。一些比较常见的事件有 Click、DoubleClick、KeyDown、KeyPress、Validating 和 Paint。

鼠标事件 Click、DoubleClick、MouseDown、MouseUp、MouseEnter、MouseLeave 和 MouseHover 处理鼠标和控件的交互操作。如果处理 Click 和 DoubleClick 事件,则每次捕获一个 DoubleClick 事件时,也会引发 Click 事件。如果处理不正确,就会出现我们不希望的结果。Click 和 DoubleClick 事件都把 EventArgs 作为其参数,而 MouseDown 和 MouseUp 事件把 MouseEventArgs 作为其参数。MouseEventArgs 包含几条有用的信息,如单击的按钮、按钮被单击的次数、鼠标轮制动器(鼠标轮上的凹槽)的数目和鼠标的当前 X、Y 坐标。如果可以访问这些信息,就必须处理 MouseDown 或 MouseUp 事件,而不是 Click 或 DoubleClick 事件。

键盘事件的工作方式与此类似:需要一些信息来确定处理什么事件。对于简单的情况,KeyPress 事件接收一个 KeyPressEventArgs,它包含表示被按键的字符值 KeyChar。Handled 属性用于确定事件是否已处理。如果把 Handled 属性设置为 true,事件就不会由操作系统进行默认处理。如果需要被按的键的更多相关信息,则处理 KeyDown 或 KeyUp 事件会比较合适。它们都接收 KeyEventArgs。KeyEventArgs 中的属性包括 Ctrl、Alt 或 Shift 键是否被按下。KeyCode 属性返回一个 Keys 枚举值,标识被按下的键。与 KeyPressEventArgs.KeyChar 属性不同,KeyCode 属性指定键盘上的每个键,而不仅仅是字母数字键。KeyData 属性返回一个 Key 值,还设置修饰符。修饰符与值进行 OR 运算,指定是否同时按 Shift 或 Ctrl 键。KeyValue 属性是 Keys 枚举的整数值。Modifiers 属性包含一个 Keys 值,它表示被按的修饰符键。如果选择了多个修饰符键,这些值就进行 OR 运算。键盘事件以下述顺序来引发:

- (1) KeyDown
- (2) KeyPress

(3) KeyUp

Validating、Validated、Enter、Leave、GotFocus 和 LostFocus 事件都处理获得焦点(或被激活)和失去焦点的控件。在用户用 tab 键选择一个控件或用鼠标选择该控件时,该控件就获得了焦点。Enter、Leave、GotFocus 和 LostFocus 事件的功能似乎非常类似。GotFocus 和 LostFocus 是低级事件,与 Windows 消息 WM_SETFOCUS 和 WM_KILLFOCUS 相关。一般应尽可能使用 Enter 和 Leave 事件。Validating 和 Validated 事件在验证控件时发生。这些事件接收一个 CancelEventArgs,利用该参数,把 Cancel 属性设置为 true,就可以取消以后的事件。如果定制了验证代码,而且验证失败,就可以把 Cancel 属性设置为 true,且控件也不会失去焦点。Validating 事件在验证过程中发生,Validated 事件在验证过程后发生。这些事件的引发顺序如下:

- (1) Enter
- (2) GotFocus
- (3) Leave
- (4) Validating
- (5) Validated
- (6) LostFocus

理解这些事件的引发顺序很重要,可以避免不小心创建递归事件。例如,在控件的 LostFocus 事件中设置控件的焦点,就会创建一个死循环,且用户将无法退出应用程序。

39.2.4 Windows 功能

System.Windows.Forms 名称空间是依赖 Windows 功能的少数几个名称空间之一。Control 类就是一个很好的示例。如果对 System.Windows.Forms.dll 进行反编译,就会看到 UnsafeNativeMethods 类的引用列表。.NET Framework 使用这个类封装许多标准的 Win32 API 调用。使用与 Win32 API 的交互操作,标准 Windows 应用程序的外观就可以通过 System.Windows.Forms 名称空间获得。

支持与 Windows 交互操作的功能包括 Handle 和 IsHandleCreated 属性。Handle 属性返回一个包含控件 HWND(Windows 句柄)的 IntPtr。窗口句柄是唯一标识窗口的一个很难懂的值。因为可以将控件看作是一个窗口,所以它有相应的 HWND。可以使用 Handle 属性进行任意数量的 Win32 API 调用。

为了访问 Windows 消息,可以重写 WndProc()方法。该方法把一个 Message 对象作为其参数。Message 对象是 Windows 消息的一个简单封装器。它包含 HWnd、LParam、WParam、Msg 和 Result 属性。如果希望由系统处理消息,就必须确保把消息传送给 base.WndProc(msg)方法。未把消息传递给 Windows 可能会带来严重的后果,因为不再进行正常的事件处理,应用程序很可能会失败。如果希望自己处理消息,就不需要传递消息。

39.2.5 其他功能

一些条目较难分类,如数据绑定功能。BindingContext 属性返回一个 BindingManagerBase 对象。DataBindings 集合维护一个 ControlBindingsCollection,它是控件的绑定对象集合,本章后面讨论数据绑定。

CompanyName、ProductName 和 Product 版本提供了控件的初始数据及其当前版本。

Invalidate()方法允许使控件的一个区域失效,以进行重新绘制。可以使整个控件失效,或指定要失效的区域或矩形。这会把一条绘制消息发送给控件的 WndProc()方法。还可以同时使任何子控件失效。

组成 Control 类的还有几十个其他属性、方法和事件。这个列表表示比较常用的成员，希望对您对可用的功能有一个大致的了解。

39.3 标准控件和组件

前一节介绍了控件常用的一些方法和属性。本节将讨论 .NET Framework 提供的各种控件，并解释每个控件提供的附加功能。

下面的示例在本书源代码下载的事件 Chapter39Code 示例中。

39.3.1 Button 控件

Button 类表示简单的命令按钮，派生自 ButtonBase 类。该类最常见的用法是编写处理按钮的 Click 事件的代码。下面的代码实现 Click 事件的事件处理程序。在单击按钮时，会弹出一个显示按钮名称的消息框。

```
private void btnTest_Click(object sender, System.EventArgs e)
{
    MessageBox.Show(((Button)sender).Name + " was clicked.");
}
```

在 PerformClick()方法中，可以模仿按钮上的 Click 事件，而实际上无需用户单击按钮，这在测试 UI 时很有用。窗口还有默认按钮，如果用户在该窗口中按回车键，就会自动单击该默认按钮。要把按钮标识为默认按钮，可以把窗体上的 AcceptButton 属性设置为按钮对象。接着，在用户按回车键时，就会引发默认按钮的 Click 事件。图 39-1 显示了标题为 Default 的默认按钮(注意黑色的边框)。还可以把按钮定义为 CancelButton，它表示即使用户按 Escape 键，这个按钮也会接收一个 Click 事件。

按钮可以包含图像和文本。图像通过 ImageList 对象或 Image 属性提供。ImageList 对象就是由放在窗体上的组件管理的一个图像列表。它们在本章后面解释。



图 39-1

Text 和 Image 都包含 Align 属性，以对齐按钮上的文本和图像。Align 属性接受 ContentAlignment 枚举的值。文本或图像可以与按钮的左、右、上、下边界对齐。

39.3.2 CheckBox 控件

CheckBox 控件也派生自 ButtonBase，用于接受来自用户的二状态或三状态响应。如果把 ThreeState 属性设置为 true，复选框的 CheckState 属性就可以是表 39-2 中 3 个 CheckState 枚举值之一：

表 39-2

值	说 明
Checked	复选框有一个选中标记
Unchecked	复选框没有选中标记
Indeterminate	在这种状态下，复选框变成灰色

如果需要告诉用户选项还未设置, 就可以使用 `Indeterminate` 状态。如果希望使用布尔值, 则还可以使用 `Checked` 属性。

`CheckedChanged` 和 `CheckStateChanged` 事件在 `CheckState` 或 `Checked` 属性改变时发生。对于 3 状态的复选框, 需要关联到 `CheckStateChanged` 事件。捕获这些事件对于根据复选框的新状态设置其他值可能很有用。示例包含的下述代码可以显示三状态复选框的状态:



```
Private void threeState_CheckStateChanged(object sender, EventArgs e)
{
    CheckBox cb = sender as CheckBox;
    if (null != cb)
        threeStateState.Text =
            string.Format("State is {0}", cb.CheckState);
}
```

代码段 03 MainExample/ButtonExample.cs

在复选框的状态改变时, 就用新状态更新标签。

39.3.3 RadioButton 控件

最后一个派生自 `ButtonBase` 的控件是 `RadioButton`(单选按钮)。单选按钮一般用作一组, 有时称为选项按钮。单选按钮允许用户从几个选项中选择。当同一个容器中有多个 `RadioButton` 控件时, 一次只能选择一个按钮。所以如果有 3 个选项, 如 `Red`、`Green` 和 `Blue`, 当 `Red` 选项被选中, 而用户单击 `Blue` 选项, 则 `Red` 选项会自动取消选中。

`Appearance` 属性接受 `Appearance` 枚举值, 即 `Button` 或 `Normal`。当选择 `Normal` 时, 单选按钮看起来像一个小圆圈, 在它的旁边有一个标签。选择按钮会填充圆圈, 选择另一个按钮会取消当前选中按钮的选择, 使圆圈为空。当选中 `Button` 时, `RadioButton` 控件看起来像一个标准按钮, 但工作方式类似于开关, 选中是指焦点在位置中, 取消选中是指正常状态或焦点在位置外。

`CheckedAlign` 属性确定圆圈与标签文本的相对位置, 它可以在标签的顶部、左右两边或下方。

只要 `Checked` 属性的值改变, 就会引发 `CheckedChanged` 事件。这样就可以根据控件的新值执行其他动作。

39.3.4 ComboBox 控件、ListBox 控件和 CheckedListBox 控件

`ComboBox`、`ListBox` 和 `CheckedListBox` 都派生自 `ListControl` 类。这个类提供了一些基本的列表管理功能。使用列表控件最重要的事是, 给列表添加数据和从列表中选择数据。使用哪个列表一般取决于列表的用法和列表中的数据类型。如果需要选择多个选项, 或用户需要在任意时刻查看列表中的几个项, 则最好使用 `ListBox` 和 `CheckedListBox`。如果一次只选择一个选项, 就可以使用 `ComboBox`。

在使用列表框之前, 必须先添加数据。为此, 应给 `ListBox.ObjectCollection` 添加对象以添加数据。这个集合由列表的 `Items` 属性提供。由于该集合存储了对象, 因此可以把任意有效的 .NET 类型添加到列表中。要标识列表项, 需要设置两个重要的属性。第一个是 `DisplayMember` 属性, 这个设置告诉 `ListControl`, 在列表中显示对象的哪个属性。另一个是 `ValueMember` 属性, 它是要作为值返回的对象的属性。如果在列表中添加了字符串, 这两个属性就默认使用字符串值。示例应用程序中的 `ListExample` 窗体显示了如何把对象加载到列表框中。该例子使用 `Vendor` 对象作为列表数据。

Vendor 对象只包含两个属性 Name 和 Id。把 DisplayMember 属性设置为 Name 属性, 这告诉列表控件, 把列表中 Name 属性的值显示给用户。

一旦在列表中加载了数据, 就可以使用 SelectedItem 和 SelectedIndex 属性获取数据。SelectedItem 返回当前选中的对象。如果把列表设置为允许选择多个选项, 就不能保证返回选中的选项。此时, 应使用 SelectItems 集合。它包含列表中当前选中的所有选项对应的一个列表。

如果需要特定索引的选项, 就可以使用 Items 属性访问 ListBox.ObjectCollection。因为这是一个标准的.NET 集合类, 所以该集合中项的访问方式与其他集合类的方式访问相同。

如果使用 DataBinding 填充列表, SelectedValue 属性就会返回选中对象设置为 ValueMember 属性的属性值。如果把 Id 设置为 ValueMember, SelectedValue 就从选中的项中返回 Id 值。要使用 ValueMember 和 SelectedValue, 列表必须通过 DataSource 属性来加载。必须先使用对象加载 ArrayList 或任何其他基于 IList 的集合, 再给列表赋予 DataSource 属性。

ComboBox 的 Items 属性返回 ComboBox.ObjectCollection。ComboBox 把编辑控件和列表框合二为一。把一个 DropDownStyle 枚举值传送给 DropDownStyle 属性, 就可以设置 ComboBox 的样式。表 39-3 列出了 DropDownStyle 的各个值。

表 39-3

值	说 明
DropDown	组合框的文本部分是可以编辑的, 用户可以输入值。用户必须单击箭头按钮, 才能显示下拉列表的值
DropDownList	文本部分不能编辑。用户必须从列表中选择
Simple	类似于 DropDown, 但列表总是可见的

如果列表中的值比较宽, 就可以使用 DropDownWidth 属性改变控件下拉部分的宽度。MaxDropDownItems 属性设置在显示列表的下拉部分时的最大项数。

FindString()和 FindStringExact()方法是列表控件的另外两个有用的方法。FindString()在列表中查找以传入字符串开头的第一个字符串。FindStringExact()查找与传入字符串匹配的第一个字符串。它们都返回找到的值的索引, 如果没有找到, 就返回 -1。它们还可以将要搜索的起始索引整数作为参数。

最后, CheckedListBox 控件类似于列表框, 但 DisplayMember 和 ValueMember 属性在 IntelliSense 中不显示, 也不显示在属性网格中。这两个属性可用于实际的控件, 但给它们添加了一个属性, 表示不使用它们。这里不是这样——可以使用 DisplayMember 确定在用户界面上显示什么内容, 或者重写添加到这个列表中的对象的 ToString()方法, 从而在屏幕上显示任意内容。

39.3.5 DataGridView 控件

.NET 的最初版本中的 DataGrid 控件有强大的功能, 但在许多方面, 它都不适用于商业应用程序, 如不能显示图像、下拉控件或锁定列等。因为该控件总是给人只完成了一半的感觉, 所以许多控件供应商都提供了自定义栅格控件, 以克服这些缺陷, 并提供更多功能。

.NET 2.0 引入了另一个栅格控件 DataGridView。它解决了 DataGrid 控件最初的许多问题, 还增加了许多以前只能在插件产品中使用的功能。

因为 DataGridView 控件具有与 DataGrid 类似的绑定功能, 所以它可以绑定到 Array、DataTable、

DataView 或 DataSet 类, 或者绑定到实现 IListSource 或 IList 接口的组件上。DataGridView 控件可以显示相同数据的多种视图。在最简单的情况下, 设置 DataSource 和 DataMember 属性, 就可以显示数据(与在 DataSet 类中一样)。注意, 因为这个控件不是 DataGrid 的插件替代品, 所以其可编程接口完全不同于 DataGrid 的可编程接口。这个控件还提供了更复杂的功能, 本章将讨论这些功能。

本节将使用 Northwind 样本数据库。如果还没有安装它, 就请从 Web 上下载安装程序, 在 SQL Server 实例或 SQL Express 实例中创建一个 Northwind 数据库。Northwind 的安装脚本在 <http://msdn.microsoft.com/enus/library/ms143221.aspx> 上。

1. 显示表格数据

第 20 章介绍了选择数据和把数据读入一个数据表中的各种方式(尽管使用 Console.WriteLine() 方法以非常基本的形式显示数据)。

下面的示例将说明如何检索某些数据, 并在 DataGridView 控件中显示它, 为此, 构建一个新的应用程序 DisplayTabularData, 如图 39-2 所示。

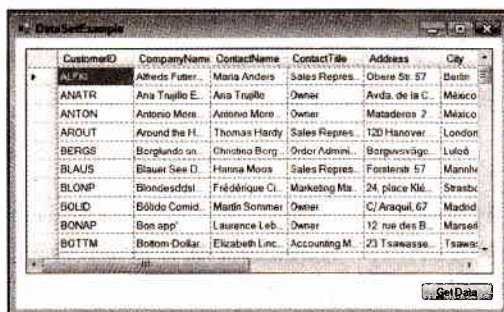


图 39-2

这个简单的应用程序从 Northwind 数据库的 Customer 表中选择每个记录, 在 DataGridView 控件中把它们显示给用户。其代码如下所示(不包含窗体和控件定义代码):

```
using System;
using System.Configuration;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Windows.Forms;

namespace DisplayTabularData
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void getData_Click(object sender, EventArgs e)
        {
            string customers = "SELECT * FROM Customers";

            using (SqlConnection con =
```

```

        new SqlConnection (ConfigurationManager.
            ConnectionStrings["northwind"].ConnectionString))
    {
        DataSet ds = new DataSet();

        SqlDataAdapter da = new SqlDataAdapter(customers, con);

        da.Fill(ds, "Customers");
        dataGridView.AutoGenerateColumns = true;
        dataGridView.DataSource = ds;
        dataGridView.DataMember = "Customers";
    }
}

```

窗体包含 `getData` 按钮，单击它会调用示例代码中的 `getData_Click()` 方法。

这段代码使用 `ConfigurationManager` 类的 `ConnectionStrings` 属性构建一个 `SqlConnection` 对象。之后使用 `DataAdapter` 对象，构建一个数据集，并用数据库表中的表填充它。然后设置 `DataSource` 和 `DataMember` 属性，由 `DataGridView` 控件显示数据。注意把 `AutoGenerateColumns` 属性也设置为 `true`，以确保给用户显示一些数据。如果这个标记没有指定，就需要自己创建所有列。

2. 数据源

`DataGridView` 控件是一种显示数据的非常灵活的方式。除了把 `DataSource` 设置为 `DataSet`，把 `DataMember` 设置为要显示的表名之外，`DataSource` 属性还可以设置为下述任何一个数据源：

- 数组(网格可以绑定到任何一个一维数组上)
- `DataTable`
- `DataView`
- `DataSet` 或 `DataViewManager`
- 实现 `ICollection` 接口的组件
- 实现 `ISource` 接口的组件
- 泛型集合类或派生于泛型集合类的对象

下面几节将给出这些数据源的示例。

3. 显示数组中的数据

初看起来，显示数组中的数据非常简单，创建一个数组，填充一些数据，再在 `DataGridView` 控件上设置 `DataSource` 属性。下面是一些示例代码：

```

string[] stuff = new string[] { "One", "Two", "Three" };
dataGridView.DataSource = stuff;

```

如果数据源包含多个表(如当使用 `DataSet` 或 `DataViewManager` 时)，就需要设置 `DataMember` 属性。

可以用上述数组代码替换上面示例中的 `getData_Click` 事件处理程序，这段代码的结果如图 39-3 所示。

网格显示出了数组中定义的字符串的长度，而不显示这些字符串。原因是在把数组用作

DataGridView 控件的数据源时, 网格会查找数组中对象的第一个公共属性, 并显示这个值, 而不会显示字符串值。因为字符串的第一个(也是唯一的)公共属性是其长度, 所以显示这个长度值。使用 TypeDescriptor 类的 GetProperties() 方法可以获得任意类的属性列表, 该方法返回一个 PropertyDescriptor 对象集合, 接着, 就可以在显示数据时使用它。NET 的 PropertyGrid 控件在显示任意对象时, 使用这个方法。

修正在 DataGridView 中显示字符串时所出现问题的一种方法是创建一个包装器类, 如下所示:

```
protected class Item
{
    public Item(string text)
    {
        _text = text;
    }
    public string Text
    {
        get{return _text;}
    }
    private string _text;
}
```

在数据源数组代码中添加这个 Item 类(对于它进行的各种处理, 它也可以是一个结构)的数组后, 结果如图 39-4 所示。

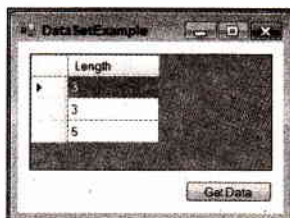


图 39-3

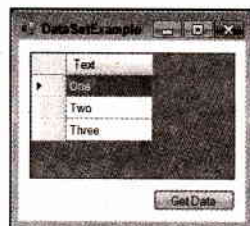


图 39-4

4. DataTable

有两种方式在 DataGridView 控件中显示 DataTable:

- 如果有一个独立的 DataTable, 就把控件的 DataSource 属性设置为这个表。
- 如果在 DataSet 中包含 DataTable, 就把控件的 DataSource 属性设置为数据集, 而且应该把 DataMember 属性设置为数据集中的 DataTable 名。

图 39-5 显示了运行 DataSourceDataTable 示例代码的结果。

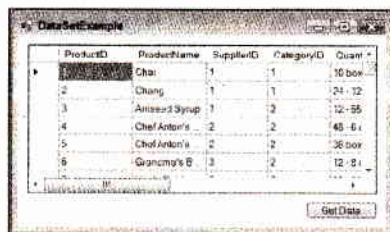


图 39-5

最后一列显示了一个复选框，而不是更常见的编辑控件。`DataGridView` 控件没有显示其他信息，而是从数据源中读取模式(在本例中是 `Products` 表)，根据列的类型推断应显示什么控件。与以前的 `DataGrid` 控件不同，`DataGridView` 控件自身还可以显示图像列、按钮和组合框。

在修改数据网格中的字段时，数据库中的数据不会改变，因为此时数据仅存储在客户端计算机上——没有与数据库建立有效的连接。本章后面将讨论更新数据库中的数据。

5. 显示 `DataGridView` 中的数据

`DataGridView` 提供了一种筛选和排序 `DataTable` 中数据的一种方式。从数据库中选择数据时，用户一般可以单击列标题，对数据排序。此外，还可以筛选数据，只显示某些行，如用户修改过的所有行。可以筛选 `DataGridView` 从而只对用户显示选择的数据行，但不允许筛选 `DataTable` 中的数据列。



`DataGridView` 不允许筛选要显示的数据列，只允许筛选要显示的数据行。

根据现有的 `DataTable` 创建 `DataGridView` 的代码如下所示：

```
DataGridView dv = new DataGridView(dataTable);
```

一旦创建 `DataGridView` 后，就可以进一步改变 `DataGridView` 上的设置，当该视图显示在数据网格中时，这些设置会影响要显示的数据，以及允许对这些数据进行的操作。例如：

- 设置 `AllowEdit = false` 表示在数据行上禁用所有列的编辑功能。
- 设置 `AllowNew = false` 表示禁用新建行功能。
- 设置 `AllowDelete = false` 表示禁用删除行的功能。
- 设置 `RowStateFilter` 只显示指定状态的行。
- 设置 `RowFilter` 可筛选行。

下一节将介绍如何使用 `RowStateFilter` 设置，其他选项都很容易理解。

(1) 通过数据筛选数据行

创建 `DataGridView` 后，就可以通过设置 `RowFilter` 属性，来改变视图显示的数据。这个属性显示为一个字符串，可用作按照给定条件筛选数据的一种方式——该字符串的值就是筛选条件。其语法类似于一般 SQL 中的 `WHERE` 子句，但主要是对已经从数据库中选择出来的数据进行操作。

筛选子句的一些示例如表 39-4 所示。

表 39-4

子 句	说 明
<code>UnitsInStock > 50</code>	只显示 <code>UnitsInStock</code> 列中对应值大于 50 的行
<code>Client = 'Smith'</code>	只返回给定客户的记录
<code>County LIKE 'C*'</code>	返回 <code>County</code> 字段以 C 开头的记录——例如，返回 <code>Corwall</code> 、 <code>Cumbria</code> 、 <code>Cheshire</code> 和 <code>Cambridgeshire</code> 所在的行。可以使用 “%” 字符表示匹配一个字符的通配符，而 “*” 表示匹配 0 个或多个字符的通配符

运行库尽可能在筛选表达式中使用与源列相匹配的数据类型。例如，在前面的示例中，使用 "UnitsInStock > '50'" 表达式完全合法，尽管该列是一个整数列。但如果提供了一个无效的筛选字符串，就会抛出 `EvaluateException` 异常。

(2) 根据状态筛选行

`DataRowView` 中的每一行都有一个定义好的行状态，它们的值如表 39-5 所示，这些状态也可以用于筛选用户查看的行。

表 39-5

DataRowViewState	说 明
Added	列出新建的所有行
CurrentRows	列出除了被删除的行以外的其他行
Deleted	列出最初被选中，且已经删除的行——不显示已经删除的新建行
ModifiedCurrent	列出所有已修改的行，并显示每一列的当前值
ModifiedOriginal	列出所有已修改的行，但显示这些列的初值，而不是当前值
OriginalRows	列出最初从数据源中选中的所有行，不包括新行。显示列的初值(即如果进行了修改，就不显示当前值)
Unchanged	列出完全没有修改的行

图 39-6 显示了两个网格，一个网格显示已添加、删除或修改的行，另一个网格列出表 39-4 中其中一种状态对应的行。

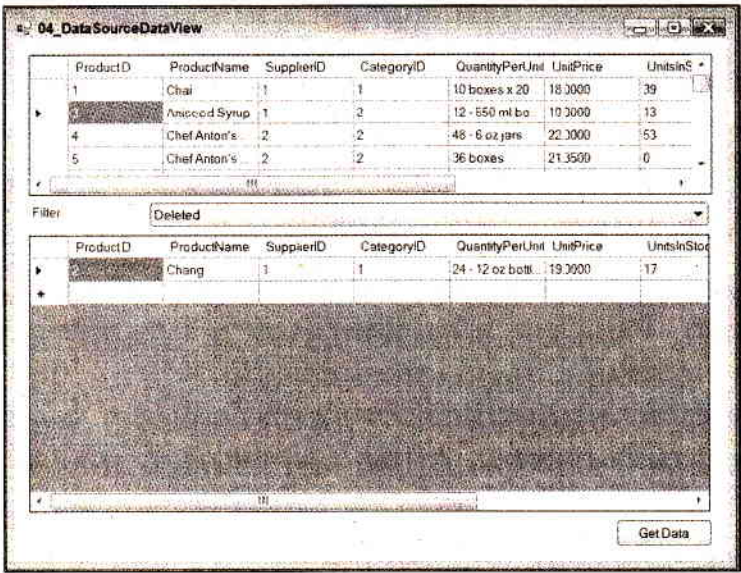


图 39-6

筛选器不仅可以用于可见的行，还可以用于这些行中列的状态。在选择 `ModifiedOriginal` 或 `ModifiedCurrent` 选项时，这很明显。这两个状态都基于 `DataRowVersion` 枚举。例如，如果用户更新了对应行中的一列，该行就会在选择 `ModifiedOriginal` 或 `ModifiedCurrent` 时显示出来；但其实际值

可以从数据库中选择出来的 **Original** 值(如果选择了 **ModifiedOriginal**)，或者 **DataColumn** 中的当前值(如果选择了 **ModifiedCurrent**)。

(3) 对行进行排序

除了筛选数据外，有时还需要对 **DataView** 中的数据进行排序。在 **DataGridView** 控件中单击列标题，就会按照升序或降序的顺序对该列进行排序，如图 39-7 所示。唯一的问题是控件只能对一列进行排序，而底层的 **DataView** 控件可以对多列进行排序。

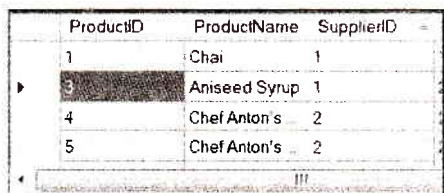
在对列进行排序时，可以单击列的标题(如上面的 **SupplierID** 列所示)，也可以通过代码排序，**DataGrid** 会显示一个箭头位图，表示对哪一列进行排序。

要以编程方式设置排列顺序，可以使用 **DataView** 的 **Sort** 属性：

```
dataView.Sort = "ProductName";
dataView.Sort = "ProductName ASC, ProductID DESC";
```

第一行按照 **ProductName** 列对数据排序，如图 39-7 所示。第二行按照 **ProductName** 列对数据进行升序排序，再按 **ProductID** 降序排序。

DataView 支持对列进行升序或降序排序——默认为升序。如果在代码中对 **DataView** 中的多列进行排序，**DataGridView** 就不会再显示任何排序箭头。



ProductID	ProductName	SupplierID
1	Chai	1
3	Aniseed Syrup	1
4	Chef Anton's	2
5	Chef Anton's	2

图 39-7

因为网格中的每一列都是强类型化的，所以其排序顺序不是基于列的字符串表示，而是基于该列的数据。结果是：如果 **DataGrid** 有一个日期列，要对它进行排序，网格就会按日期来进行排序，而不是按日期的字符串表示方式来进行排序。

6. IListSource 和 IList 接口

DataGridView 还支持任何提供 **IlistSource** 或 **IList** 接口之一的对象。**IlistSource** 接口只有一个方法 **GetList()**，它返回一个 **IList** 接口。而 **IList** 接口比较有趣，它可由运行库中的许多类实现，实现这个接口的其中一些类有 **Array**、**ArrayList** 和 **StringCollection**。

在使用 **IList** 接口时，对集合中对象的相同警告也适用于如前所示的 **Array** 实现——如果使用 **StringCollection** 作为 **DataGrid** 的数据源，网格就会显示字符串的长度，而不是我们希望显示的元素文本。

7. 显示泛型集合

除了已描述的类型之外，**DataGridView** 还可以绑定到泛型集合上。其语法与本章前面描述的示例相同，也是把 **DataSource** 属性设置为集合，控件就会生成相应的显示结果。

同样，所显示的列也基于对象的属性：在 **DataGridView** 中显示所有公共的可读字段。下面的示例显示了一个已定义的列表类：



可从
wrox.com
下载源代码

```
class PersonList: List <Person>
{
}

class Person
{
    public Person( string name, Sex sex, DateTime dob )
    {
        _name = name;
        _sex = sex;
        _dateOfBirth = dob;
    }

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

    public Sex Sex
    {
        get { return _sex; }
        set { _sex = value; }
    }

    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }

    private string _name;
    private Sex _sex;
    private DateTime _dateOfBirth;
}

enum Sex
{
    Male,
    Female
}
```

代码段 03 MainExample/Person.cs

这段代码显示了 Person 类的几个实例，它们在 PersonList 类中构建，如图 39-8 所示。

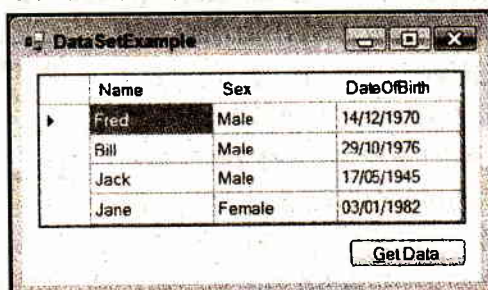


图 39-8