

该用户控件为温度转换提供了一个简单的实现方式。TempConversionType 枚举定义了这个控件可能进行的不同转换。在构造函数中设置用户控件的 DataContext 属性，使枚举值显示在两个组合框中。ToCelsiusFrom()方法把参数 t 从其原始值转换为摄氏值。温度的源类型用第二个参数 TempConversionType 定义。FromCelsiusTo()方法把摄氏值转换为所选的温度值。OnCalculate()方法是 Button.Click 事件的处理程序，它调用 ToCelsiusFrom()和 FromCelsiusTo()方法，根据用户选择的转换类型，执行转换。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.MEF
{
    public enum TempConversionType
    {
        Celsius,
        Fahrenheit,
        Kelvin
    }

    public partial class TemperatureConversion : UserControl
    {
        public TemperatureConversion()
        {
            InitializeComponent();
            this.DataContext = Enum.GetNames(typeof(TempConversionType));
        }

        private double ToCelsiusFrom(double t, TempConversionType conv)
        {
            switch (conv)
            {
                case TempConversionType.Celsius:
                    return t;
                case TempConversionType.Fahrenheit:
                    return (t - 32) / 1.8;
                case TempConversionType.Kelvin:
                    return (t - 273.15);
                default:
                    throw new ArgumentException("invalid enumeration value");
            }
        }

        private double FromCelsiusTo(double t, TempConversionType conv)
        {
            switch (conv)
            {
                case TempConversionType.Celsius:
                    return t;
                case TempConversionType.Fahrenheit:
                    return (t * 1.8) + 32;
                case TempConversionType.Kelvin:
                    return t + 273.15;
                default:
                    return t;
            }
        }
    }
}
```

```

        throw new ArgumentException("invalid enumeration value");
    }
}

private void OnCalculate(object sender, System.Windows.RoutedEventArgs e)
{
    try
    {
        TempConversionType from;
        TempConversionType to;
        if (Enum.TryParse<TempConversionType>(
            (string)comboFrom.SelectedValue, out from) &&
            Enum.TryParse<TempConversionType>(
                (string)comboTo.SelectedValue, out to))
        {
            double result = FromCelsiusTo(
                ToCelsiusFrom(double.Parse(textInput.Text), from), to);
            textOutput.Text = result.ToString();
        }
    }
    catch (FormatException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
}
}

```

代码段 TemperatureConversion/TemperatureConversion.xaml.cs

到目前为止，这个控件仅是一个简单的 WPF 用户控件。为了创建 MEF 部件，要使用 Export 特性导出 TemperatureCalculatorExtension 类。这个类实现 ICalculatorExtension 接口，同时提供 Title 和 Description 信息，从 GetUI() 方法中返回用户控件 TemperatureConversion。



可从
wrox.com
下载源代码

```

using System.ComponentModel.Composition;
using System.Windows;

namespace Wrox.ProCSharp.MEF
{
    [Export(typeof(ICalculatorExtension))]
    public class TemperatureCalculatorExtension : ICalculatorExtension
    {
        public string Title
        {
            get { return "Temperature Conversion"; }
        }

        public string Description
        {
            get { return "Convert Celsius to Fahrenheit and " +
                "Fahrenheit to Celsius"; }
        }

        public FrameworkElement GetUI()
        {
            return new TemperatureConversion();
        }
    }
}

```

另一个实现了 `ICalculatorExtension` 接口的用户控件是 `FuelEconomy`。通过这个控件，可以计算每加仑行驶的英里数或每行驶 100 千米的升数。



可从
wrox.com
下载源代码

```
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.MEF
{
    public partial class FuelEconomyUC : UserControl
    {
        private List<FuelEconomyType> fuelEcoTypes;

        public FuelEconomyUC()
        {
            InitializeComponent();
            InitializeFuelEcoTypes();
        }

        private void InitializeFuelEcoTypes()
        {
            var t1 = new FuelEconomyType
            {
                Id = "lpk",
                Text = "L/100 km",
                DistanceText = "Distance (kilometers)",
                FuelText = "Fuel used (liters)"
            };
            var t2 = new FuelEconomyType
            {
                Id = "mpg",
                Text = "Miles per gallon",
                DistanceText = "Distance (miles)",
                FuelText = "Fuel used (gallons)"
            };
            fuelEcoTypes = new List<FuelEconomyType>() { t1, t2 };
            this.DataContext = fuelEcoTypes;
        }

        private void OnCalculate(object sender, RoutedEventArgs e)
        {
            double fuel = double.Parse(textFuel.Text);
            double distance = double.Parse(textDistance.Text);
            FuelEconomyType ecoType = comboFuelEco.SelectedItem as FuelEconomyType;
            double result = 0;
            switch (ecoType.Id)
            {
                case "lpk":
                    result = fuel / (distance / 100);
                    break;
            }
        }
    }
}
```

```

        case "mpg":
            result = distance / fuel;
            break;
        default:
            break;
    }
    this.textResult.Text = result.ToString();
}
}
}

```

代码段 FuelEconomy/FuelEconomyUC.xaml.cs

还使用 Export 特性实现 ICalculatorExtension 接口并导出。



```

namespace Wrox.ProCSharp.MEF
{
    [Export(typeof(ICalculatorExtension))]
    public class FuelCalculatorExtension : ICalculatorExtension
    {
        public string Title
        {
            get { return "Fuel Economy"; }
        }

        public string Description
        {
            get { return "Calculate fuel economy"; }
        }

        public FrameworkElement GetUI()
        {
            return new FuelEconomyUC();
        }
    }
}

```

代码段 FuelEconomy/FuelCalculatorExtension.cs

在继续 WPD 计算器例子，导入用户控件之前，先看看导出的其他选项。不仅可以导出整个类型，还可以导出属性和方法，也可以给出口添加元数据信息。

28.3.1 导出属性和方法

除了导出整个类，包括属性、方法和事件之外，还可以仅导出属性或方法。给属性添加 Export 特性就可以导出属性，以使用不在自己控制之下的类(如 .NET Framework 或第三方库中的类)。为此，只需定义这个特定类型的属性，并导出该属性。

导出方法允许进行比类型更精细的控制。调用者不需要了解类型。方法通过委托来导出。下面的代码段用 Export 特性定义了 Add() 和 Subtract() 方法。导出的类型是委托 Func<double, double, double>，这个委托接收两个 double 参数，返回类型是 double。对于没有返回类型的方法，可以使用委托 Action<T>。



Func<T>和 Action<T>委托参见第 8 章。



可从
wrox.com
下载源代码

```
using System;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
    public class Operations
    {
        [Export("Add", typeof(Func<double, double, double>))]
        public double Add(double x, double y)
        {
            return x + y;
        }

        [Export("Subtract", typeof(Func<double, double, double>))]
        public double Subtract(double x, double y)
        {
            return x - y;
        }
    }
}
```

代码段 Operations/Operations.cs

导出的方法从 SimpleCalculator 插件中导入。部件本身可以使用其他部件。要使用导出的方法，需要用 Import 特性声明委托。这个特性与用 Export 特性声明的委托类型同名。



SimpleCalculator 本身是一个导出了 ICalculator 接口的部件，它由导入的部件组成。



可从
wrox.com
下载源代码

```
[Export(typeof(ICalculator))]
public class Calculator : ICalculator
{
    [Import("Add", typeof(Func<double, double, double>))]
    public Func<double, double, double> Add { get; set; }
    [Import("Subtract", typeof(Func<double, double, double>))]
    public Func<double, double, double> Subtract { get; set; }
}
```

代码段 SimpleCalculator/Calculator.cs

导入的方法用 Add 和 Subtract 委托表示，它们在 Operate()方法中通过这两个委托来调用。



可从
wrox.com
下载源代码

```
public double Operate(IOperation operation, double[] operands)
{
    double result = 0;
    switch (operation.Name)
    {
        case "+":
            result = Add(operands[0], operands[1]);
            break;
    }
}
```

```

        case "-":
            result = Subtract(operands[0], operands[1]);
            break;
        case "/":
            result = operands[0] / operands[1];
            break;
        case "*":
            result = operands[0] * operands[1];
            break;
        default:
            throw new InvalidOperationException(
                String.Format("invalid operation {0}", operation.Name));
    }
    return result;
}

```

代码段 SimpleCalculator/Calculator.cs

28.3.2 导出元数据

利用导出功能，还可以附加元数据信息。元数据允许添加除了名称和类型之外的其他信息。这些信息可用于添加功能信息，确定在入口端应使用哪些出口。

导出的 Add() 方法现在改为用 ExportMetadata 特性添加速度功能：



可从
wrox.com
下载源代码

```

[Export("Add", typeof(Func<double, double, double>))]
[ExportMetadata("speed", "fast")]
public double Add(double x, double y)
{
    return x + y;
}

```

代码段 Operations/Operation.cs

为了包含从 Add() 方法的另一种实现方式中选择的选项，实现了另一个方法，它具有不同的速度性能，但委托的类型和名称相同：



可从
wrox.com
下载源代码

```

public class Operations2
{
    [Export("Add", typeof(Func<double, double, double>))]
    [ExportMetadata("speed", "slow")]
    public double Add(double x, double y)
    {
        Thread.Sleep(3000);
        return x + y;
    }
}

```

代码段 Operations/Operation2.cs

因为有多多个导出的 Add() 方法，所以必须改变入口的定义。如果有多个出口有相同的名称和类型，就可以使用 ImportMany 特性。这个特性应用于数组或 IEnumerable<T> 接口。ImportMany 在 28.4 节详细解释。为了访问元数据，可以使用一个 Lazy<T, TMetadata> 数组。Lazy<T> 类是 .NET 4 中新增的，用于支持类型在第一次使用时的懒惰初始化。Lazy<T, TMetadata> 派生自 Lazy<T>，除了

基类支持的成员之外,它还支持通过 `Metadata` 特性访问元数据信息。在例子中,方法通过 `Func<double, double, double>` 委托来引用,它是 `Lazy<T, TMetadata>` 的第一个泛型参数。第二个泛型参数是用于收集元数据的 `IDictionary<string, object>`。`ExportMetadata` 特性可以多次使用,以添加多个功能,且总是包含字符串类型的键和对象类型的值。



可从
wrox.com
下载源代码

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, IDictionary<string, object>>[]
    AddMethods { get; set; }
//[Import("Add", typeof(Func<double, double, double>))]
//public Func<double, double, double>Add { get; set; }
```

代码段 SimpleCalculator/Calculator.cs

对 `Add()` 方法的调用现在改为遍历 `Lazy<Func<double, double, double>, IDictionary<string, object>>` 元素集合。通过 `Metadata` 属性,检查功能的键;如果速度性能的值是 `fast`,就使用 `Lazy<T>` 的 `Value` 属性调用操作,以获取委托。

```
case "+":
    // result = Add(operands[0], operands[1]);
    foreach (var addMethod in AddMethods)
    {
        if (addMethod.Metadata.ContainsKey("speed") &&
            (string)addMethod.Metadata["speed"] == "fast")
            result = addMethod.Value(operands[0], operands[1]);
    }
    // result = operands[0] + operands[1];
    break;
```

除了使用 `ExportMetadata` 特性之外,还可以创建一个派生自 `ExportAttribute` 的自定义导出特性类。`SpeedExportAttribute` 类定义了一个 `Speed` 类型的额外 `Speed` 属性:



可从
wrox.com
下载源代码

```
using System;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
    public enum Speed
    {
        Fast,
        Slow
    }

    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]
    public class SpeedExportAttribute : ExportAttribute
    {
        public SpeedExportAttribute(string contractName, Type contractType)
            : base(contractName, contractType) { }

        public Speed Speed { get; set; }
    }
}
```

代码段 CalculatorUtils/ExportAttribute.cs



如何创建自定义特性的更多内容可创建第 14 章。

有了导出的 Add() 方法, 现在可以使用 SpeedExport 特性替代 Export 和 ExportMetadata 特性:



可从
wrox.com
下载源代码

```
[SpeedExport("Add", typeof(Func<double, double, double>), Speed=Speed.Fast)]
public double Add(double x, double y)
{
    return x + y;
}
```

代码段 Operations/Operations.cs

对于入口, 需要一个包含所有元数据的接口。这样才能访问强类型化的功能。通过 SpeedExport 特性定义的性能仅是速度。ISpeedCapabilities 接口使用定义 SpeedExport 特性时使用的枚举类型 Speed, 来定义 Speed 属性:



可从
wrox.com
下载源代码

```
namespace Wrox.ProCSharp.MEF
{
    public interface ISpeedCapabilities
    {
        Speed Speed { get; }
    }
}
```

代码段 CalculatorUtils/ISpeedCapabilities.cs

现在可以使用 ISpeedCapabilities 接口替代前面定义的字典, 修改入口的定义:



可从
wrox.com
下载源代码

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, ISpeedCapabilities>[]
    AddMethods { get; set; }
```

代码段 SimpleCalculator/Calculator.cs

通过入口, 现在可以直接使用 ISpeedCapabilities 接口的 Speed 属性:

```
foreach (var addMethod in AddMethods)
{
    if ( addMethod.Metadata.Speed == Speed.Fast )
        result = addMethod.Value(operands[0], operands[1]);
}
```

28.4 导入

下面使用 WPF 用户控件和 WPF 宿主应用程序。宿主应用程序的设计视图如图 28-3 所示。WPFCalculator 应用程序是一个 WPF 应用程序, 它会加载功能完备的计算器插件(它实现 ICalculator 和 Ioperation 接口), 以及带用户界面的插件(它实现 ICalculatorExtension 接口)。为了连接到部件的出口上, 需要入口。

入口连接到出口上。使用导出的部件时, 需要一个入口来建立连接。通过 Import 特性, 可以连

接到一个出口上。如果应加载多个插件,就需要 `ImportMany` 特性,并需要把它定义一个数组类型或 `IEnumerable<T>`。因为宿主计算器应用程序允许加载实现了 `ICalculatorExtension` 接口的多个计算器扩展,所以 `CalculatorExtensionImport` 类定义了 `IEnumerable<ICalculatorExtension>` 类型的 `CalculatorExtension` 属性,以访问所有的计算器扩展部件。

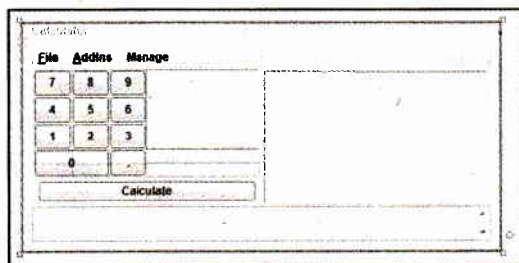


图 28-3



可从
wrox.com
下载源代码

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

namespace Wrox.ProCSharp.MEF
{
    public class CalculatorExtensionImport
    {
        [ImportMany(AllowRecomposition=true)]
        public IEnumerable< ICalculatorExtension > CalculatorExtensions { get; set; }
    }
}
```

代码段 WPFCalculator/CalculatorExtensionImport.cs

`Import` 和 `ImportMany` 特性允许使用 `ContractName` 和 `ContractType` 把入口映射到出口上。可以用这两个特性设置的其他属性有 `AllowRecomposition` 和 `RequiredCreationPolicy`。`AllowRecomposition` 属性允许在应用程序运行期间动态映射到新出口上,还允许上传出口。`RequiredCreationPolicy` 属性允许选择在请求器之间共享部件(`CreationPolicy.Shared`)、不共享部件(`CreationPolicy.NonShared`),或者是否由容器定义策略(`CreationPolicy.Any`)。

为了查看所有的导入是否成功,可以实现 `IPartImportsSatisfiedNotification` 接口。这个接口只定义了一个方法 `OnImportsSatisfied()`,在类的所有导入都成功时调用它。在 `CalculatorImport` 类中,该方法触发 `ImportsSatisfied` 事件:



可从
wrox.com
下载源代码

```
using System;
using System.ComponentModel.Composition;
using System.Windows.Controls;

namespace Wrox.ProCSharp.MEF
{
    public class CalculatorImport : IPartImportsSatisfiedNotification
    {
        public event EventHandler<ImportEventArgs> ImportsSatisfied;

        [Import(typeof(ICalculator))]
        public ICalculator Calculator { get; set; }

        public void OnImportsSatisfied()
        {
            if (ImportsSatisfied != null)
                ImportsSatisfied(this, new ImportEventArgs {
                    StatusMessage = "ICalculator import successful" });
        }
    }
}
```

代码段 WPFCalculator/CalculatorImport.cs

在创建 CalculatorImport 类时, CalculatorImport 类的事件连接到一个事件处理程序上, 把一条消息写入 TextBlock 中, 以显示状态信息:



可从
wrox.com
下载源代码

```
private void InitializeContainer()
{
    var catalog = new DirectoryCatalog(
        Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
    calcImport = new CalculatorImport();
    calcImport.ImportsSatisfied += (sender, e) =>
    {
        textStatus.Text += String.Format("{0}\n", e.StatusMessage);
    };
    container.ComposeParts(calcImport);
    InitializeOperations();
}
```

代码段 WPFCalculator/MainWindow.xaml.cs

部件的懒惰加载

部件默认从容器中加载, 例如, 调用 CompositionContainer 上的扩展方法 ComposeParts() 来加载。利用 Lazy<T> 类, 部件可以在第一次访问时加载。Lazy<T> 类型允许后期实例化任意类型 T, 并定义 IsValueCreated 和 Value 属性。IsValueCreated 属性是一个布尔值, 它返回包含的类型 T 是否已经实例化的信息。Value 属性在第一次访问包含的类型 T 时初始化它, 并返回 T 的实例。

插件的入口可以声明为类型 Lazy<T> 类型, 如示例 Lazy<ICalculator> 所示:



可从
wrox.com
下载源代码

```
[Import(typeof(ICalculator))]
public Lazy<ICalculator> Calculator { get; set; }
```

代码段 WPFCalculator/CalculatorImport.cs

调用导入的属性也需要对访问 Lazy<T> 类型的 Value 属性进行一些修改。calcImport 是一个 CalculatorImport 类型的变量, Calculator 属性返回 Lazy<ICalculator>。Value 属性以懒惰的方式实例化导入的类型, 并返回 ICalculator 接口, 在该接口中, 现在可以调用 GetOperations() 方法, 从计算器插件中获得所有支持的操作。



可从
wrox.com
下载源代码

```
private void InitializeOperations()
{
    Contract.Requires(calcImport != null);
    Contract.Requires(calcImport.Calculator != null);

    var operators = calcImport.Calculator.Value.GetOperations();
    foreach (var op in operators)
    {
        var b = new Button();
        b.Width = 40;
        b.Height = 30;
```

```

b.Content = op.Name;
b.Margin = new Thickness(2);
b.Padding = new Thickness(4);
b.Tag = op;
b.Click += new RoutedEventHandler(DefineOperation);
listOperators.Items.Add(b);
}

```

代码段 WPF Calculator/MainWindow.xaml.cs

28.5 容器和出口提供程序

部件通过容器来导入。宿主部件的类型在 `System.ComponentModel.Composition.Hosting` 名称空间中定义。`CompositionContainer` 类是部件的容器。在这个类的构造函数中，可以指定多个 `ExportProvider` 对象，以及一个 `ComposablePartCatalog`。类别是部件的源，参见 28.6 节。出口提供程序允许用重载的 `GetExport<T>()` 方法以编程方式访问所有出口。出口提供程序用于访问类别，`CompositionContainer` 类本身就是一个出口提供程序。因此可以在其他容器中嵌套容器。

调用 `Compose()` 方法时，会加载部件(假定它们不是懒惰加载)。前面一直在使用 `ComposePart()` 方法，如下面的 `InitializeContainer()` 方法所示：



可从
wrox.com
下载源代码

```

private void InitializeContainer()
{
    catalog = new DirectoryCatalog(
        Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
    calcImport = new CalculatorImport();
    calcImport.ImportsSatisfied += (sender, e) =>
    {
        textStatus.Text += String.Format("{0}\n", e.StatusMessage);
    };
    container.ComposeParts(calcImport);

    InitializeOperations();
}

```

代码段 WPF Calculator/MainWindow.xaml.cs

`ComposePart()` 是一个用 `AttributeModelServices` 类定义的扩展方法。这个类提供的方法使用特性和 .NET 反射访问部件信息，并把部件添加到容器中。除了使用这个扩展方法之外，还可以使用 `CompositionContainer` 类的 `Compose()` 方法。`Compose()` 方法与 `CompositionBatch` 类一起工作。`CompositionBatch` 类可以用于定义应在容器中添加哪些部件，删除哪些部件。`AddPart()` 和 `RemovePart()` 方法有重载版本，可以用于添加有 `Import` 属性的部件(`calcImport` 是 `CalculatorImport` 类的一个实例，包含 `Import` 特性)，或者派生自 `ComposablePart` 基类的部件。

```

var batch = new CompositionBatch();
batch.AddPart(calcImport);
container.Compose(batch);

```

宿主应用程序 WPFCalculator 加载了两种不同的部件。第一个部件实现了 ICalculator 接口, 在前面所示的 InitializeContainer() 方法中加载。实现了 ICalculatorExtension 接口的部件在 RefreshExtension() 方法中加载。Container.ComposeParts 从 CalculatorExtensionImport 类中获取入口, 并把它连接到类别中所有可用的出口上。因为这里动态加载了多个插件(用户可以选择要使用的插件), 所以 Menu 控件会动态扩展。应用程序遍历所有的计算器扩展, 并为每个扩展新建一个 MenuItem 控件, MenuItem 控件的标题包含一个设置了扩展的 Title 属性的 Label 和一个 CheckBox。CheckBox 以后用于动态删除出口。把 MenuItem 控件的 ToolTip 属性设置为扩展控件的 Description 属性, 把 Tag 属性设置为扩展控件本身, 以便于在选择菜单项时获得扩展控件。把 MenuItem 控件的 Click 事件设置为 ShowAddIn() 方法, 这个方法调用扩展控件的 GetUI() 方法, 在 TabControl 的一个新 TabItem 中显示控件。



可从
wrox.com
下载源代码

```
private void RefreshExtensions()
{
    catalog.Refresh();
    calcExtensionImport = new CalculatorExtensionImport();
    calcExtensionImport.ImportsSatisfied += (sender, e) =>
    {
        this.textStatus.Text += String.Format("{0}\n", e.StatusMessage);
    };

    container.ComposeParts(calcExtensionImport);
    menuAddins.Items.Clear();
    foreach (var extension in calcExtensionImport.CalculatorExtensions)
    {
        var menuItemHeader = new StackPanel { Orientation = Orientation.Horizontal };
        menuItemHeader.Children.Add(new Label { Content = extension.Title });
        var menuCheck = new CheckBox { IsChecked = true };
        menuItemHeader.Children.Add(menuCheck);

        var menuItem = new MenuItem {
            Header = menuItemHeader,
            ToolTip = extension.Description,
            Tag = extension
        };

        menuCheck.Tag = menuItem;
        menuItem.Click += ShowAddIn;
        menuAddins.Items.Add(menuItem);
    }
}

private void ShowAddIn(object sender, RoutedEventArgs e)
{
    var mi = e.Source as MenuItem;
    var ext = mi.Tag as ICalculatorExtension;
    FrameworkElement uiControl = ext.GetUI();
    var headerPanel = new StackPanel { Orientation = Orientation.Horizontal };
    headerPanel.Children.Add(new Label { Content = ext.Title });
    var closeButton = new Button { Content = "X" };
    var ti = new TabItem { Header = headerPanel, Content = uiControl };
    closeButton.Click += delegate
    {
        tabExtensions.Items.Remove(ti);
    };
}
```

```
headerPanel.Children.Add(closeButton);

tabExtensions.SelectedIndex = tabExtensions.Items.Add(ti);
}
```

代码段 WPFCalculator/MainWindow.xaml.cs

为了动态删除出口，在 MenuItem 标题的 CheckBox 控件中设置了 Unchecked 事件。IcalculatorExtension 接口可以从 MenuItem 控件的 Tag 属性中访问，因为它在前面的代码段中设置。要从容器中删除部件，必须创建一个 CompositeBatch 对象，其中包含了要删除的部件。CompositeBatch 类的 RemovePart()方法需要一个 ComposablePart 对象，而带特性的对象没有这个方法的重载版本。利用带特性的对象的 AttributeModelServices 类，可以使用 CreatePart()方法创建一个 ComposablePart。接着把 CompositeBatch 传递给 CompositionContainer 类的 Compose()方法，以从中删除部件：

```
menuCheck.Unchecked += (sender1, e1) =>
{
    MenuItem mi = (sender1 as CheckBox).Tag as MenuItem;
    ICalculatorExtension ext = mi.Tag as ICalculatorExtension;
    ComposablePart part =
        AttributedModelServices.CreatePart(ext);
    var batch = new CompositionBatch();
    batch.RemovePart(part);
    container.Compose(batch);
    MenuItem parentMenu = mi.Parent as MenuItem;
    parentMenu.Items.Remove(mi);
};
```



只有把 Import 特性的 AllowRecomposition 属性设置为 true，才能删除部件。

通过出口提供程序，可以实现 ExportsChanged 事件的处理程序，来获得所添加和删除的出口的相关信息。ExportsChangedEventArgs 类型的参数 e 包含所添加和删除的出口列表，这个列表会写入一个 TextBlock 控件：

```
var container = new CompositionContainer(catalog);
container.ExportsChanged += (sender, e) =>
{
    var sb = new StringBuilder();

    foreach (var item in e.AddedExports)
    {
        sb.AppendFormat("added export {0}\n", item.ContractName);
    }
    foreach (var item in e.RemovedExports)
    {
        sb.AppendFormat("removed export {0}\n", item.ContractName);
    }
    this.textStatus.Text += sb.ToString();
};
```

28.6 类别

类别定义了 MEF 搜索所请求的部件的位置。示例应用程序使用 `DirectoryCatalog` 从指定的目录中加载包含部件的程序集。利用 `DirectoryCatalog`，可以通过 `Changed` 事件获得变更信息，并遍历所有已添加和删除的定义。`DirectoryCatalog` 本身没有注册文件系统的变更。相反，必须调用 `DirectoryCatalog` 的 `Refresh()` 方法，如果自从上一次读取目录以来发生了变化，就会触发 `Changing` 和 `Changed` 事件。



可从
wrox.com
下载源代码

```
private void InitializeContainer()
{
    catalog = new DirectoryCatalog(Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
    catalog.Changed += (sender, e) =>
    {
        var sb = new StringBuilder();

        foreach (var definition in e.AddedDefinitions)
        {
            foreach (var metadata in definition.Metadata)
            {
                sb.AppendFormat(
                    "added definition with metadata - key: {0}, " +
                    "value: {1}\n", metadata.Key, metadata.Value);
            }
        }

        foreach (var definition in e.RemovedDefinitions)
        {
            foreach (var metadata in definition.Metadata)
            {
                sb.AppendFormat(
                    "removed definition with metadata - key: {0}, " +
                    "value: {1}\n", metadata.Key, metadata.Value);
            }
        }

        this.textStatus.Text += sb.ToString();
    };
    //...
```

代码段 WPFCalculator/MainWindow.xaml.cs



要获取加载到目录中的新插件的即刻通知，可以使用 `System.IO.FileSystemWatcher`，注册插件目录上的变化，再通过 `FileSystemWatcher` 的 `Changed` 事件调用 `DirectoryCatalog` 的 `Refresh()` 方法。

`CompositionContainer` 类只需一个 `ComposablePartCatalog` 就可以查找部件。`DirectoryCatalog` 派生自 `ComposablePartCatalog`。其他类别有 `AssemblyCatalog`、`TypeCatalog` 和 `AggregateCatalog`。下面比较这些类别：

- DirectoryCatalog 在目录中搜索部件。
- AssemblyCatalog 在引用的程序集中直接搜索部件。与 DirectoryCatalog 相反，目录中的程序集可能在运行期间发生变化，而 AssemblyCatalog 是不变的，部件不能改变。
- TypeCatalog 在类型列表中搜索入口。Enumerable<Type>可以传递给这个类别的构造函数。
- AggregateCatalog 是类别的类别。这个类别可以从多个 ComposablePartCatalog 对象中创建，并搜索上述所有类别。例如，可以创建一个 AssemblyCatalog，在程序集中搜索入口，再创建两个 DirectoryCatalog 对象，以搜索两个不同的目录，最后创建一个 AssemblyCatalog，合并入口搜索的 3 个类别。

运行示例应用程序时(如图 28-4 所示)，会加载 SimpleCalculator 插件，此时可以执行插件支持的一些计算操作。从 AddIns 菜单中，可以启动实现了 ICalculatorExtender 接口的插件，在 tab 控件中查看这些插件的用户界面。出口信息、目录类别中的变化，以及状态信息显示在底部，还可以从插件目录中删除 ICalculatorExtender 插件(应用程序没有运行)，在应用程序运行期间把插件复制到目录中，在运行期间刷新插件，以查看新插件。

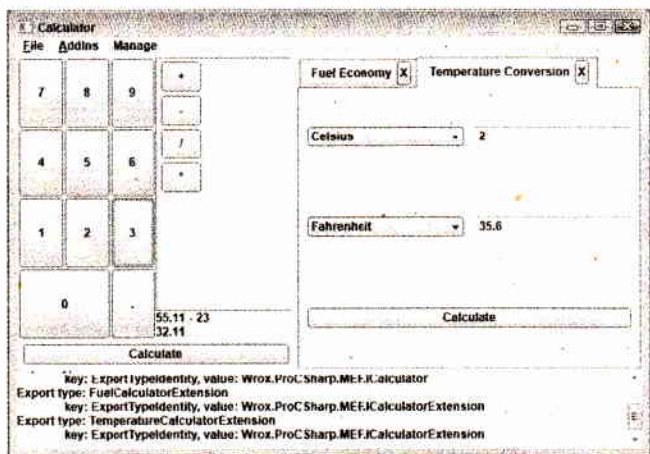


图 28-4

28.7 小结

本章介绍了.NET 4 中的一种新技术 MEF。MEF 实现方式使用特性查找和连接插件，还讨论了如何创建和加载插件，以及通过出口使用接口或方法的可能性。学习了如何使用类别动态地查找插件，如何使用容器把入口映射到出口上。

光盘中第 50 章的相关内容讨论了 MAF，它通过独立的应用程序域和独立的进程，提供了宿主与插件之间更好的分隔方式，但其开发成本也比较高。

第 29 章介绍文件、目录和注册表的读写。

第 29 章

文件和注册表操作

本章内容:

- 介绍目录结构
- 移动、复制、删除文件和文件夹
- 读写文本文件
- 读写注册表键
- 读写独立存储器

本章将介绍如何在 C# 中执行读写文件和系统注册表的任务。Microsoft 提供了非常直观的对象模型, 这些模型包括所有这些领域。本章还将介绍如何使用 .NET 基类执行上面的任务。对于文件系统操作, 相关的类几乎都在 `System.IO` 名称空间中, 而注册表操作由 `System.Win32` 名称空间中的类来处理。



.NET 基类也包含 `System.Runtime.Serialization` 名称空间中的许多类和接口, 它们都与串行化有关。串行化是把一些数据(例如, 文档的内容)转换为字节流并存储在某个地方的过程。本章不讨论这些类, 而主要讨论可直接访问文件的类。

注意, 在修改文件或注册表项时, 安全性显得更为重要。第 21 章介绍了安全性的各个方面。但在本章中, 仅假定用户有足够的访问权限运行修改文件或注册表项的所有示例, 如果在拥有管理员权限的账户下运行, 就是这种情况。

29.1 管理文件系统

图 29-1 中的类可以用于浏览文件系统和执行操作, 如移动、复制和删除文件。

这些类的作用是:

- `System.MarshalByRefObject` —— 这是 .NET 类中用于远程操作的基对象类, 它允许在应用程序域之间编组数据。这个列表中的其他项都在 `System.IO` 名称空间中。
- `FileSystemInfo` —— 这是表示任何文件系统对象的基类。
- `FileInfo` 和 `File` —— 这些类表示文件系统上的文件。
- `DirectoryInfo` 和 `Directory` —— 这些类表示文件系统上的文件夹。

- Path ——这个类包含的静态成员可以用于处理路径名。
- DriveInfo ——它的属性和方法提供了指定驱动器的信息。



在 Windows 上，包含文件并用于组织文件系统的对象称为文件夹。例如，在路径 C:\My Documents\ReadMe.txt 中，ReadMe.txt 是一个文件，My Documents 是一个文件夹。文件夹是一个 Windows 专用的术语：在其他操作系统上，用术语“目录”代替文件夹，Microsoft 为了使 .NET 具有平台无关性，对应的 .NET 基类都称为 Directory 和 DirectoryInfo。因为它有可能与 LDAP 目录(详见第 52 章，具体内容见 Wrox 网站和随书附赠光盘)混淆，而且本书与 Windows 有关，所以本章仍使用文件

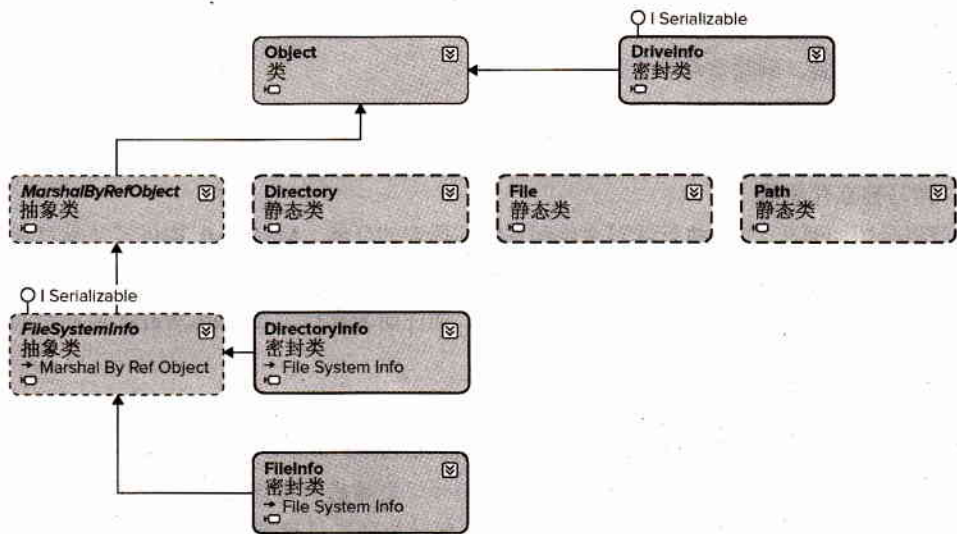


图 29-1

29.1.1 表示文件和文件夹的.NET 类

注意，上面的列表有两个用于表示文件夹的类，和两个用于表示文件的类。使用哪个类主要依赖于访问该文件夹或文件的次数：

- Directory 类和 File 类只包含静态方法，不能被实例化。只要调用一个成员方法，提供合适文件系统对象的路径，就可以使用这些类。如果只对文件夹或文件执行一个操作，使用这些类就很有效，因为这样可以省去实例化 .NET 类的系统开销。
- DirectoryInfo 类和 FileInfo 类实现与 Directory 类和 File 类大致相同的公共方法，并拥有一些公共属性和构造函数，但它们都是有状态的，并且这些类的成员都不是静态的。需要实例化这些类，之后把每个实例与特定的文件夹或文件关联起来。如果使用同一个对象执行多个操作，使用这些类就比较有效。这是因为在构造时它们将读取合适文件系统对象的身份验证和其他信息，无论对每个对象(类实例)调用了多少方法，都不需要再次读取这些信息。

比较而言，在调用每个方法时，相应的无状态类需要再次检查文件或文件夹的详细内容。本节主要使用 FileInfo 类和 DirectoryInfo 类，但我们调用的许多方法(不是全部)也可以由 File 类

和 `Directory` 类实现(尽管这些方法需要一个额外的参数——文件系统对象的路径名,这两个方法的名称略有不同)。例如:

```
FileInfo myFile = new FileInfo(@"C:\Program Files\My Program\ReadMe.txt");
myFile.CopyTo(@"D:\Copies\ReadMe.txt");
```

与下面的代码有相同的效果:

```
File.Copy(@"C:\Program Files\My Program\ReadMe.txt", @"D:\Copies\ReadMe.txt");
```

第一个代码段执行的时间略长,因为需要实例化一个 `FileInfo` 对象 `myFile`,但 `myFile` 可以对同一个文件执行进一步的操作。第二个示例不需要实例化对象来复制文件。

把包含对应文件系统对象的路径字符串传递给构造函数,就可以实例化 `FileInfo` 类或 `DirectoryInfo` 类。刚才已经介绍了文件的处理,对于文件夹,代码类似如下:

```
DirectoryInfo myFolder = new DirectoryInfo(@"C:\Program Files");
```

如果路径表示一个不存在的对象,在构造时就不会抛出异常。但如果是第一次调用方法,而该方法需要有一个对应的文件系统对象,就会抛出一个异常。检查 `Exists` 属性,可以确定对象是否存在,其类型是否合适,`FileInfo` 类和 `DirectoryInfo` 类都会实现该属性:

```
FileInfo test = new FileInfo(@"C:\Windows");
Console.WriteLine(test.Exists.ToString());
```

注意,对于这个属性,要返回 `true`,对应的文件系统对象必须是合适的类型。换言之,如果实例化了一个 `FileInfo` 对象,该对象提供了文件夹的路径,或者实例化了 `DirectoryInfo` 对象,并给它提供了文件的路径,`Exists` 的值就是 `false`。如果可能,这些对象的大多数属性和方法都会返回一个值——它们不会因为调用了错误类型的对象而抛出异常,除非要求它们完成一些确实不可能的任务。例如,上面的代码段会先显示 `false`(因为 `C:\Windows` 是一个文件夹),但接着就会显示创建文件夹的时间,因为文件夹仍拥有该信息。但如果使用 `FileInfo.Open()` 方法,以打开文件的方式打开文件夹,就会产生一个异常。

在确定了是否存在对应的文件系统对象后,就可以(如果用户正在使用 `FileInfo` 类或 `DirectoryInfo` 类)使用许多属性来确定该对象的信息,这些属性如表 29-1 所示。

表 29-1

名 称	作 用
<code>CreationTime</code>	创建文件或文件夹的时间
<code>DirectoryName</code> (仅用于 <code>FileInfo</code>)	包含文件夹的完整路径名
<code>Parent</code> (仅用于 <code>DirectoryInfo</code>)	指定子目录的父目录
<code>Exists</code>	文件或文件夹是否存在
<code>Extension</code>	文件的扩展名,对于文件夹它返回空白
<code>FullName</code>	文件或文件夹的完整路径名
<code>LastAccessTime</code>	最后一次访问文件或文件夹的时间

(续表)

名 称	作 用
LastWriteTime	最后一次修改文件或文件夹的时间
Name	文件或文件夹的名称
Root(仅用于 DirectoryInfo)	路径的根部分
Length(仅用于 FileInfo)	返回文件的大小(以字节为单位)

也可以使用表 29-2 所示的方法对文件系统对象执行操作。

表 29-2

名 称	作 用
Create()	创建给定名称的文件夹或空文件。对于 FileInfo，该方法会返回一个流对象，以便写入文件。 本章后面讨论流
Delete()	删除文件或文件夹。对于文件夹，有一个可以递归的 Delete 选项
MoveTo()	移动和/或重命名文件或文件夹
CopyTo()	(只适用于 FileInfo)复制文件，注意文件夹没有复制方法。如果复制完整的目录树，需要单独复制每个文件，创建对应于旧文件夹的新文件夹
GetDirectories()	(只适用于 DirectoryInfo)返回 DirectoryInfo 对象数组，该数组表示文件夹中包含的所有文件夹
GetFiles()	(只适用于 DirectoryInfo)返回 FileInfo 对象数组，该数组表示文件夹中包含的所有文件
GetFileSystemInfos()	(只适用于 DirectoryInfo)返回 FileInfo 和 DirectoryInfo 对象，它把文件夹中包含的所有对象表示为一个 FileSystemInfo 引用数组

注意，表 29-2 给出了主要的属性和方法，但没有列出所有的属性和方法。



表 29-2 没有列出读写文件数据的大多数属性或方法。读写文件数据实际上使用流对象完成，本章后面会介绍流对象。FileInfo 也可以实现 Open()、OpenRead()、OpenText()、OpenWrite()、Create()和 CreateText()等方法，为此它们都返回流对象。

有趣的是，创建时间、最后一次访问时间和最后一次写入时间都是可写入的。

```
// displays the creation time of a file,
// then changes it and displays it again
FileInfo test = new FileInfo(@"C:\MyFile.txt");
Console.WriteLine(test.Exists.ToString());
Console.WriteLine(test.CreationTime.ToString());
test.CreationTime = new DateTime(2010, 1, 1, 7, 30, 0);
Console.WriteLine(test.CreationTime.ToString());
```

运行这个应用程序，结果如下所示：

```
True
2/5/2009 2:59:32 PM
1/1/2010 7:30:00 AM
```

能手动修改这些属性首先看起来很奇怪,但它相当有效。例如,如果有一个程序可以通过读取、删除文件,用新内容创建新文件,来有效地修改文件,则可以修改创建日期,以匹配旧文件的最初创建日期。

29.1.2 Path 类

我们不能实例化 Path 类。然而,它提供了一些静态方法,可以更容易地对路径名执行操作。例如,假定要显示文件夹 C:\My Documents 中 ReadMe.txt 文件的完整路径名,可以用下述代码查找文件的路径:

```
Console.WriteLine(Path.Combine(@"C:\My Documents", "ReadMe.txt"));
```

使用 Path 类要比手动处理各个符号容易得多。尤其因为 Path 类在处理不同操作系统上的路径名时,能够识别不同的格式。在编写本书时,Windows 是 .NET 唯一支持的操作系统,但如果 .NET 以后要移植到 UNIX 上,Path 就要处理 UNIX 路径,Unix 把 “/” (并不是 “\”) 用作路径名中的分隔符。虽然 Path.Combine() 是这个类常常使用的一个方法,但 Path 类也实现其他方法,这些方法提供路径的信息,或者以要求的格式显示信息。

表 29-3 列出了 Path 类的一些静态字段。

表 29-3

属 性	说 明
AltDirectorySeparatorChar	提供一种与平台无关的方式,来指定分隔目录级别的另一个字符。在 Windows 上使用 “/” 符号,而在 UNIX 上使用 “\” 符号
DirectorySeparatorChar	提供一种与平台无关的方式,来指定分隔目录级别的另一个字符。在 Windows 上使用 “/” 符号,在 UNIX 上使用 “\” 符号
PathSeparator	提供一种与平台无关的方式,来指定划分环境变量的路径字符串,默认为分号
VolumeSeparatorChar	提供一种与平台无关的方式,来指定容量分隔符,默认为冒号

下一节用一个示例来说明如何浏览目录,查看文件的属性。

29.1.3 FileProperties 示例

本节要创建一个 C# 示例应用程序 FileProperties。该应用程序显示了一个简单的用户界面,可以浏览文件系统,查看文件的创建时间、最后一次访问时间,最后一次写入时间和文件的大小。这个应用程序的代码可以从 Wrox 网站或者随书附赠光盘中找到。

FileProperties 应用程序如下所述。在窗口顶部的主文本框中输入文件夹或文件的名称,再单击 Display 按钮。如果输入了文件夹的路径,其内容就显示在列表框中。如果输入文件的路径,则其详细信息显示在窗体底部的文本框中,父文件夹的内容显示在列表框中。屏幕图 29-2 显示了 FileProperties 示例应用程序。

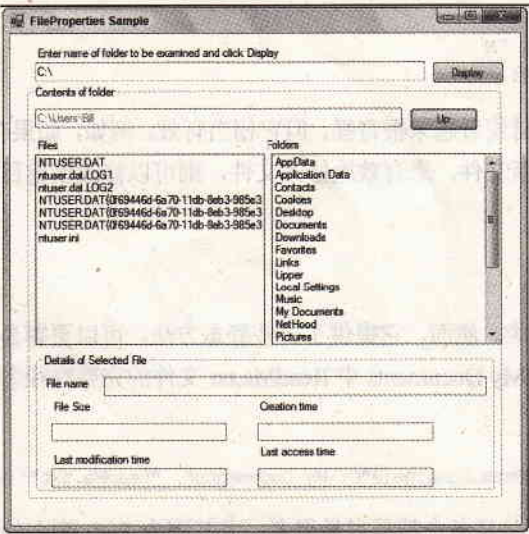


图 29-2

在定位文件系统时，单击右边列表框中的任何一个文件夹，就可以查看它下面的文件夹，或者单击 Up 按钮，查看其父文件夹。图 29-2 显示了我的 user 文件夹的内容。用户还可以单击列表框中的一个文件名，选择该文件。此时文件的属性就会显示在应用程序底部的文本框中，如图 29-3 所示。

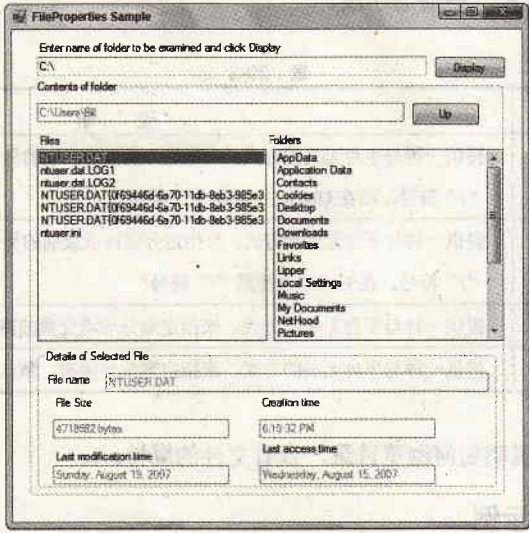


图 29-3

注意，根据需要，还可以使用 DirectoryInfo 属性显示文件夹的创建时间、最后一次访问时间和最后一次修改时间。我们只显示已选定文件的这些属性，使本例简单一些。

在 Visual Studio 2010 中创建一个标准的 C# Windows 应用程序项目，从工具箱的 Windows Forms 区域添加各种文本框和列表框。用更直观的名称来重命名这些控件：textBoxInput、textBoxFolder、buttonDisplay、buttonUp、listBoxFiles、listBoxFolders、textBoxFileName、textBoxCreationTime、textBoxLastAccessTime、textBoxLastWriteTime 和 textBoxFileSize。

然后需要指出，将使用 System.IO 名称空间：

```
using System;
using System.IO;
using System.Windows.Forms;
```

本章中所有与文件系统相关的示例都要使用该名称空间，但我们没有显式说明其余示例中的这部分代码。然后在主窗体中添加一个成员字段：



可从
wrox.com
下载源代码

```
public partial class Form1: Form
{
    private string currentFolderPath;
```

代码下载 FileProperties.sln

`CurrentFolderPath` 字符串存储了文件夹的路径，其内容显示在列表框中。

现在需要为用户生成的事件添加事件处理程序。用户可能输入的是：

- 用户单击 **Display** 按钮——此时，需要确定用户在主文本框中输入的内容是文件的路径还是文件夹的路径。如果是文件夹，列表框就会列出该文件夹中的文件和子文件夹。如果是文件，则仍要对包含该文件的文件夹进行上述操作，还要在下面的文本框中显示文件的属性。
- 用户单击 **Files** 列表框中的一个文件名——此时，在下面的文本框中显示文件的属性。
- 用户单击 **Folders** 列表框中的一个文件夹名——此时，将清除所有控件，并且在列表框中显示这个子文件夹的内容。
- 用户单击 **Up** 按钮——此时，将清除所有控件，并且在列表框中显示当前选中的文件夹的父文件夹的内容。

在查看事件处理程序的代码前，先列出实际完成所有任务的方法的代码。首先，需要清除所有控件的内容，这个方法很容易理解：



可从
wrox.com
下载源代码

```
protected void ClearAllFields()
{
    listBoxFolders.Items.Clear();
    listBoxFiles.Items.Clear();
    textBoxFolder.Text = "";
    textBoxFileName.Text = "";
    textBoxCreationTime.Text = "";
    textBoxLastAccessTime.Text = "";
    textBoxLastWriteTime.Text = "";
    textBoxFileSize.Text = "";
}
```

代码下载 FileProperties.sln

其次，定义一个方法 `DisplayFileInfo()`，该方法用于在文本框中显示给定文件的信息。它接受一个字符串参数，即文件的完整路径名，它根据该路径创建一个 `FileInfo` 对象：



可从
wrox.com
下载源代码

```
protected void DisplayFileInfo(string fileFullName)
{
    FileInfo theFile = new FileInfo(fileFullName);

    if (!theFile.Exists)
    {
        throw new FileNotFoundException("File not found: " + fileFullName);
    }
}
```

```

textBoxFileName.Text = theFile.Name;
textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
textBoxFileSize.Text = theFile.Length.ToString() + " bytes";
}

```

代码下载 [FileProperties.sln](#)

注意，如果在指定位置定位文件时有任何问题，我们将采取措施，防止抛出异常。异常在主调例程(一个事件处理程序)中处理。最后，定义一个方法 `DisplayFolderList()`，在两个列表框中显示给定文件夹的内容。该文件夹的完整路径名作为参数传递给该方法：



可从
wrox.com
下载源代码

```

protected void DisplayFolderList(string folderFullName)
{
    DirectoryInfo theFolder = new DirectoryInfo(folderFullName);

    if (!theFolder.Exists)
    {
        throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
    }

    ClearAllFields();
    textBoxFolder.Text = theFolder.FullName;
    currentFolderPath = theFolder.FullName;

    // list all subfolders in folder
    foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
        listBoxFolders.Items.Add(nextFolder.Name);

    // list all files in folder
    foreach(FileInfo nextFile in theFolder.GetFiles())
        listBoxFiles.Items.Add(nextFile.Name);
}

```

代码下载 [FileProperties.sln](#)

现在看看事件处理程序。用户单击 `Display` 按钮时会触发相应的事件，管理该事件的事件处理程序最复杂，因为它需要处理用户输入的 3 种不同的文本。用户可能输入文件夹的路径名、文件的路径名或什么也不输入：



可从
wrox.com
下载源代码

```

protected void OnDisplayButtonClick(object sender, EventArgs e)
{
    try
    {
        string folderPath = textBoxInput.Text;
        DirectoryInfo theFolder = new DirectoryInfo(folderPath);

        if (theFolder.Exists)
        {
            DisplayFolderList(theFolder.FullName);
            return;
        }

        FileInfo theFile = new FileInfo(folderPath);
        if (theFile.Exists)
        {

```

```

        DisplayFolderList(theFile.Directory.FullName);
        int index = listBoxFiles.Items.IndexOf(theFile.Name);
        listBoxFiles.SetSelected(index, true);
        return;
    }

    throw new FileNotFoundException("There is no file or folder with "
        + "this name: " + textBoxInput.Text);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
}

```

代码下载 FileProperties.sln

在上面的代码中，如果用户提供的文本表示一个文件夹或文件，就应实例化 `DirectoryInfo` 和 `FileInfo` 实例，并检查每个对象的 `Exists` 属性。如果它们都不存在，就抛出一个异常。如果它是一个文件夹，就调用 `DisplayFolderList()` 方法，给列表框填充数据。如果它是一个文件，就需要给列表框填充数据，给显示文件属性的文本框填充文本。具体处理过程是，首先给列表框填充数据，然后在 `Files` 列表框中以编程方式选择合适的文件名，这与用户选择该项的效果完全相同——它引发选中项对应的事件。然后退出当前事件处理程序，调用选中项的事件处理程序，来显示文件属性。

下面的代码是一个事件处理程序，当用户选中或以编程方式选中 `Files` 列表框中的一项时，就可以由用户或上面编写的代码调用该事件处理程序。它仅构造所选文件的完整路径名，并把该路径传递给前面给出的 `DisplayFileInfo()` 方法：



可从
wrox.com
下载源代码

```

protected void OnListBoxFilesSelected(object sender, EventArgs e)
{
    try
    {
        string selectedString = listBoxFiles.SelectedItem.ToString();
        string fullFileName = Path.Combine(currentFolderPath, selectedString);
        DisplayFileInfo(fullFileName);
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

代码下载 FileProperties.sln

处理 `Folders` 列表框中文件夹选择操作的事件处理程序以非常类似的方式实现。但此时调用 `DisplayFolderList()` 方法来更新列表框的内容：



可从
wrox.com
下载源代码

```

protected void OnListBoxFoldersSelected(object sender, EventArgs e)
{
    try
    {
        string selectedString = listBoxFolders.SelectedItem.ToString();
        string fullPathName = Path.Combine(currentFolderPath, selectedString);
        DisplayFolderList(fullPathName);
    }
}

```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

代码下载 FileProperties.sln

最后，在单击 Up 按钮时，必须调用 DisplayFolderList()方法，但这次需要获得当前显示的文件夹的父文件夹的路径。这可以通过 FileInfo.DirectoryName 属性来得到，该属性返回父文件夹的路径：



可从
wrox.com
下载源代码

```

protected void OnUpButtonClick(object sender, EventArgs e)
{
    try
    {
        string folderPath = new FileInfo(currentFolderPath).DirectoryName;
        DisplayFolderList(folderPath);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

代码下载 FileProperties.sln

29.2 移动、复制和删除文件

前面已经提到，移动和删除文件或文件夹可以使用 FileInfo 和 DirectoryInfo 类的 MoveTo()和 Delete()方法来完成。File 和 Directory 类的这两个对应方法是 Move()和 Delete()。FileInfo 和 File 类也分别实现 CopyTo()和 Copy()方法。没有复制完整文件夹的方法，而应复制文件夹中的每个文件。

这些方法的使用非常直观——SDK 文档提供了详细的解释。本节介绍在特定情况下，调用 File 类的静态方法 Move()、Copy()和 Delete()的作用。为此，把前面的 FileProperties 示例扩展为一个新示例 FilePropertiesAndMovement。这个示例有一个额外的功能：无论什么时候显示文件的属性，该应用程序都会给出删除该文件或把该文件移动或者复制到另一个地方的选项。

29.2.1 FilePropertiesAndMovement 示例

图 29-4 所示为新示例应用程序的用户界面。

可以看出，FilePropertiesAndMovement 的外观非常类似于 FileProperties 示例，但在窗口的底部添加了 3 个按钮组成的一组和一个文本框。这些控件仅在示例真正显示了文件的属性时才启用，在所有其他情况下，它们都是禁用的。我们还压缩了现有的控件，防止主窗体过大。在显示所选中文件的属性时，该示例会自动把文件的完整路径名放在底部的文本框中，供用户编辑。用户可以单击底部的任何一个按钮，执行相应的操作。此时，会显示一个相应的信息框，由用户确认该操作，如图 29-5 所示。

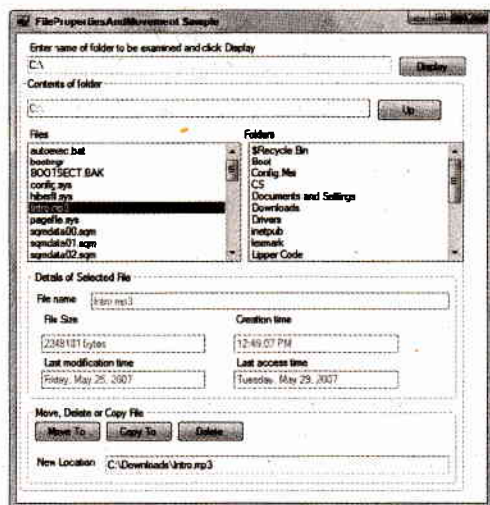


图 29-4

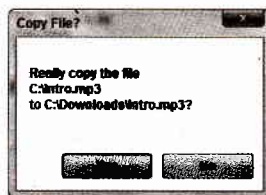


图 29-5

当用户单击 Yes 按钮后, 就可以开始执行某些动作。用户在窗体上执行的某些动作会使显示不正确。例如, 在移动和删除文件时, 显然不能在同一个地方显示该文件的内容。而且, 如果改变同一个文件夹中的文件名, 显示的信息也会过时。此时, FilePropertiesAndMovement 示例会重置其控件, 在文件的操作结束后, 只显示其中驻留文件的文件夹。

29.2.2 FilePropertiesAndMovement 示例的代码

为此, 需要在 FileProperties 示例中添加相关的控件, 及其事件处理程序代码。我们添加的控件是 buttonDelete、buttonCopyTo、buttonMoveTo 和 txtBoxNewPath。

首先看看用户单击 Delete 按钮时调用的事件处理程序:



可从
wrox.com
下载源代码

```
protected void OnDeleteButtonClick(object sender, EventArgs e)
{
    try
    {
        string filePath = Path.Combine(currentFolderPath,
                                       textBoxFileName.Text);
        string query = "Really delete the file\n" + filePath + "?";
        if (MessageBox.Show(query,
                           "Delete File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            File.Delete(filePath);
            DisplayFolderList(currentFolderPath);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to delete file. The following exception"
                        + " occurred:\n" + ex.Message, "Failed");
    }
}
```

代码下载 FilePropertiesAndMovement.sln

这个方法的代码包含在一个 try 块中,这是因为很显然会抛出一个异常,例如,在用户单击 Delete 按钮前,如果没有删除该文件的权限,或者在文件显示之后另一个进程移动了该文件,就会抛出一个异常。在 CurrentParentPath 字段中构造要删除文件的路径,其中包含父文件夹的路径,和 textBoxFileName 文本框中的文本(其中包含文件名)。

移动和复制文件的方法以类似的方式构造:



可从
wrox.com
下载源代码

```
protected void OnMoveButtonClick(object sender, EventArgs e)
{
    try
    {
        string filePath = Path.Combine(currentFolderPath,
                                       textBoxFileName.Text);
        string query = "Really move the file\n" + filePath + "\nto "
                      + textBoxNewPath.Text + "?";
        if (MessageBox.Show(query,
                           "Move File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            File.Move(filePath, textBoxNewPath.Text);
            DisplayFolderList(currentFolderPath);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to move file. The following exception"
                        + " occurred:\n" + ex.Message, "Failed");
    }
}

protected void OnCopyButtonClick(object sender, EventArgs e)
{
    try
    {
        string filePath = Path.Combine(currentFolderPath,
                                       textBoxFileName.Text);
        string query = "Really copy the file\n" + filePath + "\nto "
                      + textBoxNewPath.Text + "?";
        if (MessageBox.Show(query,
                           "Copy File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            File.Copy(filePath, textBoxNewPath.Text);
            DisplayFolderList(currentFolderPath);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to copy file. The following exception"
                        + " occurred:\n" + ex.Message, "Failed");
    }
}
```

代码下载 FilePropertiesAndMovement.sln

此外,还需要确保新按钮和文本框在合适的时间是启用的或禁用的。要在显示文件的内容时启用它们,需要把下述代码添加到 DisplayFileInfo()方法中:



可从
wrox.com
下载源代码

```
protected void DisplayFileInfo(string fileFullName)
{
    FileInfo theFile = new FileInfo(fileFullName);

    if (!theFile.Exists)
    {
        throw new FileNotFoundException("File not found: " + fileFullName);
    }

    textBoxFileName.Text = theFile.Name;
    textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
    textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
    textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
    textBoxFileSize.Text = theFile.Length.ToString() + " bytes";

    // enable move, copy, delete buttons
    textBoxNewPath.Text = theFile.FullName;
    textBoxNewPath.Enabled = true;
    buttonCopyTo.Enabled = true;
    buttonDelete.Enabled = true;
    buttonMoveTo.Enabled = true;
}
```

代码下载 FilePropertiesAndMovement.sln

还需要修改 DisplayFolderList()方法:



可从
wrox.com
下载源代码

```
protected void DisplayFolderList(string folderFullName)
{
    DirectoryInfo theFolder = new DirectoryInfo(folderFullName);

    if (!theFolder.Exists)
    {
        throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
    }

    ClearAllFields();
    DisableMoveFeatures();
    textBoxFolder.Text = theFolder.FullName;
    currentFolderPath = theFolder.FullName;

    // list all subfolders in folder
    foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
        listBoxFolders.Items.Add(nextFolder.Name);

    // list all files in folder
    foreach(FileInfo nextFile in theFolder.GetFiles())
        listBoxFiles.Items.Add(nextFile.Name);
}
```

代码下载 FilePropertiesAndMovement.sln

DisableMoveFeatures()方法是禁用新控件的一个小实用程序函数:



可从
wrox.com
下载源代码

```
void DisableMoveFeatures()
{
    textBoxNewPath.Text = "";
    textBoxNewPath.Enabled = false;
    buttonCopyTo.Enabled = false;
}
```

```

        buttonDelete.Enabled = false;
        buttonMoveTo.Enabled = false;
    }

```

代码下载 FilePropertiesAndMovement.sln

还需要给 `ClearAllFields()` 方法添加额外的代码，以清除额外的文本框：



可从
wrox.com
下载源代码

```

protected void ClearAllFields()
{
    listBoxFolders.Items.Clear();
    listBoxFiles.Items.Clear();
    textBoxFolder.Text = "";
    textBoxFileName.Text = "";
    textBoxCreationTime.Text = "";
    textBoxLastAccessTime.Text = "";
    textBoxLastWriteTime.Text = "";
    textBoxFileSize.Text = "";
    textBoxNewPath.Text = "";
}

```

代码下载 FilePropertiesAndMovement.sln

下一节看看文件的读写操作。

29.3 读写文件

读写文件在原则上非常简单，但它不是通过 `DirectoryInfo` 或 `FileInfo` 对象完成。在 .NET Framework 4 中，可以通过 `File` 对象读写文件。本章后面将学习如何使用许多其他类来读写文件，这些类表示一个通用的概念：流。

在 .NET Framework 2.0 推出之前，读写文件比较费劲，可以使用 Framework 中的类来读写文件，但不是很简单。NET Framework 2.0 扩展了 `File` 类，只需编写一行代码，就可以读写文件。NET Framework 4 也有这个功能。

29.3.1 读取文件

对于下面读取文件的示例，创建一个 Windows 窗体应用程序，它包含一个常规的文本框、一个按钮和一个多行文本框。最后，窗体如图 29-6 所示。

这个窗体的作用是，最终用户在第一个文本框中输入某个特定文件的路径，并单击 `Read` 按钮。此时应用程序就应读取指定的文件，并在多行文本框中显示文件的内容。其代码如下：



可从
wrox.com
下载源代码

```

using System;
using System.IO;
using System.Windows.Forms;

namespace ReadingFiles
{

```

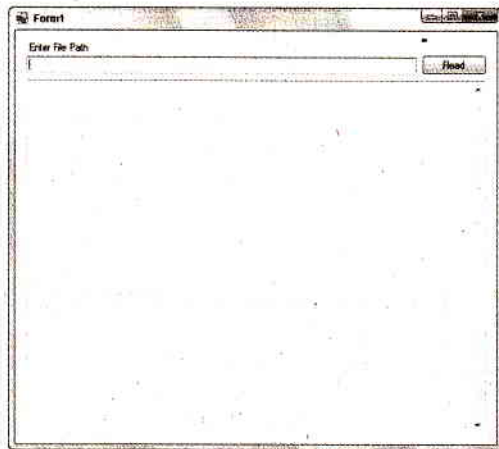


图 29-6

```

public partial class Form1: Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        textBox2.Text = File.ReadAllText(textBox1.Text);
    }
}

```

代码下载 ReadingFiles.sln

在构建这个示例时，第一步是添加 using 语句，引入 System.IO 名称空间。之后，使用窗体上 Send 按钮的 button1_Click 事件，用文件中的内容填充文本框。现在，就可以使用 File.ReadAllText() 方法获得文件的内容。显然，可以使用一条语句读取文件。ReadAllText() 方法会打开指定的文件，读取内容，然后关闭文件。ReadAllText() 方法的返回值是包含文件全部内容的字符串。最终结果如图 29-7 所示。

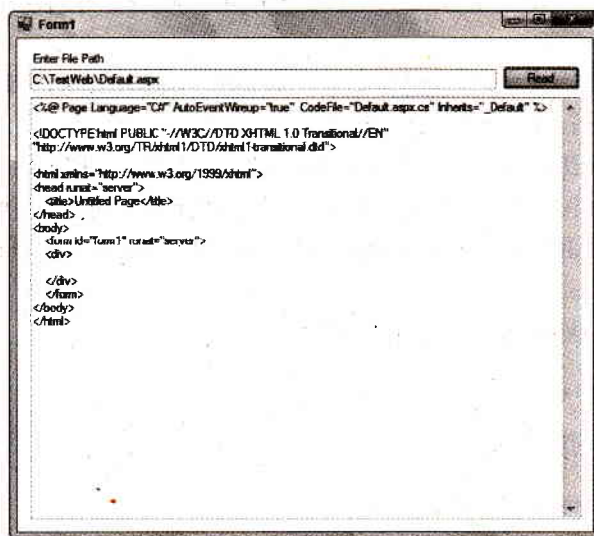


图 29-7

上面示例中的 File.ReadAllText() 签名有如下结构：

```
File.ReadAllText (FilePath);
```

另一个选项指定了要读取的文件的编码格式：

```
File.ReadAllText (FilePath, Encoding);
```

使用这个签名可以指定在打开和读取文件内容时使用的编码格式。因此，可以编写如下代码：

```
File.ReadAllText (textBox1.Text, Encoding.ASCII);
```

打开并处理文件的其他选项有 ReadAllBytes() 和 ReadAllLines() 方法。ReadAllBytes() 方法可以打

开二进制文件，把其内容读入一个字节数组中。前面列出的 `ReadAllText()` 方法会在一个字符串实例中提供指定文件的全部内容。我们对此不感兴趣，而关注以逐行方式处理文件中内容的方式。此时，应使用 `ReadAllLines()` 方法，因为它拥有这个功能，并返回一个字符串数组，以进行处理。

29.3.2 写入文件

在 .NET Framework 中，基类库除了使读取文件是一个非常简单的过程之外，也使写入文件一样简单。基类库除了提供 `ReadAllText()`、`ReadAllLines()` 和 `ReadAllBytes()` 方法，以几种不同的方式读取文件之外，还提供了写入文件的方法 `WriteAllText()`、`WriteAllBytes()` 和 `WriteAllLines()`。

为了说明如何写入文件，虽然使用同一个 Windows 窗体应用程序，但把窗体中的多行文本框用于将数据输入文件中。`button1_Click` 事件处理程序的代码如下所示：

```
private void button1_Click(object sender, EventArgs e)
{
    File.WriteAllText(textBox1.Text, textBox2.Text);
}
```

构建窗体，并启动，在第一个文本框中输入 `C:\Testing.txt`，在第二个文本框中输入一些随机内容，然后单击按钮。从视觉上什么也没有发生，但如果查看 C 盘，就会看到包含指定内容的 `Testing.txt` 文件。

如果在保存和关闭文件之前，给文件指定了一些内容，`WriteAllText()` 方法就会进入指定的位置，新建一个文本文件。只需一行代码即可完成文件的写入操作。

如果再次运行应用程序，指定同一个文件(`Testing.txt`)，但输入一些新内容，再次单击按钮，应用程序就会执行相同的任务，但注意，这次新内容不是添加到原有内容的后面，而是完全覆盖以前的内容。事实上，`WriteAllText()`、`WriteAllBytes()` 和 `WriteAllLines()` 方法都会覆盖以前的文件，所以在使用这些方法时要小心。

上面示例中的 `WriteAllText()` 方法使用如下签名：

```
File.WriteAllText(filePath, contents)
```

还可以指定新文件的编码格式：

```
File.WriteAllText(filePath, contents, encoding)
```

`WriteAllBytes()` 方法可以使用字节数组把内容写入文件，`WriteAllLines()` 方法可以把字符串数组写入文件，如下面的事件处理程序所示：

```
private void button1_Click(object sender, EventArgs e)
{
    string[] movies =
    {
        "Grease",
        "Close Encounters of the Third Kind",
        "The Day After Tomorrow"
    };

    File.WriteAllLines(@"C:\Testing.txt", movies);
}
```

现在单击按钮，应用程序就会提供 `Testing.txt` 文件，其内容如下：

Grease
Close Encounters of the Third Kind
The Day After Tomorrow

`WriteAllLines()`方法把字符串数组中的每个元素单独放在文件的一行上。

因为数据不仅可以写入磁盘，还可以放在其他地方(如命名管道或内存)，所以必须理解如何在.NET 中使用流处理文件输入输出，作为一种移动文件内容的方式，如下一节所述。

29.3.3 流

流的概念已经存在很长时间了。流是一个用于传输数据的对象，数据可以向两个方向传输：

- 如果数据从外部源传输到程序中，这就是读取流。
- 如果数据从程序传输到外部源中，这就是写入流。

外部源常常是一个文件，但也不完全都是文件。它还可能是：

- 使用一些网络协议读写网络上的数据，其目的是选择数据，或从另一个计算机上发送数据。
- 读写到命名管道上。
- 把数据读写到一个内存区域上。

在这些示例中，Microsoft 提供了一个.NET 基类 `System.IO.MemoryStream` 对象来读写内存，而 `System.Net.Sockets.NetworkStream` 对象处理网络数据。读写管道没有基本的流类，但有一个泛型流类 `System.IO.Stream`，如果要编写一个这样的类，可以从这个基类继承。流对外部数据源不做任何假定。

外部源甚至可以是代码中的一个变量。这听起来很荒谬，但使用流在变量之间传输数据的技术是一个非常有用的技巧，可以在数据类型之间转换数据。C 语言使用它在整型和字符串之间转换数据类型，或者使用函数 `sprintf()` 格式化字符串。

使用一个独立的对象来传输数据，比使用 `FileInfo` 或 `DirectoryInfo` 类更好，因为把传输数据的概念与特定数据源分离开来，可以更容易交换数据源。流对象本身包含许多泛型代码，可以在外部数据源和代码中的变量之间移动数据，把这些代码与特定数据源的概念分离开来，就更容易实现不同环境下代码的重用(通过继承)。例如，前面提到的 `StringReader` 和 `StringWriter` 类，与本章后面用于读写文本文件的两个类 `StreamReader` 和 `StreamWriter` 一样，都是同一继承树的一部分，这些类几乎一定在后台共享许多代码。

在 `System.IO` 名称空间中，与流相关的类的层次结构如图 29-8 所示。

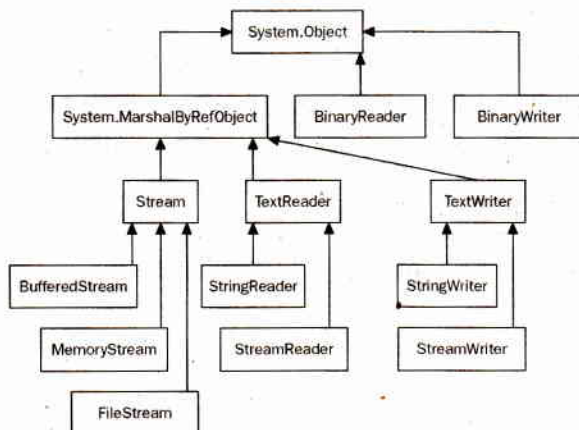


图 29-8

对于文件的读写，最常用的类如下：

- **FileStream**(文件流)——这个类主要用于在二进制文件中读写二进制数据——也可以使用它读写任何文件。
- **StreamReader**(流读取器)和 **StreamWriter**(流写入器)——这两个类专门用于读写文本文件。

虽然我们没有在示例中使用另外两个类 **BinaryReader** 和 **BinaryWriter**，但它们也很有用。**BinaryReader** 和 **BinaryWriter** 这两个类本身并不实现流，但它们能够提供其他流对象的包装器。**BinaryReader** 和 **BinaryWriter** 还可以对二进制数据进行额外的格式化，直接从相关的流中读写 C# 变量的内容。最简单的方式是把 **BinaryReader** 和 **BinaryWriter** 放在流和代码之间，进行额外的格式化，如图 29-9 所示。

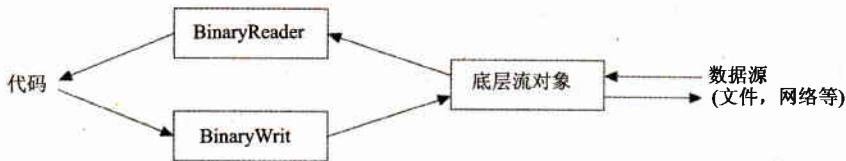


图 29-9

使用这些类和直接使用底层的流对象之间的区别是，基本流是按照字节来工作的。例如，在保存某个文档时，需要把类型为 **long** 的变量的内容写入一个二进制文件中，每个 **long** 型变量都占用 8 个字节，如果使用一般的二进制流，就必须显式地写入内存的 8 个字节中。

在 C# 代码中，必须执行一些按位操作，从 **long** 值中提取这 8 个字节。使用 **BinaryWriter** 实例，可以把整个操作封装在 **BinaryWriter.Write()** 方法的一个重载方法中，该方法的参数是 **long** 型，它把 8 个字节写入流中(如果流指向一个文件，就写入该文件)。对应的 **BinaryReader.Read()** 方法则从流中提取 8 个字节，恢复 **long** 的值。**BinaryReader** 和 **BinaryWriter** 类的更多信息详见 SDK 文档。

29.3.4 缓存的流

从性能原因上看，在读写文件时，输出结果会被缓存。如果程序要求读取文件流中下面的两个字节，该流会把请求传递给 Windows，则 Windows 不会连接文件系统，再定位文件，并从磁盘中读取文件，仅读取 2 个字节。而是在一次读取过程中，检索文件中的一个大块，把该块保存在一个内存区域即缓冲区上。以后对流中数据的请求就会从该缓冲区中读取，直到读取完该缓冲区为止。此时，Windows 会从文件中再获取另一个数据块。

写入文件的方式与此相同。对于文件，操作系统会自动完成读写操作，但需要编写一个流类，从其他没有缓存的设备中读取数据。如果是这样，就应从 **BufferedStream** 派生一个类，它实现一个缓冲区(但 **BufferedStream** 并不用于应用程序频繁切换读数据和写数据的情形)。

29.3.5 使用 FileStream 类读写二进制文件

读写二进制文件通常要使用 **FileStream** 类。如果使用 .NET Framework 1.x，就很可能使用这个类。

1. FileStream 类

FileStream 实例用于读写文件中的数据。要构造 **FileStream** 实例，需要以下 4 条信息：

- 要访问的文件。
- 表示如何打开文件的模式。例如，新建一个文件或打开一个现有的文件。如果打开一个现有的文件，写入操作是覆盖文件原来的内容，还是追加到文件的末尾？
- 表示访问文件的方式——是只读、只写，还是读写？
- 共享访问——表示是否独占访问文件。如果允许其他流同时访问文件，则这些流是只读、只写，还是读写文件？

第一条信息通常用一个包含文件的完整路径名的字符串来表示，本章只考虑需要该字符串的那些构造函数。除了这些构造函数外，一些其他的构造函数用老式的 Windows-API 风格的 Windows 句柄来处理文件。其余 3 条信息分别由 3 个 .NET 枚举 `FileMode`、`FileAccess` 和 `FileShare` 来表示，这些枚举的值很容易理解，如表 29-4 所示。

表 29-4

枚 举	值
<code>FileMode</code>	<code>Append</code> 、 <code>Create</code> 、 <code>CreateNew</code> 、 <code>Open</code> 、 <code>OpenOrCreate</code> 和 <code>Truncate</code>
<code>FileAccess</code>	<code>Read</code> 、 <code>ReadWrite</code> 和 <code>Write</code>
<code>FileShare</code>	<code>Delete</code> 、 <code>Inheritable</code> 、 <code>None</code> 、 <code>Read</code> 、 <code>ReadWrite</code> 和 <code>Write</code>

注意，对于 `FileMode`，如果要求的模式与文件的现有状态不一致，就会抛出一个异常。如果文件不存在，`Append`、`Open` 和 `Truncate` 就会抛出一个异常；如果文件存在，`CreateNew` 就会抛出一个异常。`Create` 和 `OpenOrCreate` 可以处理这两种情况，但 `Create` 会删除任何现有的文件，新建一个空文件。因为 `FileAccess` 和 `FileShare` 枚举是按位标志，所以这些值可以与 C# 的按位 OR 运算符“|”合并使用。

`FileStream` 类有许多构造函数，其中 3 个最简单的构造函数如下所示。

```
// creates file with read-write access and allows other streams read access
FileStream fs = new FileStream(@"C:\C# Projects\Project.doc",
    FileMode.Create);
// as above, but we only get write access to the file
FileStream fs2 = new FileStream(@"C:\C# Projects\Project2.doc",
    FileMode.Create, FileAccess.Write);
// as above but other streams don't get access to the file while
// fs3 is open
FileStream fs3 = new FileStream(@"C:\C# Projects\Project3.doc",
    FileMode.Create, FileAccess.Write, FileShare.None);
```

从这段代码中可以看出，这些构造函数的重载版本会根据 `FileMode` 的值，把 `FileAccess.ReadWrite` 和 `FileShare.Read` 的默认值作为第 3 个和第 4 个参数，也可以以多种方式从 `FileInfo` 实例中创建一个文件流：

```
FileInfo myFile4 = new FileInfo(@"C:\C# Projects\Project4.doc");
FileStream fs4 = myFile4.OpenRead();
FileInfo myFile5 = new FileInfo(@"C:\C# Projects\Project5.doc");
FileStream fs5 = myFile5.OpenWrite();
FileInfo myFile6 = new FileInfo(@"C:\C# Projects\Project6.doc");
FileStream fs6 = myFile6.Open(FileMode.Append, FileAccess.Write,
```

```
        FileShare.None);  
        FileInfo myFile7 = new FileInfo(@"C:\C# Projects\Project7.doc");  
        FileStream fs7 = myFile7.Create();
```

`FileInfo.OpenRead()`方法提供的流只能读取现有文件，而 `FileInfo.OpenWrite()`方法可以进行读写访问。`FileInfo.Open()`方法允许显式地指定模式、访问方式和文件共享参数。

当然，使用完一个流后，就应关闭它：

```
fs.Close();
```

关闭流会释放与它相关联的资源，允许其他应用程序为同一个文件设置流。这个操作也会刷新缓冲区。在打开和关闭流之间，可以读写其中的数据。`FileStream` 类实现了许多方法以进行这样的读写。

`ReadByte()`是读取数据的最简单的方式，它从流中读取一个字节，把结果转换为一个 0~255 之间的一个整数。如果到达该流的末尾，它就返回-1：

```
int NextByte = fs.ReadByte();
```

如果要一次读取多个字节，就可以调用 `Read()`方法，它可以把特定数量的字节读入一个数组中。`Read()`方法返回实际读取的字节数——如果这个值是 0，就表示到达了流的末尾。下面是读入一个字节数组 `ByteArray` 的一个示例：

```
int nBytesRead = fs.Read(ByteArray, 0, nBytes);
```

`Read()`方法的第二个参数是一个偏移值，使用它可以要求 `Read` 操作的数据从数组的某个元素开始填充，而不是从第一个元素开始。第 3 个参数是要读入数组的字节数。

如果要给文件写入数据，就可以使用两个并行方法 `WriteByte()`和 `Write()`。`WriteByte()`方法把一个字节写入流中：

```
byte NextByte = 100;  
fs.WriteByte(NextByte);
```

而 `Write()`方法写入一个字节数组。例如，如果用一些值初始化前面的 `ByteArray` 数组，就可以使用下面的代码输出数组的前 `nBytes` 个字节：

```
fs.Write(ByteArray, 0, nBytes);
```

与 `Read()`方法一样，第二个参数可以从数组的某个元素开始写入，而不是从第一个元素开始。`WriteByte()`方法和 `Write()`方法都没有返回值。

除了这些方法以外，`FileStream` 还实现了其他关于记账任务的方法和属性，如确定流中有多少字节，锁定流或刷新缓冲区等。其他方法通常不是基本读写数据所必需的，但如果需要它们，就可以参阅 SDK 文档。

2. BinaryFileReader 示例

下面编写一个示例 `BinaryFileReader` 说明 `FileStream` 类的用法。这个示例可以读取和显示任何文件。在 Visual Studio 2010 中作为一个 Windows 应用程序创建对应项目，添加一个菜单项，它可以打开一个标准的 `OpenFileDialog` 对话框，要求用户指定要读取的文件，然后把该文件显示为二进制码。

在读取二进制文件时，需要显示非打印字符。此时可以在多行文本框中逐个显示文件中的每个字节，每行显示 16 个字节。如果字节表示一个可打印的 ASCII 字符，就显示该字符；否则就以十六进制的格式显示该字节的值。在这两种情况下，显示的文本之间都用空格隔开，这样每个显示的字节都占用 4 列，于是显得它们的排列非常整齐。

图 29-10 是查看文本文件时 BinaryFileReader 应用程序的外观(因为 BinaryFileReader 可以查看任何文件，所以可以在文本文件和二进制文件中使用它)。对于本示例，应用程序读取一个基本的 ASP.NET 页面(.aspx)。

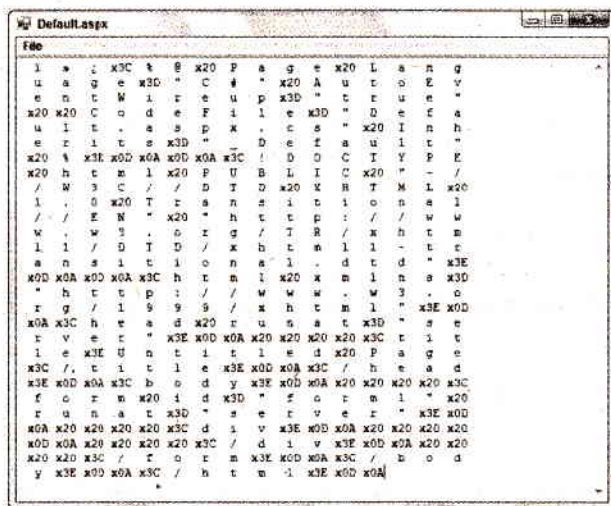


图 29-10

显然，这种格式更适合于查看单个字节的值，而不适合显示文本。本章后面会开发一个专门用于读取文本文件的示例——然后就可以查看文件中的内容。这个示例的优点是可以查看任何文件的内容。

这个示例没有说明如何写入文件。这是因为我们不希望把文本框中的内容(如图 29-10 所示)转换为二进制流，增加程序的复杂性。在后面开发可读写文本文件的示例中，将介绍如何写入文件，但只限于文本文件。

下面看看用于获得这些结果的代码。首先，需要使用 using 语句，引入 System.IO 名称空间：

```
using System.IO;
```

接着，给主窗体类添加两个字段，一个字段表示文件对话框，另一个字段表示当前查看的文件的路径：



可从
wrox.com
下载源代码

```
partial class Form1: Form
{
    private readonly OpenFileDialog chooseOpenFileDialog =
        new OpenFileDialog();
    private string chosenFile;
```

代码下载 BinaryFileReader.sln

还需要添加一些标准的 Windows 窗体代码，来处理菜单和文件对话框：



可从
wrox.com
下载源代码

```
public Form1()
{
    InitializeComponent();
    menuFileOpen.Click += OnFileOpen;
    chooseOpenFileDialog.FileOk += OnOpenFileDialogOK;
}

void OnFileOpen(object Sender, EventArgs e)
{
    chooseOpenFileDialog.ShowDialog();
}

void OnOpenFileDialogOK(object Sender, CancelEventArgs e)
{
    chosenFile = chooseOpenFileDialog.FileName;
    this.Text = Path.GetFileName(chosenFile);
    DisplayFile();
}
```

代码下载 [BinaryFileReader.sln](#)

可以看出，用户单击 OK 按钮，在文件对话框中选择一个文件后，就会调用 `DisplayFile()` 方法，该方法读取所选中的文件：



可从
wrox.com
下载源代码

```
void DisplayFile()
{
    int nCols = 16;
    FileStream inStream = new FileStream(chosenFile, FileMode.Open,
                                         FileAccess.Read);

    long nBytesToRead = inStream.Length;
    if (nBytesToRead > 65536/4)
        nBytesToRead = 65536/4;

    int nLines = (int)(nBytesToRead/nCols) + 1;
    string [] lines = new string[nLines];
    int nBytesRead = 0;

    for (int i=0; i<nLines; i++)
    {
        StringBuilder nextLine = new StringBuilder();
        nextLine.Capacity = 4*nCols;

        for (int j=0; j<nCols; j++)
        {
            int nextByte = inStream.ReadByte();
            nBytesRead++;
            if (nextByte < 0 || nBytesRead > 65536)
                break;
            char nextChar = (char)nextByte;
            if (nextChar < 16)
                nextLine.Append(" x0" + string.Format("{0,1:X}",
                                                         (int)nextChar));
            else if
                (char.IsLetterOrDigit(nextChar) ||
                 char.IsPunctuation(nextChar))
                nextLine.Append(" " + nextChar + " ");
        }
    }
}
```

```

        else
            nextLine.Append(" x" + string.Format("{0,2:X}",
                (int)nextChar));
    }
    lines[i] = nextLine.ToString();
}
inStream.Close();
this.textBoxContents.Lines = lines;
}

```

代码下载 BinaryFileReader.sln

在这个方法中进行了许多工作，首先为所选文件实例化一个 `FileStream` 对象，指定要打开一个现有文件进行读取。然后确定要读取多少个字节，和应该显示多少行。这个字节数一般是文件中的字节数。文本框中的内容最多只能显示 65 536 个字符，但以我们选择的格式，文件中的每个字节会显示 4 个字符。



用户可能希望使用 `System.Windows.Forms` 名称空间中的 `RichTextBox` 类。`RichTextBox` 类似于文本框，但它有更高级的格式化功能，此处使用 `TextBox`，是为了让示例比较简单，用户可以只考虑读取文件的过程。

在该方法中，有相当多的部分是两个嵌套的 `for` 循环，它们构造要显示的每行文本。我们使用 `StringBuilder` 类来构造每一行文本，其性能方面的原因是：循环 16 次可以把每个字节的合适文本追加到表示每行文本的字符串上。如果一行上有一个新字符串，并复制构造该行时的一半字符，则不仅要花很多时间分配字符串，还会浪费堆上的许多内存。注意，可打印的字符是字母、数字或标点符号，这与相关的静态方法 `System.Char` 指定的一样。值小于 16 的字符都不是可打印的字符，然而，这意味着回车符(13)和换行符(10)都是二进制字符(如果这些字符单独占一行，多行文本框就不能正确显示它们)。

另外，使用 `Properties` 窗口，把文本框的 `Font` 属性改为固定宽度的字体——我们选择 `Courier New 9pt regular` 字体，并把文本框设置为有水平和垂直滚动条。

最后，关闭流，把文本框的内容设置为已构建的字符串数组。

29.3.6 读写文本文件

理论上，可以使用 `FileStream` 类读取和显示文本文件。前面已经介绍了这个类。虽然上面示例中显示 `Default.aspx` 文件的格式不太容易理解，但这并不是 `FileStream` 类的内在问题——而在于我们在文本框中显示结果所使用的方式。

如果知道某个特殊文件包含文本，通常就可以使用 `StreamReader` 和 `StreamWriter` 类更方便地读写它们，而不是使用 `FileStream` 类。这是因为这些类工作的级别比较高，特别适合于读写文本。它们实现的方法可以根据流的内容，自动检测出停止读取文本较方便的位置。特别是：

- 这些类实现的方法(`StreamReader.ReadLine()`和 `StreamWriter.WriteLine()`)可以一次读写一行文本。在读取文件时，流会自动确定下一个回车符的位置，并在该处停止读取。在写入文件时，流会自动把回车符和换行符追加到文本的末尾。

- 使用 `StreamReader` 和 `StreamWriter` 类, 就不需要担心文件中使用的编码方式(文本格式)。可能的编码方式是 ASCII(一个字节表示一个字符)或者基于 Unicode 的任何格式, Unicode、UTF7、UTF8 和 UTF32。Windows 9x 系统上的文本文件总是 ASCII 格式, 因为 Windows 9x 系统不支持 Unicode, 但因为 Windows NT、2000、XP、2003、Vista、Windows Server 2008 和 Windows 7 都支持 Unicode, 所以文本文件除了包含 ASCII 数据之外, 理论上可以包含 Unicode、UTF7、UTF8 或 UTF32 数据。其约定是: 如果文件是 ASCII 格式, 它就只包含文本。如果文件是 Unicode 格式, 这就用文件的前两个或三个字节来表示, 这几个字节可以设置为表示文件中格式的值的特定组合。

这些字节称为字节码标记。在使用标准 Windows 应用程序打开一个文件时, 如 Notepad 或 WordPad, 不需要考虑这个问题, 因为这些应用程序都能识别不同的编码方法, 会自动正确地读取文件。`StreamReader` 类也是这样, 它可以正确地读取任何格式的文件, 而 `StreamWriter` 类可以使用任何一种编码技术格式化它要输出的文本。但如果要使用 `FileStream` 类读取和显示文本文件, 就必须自己处理这个过程。

1. `StreamReader` 类

`StreamReader` 实例用于读取文本文件。用某些方式构造 `StreamReader` 实例要比构造 `FileStream` 实例更简单, 因为使用 `StreamReader` 时不需要 `FileStream` 的一些选项。特别是模式和访问类型与 `StreamReader` 类不相关, 因为 `StreamReader` 只能执行读取操作。除此以外, 没有指定共享许可的直接选项, 但 `StreamReader` 有两个新选项:

- 需要指定不同的编码方法所执行的不同操作。可以构造一个 `StreamReader` 检查文件开头的字节码标记, 确定编码方法, 或者告诉 `StreamReader` 假定该文件使用某种指定的编码方法。
- 不提供要读取的文件名, 而为另一个流提供一个引用。

最后一个选项需要解释一下, 因为它说明了把读写数据的模型建立在流概念上的另一个优点。因为 `StreamReader` 工作在相对较高的级别上, 如果有另一个流在读取其他源中的数据, 就要使用 `StreamReader` 提供的工具来处理这个流, 就好像这个流包含文本, 所以此时 `StreamReader` 类非常有用。可以把这个流的输出传递给 `StreamReader`, 这样, `StreamReader` 就可以用于读取和处理任何数据源(不仅仅是文件)中的数据。前面在讨论 `BinaryReader` 类时也讨论了这种情况。但在本书中, 只使用 `StreamReader` 来直接连接文件。

因此, `StreamReader` 类有非常多的构造函数。而且, 还有一个返回 `StreamReader` 引用的 `FileInfo` 方法 `OpenText()`。下面仅说明其中一些构造函数。

最简单的构造函数只带一个文件名参数。`StreamReader` 会检查字节码标记, 以确定编码方法:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt");
```

另外, 如果指定 UTF8 编码方法, 就应假定:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt",  
Encoding.UTF8);
```

使用 `System.Text.Encoding` 类的几个属性之一, 就可以指定编码方法。这个类是一个抽象基类, 可以从这个类派生许多类, 这个类实现实际上完成文本编码的方法。每个属性都返回相应类的一个实例, 可以使用的属性包括:

- ASCII
- Unicode
- UTF7
- UTF8
- UTF32
- BigEndianUnicode

下面的示例解释了如何把 **StreamReader** 关联到 **FileStream** 上。其优点是可以指定是否创建文件和共享许可，如果直接把 **StreamReader** 附加到文件中，就不能这么做：

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
                               FileMode.Open, FileAccess.Read, FileShare.None);
StreamReader sr = new StreamReader(fs);
```

对于本例，指定 **StreamReader** 查找字节码标记，以确定使用了什么编码方法，以后的示例也是这样，从一个 **FileInfo** 实例中获得 **StreamReader**：

```
FileInfo myFile = new FileInfo(@"C:\My Documents\ReadMe.txt");
StreamReader sr = myFile.OpenText();
```

与 **FileStream** 一样，应在使用后关闭 **StreamReader**。如果没有这样做，就会致使文件一直锁定，因此不能执行其他进程(除非使用 **FileStream** 构造 **StreamReader**，并指定 **FileShare.ShareReadWrite**)：

```
sr.Close();
```

介绍完实例化 **StreamReader** 后，就可以用该实例做一些工作了。与 **FileStream** 一样，我们仅指出可以用于读取数据的许多方式，在 SDK 文档中可以到查阅其他不太常用的 **StreamReader** 类的方法。

最简单的方法使用是 **ReadLine()**，该方法一次读取一行，但返回的字符串中不包括标记该行结束的回车换行符：

```
string nextLine = sr.ReadLine();
```

另外，还可以在一个字符串中提取文件的所有剩余内容(严格地说，是流的全部剩余内容)：

```
string restOfStream = sr.ReadToEnd();
```

可以只读取一个字符：

```
int nextChar = sr.Read();
```

Read()方法的重载版本可以把返回的字符强制转换为一个整数。如果到达流的末尾，它就返回 -1。最后，可以用一个偏移值，把给定个数的字符读到数组中：

```
// to read 100 characters in.

int nChars = 100;
char [] charArray = new char[nChars];
int nCharsRead = sr.Read(charArray, 0, nChars);
```

如果要求读取的字符数多于文件中剩余的字符数，**nCharsRead** 就应小于 **nChars**。

2. StreamWriter 类

StreamWriter 类的工作方式与 **StreamReader** 类似, 但 **StreamWriter** 只能用于写入文件(或另一个流)。构造 **StreamWriter** 的方法包括:

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt");
```

上面的代码使用了 UTF8 编码方法, .NET 把这种编码方法设置为默认的编码方法。还可以指定其他的编码方法:

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt", true,
    Encoding.ASCII);
```

在这个构造函数中, 第二个参数是布尔型, 表示文件是否应以追加方式打开。奇怪的是, 构造函数的参数不能仅是一个文件名和一个编码类。

当然, 可以把 **StreamWriter** 关联到一个文件流上, 以获得打开文件的更多控制选项:

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
    FileMode.CreateNew, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
```

FileInfo 不执行返回 **StreamWriter** 类的任何方法。

另外, 如果要新建一个文件, 并开始向它写入数据, 就可以使用下面的代码:

```
FileInfo myFile = new FileInfo(@"C:\My Documents\NewFile.txt");
StreamWriter sw = myFile.CreateText();
```

与其他流类一样, 在使用完后, 要关闭 **StreamWriter** 类:

```
sw.Close();
```

写入流可以使用 **StreamWriter.Write()** 方法的 17 个重载版本来完成。最简单的方式是输出一个字符串:

```
string nextLine = "Groovy Line";
sw.Write(nextLine);
```

也可以输出单个字符:

```
char nextChar = 'a';
sw.Write(nextChar);
```

也可以输出一个字符数组:

```
char [] charArray = new char[100];
// initialize these characters
sw.Write(charArray);
```

甚至可以输出字符数组的一部分:

```
int nCharsToWrite = 50;
int startAtLocation = 25;
```

```
char [] charArray = new char[100];
// initialize these characters

sw.Write(charArray, startAtLocation, nCharsToWrite);
```

3. ReadWriteText 示例

ReadWriteText 示例说明了 StreamReader 和 StreamWriter 类的用法。它非常类似于前面的 ReadBinaryFile 示例,但假定要读取的文件是一个文本文件,并显示其内容。它还可以保存文件(包括在文本框中对文本进行的修改)。它将以 Unicode 格式保存任何文件。

图 29-11 所示的 ReadWriteText 用于显示前面的同一个 Default.aspx 文件。但这次读取内容会更容易一些。

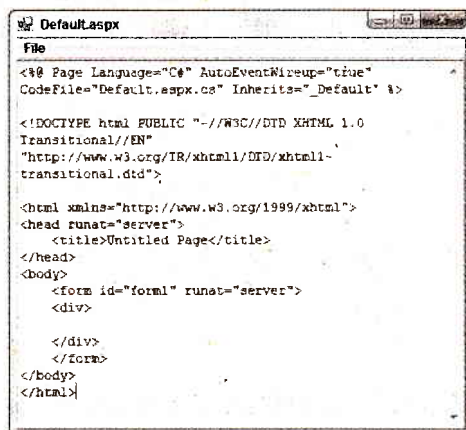


图 29-11

这里不打算介绍给 Open File 对话框添加事件处理程序的详细内容,因为它们基本上与前面的 BinaryFileReader 示例相同。与这个示例相同,打开一个新文件,将调用 DisplayFile()方法。其唯一的区别是 DisplayFile 的实现方式,以及本例有一个保存文件的选项。这由另一个菜单项 Save 来表示,这个选项的处理程序调用添加到代码中的另一个方法 SaveFile() (注意,这个新文件总是覆盖原始文件——这个示例没有写入另一个文件的选项)。

首先看看 SaveFile()方法,因为它是最简单的函数。首先利用 StreamWriter.WriteLine()方法把文本框中的每行文本依次写入 StreamWriter 流,以在每行文本的最后加上回车换行符:



可从
wrox.com
下载源代码

```
void SaveFile()
{
    StreamWriter sw = new StreamWriter(chosenFile, false, Encoding.Unicode);

    foreach (string line in textBoxContents.Lines)
        sw.WriteLine(line);

    sw.Close();
}
```

代码下载 ReadWriteText.sln

chosenFile 是主窗体的一个字符串字段,它包含已经读取的文件的名称(与前面的示例一样)。注意在打开流时指定 Unicode 编码方式。如果要以其他格式写入文件,则只需要改变该参数的值。如

果要把文本追加到文件中，这个构造函数的第二个参数就设置为 `true`，但本例不是这样。在构造时必须为 `StreamWriter` 设置编码方式，它随后可以用作只读属性 `Encoding`。

下面介绍文件的读取方式。读取过程比较复杂，因为我们不知道要读取的文件中包含多少行文本(例如，文件中包含多少个 `(char)13 (char)10` 序列，因为 `char(13) char(10)` 是行尾的回车换行符)。解决这个问题的方式是，先把文件读入 `StringCollection` 类的一个实例中，该类在 `System.Collections.Specialized` 名称空间中，主要用于保存可动态扩展的一组字符串。它实现的两个方法是我们感兴趣的：`Add()` 方法把字符串添加到集合中，和 `CopyTo()` 方法把字符串集合复制到一个正常数组(一个 `System.Array` 实例)中。`StringCollection` 对象的每个元素包含文件中的一行文本。

`DisplayFile()` 方法调用另一个方法 `ReadFileIntoStringCollection()`，后一个方法实际上读取文件。之后，就知道文件中有多少行文本，因此把 `StringCollection` 复制到大小固定的正常数组中，并把数组中的内容填充到文本框中。在进行复制时，因为只复制了字符串的引用，没有复制字符串本身，所以该过程的执行效率很高：



可从
wrox.com
下载源代码

```
void DisplayFile()
{
    StringCollection linesCollection = ReadFileIntoStringCollection();
    string [] linesArray = new string[linesCollection.Count];
    linesCollection.CopyTo(linesArray, 0);
    this.textBoxContents.Lines = linesArray;
}
```

代码下载 [ReadWriteText.sln](#)

`StringCollection.CopyTo()` 方法的第二个参数表示目标数组中的下标，我们从该下标指定的位置开始复制集合。

下面看看 `ReadFileIntoStringCollection()` 方法。使用 `StreamReader` 读取每一行文本。编译时需要计算读取的字符数，以确保不超出文本框的容量：



可从
wrox.com
下载源代码

```
StringCollection ReadFileIntoStringCollection()
{
    const int MaxBytes = 65536;
    StreamReader sr = new StreamReader(chosenFile);
    StringCollection result = new StringCollection();
    int nBytesRead = 0;
    string nextLine;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nBytesRead += nextLine.Length;
        if (nBytesRead > MaxBytes)
            break;
        result.Add(nextLine);
    }
    sr.Close();
    return result;
}
```

代码下载 [ReadWriteText.sln](#)

这就是该示例的完整代码。

如果运行 `ReadWriteText`，读取 `Default.aspx` 文件，然后保存它，该文件的格式就是 `Unicode`。任

何常用的 Windows 应用程序(Notepad, Wordpad)都不能分辨这种格式, 甚至 ReadWriteText 示例也只能在 Windows 的大多数版本下正确地读取和显示文件, 尽管因为 Windows 9x 不支持 Unicode, 像 Notepad 这样的应用程序不能识别其他平台上的 Unicode 文件(如果从 Wrox 网站 www.wrox.com 上下载了这个示例, 就可以试试)。但是, 如果使用前面的 BinaryFileReader 示例显示文件, 就会立即看出它们的区别, 如图 29-12 所示。最前面的两个字节表示 Unicode 格式的文件是可见的, 之后, 每个字符都用两个字节来表示。这非常明显, 因为在这个文件中, 每个字符的高位字节都是 0, 所以每隔一个字节就显示 x00。

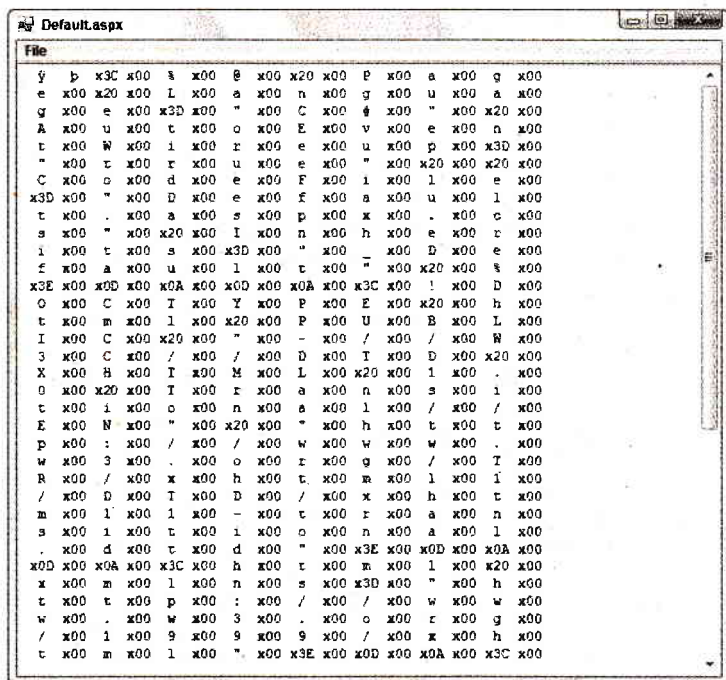


图 29-12

29.4 映射内存的文件

如果用户仅使用托管代码来编程, 映射内存的文件就可能是一个全新的概念。.NET Framework 4 现在引入了 System.IO.MemoryMappedFiles 名称空间, 把映射内存的文件包含为构建应用程序的工具集的一部分。

对底层的 Windows API 执行一些平台调用操作时, 总是会使用映射内存的文件这个概念。但现在引入了 System.IO.MemoryMappedFiles 名称空间后, 就可以使用托管代码, 而不是操作繁琐的平台调用。

应用程序需要频繁地或随机地访问文件时, 最好使用映射内存的文件和这个名称空间。使用这种方式允许把文件的一部分或者全部加载到一段虚拟内存上, 这些文件内容会显示给应用程序, 就好像这个文件包含在应用程序的主内存中一样。

有趣的是, 可以把内存中的这个文件用作多个进程的共享资源。在此之前, 用户可能使用 WCF

或命名管道与多个进程之间的共享资源通信，但现在可以在使用共享名称的进程之间共享映射内存的文件。

为了使用映射内存的文件，必须使用两个对象。第一个是映射内存的文件实例，它用于加载文件。第二个是访问器对象。下面的代码写入映射内存的文件对象，再读取它。还可以看出，在释放对象时也会执行写入操作：



可从
wrox.com
下载源代码

```
using System;
using System.IO.MemoryMappedFiles;
using System.Text;

namespace MappedMemoryFiles
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var mmFile= MemoryMappedFile.CreateFromFile(
                @"C:\Users\Bill\Document\Visual Studio 10\Projects\
                MappedMemoryFiles\MappedMemoryFiles\TextFile1.txt",
                System.IO.FileMode.Create, "fileHandle", 1024 * 1024))
            {
                string valueToWrite = "Written to the mapped-memory file on " +
                    DateTime.Now.ToString();
                var myAccessor = mmFile.CreateViewAccessor();

                myAccessor.WriteArray<byte>(0,
                    Encoding.ASCII.GetBytes(valueToWrite), 0,
                    valueToWrite.Length);

                var readOut = new byte[valueToWrite.Length];
                myAccessor.ReadArray<byte>(0, readOut, 0, readOut.Length);
                var finalValue = Encoding.ASCII.GetString(readOut);

                Console.WriteLine("Message: " + finalValue);
                Console.ReadLine();
            }
        }
    }
}
```

代码下载 [MappedMemoryFiles.sln](#)

在这个例子中，使用 `CreateFromFile()` 方法从物理文件中创建一个映射内存的文件。除了映射内存的文件之外，还需要为这个映射创建一个访问器对象，如下面的代码所示：

```
var myAccessor = mmFile.CreateViewAccessor();
```

有了访问器之后，就可以读写这个映射内存的位置，如代码示例所示。

也可以给同一个映射内存的位置创建多个访问器，如下面的代码所示：

```
var myAccessor1 = mmFile.CreateViewAccessor();
var myAccessor2 = mmFile.CreateViewAccess
```

29.5 读取驱动器信息

除了处理文件和目录之外，.NET Framework 还可以从指定的驱动器中读取信息。这使用 `DriveInfo` 类实现。`DriveInfo` 类可以扫描系统，提供可用驱动器的列表，还可以进一步提供任何驱动器的大量细节。

为了举例说明 `DriveInfo` 类的用法，创建一个简单的 Windows 窗体，列出计算机上的所有可用驱动器，再提供用户选择的驱动器的详细信息。Windows 窗体包含一个简单的 `ListBox`，如图 29-13 所示。



图 29-13

设置好窗体后，其代码包含两个事件，一个在窗体加载时引发，另一个在最终用户从列表框中选择驱动器时引发。这个窗体的代码如下所示：



可从
wrox.com
下载源代码

```
using System;
using System.IO;
using System.Windows.Forms;

namespace DriveViewer
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            DriveInfo[] di = DriveInfo.GetDrives();

            foreach (DriveInfo itemDrive in di)
            {
                listBox1.Items.Add(itemDrive.Name);
            }
        }

        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            DriveInfo di = new DriveInfo(listBox1.SelectedItem.ToString());
        }
    }
}
```

```

MessageBox.Show("Available Free Space: "
    + di.AvailableFreeSpace + "\n" +
    "Drive Format: " + di.DriveFormat + "\n" +
    "Drive Type: " + di.DriveType + "\n" +
    "Is Ready: " + di.IsReady + "\n" +
    "Name: " + di.Name + "\n" +
    "Root Directory: " + di.RootDirectory + "\n" +
    "ToString() Value: " + di + "\n" +
    "Total Free Space: " + di.TotalFreeSpace + "\n" +
    "Total Size: " + di.TotalSize + "\n" +
    "Volume Label: " + di.VolumeLabel, di.Name +
    " DRIVE INFO");
}
}

```

代码下载 DriveViewer.sln

首先, 用 `using` 关键字导入 `System.IO` 名称空间。在 `Form1_Load` 事件中使用 `DriveInfo` 类获取系统上所有可用驱动器的列表。这需要使用 `DriveInfo` 对象数组, 再用 `DriveInfo.GetDrives()` 方法填充这个数组。接着使用 `foreach` 循环遍历找到的每个驱动器, 用循环的结果填充列表框。这会生成如图 29-14 所示的结果。



图 29-14

这个窗体允许最终用户在列表框中选择一个驱动器。选择好驱动器后, 就显示一个消息框, 其中包含所选驱动器的信息。如图 29-15 所示, 在当前的计算机上有 6 个驱动器。选择其中几个驱动器会生成如图 29-15 所示的消息框。



图 29-15

在这里可以看出, 这些消息框列出了 3 个完全不同的驱动器的详细信息。第一个驱动器 C:\ 是硬盘, 因为消息框显示它的驱动器类型是 Fixed。第二个驱动器 D:\ 是 CD/DVD 驱动器, 第三个驱动器 F:\ 是 USB 笔, 其驱动器类型是 Removable。

29.6 文件的安全性

在第一次引入 .NET Framework 1.0/1.1 时, 它不能方便地访问和使用文件、目录和注册表键的访问控制列表(ACL)。那时, 要访问 ACL, 通常要进行一些 COM 交互操作, 所以需要使用 ACL 的高级编程技巧。

自 .NET Framework 2.0 发布以来, 这有了很大的变化, 因为这个版本通过 System.Security.AccessControl 名称空间, 使 ACL 的使用变得非常容易。利用这个名称空间, 可以为文件、注册表键、网络共享、Active Directory 对象等处理安全设置。

29.6.1 从文件中读取 ACL

对于一个使用 System.Security.AccessControl 名称空间的示例中, 本节将为文件和目录使用 ACL。首先了解如何查看指定文件的 ACL。这个示例在一个控制台应用程序中完成, 如下所示:



可从
wrox.com
下载源代码

```
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ReadingACLs
{
    internal class Program
    {
        private static string myFilePath;

        private static void Main()
        {
            Console.WriteLine("Provide full file path: ");
            myFilePath = Console.ReadLine();

            try
            {
                using (FileStream myFile =
                    new FileStream(myFilePath, FileMode.Open, FileAccess.Read))
                {
                    FileSecurity fileSec = myFile.GetAccessControl();

                    foreach (FileSystemAccessRule fileRule in
                        fileSec.GetAccessRules(true, true,
                            typeof (NTAccount)))
                    {
                        Console.WriteLine("{0} {1} {2} access for {3}",
                            myFilePath,
                            fileRule.AccessControlType ==
                                AccessControlType.Allow
                                ? "provides": "denies",
                            fileRule.FileSystemRights,
```

```

        fileRule.IdentityReference);
    }
}
catch
{
    Console.WriteLine("Incorrect file path given!");
}

Console.ReadLine();
}
}

```

代码下载 ReadingACLs.sln

为了使这个示例正常工作，首先引用 `System.Security.AccessControl` 名称空间，这样就可以在程序的后面访问 `FileSecurity` 和 `FileSystemAccessRule` 类。

在检索到指定的文件并把它放在 `FileStream` 对象中后，就使用 `File` 对象上的 `GetAccessControl()` 方法获取该文件的 ACL。`GetAccessControl()` 方法中的这些信息放在 `FileSecurity` 类中。这个类有对引用项的访问权限。每个访问权限都用一个 `FileSystemAccessRule` 对象表示。所以要使用 `foreach` 循环遍历在 `FileSecurity` 对象中找到的所有访问权限。

对根目录的一个简单文本文件运行这个示例，结果如下所示：

```

Provide full file path: C:\Sample.txt
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides ReadAndExecute, Synchronize access for BUILTIN\Users
C:\Sample.txt provides Modify, Synchronize access for
NT AUTHORITY\Authenticated Users

```

下一节介绍读取目录的 ACL，而不是文件的 ACL。

29.6.2 从目录中读取 ACL

读取目录(而不是文件)的 ACL 信息，与前面的示例没有什么区别，其代码如下所示：



可从
wrox.com
下载源代码

```

using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ConsoleApplication1
{
    internal class Program
    {
        private static string mentionedDir;

        private static void Main()
        {
            Console.Write("Provide full directory path: ");
            mentionedDir = Console.ReadLine();

            try

```

```

    DirectoryInfo myDir = new DirectoryInfo(mentionedDir);

    if (myDir.Exists)
    {
        DirectorySecurity myDirSec = myDir.GetAccessControl();

        foreach (FileSystemAccessRule fileRule in
            myDirSec.GetAccessRules(true, true,
                typeof (NTAccount)))
        {
            Console.WriteLine("{0} {1} {2} access for {3}",
                mentionedDir, fileRule.AccessControlType ==
                    AccessControlType.Allow
                    ? "provides": "denies",
                    fileRule.FileSystemRights,
                    fileRule.IdentityReference);
        }
    }
}
catch
{
    Console.WriteLine("Incorrect directory provided!");
}

Console.ReadLine();
}
}

```

代码下载 ReadingACLsFromDirectory.sln

这个示例的不同之处是，它使用 `DirectoryInfo` 类，该类还包含一个 `GetAccessControl()` 方法，来提取目录的 ACL 信息。在 Windows 7 下运行这个例子的结果如下所示：

```

Provide full directory path: C:\Test
C:\Test provides FullControl access for BUILTIN\Administrators
C:\Test provides 268435456 access for BUILTIN\Administrators
C:\Test provides FullControl access for NT AUTHORITY\SYSTEM
C:\Test provides 268435456 access for NT AUTHORITY\SYSTEM
C:\Test provides ReadAndExecute, Synchronize access for BUILTIN\Users
C:\Test provides Modify, Synchronize access for
    NT AUTHORITY\Authenticated Users
C:\Test provides -536805376 access for NT AUTHORITY\Authenticated Users

```

ACL 的最后一个用法是使用 `System.Security.AccessControl` 名称空间添加和删除文件的 ACL 中的项。

29.6.3 添加和删除文件中的 ACL 项

还可以使用与前面示例相同的对象处理资源的 ACL。下面的示例修改了前面读取文件的 ACL 信息的代码。在这个示例中，读取指定文件的 ACL，修改后再次读取它：

```
try
```

```

using (FileStream myFile = new FileStream(myFilePath,
    FileMode.Open, FileAccess.ReadWrite))
{
    FileSecurity fileSec = myFile.GetAccessControl();

    Console.WriteLine("ACL list before modification:");
    foreach (FileSystemAccessRule fileRule in
        fileSec.GetAccessRules(true, true,
            typeof(System.Security.Principal.NTAccount)))
    {
        Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
            fileRule.AccessControlType == AccessControlType.Allow ?
            "provides": "denies",
            fileRule.FileSystemRights,
            fileRule.IdentityReference);
    }

    Console.WriteLine();
    Console.WriteLine("ACL list after modification:");

    FileSystemAccessRule newRule = new FileSystemAccessRule(
        new System.Security.Principal.NTAccount(@"PUSHKIN\Tuija"),
        FileSystemRights.FullControl,
        AccessControlType.Allow);

    fileSec.AddAccessRule(newRule);
    File.SetAccessControl(myFilePath, fileSec);

    foreach (FileSystemAccessRule fileRule in
        fileSec.GetAccessRules(true, true,
            typeof(System.Security.Principal.NTAccount)))
    {
        Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
            fileRule.AccessControlType == AccessControlType.Allow ?
            "provides": "denies",
            fileRule.FileSystemRights,
            fileRule.IdentityReference);
    }
}
}

```

在这个示例中，给文件的 ACL 添加了一条新的访问规则。这使用 `FileSystemAccessRule` 对象完成。`FileSystemAccessRule` 类是一个抽象的访问控制项(ACE)实例。ACE 定义了要使用的用户账户，这个用户账户可以处理的访问类型，以及是允许还是拒绝这个访问。在新建这个对象的实例时，新建一个 `NTAccount` 对象，并给文件赋予 Full Control 权限。即使新建了 `NTAccount` 对象，它仍必须引用已有用户。接着使用 `FileSecurity` 类的 `AddAccessRule()` 方法赋予新规则。之后，使用 `FileSecurity` 对象引用，通过 `File` 类的 `SetAccessControl()` 方法给当前文件设置访问控制。

接着，再次列出文件的 ACL。下面是上述代码的执行结果的一个示例：

```

Provide full file path: C:\Users\Bill\Sample.txt
ACL list before modification:
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for BUILTIN\Administrators

```

```
C:\Sample.txt provides FullControl access for PUSHKIN\Bill

ACL list after modification:
C:\Sample.txt provides FullControl access for PUSHKIN\Tuija
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for PUSHKIN\Bill
```

要从 ACL 列表中删除规则，并不需要对代码进行很多修改。在上面的代码示例中，只需把下面的一行

```
fileSec.AddAccessRule(newRule);
```

改为：

```
fileSec.RemoveAccessRule(newRule);
```

就可以删除刚才添加的规则。

29.7 读写注册表

自 Windows 95 以来的所有 Windows 版本中，注册表是包含 Windows 安装、用户首选项，以及已安装软件和设备的所有配置信息的核心存储库。目前，几乎所有的商用软件都使用注册表来存储这些信息，COM 组件必须把它们的信息存储在注册表中，才能由客户端调用。 .NET Framework 及其无干扰安装概念略微减弱了注册表对于应用程序的重要性，因为程序集是完全自包含的，所以特定程序集的信息不需要放在注册表中，即使是共享程序集也是这样。另外， .NET Framework 引入了独立存储器的概念，通过它应用程序可以在文件中存储专用于每个用户的信息， .NET Framework 将确保为每个在机器上注册的用户单独地存储数据。

应用程序现在使用 Windows Installer 来安装，开发人员不再需要直接操作常常涉及安装应用程序的注册表。但是，如果发布任何完整的应用程序，则应用程序也可能要使用注册表来存储其配置信息。例如，如果应用程序要显示在控制面板的 Add/Remove Programs 对话框中，则仍需要使用相应的注册表项。还需要使用注册表处理与遗留代码的向后兼容性。

注册表的库和 .NET 库一样复杂，它包括访问注册表的类。其中有两个类涉及注册表，即 Registry 和 RegistryKey，这两个类都在 Microsoft.Win32 名称空间中。在介绍这两个类前，先简要介绍一下注册表本身的结构。

29.7.1 注册表

注册表的层次结构非常类似于文件系统的层次结构。查看和修改注册表内容的一般方式是使用 regedit 或 regedt32 实用程序。当然，自从 Windows 95 以后，regedit 在所有的 Windows 版本中都有。而 regedt32 则在 Windows NT 和 Windows 2000 中才有，其用户友好性不如 regedit，但可以访问 regedit 不能查看的安全性信息。Windows Server 2003 把 regedit 和 regedt32 合并为一个新的编辑器 regedit。这里只讨论 Windows 7 中的 regedit，在 Run 对话框或命令行提示符中输入 regedit，即可启动它。

图 29-16 显示了第一次启动 regedit 的屏幕。

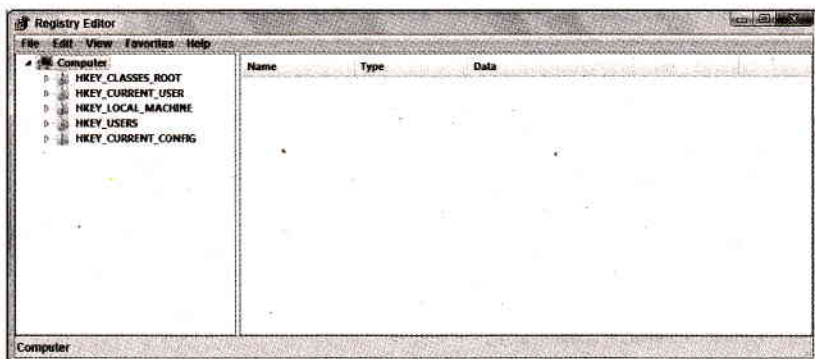


图 29-16

regedit 的树形视图/列表视图风格的用户界面非常类似于 Windows Explorer，与注册表本身的层次结构相匹配。但它们有一些重要区别。

在文件系统中，最上面的节点是磁盘的分区 C:\、D:\等。在注册表中，最上面的节点是注册表配置单元(registry hive)，已有的配置单元是不能改变的——它们是固定的，有 7 个注册表配置单元(但使用 regedit 只能看到其中的 5 个)：

- HKEY_CLASSES_ROOT(HKCR)包含系统上文件类型的细节(.txt、.doc 等)，以及使用哪些应用程序可以打开每种文件。它还包含所有 COM 组件的注册信息(后者通常是注册表中最大的一个区域，因为目前的 Windows 带有非常多的 COM 组件)。
- HKEY_CURRENT_USER(HKCU)包含用户目前登录的计算机的用户配置。这些设置包括桌面设置、环境变量、网络 and 打印机连接，以及其他定义用户的操作环境的设置。
- HKEY_LOCAL_MACHINE(HKLM)是一个很大的配置单元，其中包含所有安装到计算机上的软件和硬件信息，这些设置不是用户特有的，而是可用于所有登录到计算机上的用户。它还包含 HKCR 配置单元：HKCR 实际上并不是一个独立的配置单元，而只是一个对注册表键 HKLM/SOFTWARE/Classes 的方便映射。
- HKEY_USERS(HKUSR)包含所有用户的用户首选项。它还包含 HKCU 配置单元，HKCU 配置单元是对 HKEY_USERS 中一个键的映射。
- HKEY_CURRENT_CONFIG(HKCF)包含计算机上硬件的详细信息。

其余的两个键包含临时信息，这些信息常常会更改：

- HKEY_DYN_DATA 是一个一般容器，包含需要存储在注册表中的任何易失性数据。
- HKEY_PERFORMANCE_DATA 包含与运行应用程序的性能相关的信息。

在这些配置单元中，具有注册表键的一个树型结构。每个键在许多方面都类似于文件系统中的文件夹或文件。但是，它们有一个重要区别：文件系统可以区分文件(文件中包含数据)和文件夹(其中主要包含其他文件夹或文件)，但注册表中只有键。键可以包含数据和其他键。

如果键中包含数据，这个键就表示为一列值。每个值都有一个相关的名称、数据类型和数据，另外，键还可以有默认值，这个值没有名称。

使用 regedit 可以查看这个结构，了解其中的注册表键。图 29-17 显示了 HKCU/Control Panel/Appearance 键中的内容，其中包含当前登录用户所选的颜色主题的详细信息。regedit 在树型视图中用一个打开的文件夹图标来显示要查看的键。

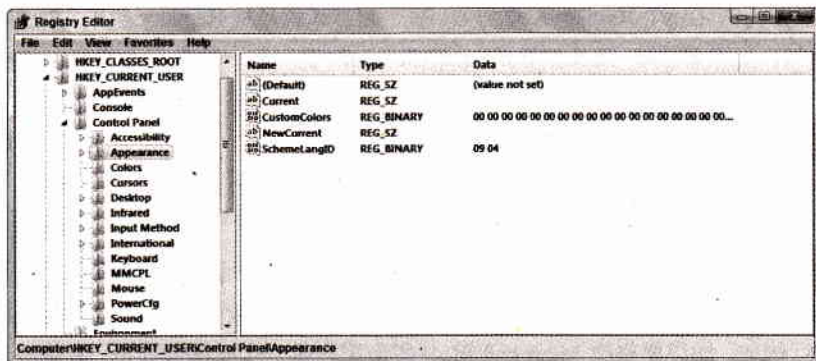


图 29-17

HKCU/Control Panel/Appearance 键有 3 个命名的值集，但其默认值不包含任何数据。在图 29-17 中，标记为 Type 的列显示了每个值的数据类型。注册表项可以格式化为这 3 个数据类型中的一个。这些类型分别是：

- REG_SZ(大致相当于.NET 字符串实例——这种匹配并不精确，因为注册表项的数据类型不是.NET 数据类型)
- REG_DWORD(大致相当于 uint)
- REG_BINARY(大致相当于字节数组)

为此，要在注册表中存储数据的应用程序会创建许多注册表键，通常把它们存储在 HKLM\Software\<CompanyName>键中。注意，这些键不一定包含数据。有时键的存在就可以给应用程序提供所需的足够信息。

29.7.2 .NET 注册表类

要访问注册表，可以使用 Microsoft.Win32 名称空间中的两个类 Registry 和 RegistryKey。RegistryKey 实例表示一个注册表键。这个类实现的方法可以浏览子键、创建新键、读取或修改键中的值。换言之，该类通常可以完成对注册表键进行的所有操作，包括设置键的安全级别。RegistryKey 是处理注册表用得最多的类。Registry 类只能对注册表键进行单一的访问，以执行简单的操作。Registry 类的另一个作用是提供表示顶级键的 RegistryKey 实例(不同的配置单元)，以便开始在注册表中定位。Registry 类通过静态属性来提供这些实例，这些属性共有 7 个，分别是 ClassesRoot、CurrentConfig、CurrentUser、DynData、LocalMachine、PerformanceData 和 Users。用户可以很快猜出它们分别与哪个配置单元相对应。

例如，要获得一个表示 HKLM 键的 RegistryKey 实例，可以编写下面的代码：

```
RegistryKey hklm = Registry.LocalMachine;
```

获得 RegistryKey 对象的引用的过程，视为打开对应键。

用户可能会认为，因为注册表的层次结构类似于文件系统的层次结构，所以 RegistryKey 类提供的方法类似于 DirectoryInfo 类实现的方法，但实际上并非如此。访问注册表的方式通常不同于使用文件和文件夹的方式，RegistryKey 类实现的方法反映了这种不同。

最明显的区别是如何在注册表的给定位置上打开一个注册表键。Registry 类没有用户可以使用公共构造函数，也没有直接通过键的名称来访问键的方法。但可以从相关的配置单元中从上至下

浏览该键。如果要实例化一个 **RegistryKey** 对象,唯一的方式就是从 **Registry** 类的对应静态属性开始,向下浏览。例如,如果要读取 **HKLM/Software/Microsoft** 键中的一些数据,就可以使用下面的代码获得它的一个引用:

```
RegistryKey hkml = Registry.LocalMachine;  
RegistryKey hkSoftware = hkml.OpenSubKey("Software");  
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft");
```

以这种方式访问注册表键是只读访问。如果要写入该键(包括写入其值,或创建和删除其子键),就需要使用 **OpenSubKey** 的另一个重写方法,该方法的第二个参数是 **bool** 类型,表示是否要对该键进行读写访问。例如,如果要修改 **Microsoft** 键(并假定用户是一个系统管理员,有修改该键的许可),就应编写如下代码:

```
RegistryKey hkml = Registry.LocalMachine;  
RegistryKey hkSoftware = hkml.OpenSubKey("Software");  
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft", true);
```

因为这个键包含 **Microsoft** 的应用程序使用的信息,所以在大多数情况下不应修改这个特殊的键。

如果希望这个键存在,就应调用 **OpenSubKey()** 方法。如果这个键不存在,它就返回一个空引用。如果要创建一个键,就应使用 **CreateSubKey()** 方法(该方法会通过返回的引用,自动提供该键的读写访问):

```
RegistryKey hkml = Registry.LocalMachine;  
RegistryKey hkSoftware = hkml.OpenSubKey("Software");  
RegistryKey hkMine = hkSoftware.CreateSubKey("MyOwnSoftware");
```

CreateSubKey() 方法的工作方式非常有趣:如果键不存在,它就创建这个键。但如果键已经存在,它就会返回一个表示该键的 **RegistryKey** 实例。这个方法采用这样的工作方式,其原因与用户通常使用这个键的方式有关。注册表整体上包含长期数据,如 **Windows** 和各种应用程序的配置信息。因此用户并不需要经常显式地创建键。

更常见的是,应用程序需要确保某些数据存在于注册表中。换言之,如果这些数据不存在,就要创建相关的键,但如果它们存在,就不需要做任何事。**CreateSubKey()** 方法就可以完成这项任务。与 **FileInfo.Open()** 方法的情况不同,例如, **CreateSubKey()** 不会偶然地删除任何数据。如果要删除注册表键,就需要调用 **RegistryKey.Delete()** 方法,因为注册表对于 **Windows** 非常重要。当调试 C# 注册表调用时,如果删除一些重要的键,就会不经意间完全中断 **Windows**。

定位了要读取或修改的注册表键后,就可以使用 **SetValue()** 或 **GetValue()** 方法设置或获取该键中的数据。这两个方法的参数都是一个字符串,其中字符串给出了参数值的名称, **SetValue()** 方法还需要另一个包含值的详细信息的对象引用。因为这个参数定义为对象引用,所以它实际上可以是任何一个类的引用。**SetValue()** 方法根据所提供的类的类型,确定把值设置为 **REG_SZ**、**REG_DWORD**, 还是 **REG_BINARY**。例如:

```
RegistryKey hkMine = HkSoftware.CreateSubKey("MyOwnSoftware");  
hkMine.SetValue("MyStringValue", "Hello World");  
hkMine.SetValue("MyIntValue", 20);
```

这段代码用两个值设置该键: `MyStringValue` 的类型是 `REG_SZ`, 而 `MyIntValue` 的类型是 `REG_DWORD`, 这里只考虑这两种类型, 在后面的示例中会使用它们。

`RegistryKey.GetValue()` 方法的工作方式也是这样。它返回一个对象引用, 如果该方法检测到值的类型为 `REG_SZ`, 它实际上就返回一个字符串引用; 如果值的类型为 `REG_DWORD`, 它就返回一个 `int` 型值。

```
string stringValue = (string)hkMine.GetValue("MyStringValue");
int intValue = (int)hkMine.GetValue("MyIntValue");
```

最后, 完成了读取或修改数据后, 应关闭该键:

```
hkMine.Close();
```

`RegistryKey` 类实现了许多方法和属性。表 29-5 和表 29-6 列出了其中最有用的方法和属性。

表 29-5

属 性	作 用
<code>Name</code>	键的名称(只读)
<code>SubKeyCount</code>	键的子键个数
<code>ValueCount</code>	键包含的值的个数

表 29-6

方 法	作 用
<code>Close()</code>	关闭键
<code>CreateSubKey()</code>	创建给定名称的子键(如果该子键已经存在, 就打开它)
<code>DeleteSubKey()</code>	删除指定的子键
<code>DeleteSubKeyTree()</code>	递归删除子键及其所有子键
<code>DeleteValue()</code>	从键中删除一个指定的值
<code>GetAccessControl()</code>	返回指定注册表键的 ACL, 该方法是 .NET Framework 2.0 中增加的
<code>GetSubKeyNames()</code>	返回包含子键名称的字符串数组
<code>GetValue()</code>	返回指定的值
<code>GetValueKind()</code>	返回指定的值, 要检索其注册表数据类型, 该方法是 .NET Framework 2.0 中增加的
<code>GetValueNames()</code>	返回一个包含所有键值名称的字符串数组
<code>OpenSubKey()</code>	返回表示给定子键的 <code>RegistryKey</code> 实例的引用
<code>SetAccessControl()</code>	把 ACL 应用于指定的注册表键
<code>SetValue()</code>	设置指定的值

29.8 读写独立存储器

除了读写注册表之外, 还可以读写独立存储器中的值。如果在写入注册表或磁盘时一般有问题, 就可以使用独立存储器。它可以用于轻松地存储应用程序状态或用户设置。

独立存储器可以看作一个虚拟磁盘，在其中可以保存只能由创建它们的应用程序或与其他应用程序实例共享的数据项。独立存储器的访问类型有两种。第一种是由用户和程序集访问。

在用户和程序集访问独立存储器时，计算机上有一个存储器位置，它可以由多个应用程序实例访问。这种访问通过用户身份和应用程序(或程序集)身份来保证，如图 29-18 所示。

这说明，同一个应用程序的多个实例可以在同一个存储器上工作。独立存储器的第二种访问类型是由用户、程序集和域来访问。在这种访问类型中，每个应用程序实例都在它自己的独立存储器中工作，详细内容如图 29-19 所示。

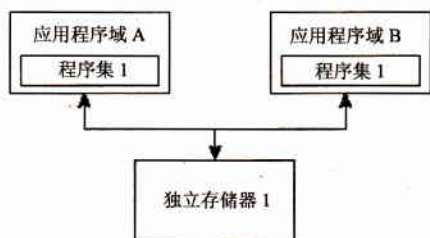


图 29-18

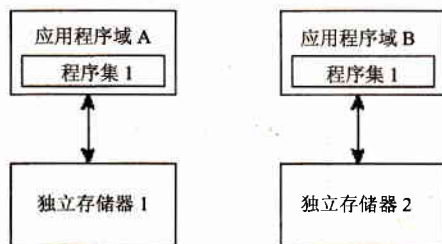


图 29-19

在这种情况下，每个应用程序实例都在它自己的存储器中工作，每个应用程序实例记录的设置都只与它自己相关。这是独立存储器的一种更精细的方法。

为了举例说明如何在 Windows 窗体应用程序中使用独立存储器(尽管也可以在 ASP.NET 应用程序中使用它)，可以使用 `ReadSettings()` 和 `SaveSettings()` 方法从独立存储器中读写值，而不是直接在注册表中读写值。



这里只列出了 `ReadSettings()` 和 `SaveSettings()` 方法中的代码，注意这一点很重要。该应用程序还有更多的代码，参见下载代码文件中示例 `SelfPlacingWindow` 的其他代码。

首先，需要重写 `SaveSettings()` 方法。为了使代码正常工作，需要添加下面的 `using` 指令：

```
using System.IO;
using System.IO.IsolatedStorage;
using System.Text;
```

`SaveSettings()` 方法详见下面的代码示例：

```
void SaveSettings()
{
    IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
    IsolatedStorageFileStream storStream = new
        IsolatedStorageFileStream("SelfPlacingWindow.xml",
            FileMode.Create, FileAccess.Write);

    System.Xml.XmlTextWriter writer = new
        System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
    writer.Formatting = System.Xml.Formatting.Indented;

    writer.WriteStartDocument();
    writer.WriteStartElement("Settings");
```

```

writer.WriteStartElement("BackColor");
writer.WriteValue(BackColor.ToKnownColor().ToString());
writer.WriteEndElement();

writer.WriteStartElement("Red");
writer.WriteValue(BackColor.R);
writer.WriteEndElement();

writer.WriteStartElement("Green");
writer.WriteValue(BackColor.G);
writer.WriteEndElement();

writer.WriteStartElement("Blue");
writer.WriteValue(BackColor.B);
writer.WriteEndElement();

writer.WriteStartElement("Width");
writer.WriteValue(Width);
writer.WriteEndElement();

writer.WriteStartElement("Height");
writer.WriteValue(Height);
writer.WriteEndElement();

writer.WriteStartElement("X");
writer.WriteValue(DesktopLocation.X);
writer.WriteEndElement();

writer.WriteStartElement("Y");
writer.WriteValue(DesktopLocation.Y);
writer.WriteEndElement();

writer.WriteStartElement("WindowState");
writer.WriteValue(WindowState.ToString());
writer.WriteEndElement();

writer.WriteEndElement();

writer.Flush();
writer.Close();

storStream.Close();
storFile.Close();
}

```

这些代码比使用注册表时略多一些，这主要是因为需要构建放在独立存储器中的 XML 文档。在这段代码中，第一个重要的地方是：

```

IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
IsolatedStorageFileStream storStream = new
    IsolatedStorageFileStream("SelfPlacingWindow.xml",
        FileMode.Create, FileAccess.Write);

```

这段代码使用访问的用户、程序集和域类型创建了 `IsolatedStorageFile` 的一个实例，使用 `IsolatedStorageFileStream` 对象创建了一个流，该对象创建虚拟文件 `SelfPlacingWindow.xml`。

之后创建一个 `XmlTextWriter` 对象，以构建 XML 文档，并将 XML 内容写入 `IsolatedStorageFileStream` 对象实例中。

```
System.Xml.XmlTextWriter writer = new
    System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
```

创建 `XmlTextWriter` 对象之后, 将所有的值逐个节点地写入 XML 文档中。当把所有内容写入 XML 文档中时, 关闭所有对象, 现在所有数据就都存储在独立存储器中。

从存储器中读取数据通过 `ReadSettings()` 方法实现。这个方法如下面的代码示例所示:

```
bool ReadSettings()
{
    IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
    string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

    foreach (string userFile in userFiles)
    {
        if(userFile == "SelfPlacingWindow.xml")
        {
            listBoxMessages.Items.Add("Successfully opened file " +
                userFile.ToString());

            StreamReader storStream =
                new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
                    FileMode.Open, storFile));
            System.Xml.XmlTextReader reader = new
                System.Xml.XmlTextReader(storStream);

            int redComponent = 0;
            int greenComponent = 0;
            int blueComponent = 0;

            int X = 0;
            int Y = 0;

            while (reader.Read())
            {
                switch (reader.Name)
                {
                    case "Red":
                        redComponent = int.Parse(reader.ReadString());
                        break;
                    case "Green":
                        greenComponent = int.Parse(reader.ReadString());
                        break;
                    case "Blue":
                        blueComponent = int.Parse(reader.ReadString());
                        break;
                    case "X":
                        X = int.Parse(reader.ReadString());
                        break;
                    case "Y":
                        Y = int.Parse(reader.ReadString());
                        break;
                    case "Width":
                        this.Width = int.Parse(reader.ReadString());
                        break;
                    case "Height":
                        this.Height = int.Parse(reader.ReadString());
                }
            }
        }
    }
}
```

```

        break;
    case "WindowState":
        this.WindowState = (FormWindowState)FormWindowState.Parse
            (WindowState.GetType(), reader.ReadString());
        break;
    default:
        break;
    }
}

this.BackColor =
    Color.FromArgb(redComponent, greenComponent, blueComponent);
this.DesktopLocation = new Point(X, Y);

listBoxMessages.Items.Add("Background color: " + BackColor.Name);
listBoxMessages.Items.Add("Desktop location: " +
    DesktopLocation.ToString());
listBoxMessages.Items.Add("Size: " + new Size(Width, Height).ToString());
listBoxMessages.Items.Add("Window State: " + WindowState.ToString());

storStream.Close();
storFile.Close();
}
}
return true;
}

```

使用 `GetFileNames()` 方法, `SelfPlacingWindow.xml` 文档会从独立存储器中弹出, 然后放在一个流中, 再使用 `XmlTextReader` 对象分析它。

```

IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

foreach (string userFile in userFiles)
{
    if(userFile == "SelfPlacingWindow.xml")
    {
        listBoxMessages.Items.Add("Successfully opened file " +
            userFile.ToString());

        StreamReader storStream =
            new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
                FileMode.Open, storFile));
    }
}

```

XML 文档包含在 `IsolatedStorageFileStream` 对象中后, 就使用 `XmlTextReader` 对象分析它:

```

System.Xml.XmlTextReader reader = new
    System.Xml.XmlTextReader(storStream);

```

之后, 通过 `XmlTextReader` 对象从流中读取它。然后元素值会放回在应用程序中。`SelfPlacingWindow` 示例使用注册表记录和检索应用程序状态值, 而使用独立存储器与使用注册表同样有效。应用程序会与以前一样记录颜色、大小和位置。

29.9 小结

本章介绍了如何在 C# 代码中使用 .NET 基类来访问注册表和文件系统。在这两种情况下，基类提供的对象模型比较简单，但功能强大，从而很容易执行这些领域中几乎所有的操作。对于文件系统，可以复制文件；移动、创建、删除文件和文件夹；读写二进制文件和文本文件。而对于注册表，可以创建、修改或读取键。

本章还介绍了独立存储器以及如何在应用程序中使用它们存储应用程序状态。

本章假定用户在有足够访问权限的账户上运行代码，以完成代码需要执行的任务。显然，安全性对于文件访问非常重要，详见第 21 章。

第Ⅳ部分

数 据

- 第 30 章 核心 ADO.NET
- 第 31 章 ADO.NET Entity Framework
- 第 32 章 数据服务
- 第 33 章 处理 XML
- 第 34 章 .NET 编程和 SQL Server

第 30 章

核心 ADO.NET

本章内容:

- 连接数据库
- 执行命令
- 调用存储过程
- ADO.NET 对象模型
- 使用 XML 和 XML 架构

本章讨论如何使用 ADO.NET 访问 C#程序中的数据, 主要介绍如何使用 `SqlConnection` 类和 `OleDbConnection` 类连接数据库, 以及断开与数据库的连接。深入讨论命令对象上的各种选项, 并说明如何为 `Sql` 类和 `OleDb` 类的每个选项使用命令。如何使用命令对象来调用存储过程, 这些存储过程的结果如何集成到缓存在客户端上的数据中。ADO.NET 对象模型与 ADO 中可用的对象完全不同, 本节将讨论 `DataSet`、`DataTable`、`DataRow` 和 `DataColumn` 类。`DataSet` 类也可以包含表之间、约束之间的关系。类层次结构在 .NET Framework 2.0 版本中有许多变化, 本章也将介绍这些变化。最后将讨论 XML 架构, 它是构建 ADO.NET 的基础。

下面首先简要介绍 ADO.NET。

30.1 ADO.NET 概述

ADO.NET 比现有 API 在技术上高出很多。它与 ADO 仅仅是名称类似, 类和访问数据的方法则完全不同。

ADO(ActiveX Data Objects)是一个 COM 组件库, 在过去的几年中, 这些组件有许多版本。ADO 主要包含 `Connection`、`Command`、`Recordset` 和 `Field` 对象。使用 ADO 时, 要打开与数据库的连接, 选择一些数据, 并把它们放在记录集中, 这些记录集由字段组成, 接着处理这些数据, 并在服务器上更新它们, 最后关闭连接。ADO 还引入了一个概念: 所谓的断开连接的记录集, 当不适合使连接打开相当长的时间时, 就可以使用该概念。

ADO 还有几个没有完全解决的问题, 其中最著名的就是笨拙的断开连接的记录集。对这个支持的需求要比以往“以 Web 为中心”的计算更高, 所以需要一种新方式。ADO.NET 和 ADO(不仅仅是名称)有许多相似之处, 所以从 ADO 升级到 ADO.NET 不会很困难。而且, 如果使用的是 SQL Server, 它有一组很好的托管类, 调整这些类可以很好地发挥出数据库的最佳性能。单是这一个理由, 就足以迁移到 ADO.NET 了。

ADO.NET 附带了 3 个数据库客户端名称空间，第 1 个用于 SQL Server，第 2 个用于 ODBC 数据源，第 3 个用于通过 OLE DB 实现的数据库。如果数据库不是 SQL Server，就应使用 OLE DB 路由，除非还能使用 ODBC。如果使用 Oracle 作为数据库，就可以访问 Oracle .NET Developer 站点，从 www.oracle.com/technology/tech/windows/odpnet/index.html 上获取其 .NET 提供程序 ODP.NET。

30.1.1 名称空间

本章所有的示例都以某种方式访问数据。表 30-1 所示的名称空间提供了在 .NET 数据访问中使用的类和接口。

表 30-1

名 称 空 间	说 明
System.Data	所有数据访问泛型类
System.Data.Common	各个数据提供程序共享(或重写)的类
System.Data.EntityClient	Entity Framework 类
System.Data.Linq.SqlClient	LINQ to SQL 提供程序类
System.Data.Odbc	ODBC 提供程序的类
System.Data.OleDb	OLE DB 提供程序的类
System.Data.ProviderBase	新的基类和连接工厂类
System.Data.Sql	用于 SQL Server 数据访问的新泛型接口和类
System.Data.SqlClient	Sql Server 提供程序的类
System.Data.SqlTypes	Sql Server 数据类型

下面两节列出 ADO.NET 中主要的类。

30.1.2 共享类

ADO.NET 包含许多类，无论是使用 SQL Server 类，还是使用 OLE DB 类，都可以使用它们。表 30-2 列出的类包含在 System.Data 名称空间中。

表 30-2

类	说 明
DataSet	这个对象主要用于断开连接，它可以包含一组 DataTable，以及这些表之间的关系
DataTable	数据的一个容器，DataTable 由一个或多个 DataColumn 组成，每个 DataColumn 由一个或多个包含数据的 DataRow 组成
DataRow	许多数值，类似于数据库表的一行，或电子表格中的一行。
DataColumn	该对象包含列的定义，如名称和数据类型
DataRelation	DataSet 类中两个 DataTable 类之间的链接，用于外键和主/从关系
Constraint	为 DataColumn 类(或一组数据列)定义规则，如唯一值

如表 30-3 所示，下面两个类包含在 System.Data.Common 名称空间中。

表 30-3

类	说 明
DataColumnMapping	将数据库中的列名映射到 DataTable 中的列名
DataTableMapping	将数据库中的表名映射到 DataSet 中的 DataTable

30.1.3 数据库专用类

除了上一节介绍的共享类外, ADO.NET 还包含许多数据库专用类。这些类实现一组在 System.Data 名称空间中定义的标准接口, 根据需要允许类按照一般形式来使用。例如, SqlConnection 类和 OleDbConnection 类派生于实现 IDbConnection 接口的 DbConnection 类。表 30-4 列出了数据库专用类。

表 30-4

类	说 明
SqlCommand、OleDbCommand 和 ODBCCommand	用作 SQL 语句或存储过程调用的包装器, SqlCommand 类的示例详见本章后面的内容
SqlCommandBuilder、OleDbCommandBuilder 和 ODBCCommandBuilder	用于从一条 SELECT 语句中生成 SQL 命令(如 INSERT、UPDATE 和 DELETE 语句)
SqlConnection、OleDbConnection 和 ODBCConnection	用于连接数据库。类似于 ADO 连接, 示例详见本章后面的内容
SqlDataAdapter、OleDbDataAdapter 和 ODBCDataAdapter	用于存储 select、insert、update 和 delete 命令的类, 因此可以用于填充 DataSet 和更新数据库, SqlDataAdapter 的示例详见本章后面的内容
SqlDataReader、OleDbDataReader 和 ODBCDataReader	用作只向前的连接数据读取器, SqlDataReader 的示例详见本章后面的内容
SqlParameter、OleDbParameter 和 ODBCParameter	用于为存储过程定义一个参数, 如何使用 SqlParameter 的示例详见本章后面的内容
SqlTransaction、OleDbTransaction 和 ODBCTransaction	用于数据库事务, 包装在一个对象中

ADO.NET 类最重要的功能是: 它们是以断开连接的方式工作, 这在目前以 Web 为中心的环境中非常重要。我们常常把服务(例如在线书店)构建为连接到一个服务器, 检索一些数据, 再在客户端上处理这些数据, 之后重新连接服务器, 并把数据传递回去, 进行处理。ADO.NET 的断开连接的本质就可以启用这种操作。

传统的 ADO 2.1 引入了断开连接的记录集, 它允许从数据库中检索数据, 把它们传递给客户端, 进行处理, 再重新连接服务器。但它们使用起来常常很繁琐, 因为断开连接的工作方式不是一开始就设计好的。ADO.NET 类则不同, 除了一种情况([provider]DataReader)外, 它们都用于脱机处理数据库。



本章将介绍在 .NET Framework 中用于数据访问的类和接口，主要论述连接数据库时使用的 SQL 类，因为 Framework SDK 示例安装了一个 SQL Server Express 数据库 (SQL Server)。在大多数情况下，OLE DB 类和 ODBC 类类似于 SQL 代码。

30.2 使用数据库连接

为了访问数据库，需要提供某种类型的连接参数，如运行数据库的计算机和登录证书。使用 ADO 的用户会很快熟悉 .NET 连接类 `OleDbConnection` 和 `SqlConnection`，图 30-1 显示了两个连接类及类的层次结构。

本章的示例使用 Northwind 数据库，可以在网络上找到它。下面的代码段说明了如何创建、打开和关闭 Northwind 数据库的连接。

```
using System.Data.SqlClient;

string source = "server=(local);" +
               "integrated security=SSPI;" +
               "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();

// Do something useful

conn.Close();
```

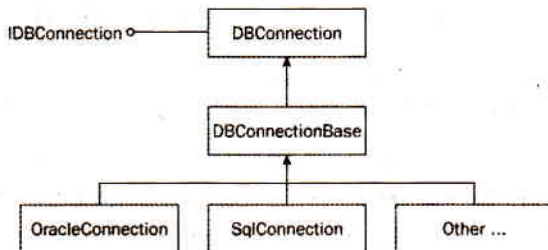


图 30-1

如果以前使用过 ADO 或 OLE DB，就会很熟悉连接字符串。如果使用的是 OleDb 提供程序，就应能剪切和粘贴旧代码。在该示例的连接字符串中，使用的参数如下所示。连接字符串中的参数用分号分隔开。

- **server=(local)**: 表示要连接到的数据库服务器。SQL Server 允许在同一台计算机上运行多个不同的数据库服务器实例，这里连接到默认的 SQL Server 实例。如果使用 SQL Express，就把服务器部分改为 `server=.\sqlexpress`。
- **integrated security=SSPI**: 这个参数使用 Windows Authentication 连接到数据库，最好在源代码中使用这个参数，而不是用户名和密码。
- **database=Northwind**: 这描述了要连接到的数据库实例。每个 SQL Server 进程都可以提供几个数据库实例。



如果忘记数据库连接字符串的格式, 就可以使用下面的 URL: www.connectionstrings.com

这个示例使用定义好的连接字符串打开数据库连接, 再关闭该连接。一旦打开连接后, 就可以对数据源执行命令, 完成后, 就可以关闭连接。

SQL Server 有另一种模式的身份验证。它可以使用 Windows 集成的安全性, 这样在登录时提供的证书就会传递给 SQL Server。为此, 应删除连接字符串的 uid 和 pwd 部分, 并添加 Integrated Security=SSPI。

在本章的下载代码中, 有一个文件 login.cs 简化了本章的示例。它链接到所有的示例代码, 包括用于这些示例的数据库连接信息; 可以修改该文件, 使用自己的服务器名称、用户名和密码。在默认情况下, 该文件使用 Windows 集成的安全性, 但是可以根据需要修改用户名和密码。

30.2.1 管理连接字符串

在以前的 .NET 版本中, 由开发人员管理数据库的连接字符串, 其方法常常是把连接字符串存储在应用程序配置文件中, 或者更常见的是, 在应用程序的某个地方硬编码连接字符串。

从 .NET 2.0 开始, 有一种预定义的方式来存储连接字符串, 甚至是以类型未知的方式使用数据库连接。例如, 现在可以编写应用程序, 然后插入各个数据库提供程序, 而这些都无需修改主应用程序。

要定义数据库连接字符串, 应使用配置文件中 <connectionStrings> 部分。在这里可以指定连接的名称、数据库连接字符串的实际参数, 还需要指定这个连接类型的提供程序。下面是一个例子:



```
<configuration>
...
<connectionStrings>
  <add name="Northwind"
        providerName="System.Data.SqlClient"
        connectionString="server=(local);integrated
                           security=SSPI;database=Northwind" />
</connectionStrings>
</configuration>
```

代码段 Connection String Example.txt

本章将在其他例子中使用这个连接字符串。

一旦在配置文件中定义了数据库连接信息, 就需要在应用程序中利用该信息。我们常常要创建一个如下的方法, 根据连接的名称检索数据库连接:



```
private DbConnection GetDatabaseConnection ( string name )
{
    ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];

    DbProviderFactory factory = DbProviderFactories.GetFactory
        ( settings.ProviderName );
    DbConnection conn = factory.CreateConnection ( );
    conn.ConnectionString = settings.ConnectionString;

    return conn;
}
```

这段代码首先读取指定的连接字符串段(使用 `ConfigurationStringSettings` 类), 再从基类 `DbProviderFactories` 中申请一个提供程序工厂, 这要使用 `ProviderName` 属性, 它在应用程序配置文件中设置为"System.Data.SqlClient"。它会映射为实际的工厂类, 用于为 SQL Server 生成数据库连接, 在本例中应使用 `System.Data.SqlClient` 中的 `SqlClientFactory` 类。必须添加对 `System.Configuration` 程序集的引用, 才能解析上述代码使用的 `ConfigurationManager` 类。

这对于获得数据库连接似乎是不必要的工作, 如果应用程序从来不运行在其他数据库(除了为它设计的数据库之外)上, 这些工作确实没有必要。但如果使用前面的工厂方法和泛型 `Db*`类(如 `DbConnection`、`DbCommand` 和 `DbDataReader`), 就会发现, 以后将该应用程序迁移到另一个数据库系统上非常简单。

30.2.2 高效地使用连接

一般情况下, 当在 .NET 中使用“稀缺”的资源时, 如数据库连接、窗口或图形对象, 最好确保每个资源在使用完后立即关闭。尽管 .NET 的设计人员实现了自动的垃圾收集, 垃圾最终都会被回收, 但仍需要尽可能早地释放资源, 以避免出现资源匮乏的情况。

当编写访问数据库的代码时, 这都非常明显, 因为使连接打开的时间略长于需要的时间, 就可能影响其他会话。在极端的情况下, 不关闭连接会使其他用户无法进入一整组数据表, 极大地降低了应用程序的性能。因为关闭数据库连接应是必需的, 所以本节讨论如何构建代码, 把一直打开资源的风险降到最低。

主要有两种方式可以确保数据库连接等类似的“稀缺”资源在使用完后立即释放。下面就介绍这两种方式。

1. 第一种方式——利用 `try...catch...finally` 语句块

确保释放资源的第一种方式是利用 `try...catch...finally` 块, 确保在 `finally` 块中关闭任何已打开的连接。下面是一个小示例:

```
try
{
    // Open the connection
    conn.Open();
    // Do something useful
}
catch ( SqlException ex )
{
    // Log the exception
}
finally
{
    // Ensure that the connection is freed
    conn.Close ( );
}
```

在 `finally` 块中, 可以释放已经使用的任何资源。这种方式的唯一麻烦是必须确保关闭连接。很

容易忘记在 **finally** 块中添加关闭连接的命令, 所以应在编码风格上添加一些不容易出现反常情况的内容。

另外, 在给定的方法中可能会打开许多资源(如两个数据库连接和一个文件), 这样 **try...catch...finally** 块的层次有时可能不容易看懂。但还有另一种方式可以确保资源的关闭——使用 **using** 语句。

2. 第二种方式——使用 using 语句块

在开发 C# 的过程中, .NET 在对象不再引用之后清理它们的方法是使用非决定性的析构方式, 这已经引起了非常热烈的讨论。

在 C++ 中, 对象只要使用完毕, 就会自动调用其析构函数。这对于基于资源的类的设计人员是非常好的消息, 因为如果用户忘记关闭资源, 则最好使用析构函数。只要对象使用完毕, 就会调用 C++ 析构函数。所以, 例如, 如果出现了异常, 但没有捕获, 有析构函数的对象就会调用它们的析构函数。

在 C# 和其他托管语言中, 没有自动的、决定性的析构的概念, 而是有一个垃圾收集器, 它会在未来的某个时刻释放资源。它是非决定性的, 因为我们不能确定这个过程在什么时候发生。忘记关闭数据库连接可能会导致 .NET 可执行程序的各种问题。幸运的是, 我们还有解决的方法。下面的代码说明了如何使用 **using** 子句确保在退出块后立即释放实现 **IDisposable** 接口(详见第 13 章)的对象。

```
string source = "server=(local);" +
               "integrated security=SSPI;" +
               "database=Northwind";

using ( SqlConnection conn = new SqlConnection ( source ) )
{
    // Open the connection
    conn.Open ( );

    // Do something useful
}
```

在这个实例中, 无论块是如何退出的, **using** 子句都会确保关闭数据库连接。

查看一下连接类的 **Dispose()** 方法的 IL 代码, 它们都检查连接对象的当前状态, 如果其状态为打开, 就调用 **Close()** 方法。浏览 .NET 程序集的一个强大工具是 **Reflector**(可以从 www.red-gate.com/products/reflector/ 上获得)。这个工具允许查看任何 .NET 方法的 IL 代码, 还可以把 IL 代码反编译为 C# 源代码, 让我们轻松地确定给定方法的作用。

在编程时, 应至少使用这两种方法中的种, 或者两种方法都使用。无论在哪里获得资源, 最好都使用 **using()** 语句, 因为尽管我们都打算编写 **Close()** 语句, 但有时会忘记, 并且出现异常时 **using** 子句就会发挥作用。因为这两种方式都没有好的异常处理方式来替代, 所以在大多数情况下, 最好组合使用这两种方法, 如下面的示例所示。

```
try
{
    using ( SqlConnection conn = new SqlConnection ( source ) )
    {
        // Open the connection
    }
}
```

```

        conn.Open ( );

        // Do something useful
        // Close it myself
        conn.Close ( );
    }
}
catch (SqlException e)
{
    // Log the exception
}

```

这里调用了 `Close()` 方法，但严格来说这是不必要的，因为 `using` 子句将确保在任何情况下都执行关闭操作。但是，应确保像这样的任何资源尽可能早地释放。因为在块的其余部分可能有更多的代码，而没有必要锁定资源。

另外，如果在 `using` 块中出现了异常，`using` 子句就会确保在资源上调用 `IDisposable.Dispose()` 方法，在本例中将确保总是关闭数据库连接。这样，与必须确保在异常子句中关闭连接相比，代码的可读性更高。还要注意，异常定义为 `SqlException`，而不是捕获所有异常的 `Exception` 类型——应总是捕获特定的异常，把不显式处理的所有其他异常放在执行栈中。如果专用的数据类可以处理错误，并执行一些操作，就应仅捕获这个异常。

最后，如果编写一个封装资源的类，那么无论该资源是什么，都应实现 `IDisposable` 接口，以关闭资源。这样，任何使用该类的代码都可以利用 `using()` 语句，并确保资源被释放。

30.2.3 事务

通常，对数据库要进行多次更新，这些更新必须在事务的范围内进行。我们常常要在代码中查找一个事务对象，它传递给许多方法，以更新数据库，但在 .NET Framework 2.0 及其更高版本中，在 `System.Transactions` 程序集中添加了 `TransactionScope` 类，它极大地简化了事务代码的编写，因为可以把几个事务方法合并到一个事务范围中，事务流会根据需要执行每个方法。

下面的代码是在 SQL Server 连接上开始事务处理：

```

string source = "server=(local);" +
    "integrated security=SSPI;" +
    "database=Northwind";

using (TransactionScope scope = new
    TransactionScope(TransactionScopeOption.Required))
{
    using (SqlConnection conn = new SqlConnection(source))
    {
        // Do something in SQL

        // Then mark complete
        scope.Complete();
    }
}

```

这段代码使用 `scope.Complete()` 方法把事务显式地标记为完成。如果不调用这个方法，事务就会回滚，以便不对数据库进行任何修改。

在使用事务作用域时，可以选择在该事务中执行的命令的独立级别。该级别确定了如何在一个数据库会话中查看在另一个数据库会话中所进行的修改，并不是所有数据库引擎都支持表 30-5 所示的 4 个级别。

表 30-5

独立级别	说 明
ReadCommitted	SQL Server 的默认级别。这个级别可以确保只有第一个事务提交后，在第二个事务中才能访问第一个事务写入的数据
ReadUncommitted	即使一个事务还没有提交数据，也允许另一个事务从数据库中读取数据。例如，如果两个用户在访问同一个数据库，第一个用户插入一些数据，但没有完成事务(通过 Commit 或 Rollback 方法)，第二个用户把自己的独立级别设置为 ReadUncommitted，因此可以访问数据
RepeatableRead	这个级别扩展了 ReadCommitted 级别，确保如果在事务中使用了相同的语句，无论是否有其他潜在的数据库更新，总是可以返回相同的数据。这个级别要求对数据进行额外的锁定，这会降低性能。这个级别可以保证，对于初始查询的每一行，都不会修改数据，但允许显示“幻象(phantom)”行——这些行是在事务运行时，由另一个事务插入的全新数据行
Serializable	这是最“高级”的事务级别，对数据库中的数据进行串行化访问。利用这种独立级别，不会显示幻象行，所以在可串行化的事务中使用的 SQL 语句总是检索相同的数据。可串行化的事务对性能的负面影响不应低估，如果不肯定是否需要使用这个独立级别，最好不要使用它

SQL Server 的默认独立级别 ReadCommitted 是数据一致性和数据可用性之间的一种很好的折衷，因为它比 RepeatableRead 或 Serializable 模式中需要的数据锁定都少。但是，有时应提高独立级别，这样在 .NET 中，才能从一种非默认的级别开始事务处理。使用哪个级别没有硬性规则，全凭经验。



如果当前使用的是不支持事务的数据库，就应转而使用支持它的数据库。一旦我们成为可以完全信任的雇员，且拥有错误数据库的全部访问权限，就可能试输入 `delete from bug where id=99999` 以删除对应的错误，但实际上输入的是“<”而不是“=”，此时会删除整个错误数据库，这可不是我们希望的。幸好 IS 小组每天晚上都会备份该数据库，可以还原它，但使用回滚命令会更简单。

30.3 命令

30.2 节简要介绍了针对数据库执行的命令。简言之，命令就是一个要在数据库上执行的包含 SQL 语句的文本字符串。命令也可以是一个存储过程，或者返回表中所有列和所有行的表的名称(换言之，SELECT * 样式的子句)。

把 SQL 子句作为一个参数传递给 Command 类的构造函数，就可以构造一条命令，如下例所示：

```
string source = "server=(local);" +
               "integrated security=SSPI;" +
               "database=Northwind";
```

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(select, conn);
```

<provider>Command 类有一个 CommandType 属性，它用来定义某条命令是 SQL 子句、存储过程的调用，还是完整的表语句(仅从给定的表中选择所有列和行)。表 30-6 总结了 CommandType 枚举。

表 30-6

命 令 类 型	样 例
Text(默认)	String select = "SELECT ContactName FROM Customers" SqlCommand cmd = new SqlCommand(select, conn);
StoredProcedure	SqlCommand cmd=new SqlCommand("CustOrderHist", conn); cmd.CommandType = CommandType.StoredProcedure; cmd.Parameters.Add("@CustomerID", "QUICK");
TableDirect	OleDbCommand cmd = new OleDbCommand("Categories", conn); cmd.CommandType = CommandType.TableDirect;

在执行存储过程时，需要把参数传送给过程。上面的示例直接设置了参数@CustomerID，尽管设置参数的值还可以使用其他方式，详见本章后面的内容。注意自从.NET 2.0 开始，给命令参数集合添加了 AddWithValue()方法，而废弃了 Add(name, value)成员。如果习惯于使用这个构建参数的原始方法来调用存储过程，在重新编译代码时，就会得到一个编译警告。最好现在就修改代码，因为 Microsoft 在.NET 的后续版本中将删除旧方法。



TableDirect 命令类型只对 OleDb 提供程序有效——如果试图把这个命令类型用于其他提供程序，就会抛出异常。

30.3.1 执行命令

定义好命令后，就需要执行它。执行语句有许多方式，这取决于要从命令中返回什么数据。

<provider>Command 类提供了下述可执行的命令：

- ExecuteNonQuery() —— 执行命令，但不返回任何结果。
- ExecuteReader() —— 执行命令，返回一个类型化的 IDataReader。
- ExecuteScalar() —— 执行命令，返回结果集中第一行第一列的值。

除了上述命令外，SqlCommand 类也提供了下面的方法：

- ExecuteXmlReader() —— 执行命令，返回一个 XmlReader 对象，它可以遍历从数据库中返回的 XML 片段。

1. ExecuteNonQuery()方法

这个方法一般用于 UPDATE、INSERT 或 DELETE 语句，其中唯一的返回值是受影响的记录个数。但如果调用带输出参数的存储过程，该方法就有返回值：



可从
wrox.com
下载源代码

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

public class ExecuteNonQueryExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            "integrated security=SSPI;" +
            "database=Northwind";
        string select = "UPDATE Customers " +
            "SET ContactName = 'Bob' " +
            "WHERE ContactName = 'Bill'";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        int rowsReturned = cmd.ExecuteNonQuery();
        Console.WriteLine("{0} rows returned.", rowsReturned);
        conn.Close();
    }
}
```

代码下载 [GetDatabaseConnection.txt](#)

ExecuteNonQuery()方法返回命令所影响的行数，它为一个整数。

2. ExecuteReader()方法

这个方法执行命令，并根据使用的提供程序返回一个类型化的 DataReader 对象，返回的对象可以用于遍历返回的记录，如下面的代码所示。



可从
wrox.com
下载源代码

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

public class ExecuteReaderExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            "integrated security=SSPI;" +
            "database=Northwind";
        string select = "SELECT ContactName,CompanyName FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        SqlDataReader reader = cmd.ExecuteReader();
        while(reader.Read())
        {
            Console.WriteLine("Contact: {0,-20} Company: {1}",
                reader[0], reader[1]);
        }
    }
}
```

代码下载 [ExecuteReaderExample.cs](#)

图 30-2 显示了这段代码的结果。

本章的后面将讨论<provider>DataReader 对象。

```
*** SqlProvider ***
Output from direct SQL statement...
```

CONTACT	COMPANY
Maria Anders	Alfreds Futterkiste
Ana Trujillo	Ana Trujillo Emparedados y helados
Antonio Moreno	Antonio Moreno Iaqueria
Thomas Hardy	Arround the Horn
Christina Berglund	Berglunds snabbköp
Hanna Moos	Blauer See Delikatessen
Frédérique Citeaux	Blondesdél père et fils
Martin Sommer	Bólido Comidas preparadas
Laurence Leblan	Bon app'
Elizabeth Lincoln	Bottom-Dollar Markets
Victoria Ashworth	B's Beverages
Patricio Simpson	Cactus Comidas para llevar
Francisco Chang	Centro comercial Moctezuma
Fang Wang	Chop-suey Chinese
Pedro Afonso	Comércio Mineiro
Elizabeth Brown	Consolidated Holdings
Evan Orellan	Drachenblut Delikatessen
Janine Labruno	Du monde entier
Ana Devon	Eastern Connection
Roland Mendel	Ernst Handel

图 30-2

3. ExecuteScalar()方法

在许多情况下，需要从 SQL 语句返回一个结果，如给定表中的记录个数，或者服务器上的当前日期/时间。ExecuteScalar()方法就可以用于这些场合：



可从
wrox.com
下载源代码

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

public class ExecuteScalarExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            "integrated security=SSPI;" +
            "database=Northwind";

        string select = "SELECT COUNT( * ) FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        object o = cmd.ExecuteScalar();
        Console.WriteLine(o);
    }
}
```

代码段 ExecuteReaderExample.cs

该方法返回一个对象，根据需要，可以把该对象强制转换为合适的类型。如果所调用的 SQL 只返回一行，则最好使用 ExecuteScalar()方法来检索这一行。这也适合于只返回一个值的存储过程。

4. ExecuteXmlReader()方法(只用于 SqlConnection 提供程序)

顾名思义，这个方法执行命令，给调用者返回一个 XmlReader 对象。SQL Server 允许使用 FOR XML 子句来扩展 SQL 的 SELECT 子句。这个子句可以带有下列 3 个选项中的一个：

- FOR XML AUTO —— 根据 FROM 子句中的表构建一棵树
- FOR XML RAW —— 把结果集中的行映射到元素，其中的列映射到属性
- FOR XML EXPLICIT —— 必须指定要返回的 XML 树的形状

对于本示例, 使用 AUTO:



可从
wrox.com
下载源代码

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

public class ExecuteXmlReaderExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            "integrated security=SSPI;" +
            "database=Northwind";

        string select = "SELECT ContactName,CompanyName " +
            "FROM Customers FOR XML AUTO";

        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        XmlReader xr = cmd.ExecuteXmlReader();
        xr.Read();
        string data;
        do
        {
            data = xr.ReadOuterXml();
            if (!string.IsNullOrEmpty(data))
            {
                Console.WriteLine(data);
            } while (!string.IsNullOrEmpty(data));
            conn.Close();
        }
    }
}
```

代码下载 ExecuteReaderExample.cs

注意, 必须导入 System.Xml 名称空间, 才能输出返回的 XML。这个名称空间和 .NET Framework 其他的 XML 功能将在第 33 章中详细论述。本例在 SQL 语句中包含了 FOR XML AUTO 子句, 然后调用 ExecuteXmlReader() 方法。代码的结果如图 30-3 所示。

在 SQL 子句中, 指定了 FROM Customers, 这样 Customers 类型的元素就显示在输出中。为它添加特性, 每个特性对应于从数据库中选择出来的每一列。这就为从数据库中选择的一行构建了 XML 片段。

```
*** SqlProvider ***
Use ExecuteXmlReader with a FOR XML AUTO SQL clause

<Customers ContactName="Maria Anders" CompanyName="Alfreds Futterkiste" />
<Customers ContactName="Antonio Moreno" CompanyName="Antonio Moreno Taqueria" />
<Customers ContactName="Christina Berglund" CompanyName="Berglunds snabbköp" />
<Customers ContactName="Frédérique Citeaux" CompanyName="Blondesddsl père et fil
e" />
<Customers ContactName="Laurence Lebihan" CompanyName="Bon app'" />
<Customers ContactName="Victoria Ashworth" CompanyName="B's Beverages" />
<Customers ContactName="Francisco Chang" CompanyName="Centro comercial Moctezuma
" />
<Customers ContactName="Pedro Afonso" CompanyName="Comércio Mineiro" />
<Customers ContactName="Even Ottilieb" CompanyName="Drachenblut Delikatessen" />
<Customers ContactName="Ann Devon" CompanyName="Eastern Connection" />
<Customers ContactName="Aria Cruz" CompanyName="Familia Arquibaldo" />
<Customers ContactName="Martine Rancé" CompanyName="Folies gourmandes" />
<Customers ContactName="Peter Franken" CompanyName="Frankenversand" />
<Customers ContactName="Paolo Accorti" CompanyName="Franchi S.p.A." />
<Customers ContactName="Eduardo Saavedra" CompanyName="Galería del gastrónomo" /
>
<Customers ContactName="André Fonseca" CompanyName="Gourmet Lanchonetes" />
<Customers ContactName="Manuel Pereira" CompanyName="GROCCERIA Restaurants" />
<Customers ContactName="Carlos Hernández" CompanyName="HILARION-Rhastos" />
```

图 30-3

30.3.2 调用存储过程

用命令对象调用存储过程，就是定义存储过程的名称，给过程的每个参数添加参数定义，然后用上一节中给出的其中一种方法执行命令。

为了使本节的示例更有说服力，下面定义一组可用于在 Northwind 样本数据库的 Region 表中插入、更新和删除记录的存储过程。尽管 Region 表很小，但它可以用于给每种常见的存储过程编写示例，它是一个很好的示例。

1. 调用没有返回值的存储过程

调用存储过程最简单的示例是不给调用者返回任何值。下面定义了两个这样的存储过程，一个用于更新先前已有的 Region 记录，另一个用于删除指定的 Region 记录。

(1) 记录的更新

更新 Region 记录很简单，因为(假定主键不能更新)只有一列可以更新。直接在 SQL Server 查询分析器中输入这些示例，或者运行本章可下载代码中的 StoredProcs.sql 文件，在该文件中包含本节的所有存储过程：



可从
wrox.com
下载源代码

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
                                @RegionDescription NCHAR(50)) AS
SET NOCOUNT OFF
UPDATE Region
SET RegionDescription = @RegionDescription
WHERE RegionID = @RegionID
GO
```

代码下载 StoredProcs.sql

给实际表执行更新命令，需要重复选择和返回全部已更新的记录。这个存储过程接受两个输入参数(@RegionID 和 @RegionDescription)，对数据库执行 UPDATE 语句。

要在 .NET 代码中运行这个存储过程，需要定义一条 SQL 命令，并执行它：



可从
wrox.com
下载源代码

```
SqlCommand cmd = new SqlCommand("RegionUpdate", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue ( "@RegionID", 23 );
cmd.Parameters.AddWithValue ( "@RegionDescription", "Something" );
```

代码下载 StoredProcs.sql

这段代码新建一个 SqlCommand 对象 aCommand，并把它定义为一个存储过程。然后，使用 AddWithValue() 方法依次添加每个参数，这会构建一个参数，并设置其值，也可以手工构建 SqlParameter 实例，并根据需要把它们添加到 Parameters 集合中。

该存储过程接受两个参数：正在更新的 Region 记录的唯一主键；给这个记录的新描述。创建命令后，就可以用下面的命令执行它：

```
cmd.ExecuteNonQuery();
```

由于该过程没有返回值，因此使用 ExecuteNonQuery() 方法就足够了。命令参数可以直接使用 AddWithValue() 方法直接设置，也可以通过构建 SqlParameter 实例来设置。注意命令集合可以按位置

或参数名来索引。

(2) 记录的删除

下一个存储过程可用于从数据库中删除一个 Region 记录:



可从
wrox.com
下载源代码

```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
SET NOCOUNT OFF
DELETE FROM Region
WHERE RegionID = @RegionID
GO
```

代码下载 StoredProcs.sql

这个过程只需要该记录的主键值。下面的代码使用 SqlCommand 对象调用这个存储过程:



可从
wrox.com
下载源代码

```
SqlCommand cmd = new SqlCommand("RegionDelete", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@RegionID", SqlDbType.Int, 0,
"RegionID"));
cmd.UpdatedRowSource = UpdateRowSource.None;
```

代码下载 StoredProcs.cs

这条命令只接收一个参数, 如下面的代码所示, 它执行 RegionDelete 存储过程, 这是一个按照名称设置参数的示例。如果有许多对同一个存储过程的类似调用, 就应构建 SqlParameter 实例, 并设置其值, 如下面的代码所示, 其性能要比为每个调用重新构建整个 SqlCommand 更好:

```
cmd.Parameters["@RegionID"].Value = 999;
cmd.ExecuteNonQuery();
```

2. 调用返回输出参数的存储过程

前面两个执行存储过程的示例都没有返回值。如果存储过程包含输出参数, 它们就需要在 .NET 客户端中定义, 以便在过程返回时填充其输出参数。下面的示例说明了如何在数据库中插入记录, 并把该记录的主键返回给调用者。

记录的插入

Region 表仅由主键(RegionID)和描述字段(RegionDescription)组成。要插入一个记录, 必须生成该数字主键, 再把新行插入到数据库中。在这个示例中, 通过在存储过程中创建一个主键, 简化了主键的生成。使用的方法未经过任何加工, 这就是本章的后面用一节的篇幅介绍键的生成的原因。现在使用这个原始示例就足够了:



可从
wrox.com
下载源代码

```
CREATE PROCEDURE RegionInsert(@RegionDescription NCHAR(50),
@RegionID INTEGER OUTPUT)AS
SET NOCOUNT OFF
SELECT @RegionID = MAX(RegionID) + 1
FROM Region
INSERT INTO Region(RegionID, RegionDescription)
VALUES(@RegionID, @RegionDescription)
GO
```

代码下载 StoredProcs.sql

插入过程新建一个 Region 记录, 在数据库本身生成主键值时, 这个值作为输出参数从过程中返

回(@RegionID)。这对于这个简单示例足够了,但对于比较复杂的表(特别是有默认值的表),通常不使用输出参数,而选择插入的整行,把该行返回给调用者。.NET 类可以处理这两种情况。



可从
wrox.com
下载源代码

```
SqlCommand cmd = new SqlCommand("RegionInsert", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@RegionDescription",
    SqlDbType.NChar,
    50,
    "RegionDescription"));
cmd.Parameters.Add(new SqlParameter("@RegionID",
    SqlDbType.Int,
    0,
    ParameterDirection.Output,
    false,
    0,
    0,
    "RegionID",
    DataRowVersion.Default,
    null));
cmd.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

代码下载 StoredProcs.cs

其中参数的定义比较复杂。第二个参数@RegionID 定义为包含其参数方向,在这个示例中是 Output。除这个标志之外,该示例还在最后一行使用 UpdateRowSource 枚举表示数据通过输出参数从这个存储过程中返回。当从一个 DataTable(详见本章后面的内容)中执行存储过程调用时,主要使用这个标志。

调用这个存储过程类似于前面的示例,但在这个实例中,需要在执行该过程后读取输出参数:

```
cmd.Parameters["@RegionDescription"].Value = "South West";
cmd.ExecuteNonQuery();
int newRegionID = (int) cmd.Parameters["@RegionID"].Value;
```

在执行该命令后,读取@RegionID 参数的值,并把它强制转换为整数。上述命令的一个缩写版本是 ExecuteScalar()方法,它返回存储过程返回的第一个值(返回为对象)。

如果调用的存储过程返回输出参数和一组记录行,该怎么办?此时,应定义合适的参数,不是调用 ExecuteNonQuery()方法,而应调用另一个方法(如 ExecuteReader()),可以遍历所有返回的记录。

30.4 快速数据访问: 数据读取器

虽然数据读取器(data reader)是从数据源中选择某些数据的最简单快捷的方法,但这也是功能最弱的一个方法。不能直接实例化数据阅读器,即调用 ExecuteReader()方法后从相应数据库的命令对象(如 SqlCommand)中返回的实例。

下面的代码说明了如何从 Northwind 数据库的 Customer 表中选择数据。这个示例连接到数据库,选择许多记录,循环所选的记录,并把它们输出到控制台上。

这个示例使用 OLE DB 提供程序作为一个来自 SQL 提供程序的简化的数据暂存器。在大多数情况下, OleDbClient 类与 SqlConnection 类是一一对应的关系,例如, OleDbConnection 对象就类似于前面

示例使用的 `SqlConnection` 对象。

要对 OLE DB 数据源执行命令，应使用 `OleDbCommand` 类。下面的代码执行一条简单的 SQL 语句，并读取记录，具体方法是返回一个 `OleDbDataReader` 对象。

注意下面的第二条 `using` 指令使 `OleDb` 类可用。

```
using System;
using System.Data.OleDb;
```

因为目前所利用的大部分数据提供程序都在同一个程序集中发布，所以只需要引用 `System.Data.dll` 程序集，就可以导入本节使用的所有类。



可从
wrox.com
下载源代码

```
public class DataReaderExample
{
    public static void Main(string[] args)
    {
        string source = "Provider=SQLOLEDB;" +
            "server=(local);" +
            "integrated security=SSPI;" +
            "database=northwind";

        string select = "SELECT ContactName,CompanyName FROM Customers";
        OleDbConnection conn = new OleDbConnection(source);
        conn.Open();
        OleDbCommand cmd = new OleDbCommand(select, conn);
        OleDbDataReader aReader = cmd.ExecuteReader();
        while(aReader.Read())
            Console.WriteLine("{0}' from {1}",
                aReader.GetString(0), aReader.GetString(1));
        aReader.Close();
        conn.Close();
    }
}
```

代码下载 [DataReaderExample.cs](#)

前面的代码包含其他章节介绍的许多熟悉的 C# 功能。要编译该示例，使用下面的命令：

```
csc /t:exe /debug+ DataReaderExample.cs /r:System.Data.dll
```

在前面的示例中，下面的代码根据源连接字符串，新建一个 OLE DB .NET 数据库连接：

```
OleDbConnection conn = new OleDbConnection(source);
conn.Open();
OleDbCommand cmd = new OleDbCommand(select, conn);
```

第 3 行根据特定的 `SELECT` 语句新建一个 `OleDbCommand` 对象，和执行命令时所使用的数据库连接。当有一条有效的命令时，就需要执行它，它返回一个初始化后的 `OleDbDataReader`：

```
OleDbDataReader aReader = cmd.ExecuteReader();
```

`OleDbDataReader` 是一个只向前的连接读取器，即只能沿着一个方向遍历记录，而使用的数据库连接一直打开，直到关闭该数据读取器为止。



OleDbDataReader 会使数据库连接一直处于打开状态, 直到显式地关闭它为止。

OleDbDataReader 类不能直接实例化, 它总是通过调用 OleDbCommand 类的 ExecuteReader() 方法来返回。一旦打开一个数据读取器, 就可以用各种方式访问包含在该读取器中的数据。

关闭 OleDbDataReader 对象(显式调用 Close() 方法或通过垃圾收集器收集对象)时, 底层的连接也会关闭, 这取决于调用了哪个 ExecuteReader() 方法。如果调用了 ExecuteReader() 方法, 并传递了 CommandBehavior.CloseConnection, 就可以在关闭读取器时强制关闭连接。

OleDbDataReader 类有一个索引器, 它可以使用常见的数组风格的语法访问任何字段(尽管不是类型安全的访问):

```
object o = aReader[0];
```

或者

```
object o = aReader["CategoryID"];
```

假定 CategoryID 字段是 SELECT 语句中用于填充读取器的第一个字段, 那么这两行语句在功能上等价, 尽管后者比前者慢一些。为了验证这一点, 编写一个简单的测试程序, 从打开的数据读取器中对同一列进行 100 万次的迭代访问, 仅为了获取一些足够大的数字来读取。虽然在一个死循环中可能并不会对同一列读取 100 万次, 但按每(微)秒来计算, 就可能编写出最佳的代码。

另外, 数字索引器平均每 0.09 秒就进行 100 万次的访问, 而文本索引器需要 0.63 秒。原因是文本方法是从模式的内部查找列号, 再使用序列号进行访问。如果知道这个区别, 就可以更好地访问数据。

是否应使用数字索引器? 也许, 但还有一种更好的方式。除了上面给出的索引器外, OleDbDataReader 还有一组类型安全的方法可以用于读取列, 这些方法很容易理解, 且都以 Get 开头。有一些方法可以读取大多数类型的数据, 如 GetInt32、GetFloat 和 GetGuid 等。

前面使用 GetInt32 的 100 万次迭代用了 0.06 秒。数字索引器中的系统开销是获取数据类型造成的, 调用与 GetInt32 相同的代码, 然后装箱(本实例是拆箱)为一个整数。如果事先知道这种模式, 希望使用加密数字而不是列名, 且允许对所有列访问使用类型安全的函数, 这样运行速度就会比使用文本格式的列名快 10 倍(选择同一列的上百万个副本)。

毫无疑问, 在可维护性和速度之间有一个折衷的问题。如果必须使用数字索引器, 就应在类范围内为所有要访问的列定义常量。上面的代码可以用于从任何 OLE DB 数据库中选择数据, 但有许多 SQL Server 专用类可以使用, 只是其可移植性有明显的损失。

下面的示例与上一示例基本相同, 但在这个实例中分别用 SQL 提供程序和 SQL 类的引用替换了 OLE DB 提供程序和对 OLE DB 类的所有引用。该示例在 04_DataReaderSql 目录下:



可从
wrox.com
下载源代码

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

public class DataReaderSql
{
    public static int Main(string[] args)
```

```

string source = "server=(local);" +
               "integrated security=SSPI;" +
               "database=northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(select, conn);
SqlDataReader aReader = cmd.ExecuteReader();
while(aReader.Read())
    Console.WriteLine("{0}' from {1}", aReader.GetString(0),
                      aReader.GetString(1));
aReader.Close();
conn.Close();
return 0;
}

```

代码下载 DataReaderSql.cs

注意一下区别是什么？如果正在输入这些代码，就用 sql 替换所有 OleDb，改变数据源字符串，并重新编译。这很容易。

对 SQL 提供程序的索引器进行相同的性能测试，这次数字索引器也使用 0.13 秒就完成了 100 万次的访问，基于索引器的字符串运行了约 0.65 秒。

30.5 管理数据和关系：DataSet 类

DataSet 类是数据的脱机容器。它不包含数据库连接的概念，实际上存储在 DataSet 类中的数据不一定来源于数据库，它可以是来自 CSV 文件、XML 文件的记录，或是从测量设备中读取的点。

DataSet 类由一组数据表组成，每个表都有一组数据列和数据行，如图 30-4 所示。除了定义数据外，还可以在 DataSet 类中定义表之间的链接。例如，我们常常要定义父/子关系(通常也称为主/从关系)。表中的一个记录(即 Order)链接到另一个表的许多记录上(即 Order_Details)，这种关系可以在 DataSet 类中定义和导航。

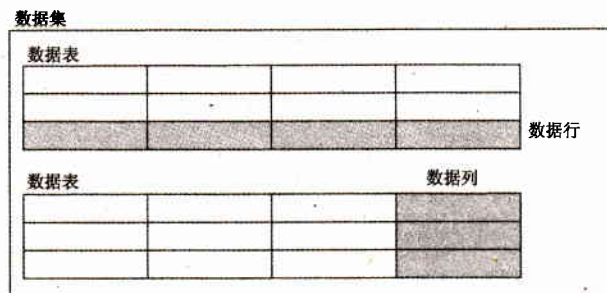


图 30-4

重点是记住，DataSet 类基本上是内存中的数据库，其中包含了所有表、关系和约束。下一节描述和 DataSet 一起使用的类。

30.5.1 数据表

数据表非常类似于物理数据库表,它由一组包含特定属性的列组成,可能包含 0 行或多行数据。数据表也可以定义主键(它可以是一列或多列),列上也可以包含约束。这些信息对应的通用术语在本章的其他部分称为“架构”。

为数据表定义架构有几种方式(实际上把 `DataSet` 类当作一个整体),这些在介绍了数据列和数据行后讨论。图 30-5 显示了一些可通过数据表访问的对象。

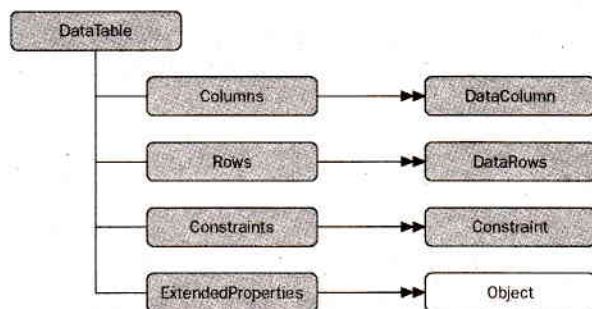


图 30-5

`DataTable` 对象(和 `DataColumn`)可以附带任意多个扩展属性。这个集合可以用与对象相关的用户自定义信息来填充。例如,某一列有一个输入掩码,用于验证列的内容是否有效,比较常见的示例是美国的社会安全号。当数据在中间层中构造,要返回给客户端,进行某些处理时,最适合使用扩展的属性。例如,可以在扩展的属性中存储数字列的有效性标准(如 `min` 和 `max`),在验证用户输入时在 UI 层使用它。

填充数据表时,可以从数据库中选择数据,从文件中读取数据,或在代码中手工填充, `Rows` 集合会包含这些检索出来的数据。

`Columns` 集合包含已经添加到表中的 `DataColumn` 实例,它们定义了数据的架构,如数据类型、是否可为空和默认值等。`Constraints` 集合可以用唯一约束或主键约束来填充。

数据表使用架构信息的一个示例是在 `DataGrid` 中显示数据时。`DataGrid` 控件使用属性(如列的数据类型)来确定该列应使用什么控件。数据库中的 `bit` 字段在 `DataGrid` 中显示为一个复选框。如果列在数据库架构中定义为 `NOT NULL`,该信息就存储在 `DataColumn` 中,以便在用户试图移出数据行时测试该列。

30.5.2 数据列

`DataColumn` 对象定义了 `DataTable` 中某列的属性,如该列的数据类型,该列是否为只读,以及其他属性。可以在代码中创建列,或者由运行库自动生成列。

在创建列时,给它指定名称也很有用;否则运行库就会为该列生成一个名称,其格式是 `Columnn`,其中 `n` 是一个递增的数字。

列的数据类型可以在构造函数中提供,也可以通过设置 `DataType` 属性来指定。一旦把数据加载到数据表中,就不能改变列的数据类型,否则会抛出一个 `ArgumentException` 异常。

创建的数据列可以包含表 30-7 所示的 .NET Framework 数据类型。

表 30-7

Boolean	Decimal	Int64	TimeSpan
Byte	Double	Sbyte	UInt16
Char	Int16	Single	UInt32
DateTime	Int32	String	UInt64

一旦创建它，就要给 `DataColumn` 对象设置其他属性，如该列是否可为空或者设置默认值。下面的代码段显示了给 `DataColumn` 对象设置的一些更常见的选项：

```

DataColumn customerID = new DataColumn("CustomerID", typeof(int));
customerID.AllowDBNull = false;
customerID.ReadOnly = false;
customerID.AutoIncrement = true;
customerID.AutoIncrementSeed = 1000;
DataColumn name = new DataColumn("Name", typeof(string));
name.AllowDBNull = false;
name.Unique = true;

```

可以给 `DataColumn` 对象设置如表 30-8 所示的属性。

表 30-8

属 性	说 明
AllowDBNull	如果为 <code>true</code> ，该列就可以设置为 <code>DBNull</code>
AutoIncrement	指定该列的值自动生成为一个递增的数字
AutoIncrementSeed	定义 <code>AutoIncrement</code> 列最初的种子值
AutoIncrementStep	定义自动生成的列值之间的递增量，默认值为 1
Caption	可以用于在屏幕上显示列名
ColumnMapping	指定当 <code>DataSet</code> 类通过调用 <code>DataSet.WriteXml</code> 来保存时，列如何映射到 XML 上
ColumnName	列名，如果没有在构造函数中设置，就由运行库自动生成列名
DataType	定义列的 <code>System.Type</code> 值
DefaultValue	可以定义列的默认值
Expression	定义在所计算的列中使用的表达式

1. 数据行

这个类构成了 `DataTable` 类的另一部分。数据表中的列根据 `DataTable` 类来定义，表中的实际数据用 `DataRow` 对象来访问。下面的示例说明了如何访问数据表中的行。首先是连接的详细信息：

```

string source = "server=(local);" +
    " integrated security=SSPI;" +
    "database=northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);

```

下面的代码引入了 `SqlDataAdapter` 类，它用于把数据置入 `DataSet` 中。`SqlDataAdapter` 类使用 SQL 子句，在 `DataSet` 类中用下面查询的结果填写 `Customers` 表。`SqlDataAdapter` 类将在 30.7 节中进一步讨论。

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

在下面的代码中，注意使用 `DataRow` 类的索引器访问数据行上的值。给定列的值可以用几个重载的索引器来检索，这样就可以通过已知的列号、列名或 `DataColumn` 来检索数据的值：

```
foreach (DataRow row in ds.Tables["Customers"].Rows)
    Console.WriteLine("'{'0}' from {1}", row[0], row[1]);
```

`DataRow` 类最吸引人的一个方面就是它的版本功能。`DataRow` 类可以接收某一特定行上指定列的各个值，其版本见表 30-9。

表 30-9

DataRow 类的 Version 值	说 明
Current	列中目前存在的值，如果没有进行编辑，该值与初值相同。如果进行了编辑，该值就是最后输入的有效值
Default	默认值(换言之，列的任何默认设置)
Original	最初从数据库中选择出来的列值。如果调用 <code>DataRow</code> 类的 <code>AcceptChanges</code> 方法，该值就更新为 <code>Current</code> 值
Proposed	对列进行修改时，可以检索到这个已改变的值。如果在行上调用 <code>BeginEdit()</code> 方法，并进行修改，每一列都会有一个推荐值，直到调用 <code>EndEdit()</code> 或 <code>CancelEdit()</code> 方法为止

可以以许多方式使用给定列的版本。例如，在数据库中更新数据行时，在这种情况下常常使用如下 SQL 语句：

```
UPDATE Products
SET Name = Column.Current
WHERE ProductID = xxx
AND Name = Column.Original;
```

显然，这段代码永远不会编译，但它说明了某一行中某一列的初值和当前值的一种用法。

要从 `DataRow` 类的索引器中检索某个版本的值，应使用其中一个索引器方法，它把 `DataRowVersion` 值作为一个参数。下面的代码段说明了如何获得 `DataTable` 对象中每一列的所有值：

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
    foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
    {
        Console.WriteLine ("{'0} Current = {1}", dc.ColumnName,
                            row[dc, DataRowVersion.Current]);
        Console.WriteLine (" Default = {0}", row[dc, DataRowVersion.Default]);
    }
}
```

```
        Console.WriteLine (" Original = {0}",
                             row[dc,DataRowVersion.Original]);
    }
}
```

整行有一个状态标志 **RowState**，它可以用于确定在持久化到数据库时需要对该行进行什么操作。把 **RowState** 属性设置为跟踪对 **DataTable** 所做的所有改变，如添加新行、删除现有行，以及改变表中的列。当数据与数据库同步时，行的状态标志用于确定应执行什么 **SQL** 操作。这些标志由 **DataRowState** 枚举定义，如表 30-10 所示。

表 30-10

DataRowState 值	说 明
Added	指出把新数据行添加到 DataTable 的 Rows 集合中。在客户端上创建的所有行都设置为这个值，在与数据库同步时，最终会使用 SQL 的 INSERT 语句
Deleted	指出通过 DataRow.Delete() 方法把 DataTable 中的数据行标记为删除。该行仍存在于 DataTable 中，但在屏幕上通常看不到它(除非显式设置 DataRowView)。在下一节讨论。在 DataTable 中标记为已删除的数据行将在与数据库同步时从数据库中删除
Detached	指出某一行在创建后立即显示为这个状态，调用 DataRow.Remove() 方法也可以返回这个状态。分离的行不是任何数据表的一部分，因此处于这种状态的行不能使用任何 SQL 语句
Modified	如果任何列中的值发生了改变，数据行就处于这个状态
Unchanged	指出自从最后一次调用 AcceptChanges() 方法以来，该行都没有发生改变

行的状态也取决于在其上调用的方法。一般在成功更新数据源(即把改变持久化到数据库后)之后调用 **AcceptChanges()** 方法。

修改 **DataRow** 中的数据最常见的方式是使用索引器，但如果对数据进行了许多修改，就需要考虑使用 **BeginEdit()** 和 **EndEdit()** 方法。

在对 **DataRow** 中的列进行了修改后，就会在该行的 **DataTable** 上引发 **ColumnChanging** 事件。该事件可以重写 **DataColumnChangeEventArgs** 类的 **ProposedValue** 属性，按照需要修改它。这是在列值上进行某些数据有效性验证的一种方式。如果在进行修改前调用 **BeginEdit()** 方法，就不会引发 **ColumnChanging** 事件，于是可以进行多次修改，再调用 **EndEdit()** 方法，持久化这些修改。如果要回退到初值，就应调用 **CancelEdit()** 方法。

DataRow 可以以某种方式链接到其他数据行上，在数据行之间能够建立可导航的链接，这在主/从数据表中非常常见。**DataRow** 包含 **GetChildRows()** 方法，该方法可以从同一个 **DataSet** 的另一个表中把一组相关行返回为当前行。这些将在 30.5.2 节中介绍。

2. 架构的生成

为 **DataTable** 创建架构有 3 种方式：

- 让运行库来完成
- 编写代码来创建表
- 使用 XML 架构生成器

下面介绍这 3 种方式。

(1) 运行库生成的架构

前面的 DataRow 示例用下面的代码从数据库中选择数据，并填充一个 DataSet 类：

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

这很容易使用，但它也有几个缺点。例如，必须利用默认的列名来处理——这是可行的，但在某些情况下，还要把物理数据库的列(如 PKID)重命名为一个用户友好性更高的名称。

自然，可以在 SQL 子句中给列指定别名，如在 SELECT PID AS PersonID FROM Person Table 中。但最好不要在 SQL 中重命名列，因为列实际上只需要在屏幕上显示一个“好”的名称即可。

自动生成 DataTable/DataColumn 的另一个潜在问题是不能控制运行库为列选择的数据类型。运行库可以确定正确的数据类型，但有时需要对此有更多的控制。例如，为给定的列定义枚举类型，以简化类的用户代码。如果接受运行库生成的默认列类型，该列就可能是一个 32 位的整数，而不是有预定义选项的枚举。

最后，也是最有可能出的问题是，在使用自动生成的表时，不能对 DataTable 中的数据进行类型安全的访问——索引器就会返回 object 的实例，而不是派生的数据类型。如果要用类型强制转换表达式对代码进行修改，就可以跳过下面的章节。

(2) 手工编码的架构

用生成的代码来创建 DataTable，再用相关联的 DataColumns 来填充相当简单。本节的示例将访问 Northwind 数据库中的 Product 表，如图 30-6 所示。

Column Name	Data Type	Allow Nulls
ProductID	int	☐
ProductName	nvarchar(40)	☐
SupplierID	int	☒
CategoryID	int	☒
QuantityPerUnit	nvarchar(20)	☒
UnitPrice	money	☒
UnitsInStock	smallint	☒
UnitsOnOrder	smallint	☒
ReorderLevel	smallint	☒
Discontinued	bit	☐

图 30-6

下面的代码生成一个 DataTable，它对应于图 30-6 的架构(但没有包含可空的列)：

```
DataTable products = new DataTable("Products");
products.Columns.Add(new DataColumn("ProductID", typeof(int)));
products.Columns.Add(new DataColumn("ProductName", typeof(string)));
products.Columns.Add(new DataColumn("SupplierID", typeof(int)));
products.Columns.Add(new DataColumn("CategoryID", typeof(int)));
products.Columns.Add(new DataColumn("QuantityPerUnit", typeof(string)));
products.Columns.Add(new DataColumn("UnitPrice", typeof(decimal)));
products.Columns.Add(new DataColumn("UnitsInStock", typeof(short)));
products.Columns.Add(new DataColumn("UnitsOnOrder", typeof(short)));
```



可从
wrox.com
下载源代码

```

products.Columns.Add(new DataColumn("ReorderLevel", typeof(short)));
products.Columns.Add(new DataColumn("Discontinued", typeof(bool)));
ds.Tables.Add(products);
}

```

代码下载 [ManufacturedDataSet.cs](#)

可以改变 `DataRow` 示例中的代码, 使用如下新生成的表定义:

```

string source = "server=(local);" +
                "integrated security=sspi;" +
                "database=Northwind";
string select = "SELECT * FROM Products";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter cmd = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
ManufactureProductDataTable(ds);
cmd.Fill(ds, "Products");
foreach(DataRow row in ds.Tables["Products"].Rows)
    Console.WriteLine("'{'0}' from {1}", row[0], row[1]);

```

`ManufactureProductDataTable()` 方法新建一个 `DataTable`, 依次添加每一列, 最后把这个表追加到 `DataSet` 中表的清单上。`DataSet` 有一个索引器, 它的参数是表名, 给调用者返回该 `DataTable`。

上面的示例仍不是类型安全的, 因为在列上使用了索引器来检索数据。最好是有一个类(或一组类)派生自 `DataSet`、`DataTable` 和 `DataRow`, 为表、行和列定义类型安全的存取器。可以自己生成这段代码, 这并不是特别乏味, 最终将得到可以进行数据类型安全访问的类。

如果不愿意自己生成这些类型安全的类, 就可以使用帮助。`.NET Framework` 对本节开头列出的第 3 种方法提供了支持: 允许使用 XML 架构来定义 `DataSet`、`DataTables` 和本节介绍的其他类。30.6 节将详细介绍这种方法。

30.5.3 数据关系

在编写应用程序时, 常常需要获取和缓存各种信息表。`DataSet` 类是这些信息的容器, 使用一般的 OLE DB, 需要提供一种奇怪的 SQL 方言来强制分层的数据关系, 提供程序本身不能没有它自己的“怪癖”。

另一方面, `DataSet` 类从一开始就用于建立数据表之间的关系。本节的代码说明了如何手工生成并填充两个数据表。如果不能访问 SQL Server 或 NorthWind 数据库, 就可以自由地运行这个示例。



可从
wrox.com
下载源代码

```

DataSet ds = new DataSet("Relationships");
ds.Tables.Add(CreateBuildingTable());
ds.Tables.Add(CreateRoomTable());
ds.Relations.Add("Rooms",
                 ds.Tables["Building"].Columns["BuildingID"],
                 ds.Tables["Room"].Columns["BuildingID"]);

```

代码下载 [DataRelationships.cs](#)

本示例使用的表如图 30-7 所示。这两个表仅包含一个主键和名称字段, `Room` 表有一个 `BuildingID` 外键。

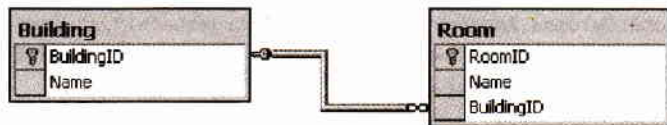


图 30-7

这些表都非常简单，下面的代码说明了如何迭代 **Buildings** 表中的行，并遍历对应的关系，以列出 **Rooms** 表中的所有子行。

```
foreach(DataRow theBuilding in ds.Tables["Building"].Rows)
{
    DataRow[] children = theBuilding.GetChildRows("Rooms");
    int roomCount = children.Length;
    Console.WriteLine("Building {0} contains {1} room{2}",
        theBuilding["Name"],
        roomCount,
        roomCount > 1 ? "s": "");
    // Loop through the rooms
    foreach(DataRow theRoom in children)
        Console.WriteLine("Room: {0}", theRoom["Name"]);
}
```

DataSet 类和其他分层的旧 **recordset** 对象之间的主要区别是关系显示的方式。在分层的 **Recordset** 对象中，关系显示为行中的一个伪列，该列本身是一个可以迭代的 **Recordset** 对象。但在 ADO.NET 中，通过调用 **GetChildRows()** 方法就可以遍历关系。

```
DataRow[] children = theBuilding.GetChildRows("Rooms");
```

该方法有许多形式，但上面的示例只使用关系的名称在父子行之间来回遍历。它返回一个行数组，使用前面示例中的索引器就可以更新这些行。

数据关系更有趣的地方是可以用两种方式遍历这些数据。在 **DataTable** 类上使用 **ParentRelations** 属性，不仅可以从父数据行中找到子数据行，还可以从子记录中找到父数据行。这个属性返回一个 **DataRelationCollection**，该集合可以使用[]数组语法来索引(例如，**DataRelations["Rooms"]**)，另外，**GetChildRows()** 方法也可以如下所示进行调用：

```
foreach(DataRow theRoom in ds.Tables["Room"].Rows)
{
    DataRow[] parents = theRoom.GetParentRows("Rooms");
    foreach(DataRow theBuilding in parents)
        Console.WriteLine("Room {0} is contained in building {1}",
            theRoom["Name"],
            theBuilding["Name"]);
}
```

GetParentRows() 方法(返回 0 行或多行数据对应的一个数组)或 **GetParentRow()** 方法(根据给定的某种关系检索一个父行)都有许多重写版本，可以检索出父行。

30.5.4 数据约束

DataTable 类仅擅长于改变在客户端上创建的列的数据类型。ADO.NET 允许在列上创建一组约

束，对数据应用一些规则。

运行库目前支持表 30-11 所示的约束类型，它们包含在 `System.Data` 名称空间的类中。

表 30-11

约 束	说 明
<code>ForeignKeyConstraint</code>	在 <code>DataSet</code> 的两个 <code>DataTable</code> 之间强制链接
<code>UniqueConstraint</code>	确保给定的列是唯一的

1. 设置主键

在关系数据库的表中，可以提供一个主键，该主键可以基于 `DataTable` 中的一列或多列。

下面的代码为 `Product` 表创建了一个主键，其架构是前面手动构建的。表上的主键只是约束的一种形式。当把主键添加到 `DataTable` 中时，运行库也会对键列生成一个唯一约束。这是因为实际上并没有 `PrimaryKey` 约束类型，主键是一列或多列上的唯一约束。



可从
wrox.com
下载源代码

```
public static void ManufacturePrimaryKey(DataTable dt)
{
    DataColumn[] pk = new DataColumn[1];
    pk[0] = dt.Columns["ProductID"];
    dt.PrimaryKey = pk;
}
```

代码下载 `ManufacturedDataSet.cs`

因为主键可以包含几列，所以它可以作为一个 `DataColumn` 数据输入。通过给表的一个列数组指定属性，就可以给这些列设置主键。

要检查表中的约束，可以迭代 `ConstraintsCollection`。上述代码自动生成的约束是 `Constraint1`，这个名称没有什么用，因此应避免这种情况，最好先在代码中创建约束，然后定义组成主键的列。

创建主键之前，下面的代码给约束命名：

```
DataColumn[] pk = new DataColumn[1];
pk[0] = dt.Columns["ProductID"];
dt.Constraints.Add(new UniqueConstraint("PK_Products", pk[0]));
dt.PrimaryKey = pk;
```

唯一约束可以应用到任意多列上。

2. 设置外键

除了唯一约束外，`DataTable` 类还可以包含外键约束，它们主要用于强制主/从关系，如果正确地建立了约束，外键约束还可用于在表之间复制列。在主/从关系的表中，常常有一个父记录(订单)和许多子记录(订单行)，它们通过父记录的主键链接起来。

因为外键约束只能作用于同一个 `DataSet` 中的表，所以下面的示例使用 `Northwind` 数据库中的 `Categories` 表，给该表和 `Products` 表之间指定约束，如图 30-8 所示。

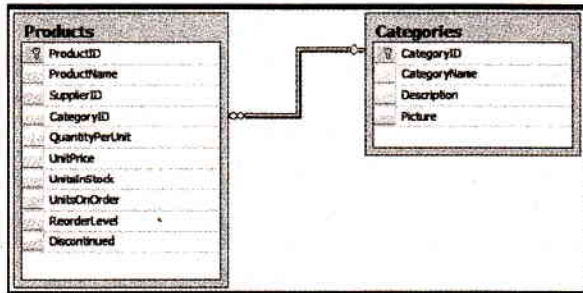


图 30-8

第一步是为 Categories 表生成一个新的数据表:

```
DataTable categories = new DataTable("Categories");
categories.Columns.Add(new DataColumn("CategoryID", typeof(int)));
categories.Columns.Add(new DataColumn("CategoryName", typeof(string)));
categories.Columns.Add(new DataColumn("Description", typeof(string)));
categories.Constraints.Add(new UniqueConstraint("PK_Categories",
    categories.Columns["CategoryID"]));
categories.PrimaryKey = new DataColumn[1]
    {categories.Columns["CategoryID"]};
```

上述代码的最后一行为 Categories 表创建主键。在本例中,主键是一个单列,但可以使用数组语法在多个列上生成一个键。

然后,需要在两个表之间创建约束:

```
DataColumn parent = ds.Tables["Categories"].Columns["CategoryID"];
DataColumn child = ds.Tables["Products"].Columns["CategoryID"];
ForeignKeyConstraint fk =
    new ForeignKeyConstraint("FK_Product_CategoryID", parent, child);
fk.UpdateRule = Rule.Cascade;
fk.DeleteRule = Rule.SetNull;
ds.Tables["Products"].Constraints.Add(fk);
```

这个约束应用到 Categories.CategoryID 和 Products.CategoryID 之间的链接上。有 4 个不同的 ForeignKeyConstraint, 但应使用可以给约束命名的 ForeignKeyConstraint。

3. 设置更新和删除约束

除了在父表和子表之间定义约束之外,还可以在更新约束中的一列时定义应执行的操作。

上面的示例设置了更新规则和删除规则,在对父表中的列(或行)执行某种操作时,使用这些规则,这些规则用来确定应对影响到的子表中的行进行什么操作。通过 Rule 枚举可以应用 4 种不同的规则:

- Cascade —— 如果更新了父键,就应把新的键值复制到所有子记录中。如果删除了父记录,则也将删除子记录,这是默认选项。
- None —— 不执行任何操作,这个选项会留下子数据表中的孤立行。
- SetDefault —— 如果定义了一个子记录,那么每个受影响的子记录都把外键列设置为其默认值。
- SetNull —— 所有子行都把键列设置为 DBNull(按照 Microsoft 使用的命名约定,键列实际上应是 SetDBNull)。



如果 DataSet 类的 EnforceConstraints 属性为 true, 就只能在 DataSet 类中强约束。

本节介绍了组成 DataSet 类中约束部分的主类, 揭示了如何在代码中手工生成这些类。还可以使用 .NET 附带的 XML 架构文件和 XSD 工具定义 DataTable、DataRow、DataColumn、DataRelation 和 Constraint。下一节将说明如何建立一个简单的架构, 以及如何生成类型安全的类, 以访问数据。

30.6 XML 架构: 用 XSD 生成代码

XML 已经在 ADO.NET 中确立了牢固的地位——实际上, 现在在对象之间远程传递数据的格式是 XML。有了 .NET 运行库, 就可以在 XML 架构定义文件(XSD)中描述 DataTable。而且, 可以定义整个 DataSet 类, 其中有许多 DataTable 类, 这些表之间有一组关系, 并可以包括全面描述数据的各种其他信息。

在定义 XSD 文件时, 运行库中有一个新工具, 该工具可以把这个架构转换为对应的数据访问类, 如类型安全的 DataTable 类, 如上所述。本节先介绍一个简单的 XSD 文件(Products.xsd), 该文件描述与前面 Products 示例相同的信息, 再扩展它, 使其包括一些额外的功能。



可从
wrox.com
下载源代码

```
<xs:schema id="Products" targetNamespace="http://tempuri.org/XMLSchema1.xsd"
xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ProductID" msdata:ReadOnly="true"
          msdata:AutoIncrement="true" type="xs:int" />
        <xs:element name="ProductName" type="xs:string" />
        <xs:element name="SupplierID" type="xs:int" minOccurs="0" />
        <xs:element name="CategoryID" type="xs:int" minOccurs="0" />
        <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0" />
        <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
        <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" />
        <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" />
        <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" />
        <xs:element name="Discontinued" type="xs:boolean" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

代码下载 Products.xsd

第 33 章将详细论述其中的一些选项。现在应知道, 这个文件基本上定义一个架构, 其中把 id 属性设置为 Products。还定义了一个比较复杂的 Product 类, 其中包含了许多元素, 每个元素对应于 Products 表中的一个字段。

这些元素映射到如下数据类上。Products 架构映射到派生自 DataSet 类的一个类上, Product 类

映射到派生自 `DataTable` 类的一个类上。每个子元素映射到派生自 `DataColumn` 的一个类上。所有列的集合映射到派生自 `DataRow` 类的一个类上。

.NET Framework 中有一个工具，只要输入 XSD 文件，该工具就可以生成这些类的代码。因为该工具唯一的工作是执行 XSD 文件上的各种功能，所以它称为 `XSD.EXE`。

假定把上面的文件另存为 `Product.xsd`，在命令提示符上输入下述命令，把该文件转换为代码：

```
xsd Product.xsd /d
```

这会创建文件 `Product.cs`。

该命令有许多选项可以和 `XSD` 一起使用，以改变生成的输出结果，其中一些比较常用的选项如表 30-12 所示。

表 30-12

选 项	说 明
/dataset (/d)	启用派生自 <code>DataSet</code> 、 <code>DataTable</code> 和 <code>DataRow</code> 的类
/language:<language>	允许选择编写输出文件的语言。C#是默认语言，也可以选择用 VB 编写 Visual Basic .NET 文件
/namespace:<namespace>	允许定义生成代码应驻留其中的名称空间，默认为没有名称空间

下面节选了 `Products` 架构在 `XSD` 中的输出结果，并做了略微的修改，以满足本书的相应格式。要查看完整的输出结果，可以在 `Products` 架构(或者自己建立的某个架构)上运行 `XSD.EXE`，查看生成的 `.cs` 文件。该示例包含所有源代码和 `Products.xsd` 文件。



可从
wrox.com
下载源代码

```
//-----  
// <autogenerated>  
//     This code was generated by a tool.  
//     Runtime Version:4.0.21006.1  
//  
//     Changes to this file may cause incorrect behavior and will be lost if  
//     the code is regenerated.  
// </autogenerated>  
//-----  
  
//  
// This source code was auto-generated by xsd, Version=4.0.21006.1  
//  
  
/// <summary>  
/// Represents a strongly typed in-memory cache of data.  
/// </summary>  
[global::System.Serializable()]  
[global::System.ComponentModel.DesignerCategoryAttribute("code")]  
[global::System.ComponentModel.ToolboxItem(true)]  
[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema")]  
[global::System.Xml.Serialization.XmlRootAttribute("Products")]  
[global::System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet")]  
public partial class Products : global::System.Data.DataSet {  
    private ProductDataTable tableProduct;
```

```

private global::System.Data.SchemaSerializationMode _schemaSerializationMode =
    global::System.Data.SchemaSerializationMode.IncludeSchema;

[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public Products() {
    this.BeginInit();
    this.InitClass();
    global::System.ComponentModel.CollectionChangeEventHandler
        schemaChangedHandler = new
            global::System.ComponentModel.CollectionChangeEventHandler(
                this.SchemaChanged);
    base.Tables.CollectionChanged += schemaChangedHandler;
    base.Relations.CollectionChanged += schemaChangedHandler;
    this.EndInit();
}

```

代码下载 product.cs

为了集中论述公共接口，这段代码删除了所有受保护成员和私有成员。**ProductDataTable** 和 **ProductRow** 定义显示了两个嵌套类的位置，后面会实现它们。下面简单地解释一下派生自 **DataSet** 的类，之后讨论这些类的代码。

Products()构造函数调用一个私有方法 **InitClass()**，该方法构造派生自 **DataTable** 类的 **ProductDataTable** 类的一个实例，并把该表添加到 **DataSet** 类的 **Tables** 集合中。**Products** 数据表可以通过下面的代码来访问：

```

DataSet ds = new Products();
DataTable products = ds.Tables["Products"];

```

或者，更简单的方式是使用 **Product** 属性来访问，该属性在派生的 **DataSet** 对象上可用：

```

DataTable products = ds.Product;

```

因为 **Product** 属性是强类型化的，所以可以使用 **ProductDataTable**，而不是上述代码中的 **DataTable** 引用。

ProductDataTable 类包含更多的代码(注意这是一段节选的代码)：



可从
wrox.com
下载源代码

```

[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")]
public partial class ProductDataTable : global::System.Data.DataTable,
    global::System.Collections.IEnumerable {
    private global::System.Data.DataColumn columnProductID;
    private global::System.Data.DataColumn columnProductName;
    private global::System.Data.DataColumn columnSupplierID;
    private global::System.Data.DataColumn columnCategoryID;
    private global::System.Data.DataColumn columnQuantityPerUnit;
    private global::System.Data.DataColumn columnUnitPrice;
    private global::System.Data.DataColumn columnUnitsInStock;
    private global::System.Data.DataColumn columnUnitsOnOrder;
    private global::System.Data.DataColumn columnReorderLevel;
    private global::System.Data.DataColumn columnDiscontinued;
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(

```

```
"System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public ProductDataTable() {
    this.TableName = "Product";
    this.BeginInit();
    this.InitClass();
    this.EndInit();
}
```

代码下载 [product.cs](#)

ProductDataTable 类派生自 **DataTable** 类, 并实现 **IEnumerable** 接口, 为表中的每一列定义了一个私有的 **DataColumn** 实例, 通过调用私有的 **InitClass()** 成员, 再次从构造函数中初始化这些实例。**DataRow** 类将使用每一列(后面介绍)。



可从
wrox.com
下载源代码

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
[global::System.ComponentModel.Browsable(false)]
public int Count {
    get {
        return this.Rows.Count;
    }
}
```

// Other row accessors removed for clarity — there is one for each column

代码下载 [product.cs](#)

给表添加数据行由 **AddProductRow()** 方法的两个重载版本实现(虽然它们的内容完全不同, 但名称相同)。第一个重载方法接受一个已经构造出来的 **DataRow**, 且没有返回值。另一个重载方法则接受一组参数值, 每个参数对应于 **DataTable** 中的一列, 该重载方法新构造一行, 设置该新行中的值, 把该行添加到 **DataTable** 对象中, 并给调用者返回该行。这些迥然不同的函数不应有相同的名称。



可从
wrox.com
下载源代码

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public ProductRow AddProductRow(string ProductName, int SupplierID,
    int CategoryID, string QuantityPerUnit, decimal UnitPrice,
    short UnitsInStock, short UnitsOnOrder, short ReorderLevel,
    bool Discontinued) {
    ProductRow rowProductRow = ((ProductRow)(this.NewRow()));
    object[] columnValuesArray = new object[] {
        null,
        ProductName,
        SupplierID,
        CategoryID,
        QuantityPerUnit,
        UnitPrice,
        UnitsInStock,
        UnitsOnOrder,
        ReorderLevel,
        Discontinued};
    rowProductRow.ItemArray = columnValuesArray;
    this.Rows.Add(rowProductRow);
}
```

```
return rowProductRow;
```

代码下载 [product.cs](#)

派生自 `DataSet` 的类中的 `InitClass()` 成员把表添加到 `DataSet` 类中, 与此相同, `ProductDataTable` 类中的 `InitClass()` 成员把列添加到 `DataTable` 类中。每一列的属性都按需要设置, 再把该列追加到列集合的尾部。



可从
wrox.com
下载源代码

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
private void InitClass() {
    this.columnProductID = new
global::System.Data.DataColumn("ProductID",
    typeof(int), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnProductID);
    this.columnProductName = new global::System.Data.DataColumn(
        "ProductName", typeof(string), null,
        global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnProductName);
    this.columnSupplierID = new global::System.Data.DataColumn(
        "SupplierID", typeof(int), null,
        global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnSupplierID);
    this.columnCategoryID = new global::System.Data.DataColumn("CategoryID",
        typeof(int), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnCategoryID);
    this.columnQuantityPerUnit = new
        global::System.Data.DataColumn("QuantityPerUnit", typeof(string); null,
        global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnQuantityPerUnit);
    this.columnUnitPrice = new global::System.Data.DataColumn("UnitPrice",
        typeof(decimal), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnUnitPrice);
    this.columnUnitsInStock = new global::System.Data.DataColumn("UnitsInStock",
        typeof(short), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnUnitsInStock);
    this.columnUnitsOnOrder = new global::System.Data.DataColumn("UnitsOnOrder",
        typeof(short), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnUnitsOnOrder);
    this.columnReorderLevel = new global::System.Data.DataColumn("ReorderLevel",
        typeof(short), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnReorderLevel);
    this.columnDiscontinued = new global::System.Data.DataColumn("Discontinued",
        typeof(bool), null, global::System.Data.MappingType.Element);
    base.Columns.Add(this.columnDiscontinued);
    this.columnProductID.AutoIncrement = true;
    this.columnProductID.AllowDBNull = false;
    this.columnProductID.ReadOnly = true;
    this.columnProductName.AllowDBNull = false;
    this.columnDiscontinued.AllowDBNull = false;
}

[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
```

```
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public ProductRow NewProductRow() {
    return ((ProductRow)(this.NewRow()));
}
```

代码下载 [product.cs](#)

NewRowFromBuilder()方法在 **DataTable** 类的 **NewRow()**方法内部调用。这里新建了强类型化的一行。**DataRowBuilder** 实例由 **DataTable** 类创建，其成员仅能在 **System.Data** 程序集中访问。

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
protected override global::System.Data.DataRow NewRowFromBuilder(
    global::System.Data.DataRowBuilder builder) {
    return new ProductRow(builder);
}
```

最后一个要讨论的类是 **ProductRow** 类，它派生自 **DataRow** 类。这个类用于提供对数据表中所有字段的类型安全的访问。它封装了特定行的存储器，并提供成员来读取(和写入)表中的每个字段。

另外，对于每个可空字段，有函数可以把该字段设置为 **null**，并检查该字段是否为 **null**。下面的示例列出了 **SupplierID** 列的函数：



可从
wrox.com
下载源代码

```
public partial class ProductRow : global::System.Data.DataRow {
    private ProductDataTable tableProduct;

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    internal ProductRow(global::System.Data.DataRowBuilder rb) :
        base(rb) {
        this.tableProduct = ((ProductDataTable)(this.Table));
    }

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    public int ProductID {
        get {
            return ((int)(this[this.tableProduct.ProductIDColumn]));
        }
        set {
            this[this.tableProduct.ProductIDColumn] = value;
        }
    }

    // Other column accessors/mutators removed for clarity

    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    public bool IsSupplierIDNull() {
        return this.IsNull(this.tableProduct.SupplierIDColumn);
    }
}
```

```

[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public void SetSupplierIDNull() {
    this[this.tableProduct.SupplierIDColumn] = global::System.Convert.DBNull;
}
}

```

代码下载 product.cs

下面的代码利用 XSD 工具中的类的输出从 Product 表中检索数据,并在控制台上显示这些数据:



可从
wrox.com
下载源代码

```

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

public class XSD_DataSet
{
    public static void Main()
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=northwind";
        string select = "SELECT * FROM Products";
        SqlConnection conn = new SqlConnection(source);
        SqlDataAdapter da = new SqlDataAdapter(select, conn);
        Products ds = new Products();
        da.Fill(ds, "Product");
        foreach(Products.ProductRow row in ds.Product )
            Console.WriteLine("{0}' from {1}",
                row.ProductID,
                row.ProductName);
    }
}

```

代码下载 XSDDataSet.cs

XSD 文件的输出结果包含一个派生自 DataSet 类的 Products 类,它使用数据适配器来创建和填充。foreach 语句使用强类型化的 ProductRow 和 Product 属性,Product 属性返回 Product 数据表。

要编译这个示例,执行下面的命令:

```
xsd product.xsd /d
```

和

```
csc /recurse: * .cs
```

第一条命令从 Products.XSD 架构中生成 Products.cs 文件,然后, csc 命令使用/recurse:*.cs 参数查找扩展名为.cs 的所有文件,并把它们添加到所生成的程序集中。

30.7 填充 DataSet 类

定义了数据集的架构,并准备好 DataTables、DataColumns、Constrain 类和一些必需内容后,就

需要用这些信息填充 DataSet 类。从外部源中读取数据,并把数据插入到 DataSet 类中,有两种方式:

- 使用数据适配器
- 把 XML 读入 DataSet 类

30.7.1 用数据适配器填充 DataSet

30.5.2 节简要介绍了 SqlDataAdapter 类,使用该类的代码如下所示:

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

加粗代码行显示了所用的 SqlDataAdapter 类——实际上,其他数据适配器类在功能上与 SqlDataAdapter 类完全相同。

为了把数据插入到 DataSet 类中,需要执行某种形式的命令以选择该数据。该命令可以是 SQL SELECT 语句,一个存储过程的调用,或者是 TableDirect 命令(用于 OLE DB 提供程序)。上面的示例使用了 SqlDataAdapter 类的一个构造函数,把传递过来的 SQL SELECT 子句转换为一个 SqlCommand,在适配器上调用 Fill()方法时执行这条命令。

在本章前面的存储过程示例中,定义了 INSERT、UPDATE 和 DELETE 过程,但没有给出 SELECT 过程,本节介绍该过程,并说明如何从 SqlDataAdapter 类上调用存储过程,从而把数据填充到 DataSet 类中。

在数据适配器上使用存储过程

首先需要定义一个存储过程,SELECT 数据的存储过程如下所示:



可从
wrox.com
下载源代码

```
CREATE PROCEDURE RegionSelect AS
SET NOCOUNT OFF
SELECT * FROM Region
GO
```

代码下载 [StoredProcs.sql](#)

这个存储过程可以直接输入到 SQL Server 查询分析器中,或者可以运行这个示例所使用的 StoredProc.sql 文件。

接着,需要定义一个执行该存储过程的 SqlCommand,同样,这段代码非常简单,并且大部分已经在前一节中出现过:



可从
wrox.com
下载源代码

```
private static SqlCommand GenerateSelectCommand(SqlConnection conn)
{
    SqlCommand aCommand = new SqlCommand("RegionSelect", conn);
    aCommand.CommandType = CommandType.StoredProcedure;
    aCommand.UpdatedRowSource = UpdateRowSource.None;
    return aCommand;
}
```

代码下载 [DataAdapter.cs](#)

这个方法生成了一个 SqlCommand,该 SqlCommand 在执行时会调用 RegionSelect 过程。最后

是把这条命令和 `SqlDataAdapter` 类关联起来, 并调用 `Fill()` 方法:



可从
wrox.com
下载源代码

```
DataSet ds = new DataSet();
// Create a data adapter to fill the DataSet
SqlDataAdapter da = new SqlDataAdapter();
// Set the data adapter 's select command
da.SelectCommand = GenerateSelectCommand (conn);
da.Fill(ds, "Region");
```

代码下载 [DataAdapter.cs](#)

其中新建了一个 `SqlDataAdapter` 类, 把生成的 `SqlCommand` 赋予数据适配器的 `SelectCommand` 属性, 然后调用执行存储过程的 `Fill()` 方法, 把返回的所有行插入到 `Region` 表的 `DataSet` 类中(在本例中, 它由运行库生成)。

数据适配器不仅仅能通过执行命令来选择数据, 30.8 节会介绍数据适配器的其他功能。

30.7.2 从 XML 中填充 DataSet 类

除了为给定的 `DataSet` 类和相关表生成架构外, `DataSet` 类还可以读写本地 XML 中的数据, 如磁盘上的文件、数据流或文本读取器。

要把 XML 加载到 `DataSet` 类中, 只需调用一个 `ReadXML()` 方法, 如下面的示例所示, 从磁盘文件中读取数据:

```
DataSet ds = new DataSet();
ds.ReadXml(".\\MyData.xml");
```

`ReadXml()` 方法试图从输入的 XML 中加载任何内联架构信息, 如果找到了某个架构, 就使用这个架构验证从该文件中加载的数据的有效性。如果没有找到内联架构, `DataSet` 类就会在加载数据时扩展其内部的结构, 这类似于前面示例中的 `Fill()` 方法的作用, 该方法检索数据, 并根据选择的数据构造 `DataTable` 类。

30.8 持久化 DataSet 类的修改

在 `DataSet` 类中编辑完数据后, 通常需要持久化这些改变。最常见的示例是从数据库中选择数据, 并把它显示给用户, 把这些更新返回给数据库。

在没有“连接数据库的”应用程序中, 这些改变可以持久化到 XML 文件中, 并传输到中间层的应用程序服务器上, 然后进行处理, 以更新几个数据源。

`DataSet` 类可以用于这两个示例, 而且这很容易完成。

30.8.1 通过数据适配器进行更新

除了 `SqlDataAdapter` 类最有可能包含的 `SelectCommand` 之外, 还可以定义 `InsertCommand`、`UpdateCommand` 和 `DeleteCommand`。顾名思义, 这些对象都是适用于相应提供程序的命令对象(如 `SqlCommand` 或 `OleDbCommand`)实例。

有了这种灵活性后, 就可以自由调整应用程序, 具体方法是对频繁使用的命令(如 `select` 和 `insert`)采用合适的存储过程来执行, 对不常使用的命令(如 `delete`)直接采用 SQL 命令来执行。一般建议为所有数据库交互操作提供存储过程, 因为这会更快, 更容易调整。

本节的示例使用 30.3.2 节中的存储过程，插入、更新和删除 Region 记录，再把这些与上面编写的 RegionSelect 过程结合起来，生成一个新示例，这个示例使用这些命令来检索和更新 DataSet 类中的数据。代码的主体在下一节介绍。

1. 新插入一行

把新行添加到 DataTable 中有两种方式。第一种方式是调用 NewRow() 方法，返回一空白行，然后向其填充数据，最后把它添加到 Rows 集合中，如下所示：

```
DataRow r = ds.Tables["Region"].NewRow();
r["RegionID"]=999;
r["RegionDescription"]="North West";
ds.Tables["Region"].Rows.Add(r);
```

第二种方式是把一个数据数组传递给 Rows.Add() 方法，如下面的代码所示：

```
DataRow r = ds.Tables["Region"].Rows.Add
    (new object [] { 999, "North West" });
```

DataTable 中的所有新行都把自己的 RowState 设置为 Added。在对数据库进行修改前，这个示例先转储记录，以便把下面的行添加到 DataTable 中(以任何一种方式)。注意右边一列显示行的状态：

New row pending inserting into database	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
999 North West	Added

要从 DataAdapter 更新数据库，可以调用其中一个 Update() 方法：

```
da.Update(ds, "Region");
```

对于 DataTable 中的新行，这将执行存储过程(在本例中是 RegionInsert)，本示例然后转储数据的状态，因此可以查看对数据库进行的修改。

New row updated and new RegionID assigned by database	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
5 North West	Unchanged

查看 DataTable 中的最后一行。把代码中的 RegionID 设置为 999，但在执行 RegionInsert 存储过程后，该值改为 5。这是有意的——数据库通常会生成主键，并且更新 DataTable 中的数据。DataTable 中数据的更新是因为源代码中的 SqlCommand 定义会把 UpdatedRowSource 属性设置为 UpdateRowSource.OutputParameters：



可从
wrox.com
下载源代码

```
SqlCommand aCommand = new SqlCommand("RegionInsert", conn);
aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
    SqlDbType.NChar,
```

```

50,
    "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID",
    SqlDbType.Int,
    0,
    ParameterDirection.Output,
    false,
    0,
    0,
    "RegionID", // Defines the SOURCE column
    DataRowVersion.Default,
    null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;

```

代码下载 DataAdapter2.cs

这段代码的作用是：无论何时数据适配器执行这条命令，输出参数都应映射到该数据行的源，在本例中它是 `DataTable` 中的一行。该标志说明了应更新什么数据——存储过程有一个输出参数映射到 `DataRow`，它应用的列是 `RegionID`，因为这是在命令的定义中定义的。

`UpdateRowSource` 的值如表 30-13 所示。

表 30-13

UpdateRowSource 值	说 明
Both	存储过程可以返回输出参数和一个完整的数据库记录。这两个数据源都用于更新源数据行
FirstReturnedRecord	该命令返回一个记录，该记录的内容应合并到最初的源 <code>DataRow</code> 中，当给定的表有许多默认(或计算)列时，使用这个值很有用，因为在执行 <code>INSERT</code> 语句之后，这些行需要与客户端上的 <code>DataRow</code> 同步。例如 <code>INSERT(列)INTO(表)WITH(主键)</code> ， <code>'SELECT(列)FROM(表)WHERE(主键)'</code> 。返回的记录应合并到源数据行上
None	丢弃从该命令返回的所有数据
OutputParameters	命令的任何输出参数都映射到 <code>DataRow</code> 的对应列上

2. 更新现有的行

更新 `DataTable` 中已有行只需使用带有一个列名或列号的 `DataRow` 类的索引器即可，如下面的代码所示：

```

r["RegionDescription"]="North West England";
r[1] = "North West England";

```

这两条语句等价(在本例中)：

Changed RegionID 5 description	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
5 North West England	Modified

在更新数据库前，被更新的行应把其状态设置为 `Modified`，如上所示。

3. 删除行

删除行需要调用 `Delete()` 方法:

```
r.Delete();
```

虽然被删除的行将其行状态设置为 `Deleted`, 但不能从被删除的 `DataRow` 中读取列, 因为它们不再有效。当调用适配器的 `Update()` 方法时, 所有被删除的行都会使用 `DeleteCommand`, 在本例中是执行 `RegionDelete` 存储过程。

30.8.2 写入 XML 输出结果

如上所述, `DataSet` 类支持在 XML 中定义其架构, 如同可以从 XML 文档中读取数据, 也可以把数据写入 XML 文档。

`DataSet.WriteXml()` 方法可以输出存储在 `DataSet` 类中的各部分数据, 可以选择只输出数据, 也可以输出数据和架构。下面是为两个 `Region` 示例编写的代码:

```
ds.WriteXml(".\\WithoutSchema.xml");
ds.WriteXml(".\\WithSchema.xml", XmlWriteMode.WriteSchema);
```

第一个文件 `WithoutSchema.xml` 如下所示:



可从
wrox.com
下载源代码

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Region>
    <RegionID>1</RegionID>
    <RegionDescription> Eastern          </RegionDescription>
  </Region>
  <Region>
    <RegionID>2</RegionID>
    <RegionDescription> Western          </RegionDescription>
  </Region>
  <Region>
    <RegionID>3</RegionID>
    <RegionDescription> Northern        </RegionDescription>
  </Region>
  <Region>
    <RegionID>4</RegionID>
    <RegionDescription> Southern        </RegionDescription>
  </Region>
</NewDataSet>
```

代码下载 WithoutSchema.xml

`RegionDescription` 上的闭合标记在页面的右边, 因为数据库列定义为 `NCHAR(50)`, 这是一个包含 50 个字符的字符串, 其中用空格填充。

`WithSchema.xml` 文件产生的输出结果包含 `DataSet` 类的 XML 架构和数据本身:



可从
wrox.com
下载源代码

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns=""
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

```

<xs:element name="NewDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="Region">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="RegionID"
              msdata:AutoIncrement="true"
              msdata:AutoIncrementSeed="1"
              type="xs:int" />
            <xs:element name="RegionDescription"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:schema>
<Region>
  <RegionID>1</RegionID>
  <RegionDescription> Eastern </RegionDescription>
</Region>
<Region>
  <RegionID>2</RegionID>
  <RegionDescription> Western </RegionDescription>
</Region>
<Region>
  <RegionID>3</RegionID>
  <RegionDescription> Northern </RegionDescription>
</Region>
<Region>
  <RegionID>4</RegionID>
  <RegionDescription> Southern </RegionDescription>
</Region>
</NewDataSet>

```

代码下载 WithoutSchema.xml

注意, 使用 msdata 架构中的文件, 它定义 DataSet 类中列的附加属性, 如 AutoIncrement 和 AutoIncrementSeed, 这些属性直接对应于 DataColumn 类上的可定义属性。

30.9 使用 ADO.NET

本节介绍使用 ADO.NET 开发数据访问应用程序时的一些常见情况。

30.9.1 分层开发

开发与数据交互的应用程序时, 常常要把应用程序分层, 常见的模型是一个应用层(前端)、一个数据服务层和数据库本身。

但使用这个模型的难题是, 确定在层之间传输什么数据, 以及应采用什么格式来传输数据。有了 ADO.NET, 就不必担心这个问题了, 这种设计一开始就支持这种体系结构。

在 ADO.NET 中, 对复制完整的记录集有比 OLE DB 更好的支持。在 .NET 中, 只需复制 DataSet 即可:

```
DataSet source = {some dataset};
DataSet dest = source.Copy();
```

这将创建源 DataSet 的一个完全相同的副本——将会复制所有 DataTable、DataColumn、DataRow 和 Relation, 所有数据都与它在源 DataSet 中所处的状态完全相同。如果只需要复制 DataSet 的架构, 就可以试试下面的代码:

```
DataSet source = {some dataset};
DataSet dest = source.Clone();
```

这也会复制所有表、关系等, 但每个复制的 DataTable 都是空的。这个过程非常简单。

在编写一个分层的系统(无论该系统是基于 Windows 客户端应用程序还是基于 Web)时, 常见的要求是在层之间附带尽可能少的数据。这减少了资源的消耗。

要达到这个要求, DataSet 类提供了 GetChanges() 方法。这个方法执行许多任务, 并返回一个 DataSet, 其中只包含从源数据集中修改过的行。这是在层之间传递数据时最理想的情况, 因为只传递一组少量的数据。

下面的示例说明了如何生成一个修改后的 DataSet:

```
DataSet source = {some dataset};
DataSet dest = source.GetChanges();
```

这也很乏味, 下面介绍的内容会比较有趣。GetChanges() 方法有两个重载版本, 一个重载版本接受 DataRowState 枚举的一个值, 并返回对应于该状态的行。GetChanges() 方法只调用 GetChanges(Deleted | Modified | Added), 该方法首先检查, 以确保调用 HasChanges() 方法进行了一些修改。如果没有修改, 就立即给调用者返回空值。

下一个操作是复制当前的 DataSet。一旦完成后, 就会创建新的 DataSet, 以忽略违反约束的情况(EnforceConstraints = false), 然后, 把每个表中所有修改的行都复制到新的 DataSet 中。

在得到一个只包含修改内容的 DataSet 后, 就可以把它们移动到数据服务层上进行处理。数据在数据库中更新后, 修改后的数据集就返回给调用者(例如, 来自存储过程的一些输出参数在某些列上有已更新的值), 然后使用 Merge() 方法, 把这些修改的内容合并到源 DataSet 中。操作的顺序如图 30-9 所示。

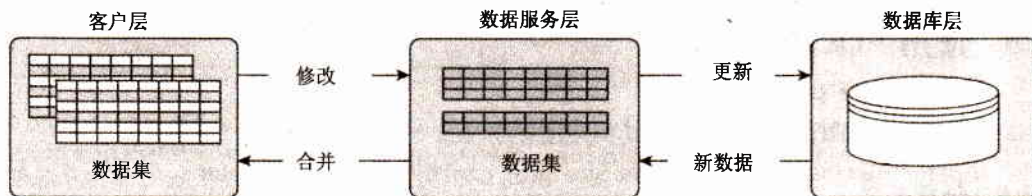


图 30-9

30.9.2 生成 SQL Server 的键

本章前面给出的 RegionInsert 存储过程在给数据库插入数据时生成了一个主键值。该示例中生

成主键的方法相当原始,不能很好地扩展,实际的应用程序应利用生成键的其他策略。

首先要定义一个 **Identity** 列,并从存储过程中返回 @@IDENTITY 值。下面的存储过程显示了如何为 Northwind 样本数据库中的 **Categories** 表定义该值。在 SQL 查询分析器中输入这个存储过程,或者运行代码下载中的 **StoredProcs.sql** 文件:



可从
wrox.com
下载源代码

```
CREATE PROCEDURE CategoryInsert(@CategoryName NVARCHAR(15),
                                @Description NTEXT,
                                @CategoryID INTEGER OUTPUT) AS
SET NOCOUNT OFF
INSERT INTO Categories (CategoryName, Description)
VALUES (@CategoryName, @Description)
SELECT @CategoryID = @@IDENTITY
GO
```

代码下载 StoredProcs.sql

这将把新的一行插入到 **Category** 表中,并给调用者返回生成的主键(**CategoryID** 列的值)。在 SQL 查询分析器中输入下述 SQL 命令,可以测试该过程:

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

当把该过程作为批处理命令来执行时,会把新的一行插入到 **Category** 表中,并返回新记录的标识,然后把该记录显示给用户。

假定过了几个月后,有人要添加一个简单的审计跟踪,它记录所有插入的记录,以及对 **Category** 名的修改。此时,定义如图 30-10 所示的一个表,记录 **Category** 的新名和旧名。

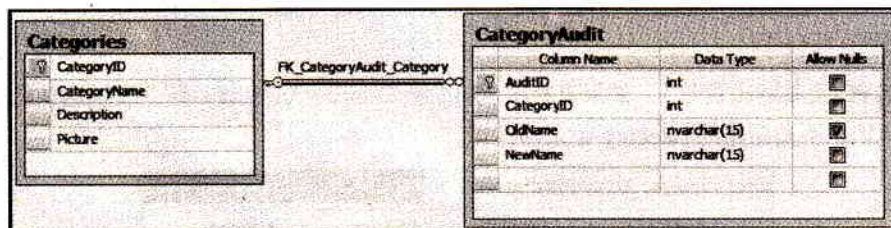


图 30-10

这个表的脚本包含在 **StoredProcs.sql** 文件中。**AuditID** 列定义为一个 **IDENTITY** 列,然后构造两个数据库触发器,数据库触发器记录对 **CategoryName** 字段的修改:



可从
wrox.com
下载源代码

```
CREATE TRIGGER CategoryInsertTrigger
ON Categories
AFTER UPDATE
AS
INSERT INTO CategoryAudit(CategoryID, OldName, NewName)
SELECT old.CategoryID, old.CategoryName, new.CategoryName
FROM Deleted AS old,
Categories AS new
WHERE old.CategoryID = new.CategoryID;
GO
```

代码下载 StoredProcs.sql

对于习惯使用 Oracle 存储过程的用户，SQL Server 其实没有 OLD 行 和 NEW 行的概念，而是使用一个插入触发器，在内存中有一个 Inserted 表可用于插入记录，在 Deleted 表中删除和更新旧记录。

该触发器检索要处理的记录的 CategoryID，把它与 CategoryName 列的新旧值一起存储。

现在，调用原始存储过程插入一个新 CategoryID 时，会得到一个标识值，但它不再是插入到 Categories 表中的行的标识值，而是 CategoryAudit 表中为该行生成的新值。

要查看这些值，可打开 SQL Server Enterprise 管理器的一个副本，查看 Categories 表的内容，如图 30-11 所示。

CategoryID	CategoryName	Description	Picture
1	Beverages	Soft drinks, coffees, teas, beers, and ales	<Binary data>
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	<Binary data>
3	Confections	Desserts, candies, and sweet breads	<Binary data>
4	Dairy Products	Cheeses	<Binary data>
5	Grains/Cereals	Breads, crackers, pasta, and cereal	<Binary data>
6	Meat/Poultry	Prepared meats	<Binary data>
7	Produce	Dried fruit and bean curd	<Binary data>
8	Seafood	Seaweed and fish	<Binary data>
NULL	NULL	NULL	NULL

图 30-11

图 30-11 列出了 Northwind 数据库中的所有类别。

因为 Categories 表中的下一个 identity 值是 9，所以执行下面的代码新插入一行，以查看返回了什么 ID。

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pastries', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

在测试 PC 上其输出值是 1。如果查看图 30-12 中 CategoryAudit 表，则会发现这是新插入的审核记录的标识，不是新建的 category 记录的标识。

Results: Query(v...TA\NORTHWIND.MDF) X				
	AuditID	CategoryID	OldName	NewName
▶	1	9	NULL	Pastries

图 30-12

问题是@@IDENTITY 实际上在起作用，它返回由会话创建的最后一个标识值，如图 30-12 所示，所以它不是 100%的可靠。

除了@@IDENTITY 外，还可以使用另外两个标识函数，它们都不会出任何问题。第一个函数是 SCOPE_IDENTITY()，它返回在当前“范围”内创建的最后一个标识值。SQL Server 把该范围定义为一个存储过程、触发器或函数。在大多数情况下，这个函数可以正常工作。但如果因某种原因，有人在存储过程中添加了另一条 INSERT 语句，就会得到这个值，而不是期望的值。

另一个函数是 IDENT_CURRENT()，它返回任何范围中为给定表生成的最后一个标识值。例如，如果两个用户同时访问 SQL Server，其中一个用户就有可能得到另一个用户生成的标识值。

可以想像，找出这个问题的原因不太容易。在 SQL Server 中使用 IDENTITY 列时要多加小心。

30.9.3 命名约定

下面的提示和约定与 .NET 并不直接相关，但应共享和遵循它们，特别是在给约束命名时。如果您对此主题已经有自己的观点，就可以跳过本节。

1. 数据库表的约定

- 总是使用单数名称——例如，`Product` 而不是 `Products`，这是一个普遍适用的约定，因为我们必须给客户解释某种数据库架构，从语法上看，“`Product` 表包含产品”要比“`Products` 表包含产品”好得多。但 `Northwind` 数据库并没有遵循这一约定。
- 给表中的字段采用某种形式的命名约定——我们采用的是表的主键 `<Table>_ID` (假定主键是一列)，字段采用 `Name`，这是考虑到记录的用户友好性，记录本身的任何文本信息采用 `Description`。采用好的命名约定意味着，只要查看一下数据库中的几乎任何表，从直觉上就知道其中的字段主要用于什么目的。

2. 数据库列的约定

- 使用单数名称，而不是复数名称。
- 链接到另一个表中的列名应与该表的主键名相同。例如，链接到 `Product` 表的列名为 `Product_ID`。链接到 `Sample` 表的列名为 `Sample_ID`。这并不总是可行的，特别是如果一个表有另一个表的多个引用，这个命名约定就无效。此时应使用其他方式命名。
- 日期字段应有一个 `_On` 后缀，如 `Modified_On`、`Created_On`。按照这种命名约定，如果读取一些 SQL 输出，就很容易从列的名称中知道该列的含义。
- 记录用户的操作的字段名应有一个 `_By` 后缀，如 `Modified_By` 和 `Created_By`，这将有助于理解。

3. 约束的约定

- 如果可能，在约束名中包含表名和列名，如 `CK_<Table>_<Field>`。例如，对于 `Person` 表中的 `Sex` 列，其检查约束可以是 `CK_Person_Sex`，而对于 `product` 和 `supplier` 之间的外键关系，对应的外键约束是 `FK_Product_Supplier_ID`。
- 约束类型的前面加一个前缀，如 `CK` 表示检查约束，`FK` 表示外键约束。也可以指定更为特殊的名称，如 `Age` 列上的 `CK_Person_Age_GT0` 表示该年龄应大于 0。
- 如果必须限制约束名的长度，则可以在其中包含表名，而不包含列名。在发现有违反约束的情况时，通常很容易推断哪个表出现错误，但有时不容易检查出是哪一列出了问题。`Oracle` 允许名称的长度最长为 30 个字符，而很容易超过这种限制。

4. 存储过程

在过去几年中，把 `C` 放在每个声明的类前面令人困惑。同样，许多 `SQL Server` 开发人员也困惑于在每个存储过程的前面加上 `sp_` 或类似的东西，这不是一个好方法。

`SQL Server` 在所有的系统存储过程前面使用 `sp_` 前缀。所以用户会对 `sp_widget` 是否为 `SQL Server` 标准存储过程产生疑问。另外，当查找存储过程时，`SQL Server` 会把带有 `sp_` 前缀的过程与没有该

前缀的过程区别对待。

如果使用这个前缀,但没有用该存储过程的数据库/所有者来限定,SQL Server 就会在当前范围内查找,然后跳到主数据库中查找存储过程。没有 `sp_` 前缀,用户就会早一些得到错误。更糟糕的是本地存储过程(在自己的数据库中创建)与系统存储过程有相同的名称和参数。应尽可能避免这种情况,如果有疑问,就不要使用前缀。

在调用存储过程时,它们应以过程的拥有者作为前缀,如 `dbo.selectWidgets`。这将比不使用该前缀略快一些,因为 SQL Server 查找该存储过程所做的工作较少。有时这不会对应用程序的执行速度有很大的影响,但它本质上是一个可以自由使用的调整技巧。

总之,在数据库或代码中命名实体时,应保持一致。

30.10 小结

数据访问是一个很大的主题,特别是在 .NET 中,因为有非常丰富的内容要涵盖。本章概述了 ADO.NET 名称空间中的主要类,揭示了在处理数据源中的数据时如何使用这些类。

首先,使用 `SqlConnection` (SQL Server 专用)和 `OleDbConnection` (用于任何 OLE DB 数据源),探讨了 `Connection` 对象的用法。这两个类的编程模型非常类似,以致于一般其中一个类可以替代另一个类,代码仍能继续运行。

接着阐述了如何正确地进行连接,这样稀缺的资源就可以尽可能早地关闭。所有连接类都实现 `IDisposable` 接口,在对象放在 `using` 子句中时调用该接口。如果本章只有一件值得注意的事,那就是尽早关闭数据库连接的重要性。

此外,通过执行没有返回数据的示例,和利用输入和输出参数调用存储过程的示例,讨论了数据库命令。本章描述了各种执行方法,包括只能在 SQL Server 提供程序上使用的 `ExecuteXmlReader()` 方法。这大大简化了基于 XML 的数据的选择和处理。

这里详细介绍了 `System.Data` 名称空间中的泛型类,包括 `DataSet`、`DataTable`、`DataColumn`、`DataRow`,以及关系和约束。`DataSet` 类是数据的最佳容器,各种方法使之成为跨层数据流的理想容器。`DataSet` 中的数据可以用 XML 来表示,以利于传输,另外,在层之间传输最少量数据的方法可用。把许多数据表放在一个 `DataSet` 中可以大大提高其可用性。

除了把架构存储在 `DataSet` 中外,.NET 还包括数据适配器,它与各种 `Command` 对象组合使用,可以把数据选择出来,放在 `DataSet` 中,以后还可以更新数据存储器中的数据。数据适配器的一个优点是可以为 4 种操作(`SELECT`、`INSERT`、`UPDATE` 和 `DELETE`)定义不同的命令。系统可以根据数据库架构信息和一条 `SELECT` 语句创建一组默认的命令,但为了得到最佳性能,可以使用一组存储过程,并相应地定义 `DataAdapter` 的命令,仅把需要的信息传送给这些存储过程。

我们还介绍了 XSD 工具(`xsd.exe`),使用一个示例来说明如何在 .NET 中基于 XML 架构使用类。产生的类可以用于应用程序,它们的自动生成减少了大量的输入工作。

最后论述了用于数据库开发的一些最佳实践和命名约定。

访问 SQL Server 数据库的更多知识,详见第 34 章。

第 31 章

ADO.NET Entity Framework

本章内容:

- ADO.NET Entity Framework
- 数据库表和实体类之间的映射
- 创建实体类及其特性
- 对象上下文
- 关系
- 对象查询
- 更新
- LINQ to Entities

ADO.NET Entity Framework 是一个对象-关系的映射架构,它提供了 ADO.NET 的一个抽象,可基于引用的数据库获取对象模型。本章使用 CSDL(Conceptual Schema Definition Language, 概念架构定义语言)、SSDL(Storage Schema Definition Language, 存储架构定义语言)和 MSL(Mapping Schema Language, 映射架构语言)给出数据库和实体类之间的映射信息。讨论实体之间的不同关系,如对象的一个层次结构一个表关系、一个类型一个表关系和 n 对 n 关系。

本章还将描述从代码中直接通过 Entity Client 访问数据库的不同方式,如何使用 Entity SQL 或帮助方法创建 Entity SQL,如何使用 LINQ to Entities,也会讨论对象跟踪,以及数据上下文如何包含变化的信息,以更新数据。



本章使用 Books、Formula1 和 Northwind 数据库。Northwind 数据库可以从 msdn.microsoft.com 上下载,Books 和 Formula1 数据库包含在 <http://www.wrox.com> 和随书附赠光盘中代码示例的下载包中。

31.1 ADO.NET Entity Framework 概述

ADO.NET Entity Framework 提供了从关系数据库架构到对象的映射。关系数据库和面向对象的语言用不同的方式定义了关联。例如,Microsoft 样本数据库 Northwind 包含 Customers 表和 Orders 表。要访问某个顾客的所有 Orders 行,需要执行一条 SQL join 语句。在面向对象的语言中,更常见

的是定义一个 Customer 类和一个 Order 类，使用 Customer 类的 Orders 属性访问顾客的订单。

对于对象-关系映射，自从.NET 1.0 以来，就可以使用 DataSet 类和类型化的数据集。DataSet 非常类似于数据库的结构，它包含 DataTable、DataRow、DataColumn 和 DataRelation 类，而不提供对象支持。ADO.NET Entity Framework 支持直接定义完全独立于数据库结构的实体类，并把它们映射到数据库的表和关系上。通过应用程序使用对象，应用程序就可以免受数据库修改的影响。

ADO.NET Entity Framework 使用 Entity SQL 为存储器定义基于实体的数据库查询。LINQ to Entities 允许使用 LINQ 语法来查询数据。对象上下文保存了变化的实体信息，从而在把实体写回存储器时，提供这些信息。

包含 ADO.NET Entity Framework 中的类的名称空间如表 31-1 所示。

表 31-1

名 称 空 间	说 明
System.Data	这是用于 ADO.NET 的主要名称空间。在 ADO.NET Entity Framework 中，这个名称空间包含了与实体相关的异常类，如 MappingException 和 QueryException 异常类
System.Data.Common	这个名称空间包含由 .NET 数据提供程序共享的类。DbProviderServices 类是一个抽象类，它必须由 ADO.NET Entity Framework 提供程序实现
System.Data.Common .CommandTrees	这个名称空间包含构建表达式树的类
System.Data.Entity.Design	这个名称空间包含由设计器用于创建 EDM (Entity Data Model, 实体数据模型)文件的类
System.Data.EntityClient	这个名称空间指定由 .NET Framework 数据提供程序访问 ADO.NET Entity Framework 的类。EntityConnection、EntityCommand 和 EntityReader 可用于访问 Entity Framework
System.Data.Objects	这个名称空间包含用于查询和更新数据库的类。ObjectContext 类封装了与数据库的连接，用作创建、读取、更新和删除方法的网关。ObjectQuery 类表示对存储器的一个查询。CompileQuery 是一个缓存的查询
System.Data.Objects .DataClasses	这个名称空间包含实体需要的类和接口

31.2 Entity Framework 映射

ADO.NET Entity Framework 提供了几个把数据库表映射到对象上的层。可以从一个数据库架构开始，使用 Visual Studio 项模板创建完整的映射。还可以先用设计器设计实体类，再把它映射到数据库上，在该数据库中，表和表之间的关系可以有完全不同的结构。

需要定义的层如下：

- 逻辑层——该层定义关系数据
- 概念层——该层定义 .NET 类
- 映射层——该层定义从 .NET 类到关系表和关联的映射。

下面先从一个简单的数据库架构开始，如图 31-1 所示，其中包含 Books 表和 Authors 表，以及

关联表 BookAuthors，它把作者映射到图书上。

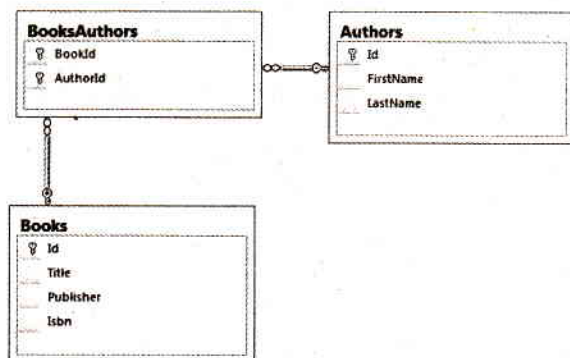


图 31-1

31.2.1 逻辑层

逻辑层由 SSDL(Store Schema Definition Language, 存储架构定义语言)定义, 描述了数据库表及其关系的结构。

下面的代码使用 SSDL 来描述 3 个表: Books、Authors 和 BookAuthors。EntityType 元素描述所有包含 EntitySet 元素的表和包含 AssociationSet 元素的表。表的部分用 EntityType 元素定义。在 EntityType Books 中, Id、Title、Publisher 和 ISBN 列用 Property 元素定义。Property 元素包含了定义数据类型的 XML 属性。Key 元素定义表的键。



可从
WROX.COM
下载源代码

```

<edmx:StorageModels>
  <Schema Namespace="BooksModel.Store" Alias="Self" Provider="System.Data.SqlClient"
    ProviderManifestToken="2008"
    xmlns:store=
      "http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
    xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
    <EntityContainer Name="BooksModelStoreContainer">
      <EntitySet Name="Authors" EntityType="BooksModel.Store.Authors"
        store:Type="Tables"
        Schema="dbo" />
      <EntitySet Name="Books" EntityType="BooksModel.Store.Books"
        store:Type="Tables"
        Schema="dbo" />
      <EntitySet Name="BooksAuthors" EntityType="BooksModel.Store.BooksAuthors"
        store:Type="Tables" Schema="dbo" />
      <AssociationSet Name="FK_BooksAuthors_Authors"
        Association="BooksModel.Store.FK_BooksAuthors_Authors">
        <End Role="Authors" EntitySet="Authors" />
        <End Role="BooksAuthors" EntitySet="BooksAuthors" />
      </AssociationSet>
      <AssociationSet Name="FK_BooksAuthors_Books"
        Association="BooksModel.Store.FK_BooksAuthors_Books">
        <End Role="Books" EntitySet="Books" />
        <End Role="BooksAuthors" EntitySet="BooksAuthors" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Authors">
      <Key>

```

```

    <PropertyRef Name="Id" />
  </Key>
  <Property Name="Id" Type="int" Nullable="false"
    StoreGeneratedPattern="Identity" />
  <Property Name="FirstName" Type="nvarchar" Nullable="false"
    MaxLength="50" />
  <Property Name="LastName" Type="nvarchar" Nullable="false"
    MaxLength="50" />
</EntityType>
<EntityType Name="Books">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Name="Id" Type="int" Nullable="false"
    StoreGeneratedPattern="Identity" />
  <Property Name="Title" Type="nvarchar" Nullable="false" MaxLength="50" />
  <Property Name="Publisher" Type="nvarchar" Nullable="false"
    MaxLength="50" />
  <Property Name="Isbn" Type="nchar" MaxLength="18" />
</EntityType>
<EntityType Name="BooksAuthors">
  <Key>
    <PropertyRef Name="BookId" />
    <PropertyRef Name="AuthorId" />
  </Key>
  <Property Name="BookId" Type="int" Nullable="false" />
  <Property Name="AuthorId" Type="int" Nullable="false" />
</EntityType>
<Association Name="FK_BooksAuthors_Authors">
  <End Role="Authors" Type="BooksModel.Store.Authors" Multiplicity="1" />
  <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors"
    Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Authors">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="BooksAuthors">
      <PropertyRef Name="AuthorId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
<Association Name="FK_BooksAuthors_Books">
  <End Role="Books" Type="BooksModel.Store.Books" Multiplicity="1" />
  <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors"
    Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Books">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="BooksAuthors">
      <PropertyRef Name="BookId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
</Schema>

```

```
</edmx:StorageModels>
```

```
代码段 BooksDemo/BooksModel.edmx
```



BooksModel.edmx 文件包含 SSDL、CSDL 和 MSL。使用 XML 编辑器可以打开这个文件，查看其内容。

31.2.2 概念层

概念层定义了 .NET 类。该层用 CSDL (Conceptual Schema Definition Language, 概念架构定义语言) 定义。

图 31-2 显示了用 ADO.NET 实体数据模型设计器定义的实体 Author 和 Book。

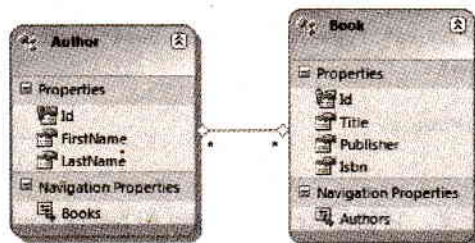


图 31-2

下面是定义实体类型 Author 和 Book 的 CSDL 内容。从 Books 数据库中创建它们。



可从
wrox.com
下载源代码

```
<edmx:ConceptualModels>
  <Schema Namespace="BooksModel" Alias="Self"
    xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
    <EntityContainer Name="BooksEntities" annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Authors" EntityType="BooksModel.Author" />
      <EntitySet Name="Books" EntityType="BooksModel.Book" />
      <AssociationSet Name="BooksAuthors" Association="BooksModel.BooksAuthors">
        <End Role="Authors" EntitySet="Authors" />
        <End Role="Books" EntitySet="Books" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Author">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="Int32" Nullable="false"
        annotation:StoreGeneratedPattern="Identity" />
      <Property Name="FirstName" Type="String" Nullable="false" MaxLength="50"
        Unicode="true" FixedLength="false" />
      <Property Name="LastName" Type="String" Nullable="false" MaxLength="50"
        Unicode="true" FixedLength="false" />
      <NavigationProperty Name="Books" Relationship="BooksModel.BooksAuthors"
        FromRole="Authors" ToRole="Books" />
    </EntityType>
    <EntityType Name="Book">
```

```

<Key>
  <PropertyRef Name="Id" />
</Key>
<Property Name="Id" Type="Int32" Nullable="false"
  annotation:StoreGeneratedPattern="Identity" />
<Property Name="Title" Type="String" Nullable="false" MaxLength="50"
  Unicode="true"
  FixedLength="false" />
<Property Name="Publisher" Type="String" Nullable="false" MaxLength="50"
  Unicode="true" FixedLength="false" />
<Property Name="Isbn" Type="String" MaxLength="18" Unicode="true"
  FixedLength="true" />
<NavigationProperty Name="Authors" Relationship="BooksModel.BooksAuthors"
  FromRole="Books" ToRole="Authors" />
</EntityType>
<Association Name="BooksAuthors">
  <End Role="Authors" Type="BooksModel.Author" Multiplicity="*" />
  <End Role="Books" Type="BooksModel.Book" Multiplicity="*" />
</Association>
</Schema>
</edmx:ConceptualModels>

```

代码段 BooksDemo/BooksModel.edmx

实体用 **EntityType** 元素定义，它包含 **Key**、**Property** 和 **NavigationProperty** 元素，以描述所创建的类的属性。**Property** 元素包含的属性描述设计器生成的类的.NET 特性的名称和类型。**Association** 元素连接 **Author** 和 **Book** 类型。**Multiplicity="***表示一个作者可以编写多本图书，和一本图书可以由多个作者编写。

31.2.3 映射层

映射层使用 **MSL**(Mapping Specification Language, 映射规范语言)把 **CSDL** 中的实体类型定义映射到 **SSDL** 上。下面的规范包含一个 **Mapping** 元素，该元素包含 **EntityTypeMapping** 元素来引用 **CSDL** 的 **Book** 类型，并定义 **MappingFragment** 来引用 **SSDL** 中的 **Authors** 表。**ScalarProperty** 把包含 **Name** 特性的.NET 类的属性映射到包含 **ColumnName** 特性的数据库表的列上。



可从
wtox.com
下载源代码

```

<edmx:Mappings>
  <Mapping Space="C - S" xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
    <EntityContainerMapping StorageEntityContainer="BooksModelStoreContainer"
      CdmEntityContainer="BooksEntities">
      <EntitySetMapping Name="Authors">
        <EntityTypeMapping TypeName="BooksModel.Author">
          <MappingFragment StoreEntitySet="Authors">
            <ScalarProperty Name="Id" ColumnName="Id" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <EntitySetMapping Name="Books">
        <EntityTypeMapping TypeName="BooksModel.Book">
          <MappingFragment StoreEntitySet="Books">
            <ScalarProperty Name="Id" ColumnName="Id" />

```

```

        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="Publisher" ColumnName="Publisher" />
        <ScalarProperty Name="Isbn" ColumnName="Isbn" />
    </MappingFragment>
</EntityTypeMapping>
</EntitySetMapping>
<AssociationSetMapping Name="BooksAuthors" TypeName="BooksModel.BooksAuthors"
    StoreEntitySet="BooksAuthors">
    <EndProperty Name="Authors">
        <ScalarProperty Name="Id" ColumnName="AuthorId" />
    </EndProperty>
    <EndProperty Name="Books">
        <ScalarProperty Name="Id" ColumnName="BookId" />
    </EndProperty>
    </AssociationSetMapping>
</EntityContainerMapping>
</Mapping>
</edmx:Mappings>

```

代码段 BooksDemo/BooksModel.edmx

31.3 Entity Client

访问 Entity Framework 的低级 API 在 System.Data.SqlClient 名称空间中。这个名称空间包含一个 ADO.NET 提供程序，它使用 EDM(Entity Data Model, 实体数据模型)来访问数据库。这个 ADO.NET 提供程序定义的类(如第 30 章所述)派生自基类 DbConnection、DbCommand、DbParameter 和 DbDataReader。在这里，这些类称为 EntityConnection、EntityCommand、EntityParameter 和 EntityDataReader。

使用这些类的方式与第 30 章相同，但需要一个特殊的连接字符串，且使用 Entity SQL 代替 T-SQL，来访问 EDM。

与数据库的连接通过 EntityConnection 类来实现，它需要一个实体连接字符串。这个字符串通过 System.Configuration 名称空间中的 ConfigurationManager 类从配置文件中读取。EntityConnection 类的 CreateCommand()方法返回一个 EntityCommand。EntityCommand 的命令文本使用 CommandText 属性来指定，且需要一条 Entity SQL 命令。BooksEntities.Books 从 BooksEntities CSDL 定义的 EntityContainer 元素中生成，Books EntitySet 从 Books 表中获取所有图书。command.ExecuteReader 返回一个 EntityDataReader，该 EntityDataReader 逐行读取数据：



可从
wrox.com
下载源代码

```

string connectionString = ConfigurationManager.ConnectionStrings["BooksEntities"].
    ConnectionString;
var connection = new EntityConnection(connectionString);
connection.Open();

EntityCommand command = connection.CreateCommand();
command.CommandText = "[BooksEntities].[Books]";
EntityDataReader reader = command.ExecuteReader(
    CommandBehavior.CloseConnection | CommandBehavior.SequentialAccess);
while (reader.Read())
{
    Console.WriteLine("{0}, {1}", reader["Title"], reader["Publisher"]);
}
reader.Close();

```

这段代码的作用与第 30 章的对应代码相同。唯一的区别是连接字符串和 Entity SQL 语句, 下面就讨论它们。

31.3.1 连接字符串

在上面的代码段中, 从配置文件中读取连接字符串。EDM 需要连接字符串, 它不同于一般的 ADO.NET 连接字符串, 因为需要映射信息。映射使用关键字 `metadata` 来定义, `metadata` 需要 3 个对象: 带分隔符的映射文件列表、不变的提供程序名 `Provider`(该提供程序用于访问数据源)和 `Provider connection string`(用于指定依赖于提供程序的连接字符串)。

带分隔符的映射文件列表引用 `BooksModel.cSDL`、`BooksModel.sSDL` 和 `BooksModel.mSL` 文件, 它们包含在程序集中用 `res:` 前缀定义的的资源里。在 Visual Studio 中, 设计人员只使用了一个文件 `BooksModel.edmx`, 它包含 CSDL、SSDL 和 MSL。把 Custom Tool 属性设置为 `EntityModelCodeGenerator`, 会创建包含在资源中的 3 个文件。

在 `connectionString` 设置中, 可以使用连接字符串设置找到数据库的连接字符串。这部分与简单的 ADO.NET 连接字符串相同, 且取决于用 `provider` 设置的提供程序。

```
<connectionStrings>
  <add name=" BooksEntities"
    connectionString="metadata=res://*/BooksModel.cSDL|res://*/BooksModel.sSDL|
      res://*/BooksModel.mSL;provider=System.Data.SqlClient;
      provider connection string="Data Source=(local);Initial Catalog=Books;
        Integrated Security=True;Pooling=False;MultipleActiveResultSets=True";"
    providerName=" System.Data.EntityClient" />
</connectionStrings>
```



利用连接字符串还可以指定没有在程序集中包含为资源的 CSDL、SSDL 和 MSL 文件。如果希望在部署项目后改变这些文件的内容, 采用这种方式就很有用。

31.3.2 Entity SQL

通过 Entity Client 查询数据时, 需要使用 Entity SQL。Entity SQL 通过添加类型, 增强了 T-SQL。这种语法不需要连接语句, 因为可以使用实体的关联来替代。

这里仅列出了几个语法选项, 它们有助于开始使用 Entity SQL。在 MSDN 文档中有完整的参考资料。

前面的示例说明了 Entity SQL 如何在 EntityContainer 和 EntitySet 中使用 CSDL 中的定义, 例如, 使用 `BooksEntities.Books` 从 `Books` 表中获取所有图书, 因为 `Books EntitySet` 映射到 SSDL 中的 `Books EntitySet` 上。

除了检索所有列之外, 还可以使用 `EntityType` 的 `Property` 元素。这看起来非常类似于上一章使用的 T-SQL 查询:

```
command.CommandText = "SELECT Books.Title, Books.Publisher FROM " +
  "BooksEntities.Books";
```



可从
wrox.com
下载源代码

Entity SQL 中没有 SELECT *。前面通过请求 EntitySet 来检索所有列。使用 SELECT VALUE 还可以获得所有列，如下一段代码所示。这条语句还使用一个包含 WHERE 的筛选器，通过查询获取特定的出版商。注意 CommandText 用 @ 字符指定参数，但添加到 Parameters 集合中的参数不使用 @ 字符把某个值写入相同的参数中：

```
command.CommandText = "SELECT VALUE it FROM BooksEntities.Books AS it WHERE " +
    "it.Publisher = @Publisher";
command.Parameters.AddWithValue("Publisher", "Wrox Press");
```

31.4 实体

用设计器和 CSDL 创建的实体类一般派生自基类 EntityObject，如下面代码中的 Book 类所示。

这个 Book 类派生自基类 EntityObject，并为其数据定义属性，如 Title 和 Publisher。这些属性的 set 访问器以两种不同的方式触发信息的改变。一种方式是调用 EntityObject 基类的 ReportPropertyChanging() 和 ReportPropertyChanged() 方法。调用这些方法会使用 INotifyPropertyChanging 和 INotifyPropertyChanged 接口，以通知每个客户端用这些接口的事件来注册。

另一种方式是使用部分方法，如 OnTitleChanging() 和 OnTitleChanged()。它们默认没有实现方式，但可以在这个类的自定义扩展中实现它们。Authors 属性使用 RelationshipManager 类给作者返回 Book。



可从
wrox.com
下载源代码

```
[EdmEntityTypeAttribute (NamespaceName="BooksModel", Name="Book")]
[Serializable()]
[DataContractAttribute (IsReference=true)]
public partial class Book : EntityObject
{
    public static Book CreateBook(int id, string title, string publisher)
    {
        Book book = new Book();
        book.Id = id;
        book.Title = title;
        book.Publisher = publisher;
        return book;
    }

    [EdmScalarPropertyAttribute (EntityKeyProperty=true, IsNullable=false)]
    [DataMemberAttribute()]
    public int Id
    {
        get
        {
            return _Id;
        }
        set
        {
            if (_Id != value)
            {
                OnIdChanging(value);
                ReportPropertyChanging("Id");
                _Id = StructuralObject.SetValidValue(value);
                ReportPropertyChanged("Id");
            }
        }
    }
}
```

```
        OnIdChanged();
    }
}

private int _Id;
partial void OnIdChanging(int value);
partial void OnIdChanged();

[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
[DataMemberAttribute()]
public string Title
{
    get
    {
        return _Title;
    }
    set
    {
        OnTitleChanging(value);
        ReportPropertyChanging("Title");
        _Title = StructuralObject.SetValidValue(value, false);
        ReportPropertyChanged("Title");
        OnTitleChanged();
    }
}

private string _Title;
partial void OnTitleChanging(string value);
partial void OnTitleChanged();

[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
[DataMemberAttribute()]
public string Publisher
{
    get
    {
        return _Publisher;
    }
    set
    {
        OnPublisherChanging(value);
        ReportPropertyChanging("Publisher");
        _Publisher = StructuralObject.SetValidValue(value, false);
        ReportPropertyChanged("Publisher");
        OnPublisherChanged();
    }
}

private string _Publisher;
partial void OnPublisherChanging(string value);
partial void OnPublisherChanged();

[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=true)]
[DataMemberAttribute()]
public string Isbn
{
    get
    {

```

```

        return _Isbn;
    }
    set
    {
        OnIsbnChanging(value);
        ReportPropertyChanging("Isbn");
        _Isbn = StructuralObject.SetValidValue(value, true);
        ReportPropertyChanged("Isbn");
        OnIsbnChanged();
    }
}
private string _Isbn;
partial void OnIsbnChanging(string value);
partial void OnIsbnChanged();

[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
[EdmRelationshipNavigationPropertyAttribute("BooksModel", "BooksAuthors", "Authors")]
public EntityCollection<Author>Authors
{
    get
    {
        return ((IEntityWithRelationships)this).RelationshipManager.
            GetRelatedCollection<Author>("BooksModel.BooksAuthors", "Authors");
    }
    set
    {
        if ((value != null))
        {
            ((IEntityWithRelationships)this).RelationshipManager.
                InitializeRelatedCollection<Author>("BooksModel.BooksAuthors",
                                                    "Authors", value);
        }
    }
}
}

```

代码段 BooksDemo/BooksModel.Designer.cs

与实体类相关的重要类和接口如表 31-2 所示。除了两个接口 `INotifyPropertyChanging` 和 `INotifyPropertyChanged` 之外，所有类型都是在 `System.Data.Objects.DataClasses` 名称空间中定义。

表 31-2

类 或 接 口	说 明
<code>StructuralObject</code>	<code>StructuralObject</code> 是 <code>EntityObject</code> 和 <code>ComplexObject</code> 类的基类。这个类实现 <code>INotifyPropertyChanging</code> 和 <code>INotifyPropertyChanged</code> 接口
<code>INotifyPropertyChanging</code> <code>INotifyPropertyChanged</code>	这些接口定义 <code>PropertyChanging</code> 和 <code>PropertyChanged</code> 事件，允许在对象的状态变化时订阅信息。这两个接口与其他类和接口不同，因为它们在 <code>System.ComponentModel</code> 名称空间中定义

(续表)

类 或 接 口	说 明
EntityObject	这个类派生自 StructuralObject，并实现 IEntityWithKey、IEntityWithChangeTracker 和 IEntityWithRelationships 接口。EntityObject 是一个常用的基类，用于映射到数据库表上的对象，该数据库表包含一个键以及与其他对象的关系
ComplexObject	这个类可以用作没有键的实体对象的基类，它派生自 StructuralObject，但没有实现与 EntityObject 类相同的其他接口
IEntityWithKey	这个接口定义 EntityKey 属性，允许快速访问对象
IEntityWithChangeTracker	这个接口定义 SetChangeTracker()方法，其中，实现 IChangeTracker 接口的变化跟踪器可以指定从对象中获得状态变化的信息
IEntityWithRelationships	这个接口定义只读属性 RelationshipManager，它返回一个用于在对象之间导航的 RelationshipManager 对象



实体类不一定派生自基类 EntityObject 或 ComplexObject，而它可以实现需要的接口。

Book 实体类很容易使用对象上下文类 BookEntities 来访问。Book 属性返回一个可以迭代的 Book 对象集合：



可从
wrox.com
下载源代码

```
using (var data = new BooksEntities())
{
    foreach (var book in data.Books)
    {
        Console.WriteLine("{0}, {1}", book.Title, book.Publisher);
    }
}
```

代码段 BooksDemo/Program.cs

31.5 对象上下文

要从数据库中检索数据，需要使用ObjectContext类。这个类定义了从实体对象到数据库的映射。在核心ADO.NET中，这个类可以和填充DataSet的数据适配器相媲美。

设计器创建的BookEntities类派生自基类ObjectContext。这个类添加了构造函数，来传递连接字符串。在默认的构造函数中，从配置文件中读取连接字符串，也可以把一个已经打开的连接以EntityConnection实例的形式传递给构造函数。如果把没有打开的连接传递给构造函数，对象上下文就会打开和关闭该连接。如果传递已打开的连接，也需要关闭它。

所创建的类定义Books和Authors属性，它们返回一个ObjectSet<TEntity>。ObjectSet<TEntity>是.NET 4新增的，它派生自ObjectQuery<TEntity>。



```

public partial class BooksEntities :ObjectContext
{
    public BooksEntities() : base("name=BooksEntities", "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    public BooksEntities(string connectionString) : base(connectionString,
        "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    public BooksEntities(EntityConnection connection) : base(connection,
        "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    partial void OnContextCreated();

    public ObjectSet<Author> Authors
    {
        get
        {
            if ((_Authors == null))
            {
                _Authors = base.CreateObjectSet<Author>("Authors");
            }
            return _Authors;
        }
    }

    private ObjectSet<Author> _Authors;

    public ObjectSet<Book> Books
    {
        get
        {
            if ((_Books == null))
            {
                _Books = base.CreateObjectSet<Book>("Books");
            }
            return _Books;
        }
    }

    private ObjectSet<Book> _Books;
}

```

代码段 BooksDemo/BooksModel.Designer.cs

ObjectContext 类给调用者提供了几个服务:

- 跟踪已经检索到的实体对象。如果再次查询该对象,就从对象上下文中提取它。
- 保存实体的状态信息。可以获得已添加、修改和删除对象的信息。

- 更新对象上下文中的实体，把改变的内容写入底层存储器中。

ObjectContext 类的方法和属性如表 31-3 所示。

表 31-3

ObjectContext 类的方法和属性	说 明
Connection	返回一个与对象上下文关联的 DbConnection 对象
MetadataWorkspace	返回一个 MetadataWorkspace 对象，该对象可用于读取元数据和映射信息
QueryTimeout	使用这个属性可以获取和设置对象上下文的查询的超时值
ObjectStateManager	这个属性返回一个 ObjectStateManager，该 ObjectStateManager 会跟踪检索到的实体对象和该对象在对象上下文中的变化
CreateQuery()	这个方法返回一个 ObjectQuery，以从存储器中获取数据。前面的 Books 和 Authors 属性就使用这个方法返回一个 ObjectQuery
GetObjectByKey() TryGetObjectByKey()	这两个方法根据键从对象状态管理器或底层存储器中返回对象。如果键不存在，GetObjectByKey() 方法就抛出一个 ObjectNotFoundException 类型的异常，而 TryGetObjectByKey() 方法返回 false
AddObject()	这个方法在对象上下文中添加一个新的实体对象
DeleteObject()	这个方法从对象上下文中删除一个对象
Detach()	这个方法从对象上下文中分离出实体对象，使不再在出现变化时被跟踪该对象
Attach() AttachTo()	Attach() 方法把分离出的对象附加到存储器中。把对象附加回对象上下文上，需要实体对象实现 IEntityWithKey 接口。AttachTo() 方法不需要对象带有键，但它需要一个实体集名称，需要附加的实体对象就放在这个实体集中
ApplyPropertyChanges()	如果对象从对象上下文中分离出来，就修改解除分离出来的对象，在把进行的修改应用于对象上下文中的对象之前，可以调用 ApplyPropertyChanges() 方法应用这些修改。当分离出来的对象从 Web 服务中返回，从客户端上修改它，再把它传递给 Web 服务时，这个方法很有用
Refresh()	实体对象存储在对象上下文中时，存储器中的数据可以修改。为了用存储器中的变化进行刷新，可以使用 Refresh() 方法。在这个方法中，可以传递一个 RefreshMode 枚举值。如果用于对象的值在存储器和对象上下文中不同，就传递 ClientWins 值，修改存储器中的数据。若传递 StoreWins 值，就修改对象上下文中的数据
SaveChanges()	从对象上下文中添加、修改和删除对象，不会改变底层存储器中的对象。使用 SaveChanges() 方法可以把这些修改持久化到存储器中
AcceptAllChanges()	这个方法把对象上下文中对象的状态改为未修改。SaveChanges() 方法隐式地调用这个方法

31.6 关系

实体类型 Book 和 Author 相互关联。一本书可以由一个或多个作者编写，一个作者也可以编写

一本或多本书。对应关系基于关联类型的个数和多样性。ADO.NET Entity Framework 支持几种关系, 这里介绍其中一些关系, 包括 TPT(Table per Type, 一种类型一个表)和 TPH(Table per Hierarchy, 一个层次结构一个表)。多样性可以是一对一、一对多和多对多。

31.6.1 一个层次结构一个表

在 TPH 中, 数据库中的一个表对应实体类的一个层次结构。数据库表 Payments(如图 31-3 所示) 包含的列对应于实体类型的一个层次结构。一些列在该层次结构中由所有实体共享, 如 Id 和 Amount, CreditCardNumber 列仅用于信用卡付费。

全部映射到同一个 Payments 表的实体类如图 31-4 所示。Payment 是一个抽象基类, 它包含的属性用于该层次结构中的所有类型。派生自 Payment 基类的具体类有 CreditCardPayment、CashPayment 和 ChequePayment。除了基类的属性之外, CreditCardPayment 类还有一个 CreditCard 属性, ChequePayment 类还有一个 BankName 属性。

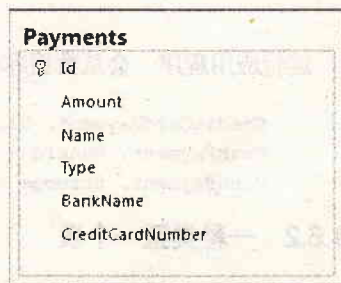


图 31-3

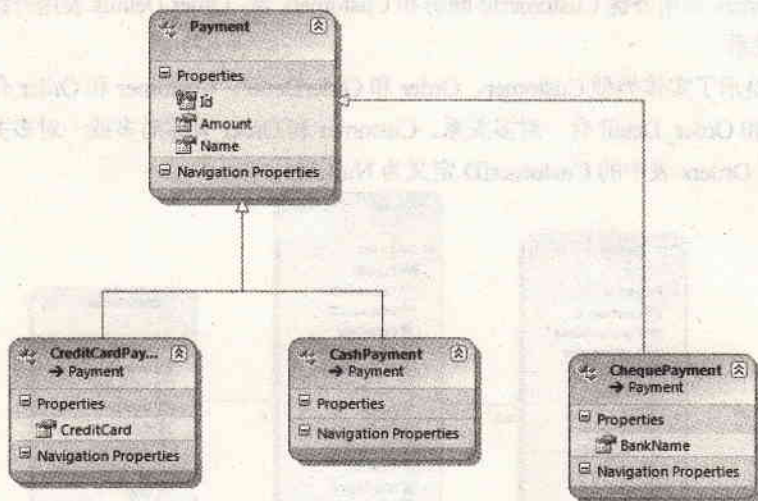


图 31-4

所有这些映射都可以使用设计器来定义。映射细节可以使用 Mapping Details 窗口配置, 如图 31-5 所示。具体类的类型选择基于一个 Condition 元素, Condition 元素用 Maps to Payments When Type = CREDIT 定义。基于 Type 列的值选择对应的类型。还可以使用其他用于选择类型的选项, 例如, 可以验证某一列是否不为空。

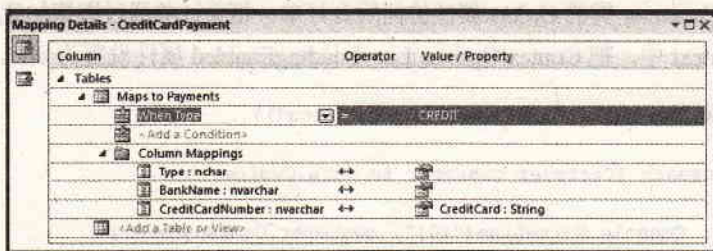


图 31-5

现在，可以迭代 `Payments` 表中的数据，根据映射返回不同的类型：



```
using (var data = new PaymentsEntities())
{
    foreach (var p in data.Payments)
    {
        Console.WriteLine("{0}, {1} - {2:C}", p.GetType().Name, p.Name, p.Amount);
    }
}
```

代码段 `PaymentsDemo/Program.cs`

运行应用程序，会从数据库中返回两个 `CashPayment` 对象和一个 `CreditCardPayment` 对象：

```
CreditCardPayment, Gladstone - $22.00
CashPayment, Donald - $0.50
CashPayment, Scrooge - $80,000.00
```

31.6.2 一种类型一个表

在 TPT 中，一个表仅映射一个类型。`Northwind` 数据库的架构是 `Customers` 类、`Orders` 类和 `Order Details` 表。`Orders` 表用外键 `CustomerId` 映射和 `Customers` 表，`Order Details` 表用外键 `OrderID` 映射和 `Orders` 表的关系。

图 31-6 显示了实体类型 `Customer`、`Order` 和 `OrderDetail`。`Customer` 和 `Order` 有零对多或一对多关系，`Order` 和 `Order_Detail` 有一对多关系。`Customer` 和 `Order` 有零对多或一对多关系，是因为在数据库架构中，`Orders` 表中的 `CustomerID` 定义为 `Nullable`。



图 31-6

用两个迭代访问顾客及其订单。首先访问 `Customer` 对象，把 `CompanyName` 属性的值写入控制台。接着使用 `Customer` 类的 `Orders` 属性访问所有订单。相关订单通过懒惰加载方式来访问属性，因为在 `ObjectContext` 中，把 `ContextOptions.LazyLoadingEnabled` 属性设置为 `true`。



```
using (var data = new NorthwindEntities())
{
    foreach (Customer customer in data.Customers)
    {
        Console.WriteLine("{0}", customer.CompanyName);
    }
}
```

```

foreach (Order order in customer.Orders)
{
    Console.WriteLine("\t{0} {1:d}", order.OrderID, order.OrderDate);
}
}
}

```

代码段 NorthwindDemo/Program.cs

在后台, 使用 `RelationshipManager` 类访问对应关系。把实体对象的类型强制转换为 `IEntityWithRelationships` 接口, 就可以访问 `RelationshipManager` 实例, 如 `Customers` 类中通过设计器生成的 `Orders` 属性所示。这个接口由 `EntityObject` 类显式实现。`RelationshipManager` 属性返回一个 `RelationshipManager`, 它与一端的实体对象关联起来。另一端调用 `GetRelatedCollection()` 方法来定义。第一个参数 `NorthwindModel.FK_Orders_Customers` 是关系的名称, 第二个参数 `Orders` 定义目标角色的名称。



可从
wrox.com
下载源代码

```

public EntityCollection<Order>Orders
{
    get
    {
        return ((IEntityWithRelationships)this).RelationshipManager.
            GetRelatedCollection<Order>("NorthwindModel.FK_Orders_Customers",
                                        "Orders");
    }
    set
    {
        if ((value != null))
        {
            ((IEntityWithRelationships)this).RelationshipManager.
                InitializeRelatedCollection<Order>(
                    "NorthwindModel.FK_Orders_Customers", "Orders", value);
        }
    }
}

```

代码段 NorthwindDemo/NorthwindModel.Designer.cs

31.6.3 懒惰加载、延迟加载和预先加载

在默认情况下, 在 `ContextOptions` 的 `LazyLoadingEnabled` 属性设置为 `true` 时, 根据请求懒惰加载(lazy loaded)关系。关系的加载还有其他选项。关系也可以预先加载(eager loaded)或延迟加载(delayed loaded)。

预先加载是指, 在加载父对象的同时加载关系。在添加对 `Include()` 方法的调用后立即加载订单。它使用 `ObjectSet<TEntity>` 传递关系名, 如 `Customers` 属性所示:

```

foreach (Customer customer in data.Customers. Include("Orders"))
{
    Console.WriteLine("{0}", customer.CompanyName);

    foreach (Order order in customer.Orders)
    {
        Console.WriteLine("\t{0} {1:d}", order.OrderID, order.OrderDate);
    }
}

```

预先加载的优点是如果需要所有相关的对象，则对数据库的请求会比较少。当然，如果并不需要所有相关的对象，懒惰加载或延迟加载会比较适合。

延迟加载需要对 EntityCollection<T>类的 Load()方法的显式调用。使用这个方法，可以把选项 LazyLoadingEnabled 设置为 false。在下面的代码段中，如果没有使用 IsLoaded 属性加载订单，就使用 Load()方法加载它们：

```
if (!customer.Orders.IsLoaded)
    customer.Orders.Load();
```

Load()方法的一个重载版本接受 MergeOption 枚举。该枚举的值如表 31-4 所示。

表 31-4

MergeOption 值	说 明
AppendOnly	这是默认值。追加新实体，对象上下文中的已有实体不修改
NoTracking	不修改 ObjectStateManager，它跟踪实体对象的变化
OverwriteChanges	实体对象的当前值用存储器中的值替代
PreserveChanges	实体对象在对象上下文中的初始值用存储器中的值替代

31.7 对象查询

查询对象是 ADO.NET Entity Framework 提供的其中一个服务。查询可以使用 LINQ to Entities、Entity SQL 和创建 Entity SQL 的 Query Builder 方法来进行。LINQ to Entities 在 31.9 节介绍，这里先介绍其他两个选项。

下面几节使用 Formula1 数据库，其中包含从设计器中创建的实体，如图 31-7 所示。

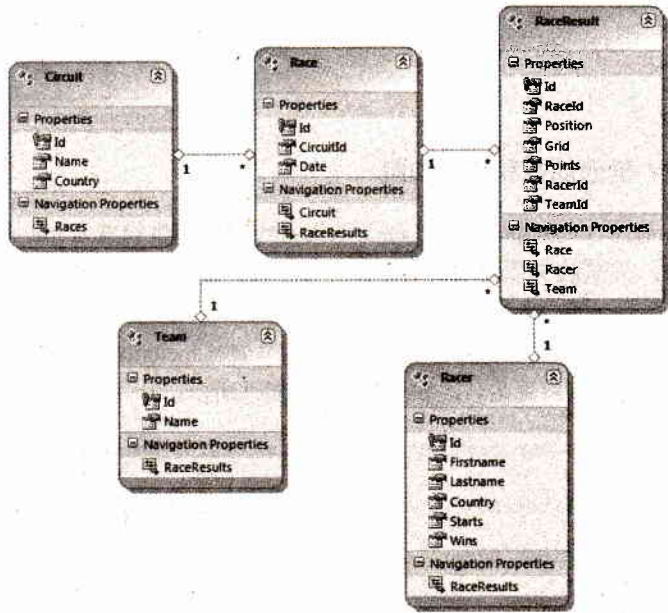


图 31-7

查询可以用 `ObjectQuery<T>` 类或其派生类 `ObjectSet<T>` 定义。下面先用一个简单的查询访问所有 `Racer` 实体。所生成的 `FormulalEntities` 类的 `Racers` 属性返回一个 `ObjectSet<Racer>`：



可从
WROX.COM
下载源代码

```
using (FormulalEntities data = new FormulalEntities(connection))
{
    ObjectSet<Racer>racers = data.Racers;
    Console.WriteLine(racers.CommandText);
    Console.WriteLine(racers.ToTraceString());
}
```

代码段 FormulalDemo/Program.cs

从 `CommandText` 属性中返回的 Entity SQL 语句如下：

```
[FormulalEntities].[Racers]
```

生成的这条 `SELECT` 语句从 `ToTraceString()` 方法显示的数据库中检索记录：

```
SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[Firstname] AS [Firstname],
  [Extent1].[Lastname] AS [Lastname],
  [Extent1].[Country] AS [Country],
  [Extent1].[Starts] AS [Starts],
  [Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
```

除了从对象上下文中访问 `Racers` 属性之外，还可以用 `CreateQuery()` 方法创建一个查询：

```
ObjectQuery<Racer>racers = data.CreateQuery<Racer>("[FormulalEntities].[Racers]");
```

这类似于使用 `Racers` 属性。

下面根据条件筛选选手。这可以使用 `ObjectQuery<T>` 类的 `Where()` 方法来完成。`Where()` 方法是其中一个 Query Builder 方法，它可以创建 Entity SQL 语句。这个方法需要把一个谓词作为字符串，其可选参数的类型是 `ObjectParameter`。这里的谓词指定只返回来自巴西的选手。`it` 指定结果项，`Country` 是 `Country` 列。`ObjectParameter()` 构造函数的第一个参数引用谓词的 `@Country` 参数，但它不带 `@` 符号。

```
string country = "Brazil";
ObjectQuery<Racer>racers = data.Racers.Where("it.Country = @Country",
    new ObjectParameter("Country", country));
```

`it` 的作用可以通过访问查询的 `CommandText` 属性来及时了解。在 Entity SQL 中，`SELECT VALUE` `it` 声明 `it` 来访问列。

```
SELECT VALUE it
FROM (
  [FormulalEntities].[Racers]
) AS it
WHERE
  it.Country = @Country
```

ToTraceString()方法显示生成的 SQL 语句如下:

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Firstname] AS [Firstname],
[Extent1].[Lastname] AS [Lastname],
[Extent1].[Country] AS [Country],
[Extent1].[Starts] AS [Starts],
[Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
WHERE [Extent1].[Country] = @Country
```

当然,也可以指定完整的 Entity SQL:

```
string country = "Brazil";
ObjectQuery<Racer>racers = data.CreateQuery<Racer>(
    "SELECT VALUE it FROM ([Formula1Entities].[Racers]) AS " +
    "it WHERE it.Country = @Country",
    new ObjectParameter("Country", country));
```

ObjectQuery<T>类提供了几个 Query Builder 方法,如表 31-5 所述。其中的许多方法非常类似于第 11 章介绍的 LINQ 扩展方法。一个重要的区别是, ObjectQuery<T>的参数类型不是委托或 Expression<T>,而通常是 string 类型。

表 31-5

ObjectQuery<T>类的 Query Builder 方法	说 明
Where()	这个方法可以根据条件筛选结果
Distinct()	这个方法创建包含唯一结果的查询
Except()	这个方法返回的结果不满足 except 筛选器指定的条件
GroupBy()	这个方法新建一个新查询,根据指定的条件组合实体
Include()	通过关系,可提前发现相关项是延迟加载的,所以需要调用 EntityCollection<T>类的 Load()方法,把相关的实体放在对象上下文中。除了使用 Load()方法,还可以用 Include()方法指定一个查询,预先提取相关实体
OfType()	这个方法指定只返回特定类型的实体,使用 TPH 关系对此很有帮助
OrderBy()	这个方法定义实体的排列顺序
Select() Selectvalue()	这些方法返回结果的一个投射。Select()方法以 DbDataRecord 的形式返回结果项,SelectValue()方法根据泛型参数 TResultType,把值返回为标量类型或复杂类型
Skip() Top()	这些方法可用于分页。Skip()可以跳过许多项,Top()方法提取指定数量的项
Intersect() Union() UnionAll()	这些方法用于合并两个查询。Intersect()方法返回的查询只包含两个查询都有的结果。Union()方法合并查询,返回没有重复的完整结果,UnionAll()方法也包含重复的结果

下面用一个例子说明如何使用这些 Query Builder 方法。其中,赛手用 Where()方法筛选,只返

回来自美国的赛手；OrderBy()方法指定先根据获奖者人数进行降序排列，再根据参加比赛的人数进行降序排列。最后，使用 Top()方法仅在结果中显示前 3 个赛手：



可从
wrox.com
下载源代码

```
using (var data = new Formula1Entities())
{
    string country = "USA";
    ObjectQuery<Racer> racers = data.Racers.Where("it.Country = @Country",
        new ObjectParameter("Country", country))
        .OrderBy("it.Wins DESC, it.Starts DESC")
        .Top("3");
    foreach (var racer in racers)
    {
        Console.WriteLine("{0} {1}, wins: {2}, starts: {3}",
            racer.Firstname, racer.Lastname, racer.Wins, racer.Starts);
    }
}
```

代码段 Formula1Demo/Program.cs

这个查询的结果如下：

```
Mario Andretti, wins: 12, starts: 128
Dan Gurney, wins: 4, starts: 87
Phil Hill, wins: 3, starts: 48
```

31.8 更新

读取、搜索和筛选数据库中的数据仅是数据密集型应用程序通常需要执行的一部分操作。把改变的数据写回存储器是另一个要执行的操作。

下面几节介绍如下主题：

- 对象跟踪
- 改变信息
- 附加和分离实体
- 存储实体的变化

31.8.1 对象跟踪

为了修改和保存从存储器中读取的数据，必须在加载实体后跟踪它们。这也要求对象上下文注意实体是否已从存储器中加载了。如果多个查询同时访问同一个记录，对象上下文就需要返回已经加载的实体。

对象上下文用 `ObjectStateManager` 来跟踪加载到上下文中的实体。

下面的示例说明了，如果两个不同的查询从数据库中返回相同的记录，状态管理器就会注意到这一点，因此它不新建实体，而是返回相同的实体。与对象上下文关联的 `ObjectStateManager` 实例可以用 `ObjectStateManager` 属性访问。`ObjectStateManager` 类定义 `ObjectStateManagerChanged` 事件，每次从对象上下文中添加或删除对象时，就调用这个事件。这里，把 `ObjectStateManager_ObjectStateManagerChanged` 方法赋予该事件，以获得改变的信息。

两个不同的查询用于返回一个实体对象。第一个查询获得来自奥地利、姓氏为 Lauda 的第一个

赛手。第二个查询请求来自奥地利的赛手，按照赢得比赛的次数排列赛手，并获取第一个结果。事实上，这是同一个赛手。为了验证返回了同一个实体对象，使用 `Object.ReferenceEquals()` 方法验证两个对象引用是否确实引用同一个实例。



可从
wrox.com
下载源代码

```
Private static void TrackingDemo()
{
    using (var data = new Formula1Entities())
    {
        data.ObjectStateManager.ObjectStateManagerChanged +=
            ObjectStateManager_ObjectStateManagerChanged;
        Racer nikil = data.Racers.Where("it.Country='Austria' && it.Lastname='Lauda'").
            First();
        Racer niki2 = data.Racers.Where("it.Country='Austria'").
            OrderBy("it.Wins DESC").First();
        if (Object.ReferenceEquals(nikil, niki2))
        {
            Console.WriteLine("the same object");
        }
    }
}

static void ObjectStateManager_ObjectStateManagerChanged(object sender,
    CollectionChangeEventArgs e)
{
    Console.WriteLine("Object State change — action: {0}", e.Action);
    Racer r = e.Element as Racer;
    if (r != null)
        Console.WriteLine("Racer {0}", r.Lastname);
}
```

代码段 Formula1Demo/Program.cs

运行这个应用程序，会看到 `ObjectStateManager` 的 `ObjectStateManagerChanged` 事件只触发了一次，`nikil` 和 `niki2` 引用的确相同：

```
Object State change — action: Add
Racer Lauda
The same object
```

31.8.2 改变信息

对象上下文也会注意到实体的改变。下面的示例添加并修改对象上下文中的一个赛手，并获得修改的信息。首先，使用 `ObjectSet<T>` 类的 `AddObject()` 方法添加一个新赛手，这个方法用 `EntityState.Added` 信息添加一个新实体。接着查询 `Lastname` 为 `Alonso` 的赛手。在这个实体类中，递增 `Starts` 属性，从而用 `EntityState.Modified` 信息标记实体。在后台，通知 `ObjectStateManager`：基于 `InotifyPropertyChanged` 接口实现方式的对象有状态改变。这个接口在实体基类 `StructuralObject` 中实现。把 `ObjectStateManager` 附加到 `PropertyChanged` 事件中，这个事件会因每个属性改变而触发。

为了获得所有添加或修改的实体对象，可以调用 `ObjectStateManager` 的 `GetObjectStateEntries()` 方法，并传递一个 `EntityState` 枚举值。这个方法返回一个 `ObjectStateEntry` 对象集合，其中保存了实体的信息。帮助方法 `DisplayState()` 迭代这个集合，以获得详细信息。

也可以把 `EntityKey` 传递给 `GetObjectStateEntry()` 方法，获得单个实体的状态信息。`EntityKey` 属

性可以用实现了 `IEntityWithKey` 接口的实体对象来获得, 即派生自基类 `EntityObject` 的实体对象。返回的 `ObjectStateEntry` 对象提供了 `GetModifiedProperties()` 方法, 在该方法中, 可以读取已改变的所有属性值, 也可以用 `OriginalValues` 和 `CurrentValues` 索引器访问属性的原始信息和当前信息。



可从
wrox.com
下载源代码

```
private static void ChangeInformation()
{
    using (var data = new Formula1Entities())
    {
        var jaime = new Racer
        {
            Firstname = "Jaime",
            Lastname = "Alguersuari",
            Country = "Spain",
            Starts = 0
        };
        data.Racers.AddObject(jaime);
        Racer fernando = data.Racers.Where("it.Lastname='Alonso'").First();
        fernando.Starts++;
        DisplayState(EntityState.Added.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(EntityState.Added));
        DisplayState(EntityState.Modified.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(EntityState.Modified));
        ObjectStateEntry stateOfFernando =
            data.ObjectStateManager.GetObjectStateEntry(fernando.EntityKey);
        Console.WriteLine("state of Fernando: {0}",
            stateOfFernando.State.ToString());
        foreach (string modifiedProp in stateOfFernando.GetModifiedProperties())
        {
            Console.WriteLine("modified: {0}", modifiedProp);
            Console.WriteLine("original: {0}",
                stateOfFernando.OriginalValues[modifiedProp]);
            Console.WriteLine("current: {0}",
                stateOfFernando.CurrentValues[modifiedProp]);
        }
    }
}

static void DisplayState(string state, IEnumerable<ObjectStateEntry> entries)
{
    foreach (var entry in entries)
    {
        var r = entry.Entity as Racer;
        if (r != null)
        {
            Console.WriteLine("{0}: {1}", state, r.Lastname);
        }
    }
}
```

代码段 Formula1Demo/Program.cs

运行这个应用程序, 会显示已添加和修改的赛车手, 已改变的属性及其原始值和当前值如下所示:

```
Added: Alguersuari
Modified: Alonso
state of Fernando: Modified
```

```
modified: Starts
original: 138
current: 139
```

31.8.3 附加和分离实体

把实体数据返回给调用者，对于从对象上下文中分离对象很重要。例如，如果实体对象从 Web 服务中返回，这就是必须的。这里，如果在客户端上改变实体对象，对象上下文并不知道对应的改变。

在示例代码中，ObjectContext 类的 Detach()方法分离实体 fernando，因此对象上下文不知道对这个实体进行了什么修改。如果把改变的实体对象从客户端应用程序传递给服务，就可以再次附加它。把它附加到对象上下文中还不够，因为这并没有给出信息，说明这个对象已经修改了。而原始对象必须在对象上下文中可用。原始对象可以使用 GetObjectByKey()或 TryGetObjectByKey()方法和键从存储器中访问。如果实体对象已经在对象上下文中，就使用已有的实体；否则就从数据库中提取新实体。调用 ApplyPropertyChanges()方法，把修改过的实体对象传递给对象上下文，如果实体对象有变化，就在已有的实体中用对象上下文中的同一个键进行修改，再把 EntityState 设置为 EntityState.Modified。ApplyPropertyChanges()方法需要对象存在于对象上下文中，否则就用 EntityState.Added 添加新实体对象。



可从
wrox.com
下载源代码

```
using (var data = new Formula1Entities())
{
    data.ObjectStateManager.ObjectStateManagerChanged +=
        ObjectStateManager_ObjectStateManagerChanged;
    ObjectQuery<Racer>racers = data.Racers.Where("it.Lastname='Alonso'");
    Racer fernando = racers.First();
    EntityKey key = fernando.EntityKey;
    data.Racers.Detach(fernando);

    // Racer is now detached and can be changed independent of the object context
    fernando.Starts++;
    Racer originalObject = data.GetObjectByKey(key) as Racer;
    data.Racers.ApplyCurrentValues(fernando);
}
```

代码段 Formula1Demo/Program.cs

31.8.4 存储实体的变化

在 ObjectStateManager 的帮助下，根据所有的变化信息，可以使用 ObjectContext 类的 SaveChanges()方法，把添加、删除和修改的实体对象写到存储器中。要验证对象上下文中的变化，可以把一个处理程序方法赋予 ObjectContext 类的 SavingChanges 事件。因为这个事件在数据写入存储器之前触发，所以可以添加一些验证逻辑，以确定是否应进行修改。SaveChanges()方法返回写入的实体对象个数。

如果数据库中用实体类表示的记录在读取之后发生了变化，该怎么办？答案取决于用模型设置的 ConcurrencyMode 属性。通过实体对象的每个属性，都可以把 ConcurrencyMode 配置为 Fixed 或 None。Fixed 值表示属性在写入时验证，以确定该值同时是否没有变化。默认值 None 表示忽略任何改变。如果把某些属性配置为 Fixed 模式，且在读写实体对象时数据有变化，就抛出一个 OptimisticConcurrencyException 异常。

调用 Refresh()方法把数据库中的实际信息读入对象上下文中，就可以处理这个异常。这个方法

接收 `RefreshMode` 枚举值配置的两端刷新模式: `ClientWins` 和 `StoreWins`。`StoreWins` 表示从数据库中提取实际信息, 并把它设置为实体对象的当前值。`ClientWins` 表示把数据库信息设置为实体对象的原始值, 因此数据库值在下一个 `SaveChanges` 事件中被覆盖。`Refresh()` 方法的第二个参数或者是实体对象的集合或者是单个实体对象。可以为每个实体确定刷新行为:



可从
wrox.com
下载源代码

```
private static void ChangeInformation()
{
    //...
    int changes = 0;
    try
    {
        changes += data.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        data.Refresh(RefreshMode.ClientWins, ex.StateEntries);
        changes += data.SaveChanges();
    }
    Console.WriteLine("{0} entities changed", changes);
    //...
```

代码段 Formula1Demo/Program.cs

31.9 LINQ to Entities

本书的几章介绍了 LINQ to Query 对象、数据库和 XML。当然, LINQ 也能用于查询实体。

在 LINQ to Entities 中, LINQ 查询的数据源是 `ObjectQuery<T>` 类。因为 `ObjectQuery<T>` 类实现了 `IQueryable` 接口, 所以用于查询的扩展方法用 `System.Linq` 名称空间中的 `Queryable` 类定义。用这个类定义的扩展方法有一个参数 `Expression<T>`, 这就是编译器把表达式树写入程序集的原因。第 11 章介绍了表达式树, 表达式树从 `ObjectQuery<T>` 类中解析到 SQL 查询中。

可以使用如下简单的 LINQ 查询返回赢得超过 40 场比赛的赛车手:



可从
wrox.com
下载源代码

```
using (var data = new Formula1Entities())
{
    var racers = from r in data.Racers
                  where r.Wins > 40
                  orderby r.Wins descending
                  select r;
    foreach (Racer r in racers)
    {
        Console.WriteLine("{0} {1}", r.Firstname, r.Lastname);
    }
}
```

代码段 Formula1Demo/Program.cs

访问 Formula1 数据库的结果如下:

```
Michael Schumacher
Alain Prost
Ayrton Senna
```

还可以定义一个 LINQ 查询来访问关系, 如下所示。变量 *r* 表示赛手, 变量 *rr* 表示所有比赛结果。筛选器用 **where** 子句定义, 只检索榜上有名的瑞士赛手。要完成这个排行榜, 应组合结果, 并计算排行榜的个数。根据排行榜的完成情况进行排序:

```
using (var data = new Formula1Entities())
{
    var query = from r in data.Racers
                from rr in r.RaceResults
                where rr.Position <= 3 && rr.Position >= 1 &&
                    r.Country == "Switzerland"
                group r by r.Id into g
                let podium = g.Count()
                orderby podium descending
                select new
                {
                    Racer = g.FirstOrDefault(),
                    Podiums = podium
                };
    foreach (var r in query)
    {
        Console.WriteLine("{0} {1} {2}", r.Racer.Firstname, r.Racer.Lastname,
                            r.Podiums);
    }
}
```

运行这个应用程序, 会返回瑞士的 3 个赛手姓名:

```
Clay Regazzoni 28
Jo Siffert 6
Rudi Fischer 2
```

31.10 小结

本章介绍了 ADO.NET Entity Framework 的特性, ADO.NET Entity Framework 基于 CSDL、MSL 和 SSDL 定义的映射, 这些 XML 信息描述实体、映射和数据库架构。使用这种映射技术, 可以创建不同的关系类型, 把实体类映射到数据库表上。

本章还介绍了对象上下文如何保存所检索和更新的实体信息, 如何把改变的内容写入存储器中。

LINQ to Entities 仅是 ADO.NET Entity Framework 的一部分, 它允许使用新的查询语法访问实体。

第 32 章

数据服务

本章内容:

- WCF 数据服务概述
- 包含 CLR 对象的 WCF 数据服务宿主
- 访问 WCF 数据服务的 HTTP 客户端
- WCF 数据服务的 URL 查询
- WCF 数据服务和 ADO.NET Entity Framework
- 使用 WCF 数据服务.NET 客户端提供程序
- 跟踪、更新和批处理

上一章介绍了 ADO.NET Entity Framework, 通过它可方便地创建映射到数据库结构的对象模型。Entity Framework 没有提供获取不同层上的对象的方式, 而 WCF 数据服务提供了这个功能。WCF 数据服务提供了一个 WCF 服务, 以便于访问实体数据模型提供的的数据, 或者仅通过实现了 IQueryable<T>接口的简单 CLR 对象来访问数据。

32.1 概述

ADO.NET Entity Framework 提供了映射, 并创建表示数据库的实体类。通过 Entity Framework 中的数据上下文, 数据上下文可以了解数据的变化, 以确定它应更新什么数据。但 Entity Framework 无助于创建多层上的解决方案。

通过 WCF 数据服务, 可以在服务器端使用 Entity Framework(或简单的 CLR 对象模型), 并把 HTTP 查询从客户端发送给服务, 以检索和更新数据。图 32-1 显示了 Windows 客户端或 Web 页面使用 HTML 和 JavaScript 给服务器发送 HTTP 请求的典型场景。返回的信息可以是 AtomPub 或 JSON 格式。AtomPub 是基于 XML 的 Atom Publisher 格式, JSON(JavaScript Object Notation, JavaScript 对象标记)最好从 JavaScript 客户端中访问。

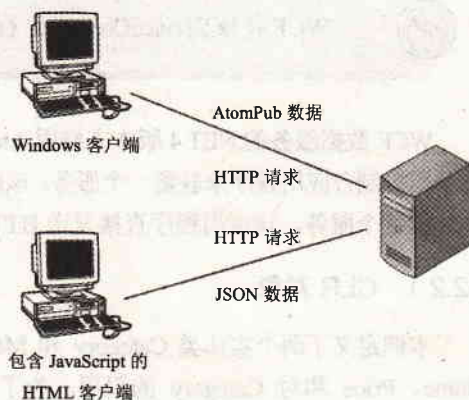


图 32-1

WCF 数据服务给通信部分使用 WCF(Windows Communication Foundation, Windows 通信基础), 还使用了 WebHttpBinding。



Atom 协议的更多信息可参见第 47 章。第 43 章提供了 WCF 的详细信息。

利用 WCF 数据服务, 不仅可以获得服务器上的功能, 以及通过 AtomPub 或 JSON 使用 HTTP Web 请求的功能, 而且 WCF 数据服务还有一个客户端部分。对于客户端, 有一个数据服务上下文, 可以创建以 AtomPub 或 JSON 格式转换的查询。HTTP 协议是无状态的, 而客户端的数据服务上下文是有状态的。通过这个上下文, 客户端可以记录哪些实体改变了、添加了或删除了, 并把包含所有变更信息的请求发送给服务。

下面详细讨论 WCF 数据服务, 首先创建一个简单的服务, 从客户端使用 HTTP Web 请求访问它。

32.2 包含 CLR 对象的自定义宿主

WCF 数据服务的核心是 `DataService<T>` 类, 它是 WCF 服务的实现方式。`DataService<T>` 类实现如下定义的 `IRequestHandler` 接口。把 `WebInvoke` 特性指定为接受任意 URI 参数和任意 HTTP 方法。`ProcessRequestForMessage()` 方法因为其参数和返回类型而非常灵活。它接受任意流, 返回一个 `Message`。这是数据具有灵活性所必须的。

```
[ServiceContract]
public interface IRequestHandler
{
    [OperationContract]
    [WebInvoke(UriTemplate="*", Method="**")]
    Message ProcessRequestForMessage(Stream messageBody);
}
```



WCF 特性 `ServiceContract`、`OperationContract` 和 `WebInvoke` 参见第 43 章。

WCF 数据服务的 .NET 4 版本支持用 AtomPub 或 JSON 格式收发请求。下面举一个简单的例子, 它使用控制台应用程序来驻留一个服务, 该服务提供了一个 CLR 对象列表。接着从客户端应用程序中使用这个服务, 该应用程序直接发送 HTTP 请求, 以检索数据。

32.2.1 CLR 对象

本例定义了两个实体类 `Category` 和 `Menu`。这些类都是简单的数据存储器。`Menu` 实体类包含 `Name`、`Price` 和对 `Category` 的引用。为了使数据服务可唯一地标识不同的实例, 必须添加特性 `DataServiceKey` 特性, 以引用唯一标识符。这个特性在 `System.Data.Services.Common` 名称空间中定义。除了把一个属性定义为标识, 它还可以指定一组属性, 以进行唯一标识。



可从
wrox.com
下载源代码

```
[DataServiceKey("Id")]
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Category() { }
    public Category(int id, string name)
    {
        this.Id = id;
        this.Name = name;
    }
}
```

代码段 DataServicesHost/Category.cs



可从
wrox.com
下载源代码

```
[DataServiceKey("Id")]
public class Menu
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public Category Category { get; set; }

    public Menu() { }
    public Menu(int id, string name, decimal price, Category category)
    {
        this.Id = id;
        this.Name = name;
        this.Price = price;
        this.Category = category;
    }
}
```

代码段 DataServicesHost/Menu.cs

MenuCard 类管理 Menu 和 Category 项的集合，它包含 Menu 和 Category 项的一个列表，这些项可以从公共 Menus 和 Categories 属性中迭代。因为这个类实现单一模式，所以每个集合只存在一个列表。



可从
wrox.com
下载源代码

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

namespace Wrox.ProCSharp.DataServices
{
    public class MenuCard
    {
        private static object sync = new object();
        private static MenuCard menuCard;
        public static MenuCard Instance
        {
            get
            {
                lock (sync)
            }
        }
    }
}
```

```

        {
            if (menuCard == null)
                menuCard = new MenuCard();
        }
        return menuCard;
    }

    private List<Category> categories;
    private List<Menu> menus;

    private MenuCard()
    {
        categories = new List<Category>
        {
            new Category(1, "Main"),
            new Category(2, "Appetizer")
        };

        menus = new List<Menu>() {
            new Menu(1, "Roasted Chicken", 22, categories[0]),
            new Menu(2, "Rack of Lamb", 32, categories[0]),
            new Menu(3, "Pork Tenderloin", 23, categories[0]),
            new Menu(4, "Fried Calamari", 9, categories[1])
        };
    }

    public IEnumerable<Menu> Menus
    {
        get
        {
            return menus;
        }
    }

    public IEnumerable<Category> Categories
    {
        get
        {
            return categories;
        }
    }
}

```

代码段 DataServicesHost/MenuCard.cs

32.2.2 数据模型

下面是真正有趣的类 `MenuCardDataModel`。这个类通过指定返回 `IQueryable<T>` 的属性，来定义从数据服务中提供哪些实体。`IQueryable<T>` 由 `DataService<T>` 类用于传递查询对象列表的表达式。



可从
wrox.com
下载源代码

```

public class MenuCardDataModel
{
    public IQueryable<Menu> Menus
    {
        get
    }
}

```

```

        {
            return MenuCard.Instance.Menus.AsQueryable();
        }
    }

    public IQueryable<Category> Categories
    {
        get
        {
            return MenuCard.Instance.Categories.AsQueryable();
        }
    }
}

```

代码段 DataServicesHost/MenuCardDataModel.cs

32.2.3 数据服务

数据服务 `MenuDataService` 的实现代码派生自基类 `DataService<T>`。`DataService<T>` 类的泛型参数是 `MenuCardDataModel` 类，`Menus` 和 `Categories` 属性返回 `IQueryable<T>`。

在 `InitializeService()` 方法中，需要使用 `DataServiceConfiguration` 类配置实体和服务操作访问规则。可以给实体和操作访问规则传递 “*”，以允许访问每个实体和操作。利用枚举 `EntitySetRights` 和 `ServiceOperationsRights`，可以指定是否启用读写访问功能。`DataServiceBehavior` 的 `MaxProtocolVersion` 属性定义应支持哪个版本的 AtomPub 协议。.NET 4 支持版本 2，版本 2 还支持其他一些功能，如获取列表中的项数。



可从
wrox.com
下载源代码

```

using System.Data.Services;
using System.Data.Services.Common;
using System.Linq;
using System.ServiceModel.Web;

namespace Wrox.ProCSharp.DataServices
{
    public class MenuDataService : DataService<MenuCardDataModel>
    {
        public static void InitializeService(DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
            config.SetEntitySetAccessRule("Categories", EntitySetRights.All);
            config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);

            config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
        }
    }
}

```

代码段 DataServicesHost/MenuDataService.cs

32.2.4 驻留服务

最后，需要一个进程来驻留应用程序。本章后面将把一个 Web 应用程序用作宿主。还可以在 WCF 支持的任意应用程序类型中驻留服务，如简单的控制台应用程序或 Windows 服务。这个示例使用控制台应用程序，该应用程序很容易改为使用其他宿主类型。

在控制台应用程序的 `Main()` 方法中，实例化一个 `DataServiceHost`。`DataServiceHost` 派生自基类

ServiceHost, 以提供 WCF 功能。还可以使用 DataServiceHostFactory 创建 DataServiceHost。调用 Open() 方法, DataServiceHost 就实例化 MenuDataService 类的一个实例, 以提供服务的功能。该服务的地址用 http://localhost:9000/Samples 定义。



可从
wrox.com
下载源代码

```
using System;
using System.Data.Services;

namespace Wrox.ProCSharp.DataServices
{
    class Program
    {
        static void Main()
        {
            DataServiceHost host = new DataServiceHost(typeof(MenuDataService),
                new Uri[] { new Uri("http://localhost:9000/Samples ") });

            host.Open();

            Console.WriteLine("service running");
            Console.WriteLine("Press return to exit");
            Console.ReadLine();

            host.Close();
        }
    }
}
```

代码段 DataServicesHost/Program.cs

现在可以启动可执行程序, 在 Internet Explorer 中用链接 http://localhost:9000/Samples/Menus 和 http://localhost:9000/Samples/Category 请求服务。为了使用 Internet Explorer 查看服务返回的数据, 需要取消 Turn on feed reading view 选项。



为了不提升的管理员权限启动一个侦听器, 应使用 netsh http add urlacl url=http://+:9000/Samples user=username listen=yes 配置端口和用户的 ACL。当然, 要修改这些管理设置, 需要提升的管理员权限。

32.2.5 其他服务操作

除了从数据模型中提供属性之外, 还可以给数据服务添加其他服务操作。在示例代码中, 包含 GetMenusByName() 方法, 它从客户端获取一个请求参数, 并通过 IQueryable<Menu> 集合返回所有以请求的字符串开头的菜单。这类操作需要在 InitializeService() 方法中添加到服务操作访问规则中, 以防没有使用 “*” 提供所有服务操作。

```
public class MenuDataService : DataService<MenuCardDataModel>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
        config.SetEntitySetAccessRule("Categories", EntitySetRights.All);
    }
}
```

```

config.SetServiceOperationAccessRule("GetMenusByName",
                                     ServiceOperationRights.All);

config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
}

[WebGet(UriTemplate="GetMenusByName?name={name}",
        BodyStyle=WebMessageBodyStyle.Bare)]
public IQueryable<Menu> GetMenusByName(string name)
{
    return (from m in CurrentDataSource.Menus
            where m.Name.StartsWith(name)
            select m).AsQueryable();
}
}

```

代码段 DataServiceHost/MenuDataService.cs

32.3 HTTP 客户端应用程序

客户端应用程序可以是一个简单的应用程序，仅给服务发送 HTTP 请求，并接收 AtomPub 或 JSON 响应。第一个客户端应用程序示例是一个 WPF 应用程序，它使用 System.Net 名称空间中的 `HttpWebRequest` 类。

图 32-2 显示了 Visual Studio Designer 中的 UI。textUrl 文本框用于输入 HTTP 请求，其默认值是 `http://localhost:9000/Samples/Menus`。只读文本框 textReturn 接收来自服务的回应。还有一个复选框 checkJSON，其中表示响应可以用 JSON 格式请求。Call Data Service 按钮的 Click 事件关联到 OnRequest()方法上。

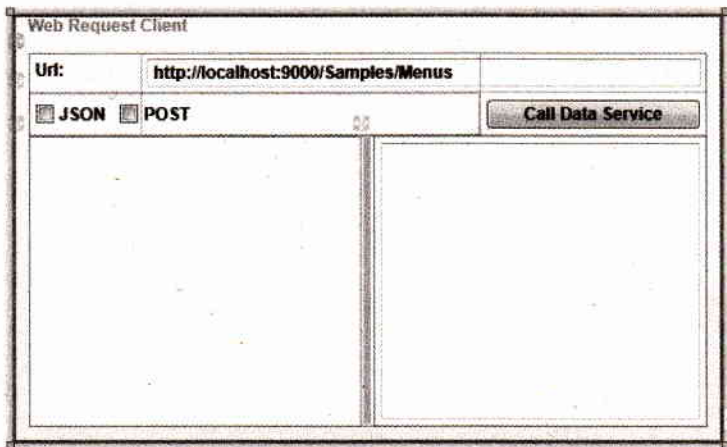


图 32-2

在 `OnRequest()` 处理程序的实现代码中，用 `WebRequest.Create()` 工厂方法创建了一个 `HttpWebRequest` 对象。所发送的 HTTP 请求用 `textUrl.Text` 属性定义。如果勾选 JSON 复选框，`HttpWebRequest` 对象的 `Accept` 属性就设置为 `application/json`。这样，数据服务就返回一个 JSON 响应，而不是默认的 AtomPub 格式的响应。到服务器的请求用异步方法 `BeginGetRequest()` 发送。接收到来自服务的响应时，就调用把第一个参数定义为 Lambda 表达式的方法。响应应与 `HttpWebRequest`

对象关联的流中读取，并把它转换为一个字符串，再传递给文本框 `textResult`。



可从
wrox.com
下载源代码

```
private void OnRequest(object sender, RoutedEventArgs e)
{
    HttpRequest request = WebRequest.Create(textUrl.Text)
        as HttpRequest;
    if (checkJSON.IsChecked == true)
    {
        request.Accept = "application/json";
    }

    request.BeginGetResponse((ar) =>
    {
        try
        {
            using (MemoryStream ms = new MemoryStream())
            {
                const int bufferSize = 1024;
                byte[] buffer = new byte[bufferSize];
                HttpResponse response =
                    request.EndGetResponse(ar) as HttpResponse;

                Stream responseStream = response.GetResponseStream();
                int count;
                while ((count = responseStream.Read(buffer, 0,
                    bufferSize)) > 0)
                {
                    ms.Write(buffer, 0, count);
                }
                responseStream.Close();
                byte[] dataRead = ms.ToArray();
                string data = UnicodeEncoding.ASCII.GetString(
                    dataRead, 0, dataRead.Length);
                Dispatcher.BeginInvoke(new Action<string> (s =>
                {
                    textResult.Text = data;
                }), data);
            }
        }
        catch (WebException ex)
        {
            Dispatcher.Invoke(new Action<string> (s =>
            {
                textResult.Text = s;
            }), ex.Message);
        }
    }, null);
}
```

代码段 WebRequestClient/WebRequestClient.xaml.cs

现在就可以给服务发送几个 HTTP 请求，并查看返回的数据。运行的应用程序如图 32-3 所示。

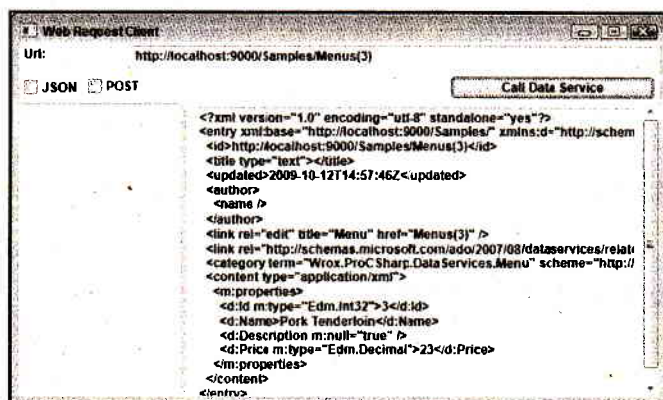


图 32-3

使用请求 `http://localhost:9000/Samples/Menus(3)`，会得到唯一标识符为 3 的菜单，接收到的 AtomPub 信息如下：

```
<?xml version="1.0" encoding="utf - 8" standalone="yes"?>
<entry xml:base="http://localhost:9000/Samples/"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
      xmlns="http://www.w3.org/2005/Atom">
  <id> http://localhost:9000/Samples/Menus(3) </id>
  <title type="text"></title>
  <updated>2009-07-30T13:20:07Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="Menu" href="Menus(3)" />
  <link rel=
    "http://schemas.microsoft.com/ado/2007/08/dataservices/related/Category"
    type="application/atom+xml;type=entry" title="Category"
    href="Menus(3)/Category" />
  <category term="Wrox.ProCSharp.DataServices.Menu"
    scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:Id m:type="Edm.Int32">3</d:Id>
      <d:Name > Pork Tenderloin</d:Name>
      <d:Description m:null="true" />
      <d:Price m:type="Edm.Decimal">23</d:Price>
    </m:properties>
  </content>
</entry>
```

如果选择 JSON 格式，就会返回相同的信息，但采用 JSON 表示方式返回，以便于从 JavaScript 中读取：

```
{ "d" :
  { "_metadata":
    { "uri": "http://localhost:9000/Samples/Menus(3)",
      "type": "Wrox.ProCSharp.DataServices.Menu"
    },
```

```

      "Id": 3, "Name": "Pork Tenderloin", "Price": "23",
      "Category":
        { "__deferred":
          { "uri": "http://localhost:9000/Samples/Menus(3)/Category"
          }
        }
    }
  }
}

```

下面看看构建查询所需要的所有寻址选项。

使用 URL 查询

由于数据服务接口非常灵活，因此可以从服务中请求所有对象，或者获取特定的对象和特定属性的值。



为了便于阅读，在下面的查询中，省略了服务的地址 `http://localhost:9000/Samples`。对于所有查询都必须加上这个前缀。

前面介绍过，可以获取实体集中所有实体的列表。查询

```
Menus
```

返回所有菜单实体，而

```
Catagories
```

返回所有类别实体。根据 AtomPub 协议，返回的根元素是 `<feed>`，它包含每个元素的 `<entry>` 元素。这个查询没有跨越引用，例如，用 `Menus` 获取所有菜单，并不返回类别的内容，只返回对它的引用。要获得菜单中的类别信息，可以使用 `$expand` 查询字符串：

```
Menus?$expand= Catagory
```

在括号中传递主键值仅返回一个实体。这里访问的是标识符为 3 的菜单。它需要前面使用的 `DataServiceKey` 特性的定义：

```
Menus(3)
```

使用导航属性(`/`)，可以访问实体的属性：

```
Menus(3)/Price
```

这种语法可用于关系，来访问相关实体中的属性：

```
Menus(3)/Category/Name
```

为了仅获取值，不获取实体的其他 XML 内容，可以使用 `$value` 查询函数：

```
Menus(3)/Category/Name/$value
```

回到完整的列表，使用 `$count` 可以获得列表中的实体个数。`$count` 只能用于 AtomPub 协议的版本 2。

```
Menus/$count
```

使用查询字符串选项\$top, 可以仅获得列表的第一个实体:

```
Menus?$top=2
```

使用\$skip 可以跳过一些实体。\$skip 和\$top 可以联合使用, 实现分页功能:

```
Menus?$skip=2
```

使用\$filter 查询字符串选项, 以及逻辑运算符 eq(等于)、ne(不等于)、gt(大于)、ge(大于等于 0), 就可以筛选实体:

```
Menus?$filter=Category/Name eq 'Appetizer'
```

结果可以使用\$orderby 查询字符串选项排序:

```
Menus?$filter=Category/Name eq 'Appetizer' & orderby=Price desc
```

若仅需要获取一个投射, 即可用属性的一个子集, 则可以使用\$select 指定要访问的属性:

```
Menus?$select=Name, Price
```

32.4 使用 WCF 数据服务和 ADO.NET Entity Framework

学习了数据服务的基本概念, 并把 AtomPub 或 JSON 数据传递给简单的 HTTP 请求后, 下面介绍一个较复杂的示例, 它对数据模型使用 ADO.NET Entity Framework, 用一个 Web 应用程序作为宿主, 客户在网络上执行 LINQ 查询, 以使用 System.Data.Services.Client 名称空间中的类。

32.4.1 ASP.NET 宿主和 EDM

首先需要新建一个项目, 这次创建一个 Web 应用程序项目 RestaurantDataServiceWeb, 用于驻留服务。新的数据模型用 ADO.NET 实体数据模型(EDM)模板创建, 并使用 Restaurant 数据库中的 Menus 和 Categories 表, 如图 32-4 所示, 以创建实体类 Menu 和 Category。

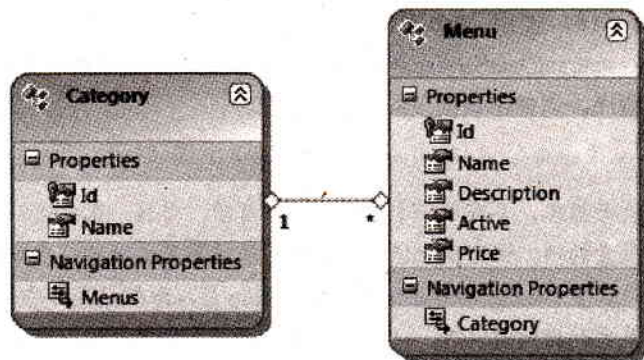


图 32-4



第31章介绍了如何创建和使用 ADO.NET 实体数据模型。

现在使用 Data Service 模板创建 RestaurantDataService.svc。svc 文件包含 ASP.NET 的指令 ServiceHost，并使用 DataServiceHostFactory 根据请求实例化数据服务。



可从
wrox.com
下载源代码

```
<%@ ServiceHost Language="C#" Factory="System.Data.Services.DataServiceHostFactory,
System.Data.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
Service="Wrox.ProCSharp.DataServices.RestaurantDataService" %>
```

代码段 RestaurantDataServiceWeb/RestaurantDataService.svc

通过代码隐藏，需要把 DataService<T>类的模板参数改为引用前面创建的实体数据服务上下文类，并修改实体集访问规则和服务操作访问规则，以允许访问它们：



可从
wrox.com
下载源代码

```
using System.Data.Services;
using System.Data.Services.Common;

namespace Wrox.ProCSharp.DataServices
{
    public class RestaurantDataService : DataService<RestaurantEntities>
    {
        // This method is called only once to initialize service-wide policies.
        public static void InitializeService(DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
            config.SetEntitySetAccessRule("Categories", EntitySetRights.All);

            config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
        }
    }
}
```

代码段 RestaurantDataServiceWeb/RestaurantDataService.svc.cs

现在可以像以前那样，使用 Web 浏览器调用这个数据服务的查询了；例如，可以使用 <http://localhost:13617/RestaurantDataService.svc/Menus>，从数据库中检索所有菜单的 AtomPub。接着创建一个客户端应用程序，以使用数据服务的客户端部分。



对于大型数据库，不应用一个查询返回所有项。当然，客户端可以把查询限制为仅请求一定数量的项。但可以相信客户端吗？通过配置选项可以在服务器上设置这个限制。例如，设置 config.MaxResultsPerCollection 可以限制从集合中返回的最大项数。还可以配置最大的批处理次数、一次最多插入的对象数以及树中对象的最大深度。另外，为了允许执行任意查询，还可以定义服务操作，如 32.2.5 节所述。

32.4.2 使用 System.Data.Service.Client 的 .NET 应用程序

本章前面创建了一个 .NET 客户端应用程序，它使用 `HttpWebRequest` 类仅发送 HTTP 请求。数据服务的客户端部分使用 `System.Data.Service.Client` 名称空间，为客户端提供了构建 HTTP 请求的功能。这个名称空间中最重要的两个类是 `DataServiceContext` 和 `DataServiceQuery<TElement>`。`DataServiceContext` 类表示在客户端上管理的状态。这个状态跟踪从服务器上加载的对象和在客户端上进行的所有修改。`DataServiceQuery<TElement>` 类表示对数据服务的一个 HTTP 查询。

为了调用数据服务，需要创建一个 WPF 应用程序。图 32-5 显示了该 WPF 应用程序的设计视图。第 1 行包含一个 `ComboBox` 控件，它用于显示所有类别。第 2 行包含一个 `StackPanel`，它有 4 个 `Button` 控件。第 3 行包含一个 `DataGrid` 控件，以显示菜单；第 4 行包含一个 `TextBlock` 元素，以显示一些状态信息。

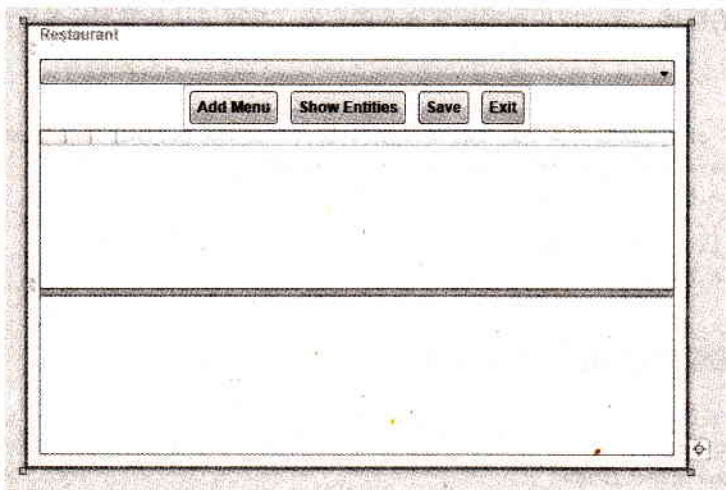


图 32-5

要创建客户端代理和在客户端使用的实体类，需要元数据信息。数据服务使用 `$metadata` 查询字符串 `http://localhost:13617/RestaurantDataService.svc/$metadata` 来提供元数据。

有了这些信息，就可以把一个服务引用添加到客户端应用程序项目中，以创建代理类和实体类。对于 `Restaurant` 数据服务，创建一个派生自基类 `DataServiceContext` 的 `RestaurantEntities` 类，它可用作代理。还要创建保存数据的实体类 `Menu` 和 `Category`。实体类实现 `INotifyPropertyChanged` 接口，以便于获得在 UI 上进行的变更修改通知。

1. 数据服务上下文

现在可以使用数据服务上下文 `RestaurantEntities`，给数据服务发送一个查询。在 WPF 窗口(即 `MainWindow` 类)的代码隐藏中定义该服务上下文的一个变量。为了避免给数据服务生成的 `Menu` 类与 WPF 中的 `Menu` 控件冲突，用别名 `R`(表示 `Restaurant`)定义一个名称空间别名，来引用从服务引用中生成的类。



可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.Data.Services.Client;
using System.Linq;
```

```

using System.Text;
using System.Windows;
using System.Windows.Controls;
using R = Wrox.ProCSharp.DataServices.RestaurantService.RestaurantModel;

namespace Wrox.ProCSharp.DataServices
{
    public partial class MainWindow : Window
    {
        private R.RestaurantEntities data;
        private DataServiceCollection<R.Menu> trackedMenus;
    }
}

```

代码段 ClientApp/MainWindow.xaml.cs

RestaurantEntities 的实例在 **MainWindow** 类的构造函数中创建。数据服务上下文的构造函数需要一个到服务根的链接。这在应用程序配置文件中定义，并可以从强类型化的设置中访问。

```

public MainWindow()
{
    Uri serviceRoot = new Uri(Properties.Settings.Default.RestaurantServiceURL);
    data = new R.RestaurantEntities(serviceRoot);
    data.SendingRequest += data_SendingRequest;

    InitializeComponent();
    this.DataContext = this;
}

```

用于引用数据服务的应用程序配置文件的内容如下：



可从
wrox.com
下载源代码

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
      type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089">
      <section name="Wrox.ProCSharp.DataServices.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
          Culture=neutral, PublicKeyToken=b77a5c561934e089"
          requirePermission="false" />
    </sectionGroup>
  </configSections>
  <applicationSettings>
    <Wrox.ProCSharp.DataServices.Properties.Settings>
      <setting name="RestaurantServiceURL" serializeAs="String">
        <value>http://localhost:13617/RestaurantDataService.svc</value>
      </setting>
    </Wrox.ProCSharp.DataServices.Properties.Settings>
  </applicationSettings>
</configuration>

```

代码段 ClientApp/app.config

每次把请求发送给服务时，数据服务上下文就通过 **SendingRequest** 事件调用处理程序。把 **data_sendingRequest()** 方法关联到 **SendingRequest** 事件上，它接收 **SendingRequestEventArgs** 参数中的请求信息。通过 **SendingRequestEventArgs** 参数，可以访问请求信息和标题信息。把从 **Method** 和

RequestUri 属性中检索的请求方法和 URI 写入到 UI 的 textStatus 控件中。



可从
wrox.com
下载源代码

```
void data_SendingRequest(object sender, SendingRequestEventArgs e)
{
    var sb = new StringBuilder();
    sb.AppendFormat("Method: {0}\n", e.Request.Method);
    sb.AppendFormat("Uri: {0}\n", e.Request.RequestUri.ToString());
    this.textStatus.Text = sb.ToString();
}
```

代码段 ClientApp/MainWindow.xaml.cs

数据服务上下文 RestaurantEntities 允许从服务中检索实体；跟踪已检索到的实体；还可以添加、输出和修改数据上下文中的实体；保存改变的状态，以发送更新请求。

2. LINQ 查询

数据服务上下文实现了一个 LINQ 提供程序，用于把 LINQ 请求转换为 HTTP 请求。MainWindow 类的 Categories 属性使用数据上下文定义了一个 LINQ 查询，以返回所有类别：



可从
wrox.com
下载源代码

```
public IEnumerable<R.Category> Categories
{
    get
    {
        return from c in data.Categories
               orderby c.Name
               select c;
    }
}
```

代码段 ClientApp/MainWindow.xaml.cs

XAML 代码中的 ComboBox 定义了到这个属性的绑定，以显示所有类别：



可从
wrox.com
下载源代码

```
<ComboBox x:Name="comboCategories" Grid.Row="0"
           ItemsSource="{Binding Path=Categories}" SelectedIndex="0"
           SelectionChanged="OnCategorySelection">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Path=Name}" />
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

代码段 ClientApp/MainWindow.xaml

利用 SendingRequest 事件，很容易通过下面的 URI，把访问所有类别的 LINQ 查询转换为 HTTP 的 GET 请求：

```
Method: GET
Uri: http://localhost:13617/RestaurantDataService.svc/Categories()?orderby=Name
```

RestaurantEntities 类的 Categories 属性返回一个 DataServiceQuery<Category> 类。因为 DataServiceQuery<T> 类实现 IQueryable 接口，所以编译器会从已分析并转换为 HTTP 的 GET 请求的 LINQ 查询中创建表达式树。



LINQ 查询参见第 11 章。

还可以创建一个 LINQ 查询，仅获取 Soups 类别中的菜单：

```
var q = from m in data.Menus
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

这会转换为 URI `Menus()?$filter=Category/Name eq 'Soups'&$orderby=Name`。

使用数据服务查询选项，可以扩展查询。例如，通过所选菜单，要包含关系并获取所有类别，应给 `AddQueryOption()` 方法添加 `$expand` 和参数值 `Category`：

```
var q = from m in data.Menus.AddQueryOption("$expand", "Category")
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

这会把查询修改为 `Menus()?$filter=Category/Name eq 'Soups'&$orderby=Name&$expand=Category`。

`DataServiceQuery<T>` 类还给这个特殊的查询选项提供 `Expand()` 方法。

```
var q = from m in data.Menus.Expand("Category")
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

3. 可观察的集合

为了使用户界面了解集合的变化，数据服务包含集合类 `DataServiceCollection<T>`。这个集合类基于 `ObservableCollection<T>`，`ObservableCollection<T>` 实现 `INotifyCollectionChanged` 接口。WPF 控件利用这个接口的事件来注册，以获得集合变更的通知，从而使 UI 能立即更新。

为了创建 `DataServiceCollection<T>` 实例，`DataServiceCollection` 类定义静态方法 `Create()` 和 `CreateTracked()`。`Create()` 方法会创建一个独立于数据服务上下文的列表，`CreateTracked()` 方法会跟踪从数据服务上下文返回的对象，以便给该服务发送变更的内容，从而保存到该对象中。

`Menus` 属性调用 `DataServiceCollection.CreateTracked<T>()` 方法，来填充 `DataServiceCollection<R.Menu>` 列表 `trackedMenus`，并返回这个列表。把通过 LINQ 查询检索到的实体与数据服务上下文 `data` 关联起来。



可从
wrox.com
下载源代码

```
public IEnumerable <R.Menu> Menus
{
    get
    {
        if (trackedMenus == null)
            trackedMenus = DataServiceCollection.CreateTracked<R.Menu>(
                data,
                from m in data.Menus
                where m.CategoryId == (comboBoxCategories.SelectedItem as R.Category).Id
                & & m.Active
                select m);
    }
}
```

```
return trackedMenus;
```

代码段 ClientApp/MainWindow.xaml.cs

XAML 代码中的 DataGrid 通过 Binding 标记扩展映射到 Menus 属性上:



可从
wrox.com
下载源代码

```
<DataGrid Grid.Row="2" ItemsSource="{Binding Path=Menus}"
           AutoGenerateColumns="False">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=Name}" />
        <DataGridTextColumn Binding="{Binding Path=Description}" />
        <DataGridTextColumn Binding="{Binding Path=Price}" />
        <DataGridTextColumn Binding="{Binding Path=CategoryId}" />
    </DataGrid.Columns>
</DataGrid>
```

代码段 ClientApp/MainWindow.xaml

使用 ComboBox 的 SelectionChanged 事件选择一个新类别, 再在处理程序方法 OnCategorySelection() 中用新选择的类别检索菜单:



可从
wrox.com
下载源代码

```
private void OnCategorySelection(object sender, SelectionChangedEventArgs e)
{
    var selectedCategory = comboCategories.SelectedItem as R.Category;
    if (selectedCategory != null && trackedMenus != null)
    {
        trackedMenus.Clear();
        trackedMenus.Load(from m in data.Menus
                          where m.CategoryId == selectedCategory.Id
                          select m);
    }
}
```

代码段 ClientApp/MainWindow.xaml.cs



关于可观察的集合和 ObservableCollection<T>类的更多信息参见第 10 章。

4. 对象跟踪

数据上下文跟踪已检索到的所有对象。迭代 Entities 属性的返回值, 可以获得这些对象的信息。Entities 属性返回 EntityDescriptor 对象的只读集合。EntityDescriptor 对象包含可通过 Entity 属性访问的实体本身和状态信息。EntityStates 类型的状态是一个枚举, 其可能的值有 Added、Deleted、Detached、Modified 和 Unchanged。这些信息用于跟踪变化, 并给对应服务发送一个变更请求。

要获得与数据上下文相关联的当前实体的信息, 把处理程序方法 OnShowEntities() 关联到 Show Entities 按钮的 Click 事件上。这里使用 State、Identity 和 Entity 属性把状态信息写入到 UI 上。



可从
wrox.com
下载源代码

```
private void OnShowEntities(object sender, RoutedEventArgs e)
{
    var sb = new StringBuilder();
    foreach (var entity in data.Entities)
    {
        sb.AppendFormat("state = {0}, Uri = {1}, Element = {2}\n",
            entity.State, entity.Identity, entity.Entity);
    }
    this.textStatus.Text = sb.ToString();
}
```

代码段 ClientApp/MainWindow.xaml.cs

图 32-6 显示了正在运行的应用程序和所跟踪对象的信息。

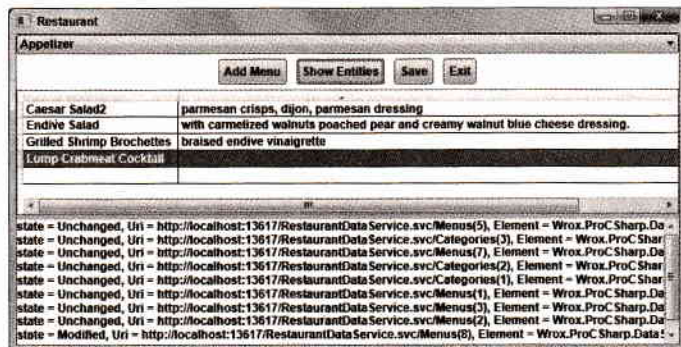


图 32-6

5. 添加、修改和删除实体

要给数据服务上下文添加实体，以便以后把新对象发送给数据服务，可以使用 `AddObject()` 方法给数据服务上下文添加实体，或者使用该方法的强类型化变体，如 `AddToMenus()` 和 `AddToCategories()`。只需填充必选属性；否则，就不能成功保存状态。添加新对象会把状态设置为 `Added`。

数据服务上下文的 `DeleteObject()` 方法把对象的状态设置为 `Deleted`。

如果修改了对象的属性，该状态就从 `Unchanged` 改为 `Modified`。

现在可以调用 `SaveChanges()` 方法，它发送 HTTP MERGE 请求，以更新实体；或者发送 HTTP DELETE 请求，以删除实体；或者发送 HTTP POST 请求，以添加新实体。



可从
wrox.com
下载源代码

```
private void OnSave(object sender, RoutedEventArgs e)
{
    try
    {
        DataServiceResponse response = data.SaveChanges();
    }
    catch (DataServiceRequestException ex)
    {
        textStatus.Text = ex.ToString();
    }
}
```

代码段 ClientApp/MainWindow.xaml.cs

6. 操作的批处理

除了给集合中的每个实体发送 DELETE 和 MODIFY 请求之外, 还可以批处理多个变更请求, 以作为一个网络请求。默认情况下, 在应用 SaveChanges() 方法时, 每个变更都使用一个请求来发送。给 SaveChanges() 方法添加 SaveChangesOptions.Batch 参数, 就可以使用 \$batch 查询选项, 把所有变更请求合并为一个网络调用。



可从
WROX.COM
下载源代码

```
private void OnSave(object sender, RoutedEventArgs e)
{
    try
    {
        DataServiceResponse response = data.SaveChanges(SaveChangesOptions.Batch);
    }
    catch (DataServiceRequestException ex)
    {
        textStatus.Text = ex.ToString();
    }
}
```

代码段 ClientApp/MainWindow.xaml.cs

如传输的数据所示, 把多个 HTTP 标题合并到一个 HTTP POST 请求中, 之后在服务器端分解开。使用下面的 HTTP POST 请求, 会合并 DELETE 和 MERGE 请求。DELETE 请求会删除 id 为 4 的菜单, MERGE 请求包含 AtomPub 信息, 以更新 id 为 2 的菜单。

```
POST /RestaurantDataService.svc/$batch HTTP/1.1
User-Agent: Microsoft WCF Data Services
DataServiceVersion: 1.0;NetFx
MaxDataServiceVersion: 2.0;NetFx
Accept: application/atom+xml,application/xml
Accept-Charset: UTF-8
Content-Type: multipart/mixed; boundary=batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd
Host: 127.0.0.1:13617
Content-Length: 1742
Expect: 100-continue

-- batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd
Content-Type: multipart/mixed;
boundary=changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7

-- changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7
Content-Type: application/http
Content-Transfer-Encoding: binary

DELETE http://127.0.0.1:13617/RestaurantDataService.svc/Menus(4) HTTP/1.1
Host: 127.0.0.1:13617
Content-ID: 18

-- changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7
Content-Type: application/http
Content-Transfer-Encoding: binary
MERGE http://127.0.0.1:13617/RestaurantDataService.svc/Menus(2) HTTP/1.1
Host: 127.0.0.1:13617
Content-ID: 19
```

```

Content-Type: application/atom+xml;type=entry
Content-Length: 965

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
  <category scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme"
    term="RestaurantModel.Menu" />
  <title />
  <author>
    <name />
  </author>
  <updated> 2009-08-01T19:17:43.4741882Z</updated>
  <id> http://127.0.0.1:13617/RestaurantDataService.svc/Menus(2)</id>
  <content type="application/xml">
    <m:properties>
      <d:Active m:type="Edm.Boolean">true</d:Active>
      <d:Description> Lean and tender 8 oz. sirloin seasoned perfectly
        with our own special seasonings and topped with seasoned
        butter.
      </d:Description>
      <d:Id m:type="Edm.Int32">2</d:Id>
      <d:Name> Sirloin Steak</d:Name>
      <d:Price m:type="Edm.Decimal">44.0</d:Price>
    </m:properties>
  </content>
</entry>
-- changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7--
-- batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd--

```

32.5 小结

本章介绍了 WCF 数据服务的功能。WCF 数据服务把 ADO.NET Entity Framework 中的数据模型引入多层上，它的底层技术是 WCF，使用无连接、无状态的通信发送 AtomPub 或 JSON 查询。

本章介绍了这种技术的服务器端部分和客户端部分，在客户端的一个数据服务上下文中跟踪变更信息。因为 WCF 数据服务的客户端部分实现一个 LINQ 提供程序，所以可以创建简单的 LINQ 请求，LINQ 请求可以转换为 HTTP GET/POST/PUT/DELETE 请求。

第 33 章

处 理 XML

本章内容:

- XML 标准
- XmlReader 和 XmlWriter
- XmlDocument
- XPathDocument
- XmlNavigator
- LINQ to XML
- 使用 System.Xml.Linq 名称空间中的对象
- 使用 LINQ 查询 XML 文档
- 使用 LINQ to SQL 和 LINQ to XML

XML 在 .NET Framework 中有重要作用。 .NET Framework 不仅允许在应用程序中使用 XML, .NET Framework 本身也在配置文件和源代码文档中使用 XML。另外, SOAP、Web 服务和 ADO.NET 也使用 XML。

为了涵盖 XML 的扩展用法, .NET Framework 包含了 System.Xml 名称空间。这个名称空间包含许多用于处理 XML 的类。本章将讨论这些类。

本章介绍如何使用 XmlDocument 类, 这是 DOM(Document Object Model, 文档对象模型)的实现方式, 以及 .NET 为 SAX 提供的一种替代品(XmlReader 和 XmlWriter 类)。本章还要讨论 XPath 和 XSLT 的类实现方式, 接着介绍 XML 和 ADO.NET 如何一起工作, 如何把其中一种格式转换为另一种格式。还将介绍如何把对象序列化为 XML, 使用 System.Xml.Serialization 名称空间中的类从 XML 文档中创建一个对象(或者反序列化)。更重要的是, 要介绍如何把 XML 合并到 C# 应用程序中。

注意 XML 名称空间可以用许多不同的方式得到类似的结果。因为我们不可能把所有这些不同方式都放在一章中介绍, 所以这里仅探讨其中一种方式, 并尽量提及完成同一任务的其他方式。

因为篇幅有限, 不能从头开始介绍 XML, 所以本章假定读者已经熟悉 XML 技术。因此, 您应知道元素、属性和节点, 还应知道格式良好的文档的含义, 您也应熟悉 SAX 和 DOM。



如果要更多地了解 XML, 可以参阅 Wrox 出版社的 *Professional XML* (Wiley 出版社, 2007 年, ISBN: 978-0-471-77777-9)。

除了一般的 XML 用法之外, .NET Framework 还可以通过 LINQ to XML 使用 XML。Wrox 网站和随书附赠光盘上的第 56 章介绍了使用 LINQ 查询 SQL Server 数据库。本章介绍使用 LINQ 查询 XML 数据源。

首先简要介绍目前使用的 XML 标准。

33.1 .NET 支持的 XML 标准

W3C(World Wide Web Consortium, 万维网联合会)开发了一组标准, 它给 XML 提供了强大的功能和潜力。如果没有这些标准, XML 就不会对开发领域有它应有的影响。W3C 网站(www.w3.org)包含 XML 的所有有用信息。

.NET Framework 支持下述 W3C 标准:

- XML 1.0(www.w3.org/TR/1998/REC-xml-19980210), 包括 DTD 支持
- XML 名称空间(www.w3.org/TR/REC-xml-names), 包括流级和 DOM
- XML 架构(www.w3.org/2001/XMLSchema)
- XPath 表达式(www.w3.org/TR/xpath)
- XSLT 转换(www.w3.org/TR/xslt)
- DOM Level 1 核心(www.w3.org/TR/REC-DOM-Level-1)
- DOM Level 2 核心(www.w3.org/TR/DOM-Level-2-Core)
- Soap 1.1(www.w3.org/TR/SOAP)

随着 Framework 走向成熟和 W3C 更新所推荐的标准, 标准支持的级别也会改变, 因此, 必须确保标准和 Microsoft 提供的支持级别都是最新的。

33.2 System.Xml 名称空间

对 XML 处理的支持由 .NET 中 System.Xml 名称空间中的类提供。本节介绍(没有特定的顺序)System.Xml 名称空间中一些比较重要的类。表 33-1 列出了主要的 XML 读取器类和写入器类。

表 33-1

类 名	说 明
XmlReader	抽象的读取器类, 提供快速、没有缓存的 XML 数据。XmlReader 是只向前的, 类似于 SAX 分析器
XmlWriter	抽象的写入器类, 以流或文件的格式提供快速、没有缓存的 XML 数据
XmlTextReader	扩展 XmlReader, 提供访问 XML 数据的快速只向前流
XmlTextWriter	扩展 XmlWriter, 快速生成只向前的 XML 流

表 33-2 列出了用于处理 XML 的其他一些重要的类。

表 33-2

类 名	说 明
XmlNode	抽象类, 表示 XML 文档中的一个节点。它是 XML 名称空间中几个类的基类
XmlDocument	扩展 XmlNode, 这是 W3C DOM 的实现方式, 它给出 XML 文档在内存中的树型表示, 可以浏览和编辑它们
XmlDataDocument	扩展 XmlDocument, 即从 XML 数据中加载的文档, 或从 ADO.NET DataSet 的关系数据中加载的文档, 允许把 XML 和关系数据混合在同一个视图中
XmlResolver	抽象类, 分析基于 XML 的外部资源, 如 DTD 和架构引用, 也可以用于处理<xsl:include>和 <xsl:import>元素
XmlNodeList	可以迭代的一个 XmlNode 列表
XmlUrlResolver	扩展 XmlResolver, 用 URI(Uniform Resource Identifier, 统一资源标识符)解析外部资源

System.Xml 名称空间中的许多类都提供了管理 XML 文档和流的方式, 而其他类(例如 XmlDataDocument 类)则提供了 XML 数据存储器和存储在 DataSet 中的关系数据之间的桥梁。



XML 名称空间可用于属于 .NET 系列的任何语言, 这表示, 本章中所有的示例也可以用 VB.NET、托管 C++ 等来编写。

33.3 使用 System.Xml 类

下面几个示例将使用 books.xml 作为数据源。books.xml 和本章的其他代码示例可以从 Wrox 网站(www.wrox.com)和随书附赠光盘中找到, books.xml 也包含在 .NET SDK 的几个示例中。books.xml 文件是假想书店的书目清单, 它包含类型、作者姓名、价格和 ISBN 号等信息。

下面是 books.xml 文件:



可从
wrox.com
下载源代码

```
<?xml version='1.0'?>
<!--This file represents a fragment of a book store inventory database-->
<bookstore>
  <book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
    <title> The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title> The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
</bookstore>
```

```
</book>
<book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
  <title>The Gorgias</title>
  <author>
    <name>Plato</name>
  </author>
  <price>9.99</price>
</book>
</bookstore>
```

代码段 books.xml

33.4 读写流格式的 XML

如果您曾经使用过 SAX, 就应很熟悉 `XmlReader` 类和 `XmlWriter` 类。基于 `XmlReader` 的类提供了一种非常迅速、只向前的只读光标来处理 XML 数据。因为它是一个流模型, 所以内存要求不是很高。但是, 它没有提供基于 DOM 模型的导航功能和读写功能。基于 `XmlWriter` 的类可以生成遵循 W3C 的 XML 1.0 Namespace Recommendations 的 XML 文档。

`XmlReader` 和 `XmlWriter` 都是抽象类。下面的类派生自 `XmlReader`:

- `XmlNodeReader`
- `XmlTextReader`
- `XmlValidatingReader`

下面的类派生自 `XmlWriter` 的类:

- `XmlTextWriter`
- `XmlQueryWriter`

`XmlTextReader` 类和 `XmlTextWriter` 类或者与 `System.IO` 名称空间中一个基于流的对象或者与 `TextReader/TextWriter` 对象一起使用。`XmlNodeReader` 类把 `XmlNode` 作为其源, 而不是一个流。`XmlValidatingReader` 类添加了 DTD 和架构验证, 因此提供了数据的有效性验证。本章后面会详细介绍这些类。

33.4.1 使用 `XmlReader` 类

`XmlReader` 类非常类似于 MSXML SDK 中的 SAX。它们最大的一个区别是 SAX 是一种推模型(push model), 它把数据推入应用程序中, 开发人员必须准备接受它, 而 `XmlReader` 是一种拉模型(pull model), 它把应用程序请求的数据拉入该应用程序。这样就有一种更简单、更直观的编程模型。另一个优点是拉模型可以选择把什么数据发送给应用程序。如果不需要所有数据, 就不需要处理它。而在推模型中, 所有 XML 数据都必须由应用程序处理, 无论是否需要这些数据。

下面介绍一个非常简单的示例, 以读取 XML 数据, 后面将详细介绍 `XmlReader` 类, 这些代码在 `XmlReaderSample` 文件夹中。下面的代码将读取 `book.xml` 文档中的数据。在读取每个节点时, 都要检查 `NodeType` 属性。如果节点是一个文本节点, 就把其值追加到文本框中:



可从
wrox.com
下载源代码

```
using System.Xml;

private void button3_Click(object sender, EventArgs e)
{
```

```

richTextBox1.Clear();
XmlReader rdr = XmlReader.Create("books.xml");
while (rdr.Read())
{
    if (rdr.NodeType == XmlNodeType.Text)
        richTextBox1.AppendText(rdr.Value + "\r\n");
}

```

代码段 XMLReaderSample.sln

如前所述, `XmlReader` 是一个抽象类。所以, 要直接使用 `XmlReader` 类, 必须添加静态方法 `Create()`, 该方法返回一个 `XmlReader` 对象。`Create()` 方法有 9 个重载版本。在上面的例子中, 该方法有一个字符串参数, 表示 `XmlDocument` 的文件名。还可以给该方法传送基于流的对象和基于 `TextReader` 的对象。

另一个可以使用的对象是 `XmlReaderSettings`, 它指定读取器的功能。例如, 可以使用架构来验证数据流。把 `Schemas` 属性设置为一个有效的 `XMLSchemaSet` 对象, 来缓存 XSD 架构。接着把 `XmlReaderSettings` 对象的 `XsdValidate` 属性设置为 `true`。

有几个 `Ignore` 属性可用于控制读取器处理某些节点和值的方式。这些属性包括 `IgnoreComments`、`IgnoreIdentityConstraints`、`IgnoreInlineSchema`、`IgnoreProcessingInstructions`、`IgnoreSchemaLocation` 和 `IgnoreWhitespace`, 它们可以从文档中提取某些项。

1. Read()方法

遍历文档有几种方式, 如前面的示例所示, `Read()` 方法可以进入下一个节点。然后验证该节点是否有一个值(`HasValue()`)或者快速查看该节点是否有特性(`HasAttributes()`)。也可以使用 `ReadStartElement()` 方法, 该方法验证当前节点是否是起始元素, 如果是起始元素, 就可以定位到下一个节点上。如果不是起始元素, 就引发一个 `XmlException` 异常。调用这个方法与调用 `Read()` 方法后再调用 `IsStartElement()` 方法是一样的。

`ReadElementString()` 类似于 `ReadString()`, 但它可以选择以元素名作为参数。如果下一个内容节点不是起始标记, 或者如果 `Name` 参数不匹配当前的节点 `Name`, 就会引发异常。

下面的示例说明了如何使用 `ReadElementString()` 方法。注意因为这个示例使用 `FileStream`, 所以需要利用 `using` 语句来包括 `System.IO` 名称空间:



可从
wrox.com
下载源代码

```

private void button6_(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader rdr = XmlReader.Create("books.xml");
    while (!rdr.EOF)
    {
        //if we hit an element type, try and load it in the listbox
        if (rdr.MoveToContent() == XmlNodeType.Element && rdr.Name == "title")
        {
            richTextBox1.AppendText(rdr.ReadElementString() + "\r\n");
        }
        else
        {
            //otherwise move on
            rdr.Read();
        }
    }
}

```

在 while 循环中, 使用 `MoveToContent()` 方法查找类型为 `XmlNodeType.Element`、名称为 `title` 的节点。我们使用 `XmlTextReader` 类的 `EOF` 属性作为循环条件。如果节点的类型不是 `Element`, 或者名称不是 `title`, `else` 子句就会调用 `Read()` 方法进入下一个节点。当查找到一个满足条件的节点时, 就把 `ReadElementString()` 方法的结果添加到列表框中。这样就在列表框中添加一个书名。注意, 在成功执行 `ReadElementString()` 方法后; 不需要调用 `Read()` 方法, 因为 `ReadElementString()` 方法已经使用了整个 `Element`, 并定位到下一个节点上。

如果删除了 `if` 子句中的 `&& rdr.Name=="title"`, 在抛出 `XmlException` 异常时, 就必须捕获它。如果查看一下数据文件, 就会发现 `MoveToContent()` 方法查找到的第一个元素是 `<bookstore>`; 因为它是一个元素, 所以通过了 `if` 语句中的检查。但是, 由于它不包含简单的文本类型, 因此它会导致 `ReadElementString()` 方法引发一个 `XmlException` 异常。解决这个问题的一种方式是把 `ReadElementString()` 调用放在它自己的函数中。现在, 如果在这个函数中 `ReadElementString()` 调用失败, 就可以处理错误, 并返回主调函数。

下面就调用这个新方法 `LoadTextBox()`, 把 `XmltextReader` 类作为参数。进行这些修改后, `LoadTextBox()` 方法如下所示:

```
private void LoadTextBox(XmlReader reader)
{
    try
    {
        richTextBox1.AppendText (reader.ReadElementString() + "\r\n");
    }
    // if an XmlException is raised, ignore it.
    catch(XmlException ex){}
}
```

上面示例中的下述代码

```
if (tr.MoveToContent() == XmlNodeType.Element && tr.Name == "title")
{
    richTextBox1.AppendText (tr.ReadElementString() + "\r\n");
}
else
{
    //otherwise move on
    tr.Read();
}
```

就会变成:

```
if (tr.MoveToContent() == XmlNodeType.Element)
{
    LoadTextBox(tr);
}
else
{
    //otherwise move on
```

```
tr.Read();
}
```

运行这段代码，结果应与前面示例的结果一样。因此，完成同一个任务有多种不同的方式。这体现了 `System.Xml` 名称空间中类的灵活性。

`XmlReader` 类还可以读取强类型化的数据，它有几个 `ReadElementContentAs()` 方法，如 `ReadElementContentAsDouble()`、`ReadElementContentAsBoolean()` 等。下面的示例说明了如何把对应值读取为小数，并对该值进行数学处理。在本例中，要给价格元素中的值增加 25%：



可从
wrox.com
下载源代码

```
private void button5_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader rdr = XmlReader.Create("books.xml");
    while (rdr.Read())
    {
        if (rdr.NodeType == XmlNodeType.Element)
        {
            if (rdr.Name == "price")
            {
                decimal price = rdr.ReadElementContentAsDecimal();
                richTextBox1.AppendText("Current Price = " + price + "\r\n");
                price += price * (decimal).25;
                richTextBox1.AppendText("New Price = " + price + "\r\n\r\n");
            }
            else if (rdr.Name == "title")
            {
                richTextBox1.AppendText(rdr.ReadElementContentAsString() + "\r\n");
            }
        }
    }
}
```

代码段 XMLReaderSample.sln

如果不能把该值转换为小数，就引发一个 `FormatException` 异常。与把该值读取为一个字符串，再把它转换为合适的数据类型相比，这个方法的效率较高。

2. 检索特性数据

在运行示例代码时，可能注意到在读取节点时，没有看到特性。这是因为特性不是文档的结构的一部分。针对元素节点，可以检查特性是否存在，并可选择性地检索特性值。

例如，如果有特性，`HasAttributes` 就返回 `true`；否则它就返回 `false`。`AttributeCount` 属性确定特性的个数。`GetAttribute()` 方法按照名称或索引来获取特性。如果要一次迭代一个特性，就可以使用 `MoveToFirstAttribute()` 和 `MoveToNextAttribute()` 方法。

下面的示例迭代 `book.xml` 文档中的特性：



可从
wrox.com
下载源代码

```
private void button7_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader tr = XmlReader.Create("books.xml");
    //Read in node at a time
    while (tr.Read())
    {

```

```
//check to see if it's a NodeType element
if (tr.NodeType == XmlNodeType.Element)
{
    //if it's an element, then let's look at the attributes.
    for (int i = 0; i < tr.AttributeCount; i++)
    {
        richTextBox1.AppendText(tr.GetAttribute(i) + "\r\n");
    }
}
```

代码段 XMLReaderSample.sln

这次查找元素节点。找到一个节点后，就迭代其所有的特性，使用 `GetAttribute()` 方法把特性值加载到列表框中。在本例中，这些特性是 `genre`、`publicationdate` 和 `ISBN`。

33.4.2 使用 XmlReader 类进行验证

有时不但要知道文档的格式是良好的，还要确定文档是有效的。`XmlReader` 类可以使用 `XmlReaderSettings` 类，根据 XSD 架构验证 XML。把 XSD 架构添加到 `XMLSchemaSet` 中，通过 `Schemas` 属性可以访问 `XMLSchemaSet`。`XsdValidate` 属性还必须设置为 `true`，这个属性默认为 `false`。

下面的示例说明了 `XmlReaderSettings` 类的用法。这个 XSD 架构用于验证 `book.xml` 文档：



```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="bookstore">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="book">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="title" type="xs:string" />
                            <xs:element name="author">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element minOccurs="0" name="name"
                                            type="xs:string" />
                                        <xs:element minOccurs="0" name="first-name"
                                            type="xs:string" />
                                        <xs:element minOccurs="0" name="last-name"
                                            type="xs:string" />
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="price" type="xs:decimal" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:attribute name="genre" type="xs:string" use="required" />
                <!-- < xs:attribute name="publicationdate"
                    type="xs:unsignedShort" use="required" /> -->
                <xs:attribute name="ISBN" type="xs:string" use="required" />
            </xs:complexType>
        </xs:element>
    </xs:element>
</xs:schema>
```

```

        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>

```

代码段 books.xsd

在 Visual Studio 中这个架构从 book.xml 中生成。注意 publicationdate 属性被注释掉了，这样在验证到该属性时会失败。

下面的代码使用该架构验证 books.xml 文档：



可从
wrox.com
下载源代码

```

private void button8_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReaderSettings settings = new XmlReaderSettings();
    settings.Schemas.Add(null, "books.xsd");
    settings.ValidationType = ValidationType.Schema;
    settings.ValidationEventHandler +=
        new System.Xml.Schema.ValidationEventHandler(settings_ValidationEventHandler);
    XmlReader rdr = XmlReader.Create("books.xml", settings);
    while (rdr.Read())
    {
        if (rdr.NodeType == XmlNodeType.Text)
            richTextBox1.AppendText(rdr.Value + "\r\n");
    }
}

```

代码段 XMLReaderSample.sn

创建 XmlReaderSettings 对象设置后，就把 books.xsd 架构添加到 XmlSchemaSet 对象中。XmlSchemaSet 对象的 Add() 方法有 4 个重载版本，第一个重载版本把 XmlSchema 对象作为参数，XmlSchema 对象可以用于快速创建架构，而无需在磁盘上创建架构文件。另一个重载版本把另一个 XmlSchemaSet 对象作为参数。第 3 个重载版本接受两个字符串参数。第一个字符串是目标名称空间，第二个字符串是 XSD 文档的 URL。如果目标名称空间参数为空，就使用架构的 targetNamespace。最后一个重载版本也把 targetNamespace 作为第一个参数，但它使用基于 XmlReader 的对象读取架构。XmlSchemaSet 对象在处理要验证的文档之前预处理架构。

引用该架构后，XsdValidate 属性就设置为 ValidationType 枚举的一个值，该枚举的有效值是 DTD、Schema 和 None。如果把选中的值设置为 None，就不进行验证。

因为我们使用的是 XmlReader 对象，所以，如果文档有验证问题，在读取器读取属性或元素之前，就不会发现该问题。验证失败时，会引发一个 XmlSchemaValidationException 异常。这个异常可以在 catch 块中处理，但处理异常会很难控制数据流。为了解决这个问题，可以使用 XmlReaderSettings 类中的 ValidationEvent。这样，就可以处理验证失败，且无需使用异常处理。该事件还可以由验证警告引发，验证警告不会引发异常。ValidationEvent 传递一个 ValidationEventArgs 对象，该对象包含 Severity 属性。这个属性确定事件是由错误还是警告引发。如果该事件由错误引发，则还会传递引发该事件的异常。还有一个消息属性。在本例中，消息显示在 MessageBox 中。

33.4.3 使用 XmlWriter 类

XmlWriter 类可以把 XML 写入一个流、文件、StringBuilder、TextWriter 或另一个 XmlWriter 对

象中。与 `XmlTextReader` 类一样, `XmlWriter` 类以只向前、未缓存的方式进行写入。`XmlWriter` 类的可配置性很高, 可以指定是否缩进内容、缩进量、在属性值中使用什么引号, 以及是否支持名称空间等信息。与 `XmlReader` 类一样, 这个配置使用 `XmlWriterSettings` 对象进行。

下面是一个简单的示例, 它说明了如何使用 `XmlTextWriter` 类:



可从
wrox.com
下载源代码

```
private void button9_Click(object sender, EventArgs e)
{
    XmlWriterSettings settings = new XmlWriterSettings();
    settings.Indent = true;
    settings.NewLineOnAttributes = true;
    XmlWriter writer = XmlWriter.Create("newbook.xml", settings);
    writer.WriteStartDocument();
    //Start creating elements and attributes
    writer.WriteStartElement("book");
    writer.WriteAttributeString("genre", "Mystery");
    writer.WriteAttributeString("publicationdate", "2001");
    writer.WriteAttributeString("ISBN", "123456789");
    writer.WriteElementString("title", "Case of the Missing Cookie");
    writer.WriteStartElement("author");
    writer.WriteElementString("name", "Cookie Monster");
    writer.WriteEndElement();
    writer.WriteElementString("price", "9.99");
    writer.WriteEndElement();
    writer.WriteEndDocument();
    //clean up
    writer.Flush();
    writer.Close();
}
```

代码下载 [XMLReaderSample.sln](#)

这里编写一个新的 XML 文件 `newbook.xml`, 并给一本新书添加数据。注意 `XmlWriter` 类会用新文件覆盖已有文件。本章的后面会把一个新元素或新节点插入到已有文档中, 使用 `Create()` 静态方法实例化 `XmlWriter` 对象。在本例中, 把一个表示文件名的字符串和 `XmlWriterSettings` 类的一个实例传递为参数。

`XmlWriterSettings` 类的属性控制生成 XML 的方式。`CheckedCharacters` 属性是一个布尔值, 如果 XML 中的字符不遵循 W3C XML 1.0 建议, 该属性就会引发一个异常。`Encoding` 类设置生成 XML 所使用的编码, 默认为 `Encoding.UTF8`。`Indent` 属性是一个布尔值, 它确定元素是否应缩进。把 `IndentChars` 属性设置为用于缩进的字符串, 默认为两个空格。`NewLine` 属性用于确定换行符。在上面的示例中, 把 `NewLineOnAttribute` 属性设置为 `true`, 所以把每个属性单独放在一行上, 更便于读取生成的 XML。

`WriteStartDocument()` 方法添加文档声明。现在开始写入数据。首先是 `book` 元素, 接下来添加 `genre`、`publicationdate` 和 `ISBN` 属性。然后写入 `title`、`author` 和 `price` 元素。注意 `author` 元素有一个子元素 `name`。

单击对应按钮, 生成 `booknew.xml` 文件, 如下所示:

```
<?xml version="1.0" encoding="utf-8"?>
<book
  genre="Mystery"
  publicationdate="2001"
```

```

ISBN="123456789">
<title> Case of the Missing Cookie</title>
<author>
    <name>Cookie Monster</name>
</author>
<price>9.99</price>
</book>

```

在开始和结束写入元素和属性时,要注意控制元素的嵌套。在给 `authors` 元素添加 `name` 子元素时,就可以看到这种嵌套。注意 `WriteStartElement()` 和 `WriteEndElement()` 方法调用是如何安排的,以及它们如何在输出文件中生成嵌套的元素。

除了 `WriteElementString()` 和 `WriteAttributeString()` 方法外,还有其他几个专用的写入方法。`WriteCData()` 方法可以输出一个 `CData` 部分(`<![CDATA[...]]>`), 输出它接受的文本参数。`WriteComment()` 方法以正确的 XML 格式输出注释。`WriteChars()` 方法输出字符缓冲区的内容,其工作方式类似于前面的 `ReadChars()` 方法,它们都使用相同类型的参数。`WriteChars()` 方法需要一个缓冲区(一个字符数组)、写入的起始位置(一个整数)和要写入的字符个数(一个整数)。

使用基于 `XmlReader` 和 `XmlWriter` 的类读写 XML 非常灵活,使用起来也很简单。下面介绍如何使用 `System.Xml` 名称空间中的 `XmlDocument` 类和 `XmlNode` 类实现 DOM。

33.5 在.NET 中使用 DOM

.NET 中的文档对象模型(Document Object Model, DOM)支持 W3C DOM Level 1 和 Core DOM Level 2 规范。DOM 通过 `XmlNode` 类来实现。`XmlNode` 是一个抽象类,它表示 XML 文档的一个节点。

还有一个 `XmlNodeList` 类,它是节点的一个有序列表。这是一个实时的节点列表,对节点的任何修改都会立即反映在列表中。`XmlNodeList` 类支持索引访问或迭代访问。

`XmlNode` 类和 `XmlNodeList` 类组成了 .NET Framework 中 DOM 实现的核心,表 33-3 列出了基于 `XmlNode` 的一些类。

表 33-3

类 名	说 明
<code>XmlLinkedNode</code>	返回当前节点之前或之后的节点。给 <code>XmlNode</code> 类添加 <code>NextSibling</code> 和 <code>PreviousSibling</code> 属性
<code>XmlDocument</code>	表示整个文档,实现 DOM Level 1 和 Level 2 规范
<code>XmlDocumentFragment</code>	表示文档树的一个片段
<code>XmlAttribute</code>	表示 <code>XmlElement</code> 对象的一个属性对象
<code>XmlEntity</code>	表示一个已分析或未分析的实体节点
<code>XmlNotation</code>	包含在 DTD 或架构中声明的记号

表 33-4 列出了扩展 `XmlCharacterData` 的类。

表 33-4

类 名	说 明
XmlCDataSection	表示文档中的一个 CData 部分
XmlComment	表示一个 XML 注释对象
XmlSignificantWhitespace	表示带有空白的节点。只有 PreserveWhiteSpace 标志为 true 时, 才能创建节点
XmlWhitespace	表示元素内容中的空白, 只有 PreserveWhiteSpace 标志为 true 时, 才能创建节点
XmlText	表示元素或属性的文本内容

最后, 表 33-5 列出了扩展 XmlLinkedNode 的类。

表 33-5

类 名	说 明
XmlDeclaration	表示声明节点(<?xml version='1.0'...>)
XmlDocumentType	表示与文档类型声明相关的数据
XmlElement	表示一个 XML 元素对象
XmlEntityReferenceNode	表示一个实体引用节点
XmlProcessingInstruction	包含 XML 处理指令

可以看出, .NET 使其类适合于可能遇到的任何 XML 类型。因此, 该工具集非常灵活和强大。本节不打算详细介绍每个类, 而是用几个示例来说明可以完成什么任务。

使用 XmlDocument 类

XmlDocument 类及其派生类 XmlDataDocument(详见本章后面的内容)是用于在 .NET 中表示 DOM 的类。与 XmlReader 类和 XmlWriter 类不同, XmlDocument 类具有读写功能, 并可以随机访问 DOM 树。XmlDocument 类非常类似于 MSXML 中的 DOM 实现。如果您用 MSXML 编过程序, 就会觉得使用 XmlDocument 类很合适。

下面介绍的示例创建一个 XmlDocument 对象, 加载磁盘上的一个文档, 再从标题元素中加载带有数据的文本框, 这类似于 33.4.1 节的示例, 区别是本例选择要使用的节点, 而不是像基于 XmlReader 类的示例那样浏览整个文档。

下面是创建 XmlDocument 对象的代码, 与 XmlReader 示例相比, 这个示例比较简单:



可从
wrox.com
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    //doc is declared at the module level
    //change path to match your path structure
    _doc.Load("books.xml");
    //get only the nodes that we want.
    XmlNodeList nodeList = _doc.GetElementsByTagName("title");
    //iterate through the XmlNodeList
    textBox1.Text = "";
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.OuterXml + "\r\n";
    }
}
```

代码段 frmXMLDOM.cs

注意，本节的示例添加了以下模块级的声明：

```
private XmlDocument doc=new XmlDocument();
```

如果这就是我们需要完成的全部工作，使用 `XmlReader` 类就是加载文本框的一种非常高效的方式，原因是我们只浏览一次文档，就完成了处理。这就是 `XmlReader` 类的工作方式。但如果要重新查看某个节点，则最好使用 `XmlDocument` 类。

下面的示例使用 XPath 语法从文档中检索一组节点：



可从
wrox.com
下载源代码

```
private void button2_Click(object sender, EventArgs e)
{
    //doc is declared at the module level
    //change path to match your path structure
    doc.Load("books.xml");
    //get only the nodes that we want.
    XmlNodeList nodeList = _doc.SelectNodes("/bookstore/book/title");
    textBox1.Text = "";
    //iterate through the XmlNodeList
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.OuterXml + "\r\n";
    }
}
```

代码段 frmXMLDOM.cs

`SelectNodes()` 方法返回一个 `NodeList` 或一个 `XmlNodes` 集合。这个列表只包含匹配作为 `SelectNodes()` 方法的参数传递的 XPath 语句。在这个示例中，只需查看 `title` 节点。如果调用了 `SelectSingleNode()` 方法，就会接收到一个节点对象，它包含 `XmlDocument` 中满足 XPath 条件的第一个节点。

下面简要介绍一下 `SelectSingleNode()` 方法，它是 `XmlDocument` 类的 XPath 实现方式，`SelectSingleNode()` 和 `SelectNodes()` 都是在 `XmlNode` 类中定义的方法，而 `XmlDocument` 类基于 `XmlNode` 类。`SelectSingleNode()` 方法返回一个 `XmlNode`，`SelectNodes()` 方法返回一个 `XmlNodeList`。`System.Xml.XPath` 名称空间包含许多 XPath 实现方式。后面一节会介绍它们。

插入节点

前面的示例使用 `XmlTextWriter` 类新建一个文档。其局限性是它不能把节点插入到当前文档中。而使用 `XmlDocument` 类可以做到这一点。把上一个示例中的 `button1_Click()` 事件处理程序作如下改动：



可从
wrox.com
下载源代码

```
private void button4_Click(object sender, System.EventArgs e)
{
    //change path to match your structure
    _doc.Load("books.xml");
    //create a new 'book' element
    XmlElement newBook = _doc.CreateElement("book");
    //set some attributes
```

```

newBook.SetAttribute("genre", "Mystery");
newBook.SetAttribute("publicationdate", "2001");
newBook.SetAttribute("ISBN", "123456789");
//create a new 'title' element
XmlElement newTitle = _doc.CreateElement("title");
newTitle.InnerText = "Case of the Missing Cookie";
newBook.AppendChild(newTitle);
//create new author element
XmlElement newAuthor = _doc.CreateElement("author");
newBook.AppendChild(newAuthor);
//create new name element
XmlElement newName = _doc.CreateElement("name");
newName.InnerText = "Cookie Monster";
newAuthor.AppendChild(newName);
//create new price element
XmlElement newPrice = _doc.CreateElement("price");
newPrice.InnerText = "9.95";
newBook.AppendChild(newPrice);
//add to the current document
_doc.DocumentElement.AppendChild(newBook);
//write out the doc to disk
XmlTextWriter tr = new XmlTextWriter("booksEdit.xml", null);
tr.Formatting = Formatting.Indented;
_doc.WriteContentTo(tr);
tr.Close();
//load listBox1 with all of the titles, including new one
XmlNodeList nodeList = _doc.GetElementsByTagName("title");
textBox1.Text = "";
foreach (XmlNode node in nodeList)
{
    textBox1.Text += node.OuterXml + "\r\n";
}

```

代码段 frmXMLDOM.cs

在执行这段代码后，会实现与上一个示例相同的功能，但本例在文本框中添加了一本书：*The Case of the Missing Cookie*（即将上市的经典著作）。仔细查看代码，就会发现这是一个相当简单的过程。首先，新建一个 **book** 元素：

```
XmlElement newBook = doc.CreateElement("book");
```

CreateElement()方法有 3 个重载版本，它们可以指定：

- 元素名
- 名称和名称空间 URI
- 前缀、本地名和名称空间

一旦创建该元素，就要添加属性：

```

newBook.SetAttribute("genre", "Mystery");
newBook.SetAttribute("publicationdate", "2001");
newBook.SetAttribute("ISBN", "123456789");

```

既然创建了属性，就要添加书籍的其他元素：

```
XmlElement newTitle = doc.CreateElement("title");
newTitle.InnerText = "The Case of the Missing Cookie";
newBook.AppendChild(newTitle);
```

再次新建一个基于 `XmlElement` 的对象(`newTitle`)，然后把 `InnerText` 属性设置为新经典著作的书名，把该元素追加为 `book` 元素的一个子元素。对 `book` 元素中的其他元素重复这一操作。注意把 `name` 元素添加为 `author` 元素的一个子元素。这样就可以像其他 `book` 元素一样得到合适的嵌套关系。

最后把 `newBook` 元素追加到 `doc.DocumentElement` 节点上，它与所有其他 `book` 元素同级。现在用新元素更新已有文档。

最后，把新 XML 文档写入到磁盘中。在这个示例中，新建一个 `XmlTextWriter`，把它传递给 `WriteContentTo()` 方法。`WriteContentTo()` 和 `WriteTo()` 方法都接受一个 `XmlTextWriter` 参数。`WriteContentTo()` 方法把当前节点及其所有子节点都保存到 `XmlTextWriter` 中，而 `WriteTo()` 方法只保存当前节点。因为 `doc` 是一个基于 `XmlDocument` 的对象，它表示整个文档，所以应保存它。还可以使用 `Save()` 方法，它总是保存整个文档，`Save()` 方法有 4 个重载版本，其参数分别是一个包含文件名和路径的字符串、一个基于 `Stream` 的对象、一个基于 `TextWriter` 的对象和一个基于 `XmlWriter` 的对象。

我们还在 `XmlTextWriter` 上调用了 `Close()` 方法，刷新内部缓存，并关闭文件。

在运行这个示例时，会得到如图 33-1 所示的对话框。注意列表框底部的新项。

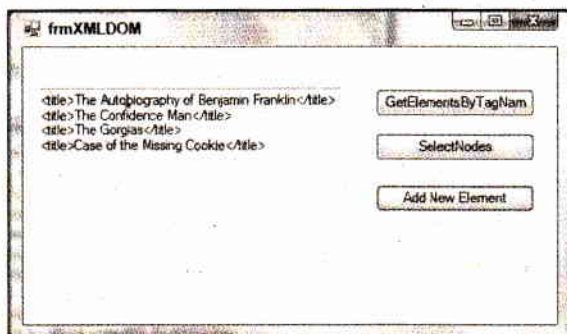


图 33-1

本章的前面说明了如何使用 `XmlTextWriter` 类创建一个文档，还可以使用 `XmlDocument` 类。使用哪个类比较好？如果要写入 XML 流的数据可用且准备写入，那么最好选择 `XmlTextWriter` 类。但是，如果需要一次构建 XML 文档的一小部分，在不同的地方插入节点，那么用 `XmlDocument` 类创建文档比较好。为此，可以把下面的代码：

```
doc.Load("books.xml");
```

改为：

```
//create the declaration section
XmlDeclaration newDec = doc.CreateXmlDeclaration("1.0",null,null);
doc.AppendChild(newDec);
//create the new root element
XmlElement newRoot = doc.CreateElement("newBookstore");
doc.AppendChild(newRoot);
```

首先新建一个 `XmlDeclaration`，其参数是版本(目前是"1.0")、编码和独立标志。如果没有使用 `null`，编码参数就应设置为一个字符串，该字符串应是 `System.Text.Encoding` 类的一部分。`null` 默认为 UTF-8。独立标志可以是 `yes`、`no` 或 `null`。如果它是 `null`，就不使用该特性，也不包含在文档中。

要创建的下一个元素是 `DocumentElement`。在本例中，它称为 `newBookstore`，这样区别就比较明显。代码的其余部分与前面的示例相同，工作原理也相同。下面是从以下代码中生成的 `booksEdit.xml`：

```
<?xml version="1.0"?>
<newBookstore>
  <book genre="Mystery" publicationdate="2001" ISBN="123456789">
    <title>The Case of the Missing Cookie</title>
    <author>
      <name>C. Monster</name>
    </author>
    <price> 9.95</price>
  </book>
</newBookstore>
```

在希望随机访问文档时，可以使用 `XmlDocument` 类。在希望有一个流类型的模型时，可以使用基于 `XmlReader` 的类。基于 `XmlNode` 的 `XmlDocument` 类的灵活性要求的内存比较多，读取文档的性能也没有使用 `XmlReader` 类好，遍历 XML 文档还有另一种方式：使用 `XPathNavigator` 类。

33.6 使用 XPathNavigator 类

`XPathNavigator` 类用于从 XML 文档中选择、迭代和偶尔编辑数据。`XPathNavigator` 类可以从 `XmlDocument` 中创建，以支持编辑功能；它也可以从 `XpathDocument` 类中创建，此时只能用于读取。因为 `XPathDocument` 类是只读的，所以它执行得很好。与 `XmlReader` 类不同，`XPathNavigator` 类不是一个流模型，所以同一个文档不需要重新读取和分析，也能使用。

`XPathNavigator` 类在 `System.Xml.XPath` 名称空间中，`XPath` 是一种查询语言，可以从 XML 文档中选择特定的节点或元素，以进行处理。

33.6.1 System.Xml.XPath 名称空间

`System.Xml.XPath` 名称空间建立在速度的基础上，由于它提供了 XML 文档的一种只读视图，因此它没有编辑功能。这个名称空间中的类可以采用光标的方式在 XML 文档上进行快速迭代和选择操作。

表 33-6 列出了 `System.Xml.XPath` 名称空间中的重要类，并对每个类的功能进行了简单的说明。

表 33-6

类 名	说 明
<code>XPathDocument</code>	提供整个 XML 文档的视图，只读
<code>XPathNavigator</code>	提供 <code>XPathDocument</code> 的导航功能
<code>XPathNodeIterator</code>	提供节点集的迭代功能
<code>XPathExpression</code>	编译好的 XPath 表达式，由 <code>SelectNodes</code> 、 <code>SelectSingleNodes</code> 、 <code>Evaluate</code> 和 <code>Matches</code> 使用
<code>XPathException</code>	XPath 异常类

1. XPathDocument 类

XpathDocument 类没有提供 XmlDocument 类的任何功能,它唯一的功能是创建 XPathNavigator。因此,这是 XPathDocument 类上唯一可用的方法(除了其他由 Object 提供的方法)。

XpathDocument 类可以用许多不同的方式创建。可以给构造函数传递 XmlReader、XML 文档的文件名或基于流的对象,其灵活性非常大。例如,可以使用 XmlValidatingReader 验证 XML,然后使用同一个对象创建 XPathDocument。

2. XPathNavigator 类

XpathNavigator 类包含移动和选择所需元素的所有方法,在该类中定义的其中一些“移动”方法如表 33-7 所示。

表 33-7

方 法 名	说 明
MoveTo()	把 XPathNavigator 作为参数,移动当前位置到 XPathNavigator 指定的地方
MoveToAttribute()	移动到指定的属性,其参数是属性名和名称空间
MoveToFirstAttribute()	移动到当前元素中的第一个属性上,如果成功,就返回 true
MoveToNextAttribute()	移动到当前元素中的下一个属性上,如果成功,就返回 true
MoveToFirst()	移动到当前节点中的第一个同级节点上,如果成功,就返回 true
MoveToLast()	移动到当前节点中的最后一个同级节点上,如果成功,就返回 true
MoveToNext()	移动到当前节点中的下一个同级节点上,如果成功,就返回 true
MoveToPrevious()	移动到当前节点中的上一个同级节点上,如果成功,就返回 true
MoveToFirstChild()	移动到当前元素中的第一个子元素上,如果成功,就返回 true
MoveToId()	移动到 ID 参数提供的元素上,文档中需要有一个架构,元素的数据类型必须是 ID 类型
MoveToParent()	移动到当前节点的父节点上,如果成功,就返回 true
MoveToRoot()	移动到文档的根节点上

要选择文档的一个子集,可以使用表 33-8 所示的其中一个 Select()方法。

表 33-8

方 法 名	说 明
Select()	使用 XPath 表达式选择一个节点集
SelectAncestors()	根据 XPath 表达式选择当前节点的所有上级节点
SelectChildren()	根据 XPath 表达式选择当前节点的所有子节点
SelectDescendants()	根据 XPath 表达式选择当前节点的所有下级节点
SelectSingleNode()	使用 XPath 表达式选择一个节点

如果 XPathNavigator 类是从 XPathDocument 类中创建的,它就是只读的。如果 XPathNavigator 类是从 XmlDocument 类中创建的,它就可以用于编辑文档。查看 CanEdit 属性就可以验证这一点。

如果该属性是 `true`，就可以使用某个“插入”方法。`InsertBefore()`和 `InsertAfter()`会分别在当前节点的前面和后面新建一个节点。新节点的源可以来自 `XmlReader` 类或字符串。还可以返回一个 `XmlWriter` 类，它用于写入新节点信息。

使用 `ValueAs` 属性可以从节点中读取强类型化的值。注意这与 `XmlReader` 类不同，`XmlReader` 类使用 `ReadValue()`方法。

3. XPathNodeIterator 类

`XpathNodeIterator` 类可以看作是 `XPath` 中的一个 `NodeList` 或一个 `NodeSet`，这个对象有两个属性和 3 个方法：

- `Clone()` —— 新建它本身的一个副本。
- `Count` —— `XpathNodeIterator` 对象中的节点数。
- `Current` —— 返回指向当前节点的 `XpathNavigator`。
- `CurrentPosition()` —— 返回表示当前位置的一个整数。
- `MoveNext()` —— 移动到匹配 `XPath` 表达式的下一个节点上，`XPath` 表达式用于创建 `XpathNodeIterator`。

`XpathNodeIterator` 类由 `XpathNavigators` 类的 `Select()`方法返回，使用它可以迭代 `XpathNavigator` 类的“选择”方法返回的节点集。使用 `XpathNodeIterator` 类的 `MoveNext()`方法不会改变创建它的 `XpathNavigator` 类的位置。

4. 使用 XPath 名称空间中的类

要理解这些类的用法，最好是查看一下迭代 `books.xml` 文档的代码，弄清楚导航是如何工作的。为了使用这些示例，首先需要添加对 `System.Xml.Xsl` 和 `System.Xml.XPath` 名称空间的引用，如下所示：

```
using System.Xml.XPath;
using System.Xml.Xsl;
```

对于这个示例，使用 `booksxpath.xml` 文件，它类似于前面使用的 `books.xml` 文件，但在 `booksxpath.xml` 文件中添加了两本书。下面是窗体代码，这段代码是 `XmlSample` 项目的一部分：



可从
wrox.com
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    //modify to match your path structure
    XPathDocument doc = new XPathDocument("books.xml");
    //create the XPath navigator
    XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
    //create the XPathNodeIterator of book nodes
    // that have genre attribute value of novel
    XPathNodeIterator iter = nav.Select("/bookstore/book[@genre='novel']");
    textBox1.Text = "";
    while (iter.MoveNext())
    {
        XPathNodeIterator newIter =
            iter.Current.SelectDescendants(XPathNodeType.Element, false);
        while (newIter.MoveNext())
```

```

        textBox1.Text += newIter.Current.Name + ": " +
            newIter.Current.Value + "\r\n";
    }
}

```

代码段 frmNavigator.cs

在 `button1_Click()` 方法中, 首先创建 `XPathDocument` (命名为 `doc`), 其参数是要打开的文档的文件和路径字符串。下面一行代码创建 `XPathNavigator`:

```
XPathNavigator nav = doc.CreateNavigator();
```

本例用 `Select()` 方法检索所有 `genre` 属性值为 `novel` 的一组节点, 然后使用 `MoveNext()` 方法迭代书籍列表中的所有小说。

要把数据加载到列表框中, 使用 `XPathNodeIterator.Current` 属性。根据 `XPathNodeIterator` 指向的节点, 新建一个 `XPathNavigator` 对象。在本例中, 为文档中的一个 `book` 节点创建 `XPathNavigator`。

之后的循环接受这个 `XPathNavigator`, 调用 `Select()` 方法的另一个重载版本 `SelectDescendants()` 创建另一个 `XpathNodeIterator`。这样, `XPathNodeIterator` 就包含了 `book` 节点的所有子节点。

然后, 在这个 `XPathNodeIterator` 上执行另一个 `MoveNext()` 循环, 给文本框加载元素名和元素值。

在运行代码后, 屏幕显示 33-2 所示的对话框, 注意只列出了小说。

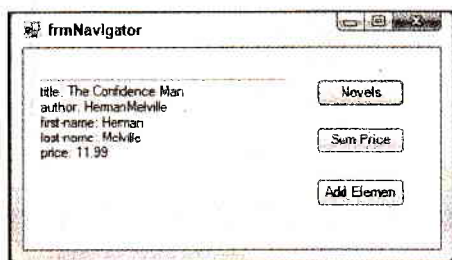


图 33-2

如果要把这些书的成本相加, 该怎么办? `XpathNavigator` 类为此包含了 `Evaluate()` 方法。`Evaluate()` 有 3 个重载版本, 第 1 个版本包含一个字符串, 该字符串就是 `XPath` 函数调用。第 2 个重载版本的参数是 `XPathExpression` 对象, 第 3 个重载版本的参数是 `XPathExpression` 和 `XPathNodeIterator`。下面的代码类似于前面的示例, 但这次迭代文档中的所有节点。最后调用的 `Evaluate()` 方法汇总了所有书籍的成本:



```

private void button2_Click(object sender, EventArgs e)
{
    //modify to match your path structure
    XPathDocument doc = new XPathDocument("books.xml");
    //create the XPath navigator
    XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
    //create the XPathNodeIterator of book nodes
    XPathNodeIterator iter = nav.Select("/bookstore/book");
    textBox1.Text = "";
    while (iter.MoveNext())
    {

```

```

XPathNodeIterator newIter =
    iter.Current.SelectDescendants(XPathNodeType.Element, false);
while (newIter.MoveNext())
{
    textBox1.Text += newIter.Current.Name + ": " + newIter.Current.Value +
        "\r\n";
}
textBox1.Text += "=====" + "\r\n";
textBox1.Text += "Total Cost = " +
    nav.Evaluate("sum(/bookstore/book/price)");
}

```

代码段 frmNavigator.cs

这次，可以看到文本框中书籍的总成本，如图 33-3 所示。

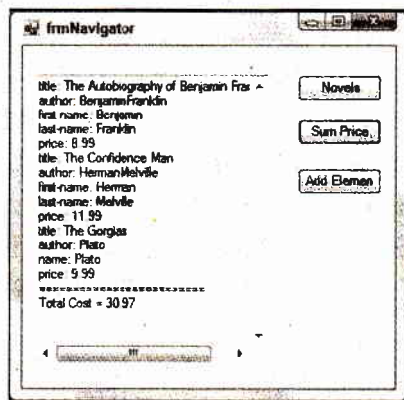


图 33-3

现在，假定需要添加一个折扣节点。使用 `InsertAfter()` 方法可以很容易插入该节点。下面是代码：



可从
wrox.com
下载源代码

```

private void button3_Click(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();
    doc.Load("books.xml");
    XPathNavigator nav = doc.CreateNavigator();

    if (nav.CanEdit)
    {
        XPathNodeIterator iter =
            nav.Select("/bookstore/book/price");
        while (iter.MoveNext())
        {
            iter.Current.InsertAfter("<disc>5</disc>");
        }
    }
    doc.Save("newbooks.xml");
}

```

代码段 frmNavigator.cs

这段代码在价格元素的后面添加了 `<disc>5</disc>` 元素。首先选择前面所有的价格节点，使用 `XPathNodeIterator` 迭代节点，并插入新节点。修改后的文档保存为一个新名称 `newbooks.xml`。下面

是新文档的内容:

```
<?xml version="1.0"?>
<!--This file represents a fragment of a book store inventory database-->
<bookstore>
  <book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
    <disc>5</disc>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
    <disc>5</disc>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
    <disc>5</disc>
  </book>
</bookstore>
```

节点可以插入到选中的节点之前或之后。还可以修改节点，并删除它们。如果对许多节点进行了改动，那么最好使用从 `XmlDocument` 中创建的 `XPathNavigator`。

33.6.2 System.Xml.Xsl 名称空间

`System.Xml.Xsl` 名称空间包含 .NET Framework 用于支持 XSL 转换的类。这个名称空间中的类可以和任何实现 `IXPathNavigable` 接口的存储器一起使用。在目前的 .NET Framework 中，包含 `XmlDocument`、`XmlDataDocument` 和 `XPathDocument`。与 `XPath` 一样，应使用最有效的存储器。如果计划创建一个自定义存储器，如文件系统的存储器，并希望能够进行一定的转换，就应在类中实现 `IXPathNavigable` 接口。

XSL 基于一个流式上拉模式(streaming pull mode)上。因此，可以把几个转换链接在一起。根据需要，甚至可以在转换之间应用一个自定义读取器，这样在设计时就会有有很大的灵活性。

1. 转换 XML

第一个示例接受 `books.xml` 文档，并使用 XSLT 文件 `book.xsl` 把它转换为一个简单的 HTML 文档，以进行显示(这段代码在 `XSLSample01` 文件夹中)，需要添加如下 `using` 语句：

```
using System.IO;
using System.Xml.Xsl;
using System.Xml.XPath;
```

下面是执行转换的代码:



可从
wrox.com
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    XslCompiledTransform trans = new XslCompiledTransform();
    trans.Load("books.xml");
    trans.Transform("books.xml", "out.html");
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "out.html");
}
```

代码下载 XslSample01.sln

这就是一个最简单的转换。首先新建一个 `XslCompiledTransform` 对象。它加载 `books.xml` 转换文档, 然后执行转换。在本例中, 把带文件名的字符串用作输入。输出是 `out.html`。之后把这个文件加载到窗体使用的 Web 浏览器控件上。这里不把文件名 `books.xml` 用作输入文档, 而是使用一个基于 `IXPathNavigable` 的对象。它可以是创建 `XpathNavigator` 的任何对象。

创建 `XslCompiledTransform` 对象, 加载样式表后, 就执行转换。`Transform()` 方法的参数可以是 `IXPathNavigable` 对象、流、`TextWriter`、`XmlWriter` 和 `URI` 的任意组合, 因此转换流上的灵活性很大。可以把转换的结果作为输入传递给下一个转换操作。

`XsltArgumentLists` 和 `XmlResolver` 对象也包含在参数选项中。下一节介绍 `XsltArgumentList` 对象。基于 `XmlResolver` 的对象用于解析当前文档外部的数据项, 如架构、证书或样式表。

`books.xml` 文档是一个很简单的样式表, 如下所示:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Price List</title>
      </head>
      <body>
        <table>
          <xsl:apply-templates/>
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <tr><td>
      <xsl:value-of select="title"/>
    </td><td>
      <xsl:value-of select="price"/>
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

2. 使用 XsltArgumentList

XsltArgumentList 是把对象和方法绑定到名称空间上的一种方式。绑定之后,就可以在转换过程中调用该方法。下面是一个示例:



可从
wrox.com
下载源代码

```
private void button3_Click(object sender, EventArgs e)
{
    //new XPathDocument
    XPathDocument doc = new XPathDocument("books.xml");
    //new XsltTransform
    XslCompiledTransform trans = new XslCompiledTransform();
    trans.Load("booksarg.xsl");
    //new XmlTextWriter since we are creating a new xml document
    XmlWriter xw = new XmlTextWriter("argSample.xml", null);
    //create the XsltArgumentList and new BookUtils object
    XsltArgumentList argBook = new XsltArgumentList();
    BookUtils bu = new BookUtils();
    //this tells the argumentlist about BookUtils
    argBook.AddExtensionObject("urn:XslSample", bu);
    //new XPathNavigator
    XPathNavigator nav = doc.CreateNavigator();
    //do the transform
    trans.Transform(nav, argBook, xw);
    xw.Close();
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "argSample.xml");
}
```

代码段 XslSample01.sln

下面是 BooksUtil 类的代码,这是将在转换过程中调用的类:



可从
wrox.com
下载源代码

```
class BookUtils
{
    public BookUtils() { }
    public string ShowText()
    {
        return "This came from the ShowText method!";
    }
}
```

代码段 BookUtils.cs

下面是转换的结果。其结果已进行了格式化,以便于查看(argSample.xml):

```
<books>
  <discbook>
    <booktitle>The Autobiography of Benjamin Franklin</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Confidence Man</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Gorgias</booktitle>
```

```

    <showtext>This came from the ShowText method!</showtext>
  </discbook>
</discbook>
  <booktitle>The Great Cookie Caper</booktitle>
  <showtext>This came from the ShowText method!</showtext>
</discbook>
</discbook>
  <booktitle>A Really Great Book</booktitle>
  <showtext>This came from the ShowText method!</showtext>
</discbook>
</books>

```

本例定义一个新的 `BookUtils` 类。在这个类中有一个无用的方法，它返回字符串“This came from the ShowText method!”。在 `button3_Click` 事件中，创建 `XPathDocument` 和 `XsltTransform` 对象，前面的示例把 XML 文档和转换文档直接加载到 `XslCompiledTransform` 对象中，这次使用 `XPathNavigator` 来加载文档。

接着编写下面的代码：

```

XsltArgumentList argBook=new XsltArgumentList();
BookUtils bu=new BookUtils();
argBook.AddExtensionObject("urn:XslSample",bu);

```

这段代码创建 `XsltArgumentList` 对象。接着创建 `BookUtils` 对象的一个实例，在调用 `AddExtensionObject()` 方法时，其参数是扩展的名称空间和要从中调用方法的对象。在调用 `Transform()` 时，其参数是 `XsltArgumentList(argBook)`，以及前面创建的 `XPathNavigator` 和 `XmlWriter` 对象。

下面是 `booksarg.xml` 文档(基于 `books.xml`)：

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:bookUtil="urn:XslSample">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <xsl:element name="books">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <xsl:element name="discbook">
      <xsl:element name="booktitle">
        <xsl:value-of select="title"/>
      </xsl:element>
      <xsl:element name="showtext">
        <xsl:value-of select="bookUtil:ShowText()" />
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>

```

两行重要的代码已突出显示。首先，在给 `XsltArgumentList` 添加对象时，添加前面创建的名称

空间。然后,在调用对应方法时,使用标准的 XSLT 名称空间前缀语法。

另一种方式是使用 XSLT 脚本完成。可以在该样式表中包含 C#、VB 和 JavaScript 代码。最重要的是在 Transform.Load()调用中编译该脚本,这与目前的非.NET 实现方式不同。这样就可以执行已经编译的脚本。

下面对前面的 XSLT 文件也进行这样的修改。首先给样式表添加脚本,在 bookscript.xml 中进行的修改如下所示:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:msxsl="urn:schemas-microsoft-com:xslt"
                xmlns:user="http://wrox.com">

  <msxsl:script language="C#" implements-prefix="user">

    string ShowText()
    {
      return "This came from the ShowText method!";
    }
  </msxsl:script>

  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:element name="books">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <xsl:element name="discbook">
      <xsl:element name="booktitle">
        <xsl:value-of select="title"/>
      </xsl:element>
      <xsl:element name="showtext">
        <xsl:value-of select="user:ShowText()"/>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

同样,其中的改变已突出显示。设置脚本的名称空间,添加代码(可以从 VS.NET IDE 中复制和粘贴代码);在样式表上执行调用。输出结果与前面示例的输出结果相同。

33.6.3 调试 XSLT

Visual Studio 2010 有调试转换功能。我们可以单步逐行地执行转换代码,查看变量,访问主调栈,设置断点,这些都与调试 C#源代码相同。调试转换代码有两种方式:仅使用样式表和 XML 输入文件,或者运行转换代码所在的应用程序。

1. 在不运行应用程序的情况下调试

第一次创建转换操作时,有时并不希望运行整个应用程序,只希望使样式表工作。Visual Studio

2010 允许使用 XSLT 编辑器进行这个操作。

把 books.xml 样式表加载到 Visual Studio 2010 的 XSLT 编辑器中。在下面的代码上设置一个断点：

```
<xsl:value-of select="title"/>
```

现在选择 XML 菜单，再选择 Debug XSLT 命令。此时需要指定 XML 输入文档，这就是我们要转换的 XML。在默认配置下，结果如图 33-4 所示。

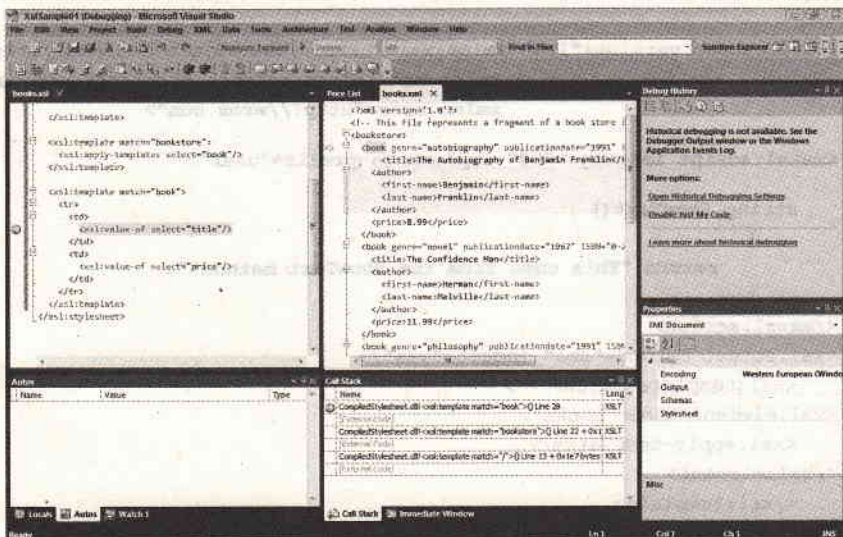


图 33-4

既然已经暂停转换，就可以查看与调试源代码时相同的几乎所有调试信息。注意调试器显示了 XSLT、输入文档、当前突出显示的元素和转换的结果。现在就可以单步执行转换代码。如果 XSLT 包含脚本代码，则还可以在脚本中设置断点，使用相同的调试功能。

2. 在运行应用程序的情况下调试

如果要同时调试转换代码和应用程序，就必须在创建 `XslCompiledTransform` 对象时进行一个小小的改动。构造函数的一个重载版本把一个布尔值作为参数，这个参数是 `enableDebug`，默认为 `false`，表示即使在转换代码中设置一个断点，如果运行调用转换代码的应用程序，它也不会中断。如果把该参数设置为 `true`，就会生成 XSLT 的调试信息，并在断点处暂停。所以在前面的示例中，创建 `XslCompiledTransform` 的代码行应改为：

```
XslCompiledTransform trans = new XslCompiledTransform(true);
```

现在当应用程序运行在调试模式时，甚至 XSLT 会生成调试信息，同样可以在样式表中获得完整的 Visual Studio 调试功能。

总之，在进行转换时，一定要记住使用正确的 XML 数据存储器。如果不需要编辑功能，就使用 `XPathDocument`；如果要从 ADO.NET 中获得数据，就使用 `XmlDataDocument`；如果需要编辑数据，就使用 `XmlDocument`。其他过程都相同。

33.7 XML 和 ADO.NET

XML 是把 ADO.NET 绑定到其他语言中的纽带。ADO.NET 从一开始就在 XML 环境中工作。XML 用于在数据存储器 and 应用程序或网页之间来回传输数据。因为 ADO.NET 使用 XML 进行远程传输,所以数据可以在甚至不能识别 ADO.NET 的应用程序和系统之间交换。因为 XML 在 ADO.NET 中非常重要,所以 ADO.NET 提供了一些强大的功能来读写 XML 文档。System.Xml 名称空间也包含可以使用 ADO.NET 关系数据的类。

用于示例的数据库来自于 AdventureWorksLT 示例应用程序。示例数据库可以从 codeplex.com/SqlServerSamples 上下载。注意 AdventureWorks 数据库有几个版本,大多数版本都可以工作,但 LT 版本是简化版本,足以达到本章的目的。

33.7.1 将 ADO.NET 数据转换为 XML 文档

第一个示例使用 ADO.NET、流和 XML 把数据库中的一些数据推入 DataSet 中,从 DataSet 中加载带有 XML 的 XmlDocument 对象,并把 XML 加载到文本框中。为了运行下面几个示例,需要添加如下 using 语句:

```
using System.Data;
using System.Xml;
using System.Data.SqlClient;
using System.IO;
```

把连接字符串定义为模块级变量:

```
string _connectString = "Server=.\SQLExpress;
                        Database=adventureworksLT;Trusted_Connection=Yes";
```

对于 ADO.NET 示例,在窗体上添加一个 DataGrid 对象,这样就可以在 ADO.NET 的 DataSet 中查看数据,因为数据被绑定到网格上。还可以查看生成的 XML 文档中的数据,这些数据已加载到文本框中。下面是第一个示例的代码。本示例的第一步是创建标准的 ADO.NET 对象,以生成一个 DataSet 对象。之后把该数据集绑定到网格上。



可从
wrox.com
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();
    DataSet ds = new DataSet("XMLProducts");
    SqlConnection conn = new SqlConnection(_connectString);
    SqlDataAdapter da = new SqlDataAdapter
        ("SELECT Name, StandardCost FROM SalesLT.Product", conn);

    //fill the dataset
    da.Fill(ds, "Products");
    //load data into grid
    dataGridView1.DataSource = ds.Tables["Products"];
}
```

代码段 frmADOXML.cs

在创建了 ADO.NET 对象,并绑定到网格上后,再实例化 MemoryStream、StreamReader 和 StreamWriter 对象。StreamReader 和 StreamWriter 对象使用 MemoryStream 对象来浏览 XML 文档:

```
MemoryStream memStrm=new MemoryStream();
```

```
StreamReader strmRead=new StreamReader(memStrm);
StreamWriter strmWrite=new StreamWriter(memStrm);
```

使用 `MemoryStream` 的原因是不必把数据写入磁盘中。但还可以使用基于 `Stream` 类的其他对象，如 `FileStream`。

下一步是生成 XML。调用 `DataSet` 类的 `WriteXml()` 方法，它生成一个 XML 文档。`WriteXml()` 方法有两个重载版本，一个重载版本的参数是带有文件路径和名称的字符串，另一个重载版本还有一个模式参数。这个模式是一个 `XmlWriteMode` 枚举，其值可以是

- `IgnoreSchema`
- `WriteSchema`
- `DiffGram`

如果不希望 `WriteXml()` 方法把内联架构写入 XML 文件的开头，就使用 `IgnoreSchema` 参数；如果要写入内联架构，就使用 `WriteSchema` 参数。`DiffGram` 显示在 `DataSet` 中编辑前后的数据。

 可从 wrox.com 下载源代码

```
//write the xml from the dataset to the memory stream
ds.WriteXml(strmWrite, XmlWriteMode.IgnoreSchema);
memStrm.Seek(0, SeekOrigin.Begin);
//read from the memory stream to a XmlDocument object
doc.Load(strmRead);
//get all of the products elements
XmlNodeList nodeList = doc.SelectNodes("//XMLProducts/Products");
textBox1.Text = "";

foreach (XmlNode node in nodeList)
{
    textBox1.Text += node.InnerXml + "\r\n";
}
```

代码段 frmADOXML.cs

在 33-5 所示的对话框中，可以查看列表中的数据 and 绑定的数据网格。

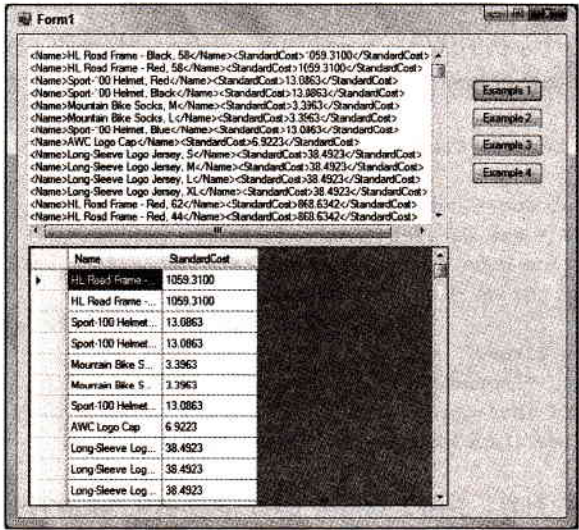


图 33-5

如果只需要该架构,就本应调用 `WriteXmlSchema()`方法而不是 `WriteXml()`方法。这个方法有 4 个重载版本,第 1 个重载版本的参数是一个字符串,其中包含了写入 XML 文档的位置对应的路径和文件名,第 2 个重载版本使用基于 `XmlWriter` 类的一个对象。第 3 个重载版本使用基于 `TextWriter` 类的对象。第 4 个重载版本使用派生自 `Stream` 类的对象。

此外,如果要把 XML 文档持久化到磁盘中,就应执行下述操作:

```
string file = "c:\\test\\product.xml";
ds.WriteXml(file);
```

在磁盘上会生成一个格式良好的 XML 文档,它可以由另一个流或 `DataSet` 来读取,或者由另一个应用程序或 Web 站点使用。因为没有指定 `XmlMode` 参数,所以这个 `XmlDocument` 包含了该架构。在本例中,把流作为 `XmlDocument.Load()`方法的参数。

现在数据有两个视图,但更重要的是,可以使用两个不同的模型来处理数据。可以使用 `System.Data` 名称空间来操纵数据,也可以使用 `System.Xml` 名称空间来处理数据。这样应用程序就可以有某些非常灵活的设计,因为现在不必把一个对象模型绑定到程序上。这是 ADO.NET 和 `System.Xml` 组合的强大之处。相同数据可以有多个视图,访问数据也有多种方式。

下一个示例消除 3 个流,并使用内置于 `System.Xml` 名称空间中的一些 ADO 功能来简化过程。但需要修改模块级的代码行:

```
private XmlDocument doc = new XmlDocument();
```

为:

```
private XmlDataDocument doc;
```

这么做的原因是现在使用的是 `XmlDataDocument`。下面是代码:



可从
wrox.com
下载源代码

```
private void button3_Click(object sender, EventArgs e)
{
    XmlDataDocument doc;
    //create a dataset
    DataSet ds = new DataSet("XMLProducts");
    //connect to the northwind database and
    //select all of the rows from products table
    SqlConnection conn = new SqlConnection(_connectString);
    SqlDataAdapter da = new SqlDataAdapter
        ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
    //fill the dataset
    da.Fill(ds, "Products");
    ds.WriteXml("sample.xml", XmlWriteMode.WriteSchema);
    //load data into grid
    dataGridView1.DataSource = ds.Tables[0];
    doc = new XmlDataDocument(ds);
    //get all of the products elements
    XmlNodeList nodeList = doc.GetElementsByTagName("Products");
    textBox1.Text = "";
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.InnerXml + "\r\n";
    }
}
```

代码段 frmADOXML.cs

可以看出,把 DataSet 对象加载到 XML 文档中的代码已经进行了简化。它没有使用 XmlDocument 类,而是使用 XmlDataDocument 类,这个类是为了使用 DataSet 对象的数据而专门设计的。

因为 XmlDataDocument 类基于 XmlDocument 类,所以它拥有 XmlDocument 类的所有功能。一个主要区别是 XmlDataDocument 类有重载的构造函数。注意下面的代码实例化 XmlDataDocument 对象 doc:

```
doc = new XmlDataDocument(ds);
```

它传递的参数是我们创建的 DataSet 对象 ds,从 DataSet 对象中创建 XML 文档,而且不必使用 Load()方法。实际上,如果实例化一个新的 XmlDataDocument 对象而不把 DataSet 作为参数,该 XmlDataDocument 就会包含一个名为 NewDataSet 的 DataSet,其 table 集中没有任何 DataTable。在创建了基于 XmlDataDocument 的对象后,还可以设置一个 DataSet 属性。

如果把下面一行代码添加到 DataSet.Fill()调用的后面:

```
ds.WriteXml("c:\\test\\sample.xml", XmlWriteMode.WriteSchema);
```

就会在文件夹 c:\test 中生成下面的 XML 文件 sample.xml:

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
  <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="XMLProducts" msdata:IsDataSet="true"
      msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Products">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Name" type="xs:string" minOccurs="0" />
                <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  <Products>
    <Name> HL Road Frame-Black, 58 </Name>
    <StandardCost> 1059.3100 </StandardCost>
  </Products>
  <Products>
    <Name> HL Road Frame-Red, 58 </Name>
    <StandardCost> 1059.3100 </StandardCost>
  </Products>
  <Products>
    <Name> Sport-100 Helmet, Red</Name>
    <StandardCost>13.0863</StandardCost>
  </Products>
</XMLProducts>
```

这里只显示了前几个 **Products** 元素。实际的 XML 文件应包含 Northwind 数据库中 **Products** 表的所有 **Products** 元素。

转换关系数据

这看起来非常简单，因为只有一个表。但对于关系数据，如 **DataSet** 中有多个 **DataTable** 和 **Relation**，该怎么办？其工作方式仍旧是这样。下面的示例使用了两个关系表：



可从
wrox.com
下载源代码

```
private void button5_Click(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();
    DataSet ds = new DataSet("XMLProducts");
    SqlConnection conn = new SqlConnection(_connectString);
    SqlDataAdapter daProduct = new SqlDataAdapter
        ("SELECT Name, StandardCost, ProductCategoryID FROM SalesLT.Product", conn);
    SqlDataAdapter daCategory = new SqlDataAdapter
        ("SELECT ProductCategoryID, Name from SalesLT.ProductCategory", conn);
    //Fill DataSet from both SqlAdapters
    daProduct.Fill(ds, "Products");
    daCategory.Fill(ds, "Categories");
    //Add the relation
    ds.Relations.Add(ds.Tables["Categories"].Columns["ProductCategoryID"],
        ds.Tables["Products"].Columns["ProductCategoryID"]);
    //Write the Xml to a file so we can look at it later
    ds.WriteXml("Products.xml", XmlWriteMode.WriteSchema);
    //load data into grid
    dataGridView1.DataSource = ds.Tables[0];
    //create the XmlDataDocument
    doc = new XmlDataDocument(ds);
    //Select the productname elements and load them in the grid
    XmlNodeList nodeList = doc.SelectNodes("//XMLProducts/Products");
    textBox1.Text = "";
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.InnerXml + "\r\n";
    }
}
```

代码段 frmADOXML.cs

在这个示例中，在 **XMLProducts** 数据集中创建了两个 **DataTable**：**Products** 和 **Categories**。在两个表的 **ProductCategoryID** 列上新建一个关系。

使用与上一个示例相同的 **WriteXml()** 方法调用，得到如下 XML 文件(SuppProd.xml)：

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
  <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="XMLProducts" msdata:IsDataSet="true"
      msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Products">
            <xs:complexType>
```

```

        <xs:sequence>
            <xs:element name="Name" type="xs:string" minOccurs="0" />
            <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
            <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Categories">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
            <xs:element name="Name" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:unique name="Constraint1">
    <xs:selector xpath="."//Categories" />
    <xs:field xpath="ProductCategoryID" />
</xs:unique>
<xs:keyref name="Relation1" refer="Constraint1">
    <xs:selector xpath="."//Products" />
    <xs:field xpath="ProductCategoryID" />
</xs:keyref>
</xs:element>
</xs:schema>
<Products>
    <Name>HL Road Frame-Black, 58</Name>
    <StandardCost>1059.3100</StandardCost>
    <ProductCategoryID>18</ProductCategoryID>
</Products>
<Products>
    <Name>HL Road Frame-Red, 58</Name>
    <StandardCost>1059.3100</StandardCost>
    <ProductCategoryID>18</ProductCategoryID>
</Products>
</XMLProducts>

```

该架构包含 DataSet 中的两个 DataTable。此外，数据包含两个表中的所有数据。简洁起见，这里只显示第一个 Products 和 ProductCategory 记录。与以前一样，使用正确的 XmlWriteMode 参数可以只保存架构或数据。

33.7.2 把 XML 文档转换为 ADO.NET 数据

假设有一个 XML 文档，准备把它转换为 ADO.NET 的 DataSet。这样就可以把 XML 加载到数据库中，或者把数据绑定到 .NET 数据控件上，如 DataGrid。此时实际上可以把 XML 文档用作数据存储，完全消除数据库的系统开销。如果数据比较少，这完全有可能。看看下面这些代码：



可从
wrox.com
下载源代码

```

private void button7_Click(object sender, EventArgs e)
{
    //create the DataSet
    DataSet ds = new DataSet("XMLProducts");
}

```

```

//read in the xml document
ds.ReadXml("Products.xml");

//load data into grid
dataGridView1.DataSource = ds.Tables[0];

textBox1.Text = "";

foreach (DataTable dt in ds.Tables)
{
    textBox1.Text += dt.TableName + "\r\n";
    foreach (DataColumn col in dt.Columns)
    {
        textBox1.Text += "\t" + col.ColumnName + "-" + col.DataType.FullName + "\r\n";
    }
}

```

代码段 frmADOXML.cs

这很简单。这个示例实例化一个新的 `DataSet` 对象，然后，从此处调用 `ReadXml()` 方法，把 XML 放在 `DataSet` 的一个 `DataTable` 中。与 `WriteXml()` 方法一样，`ReadXml()` 方法的参数是 `XmlReadMode`。`ReadXml()` 方法还可以使用 `XmlReadMode` 中的更多选项(如表 33-9 所示)。

表 33-9

选 项	说 明
Auto	把 <code>XmlReadMode</code> 设置为最合适值。如果数据是 <code>DiffGram</code> 格式，就选择 <code>DiffGram</code> ；如果已经读取了架构，或者检测到某个内联架构，就选择 <code>ReadSchema</code> 。如果没有为 <code>DataSet</code> 指定模式，也没有检测到内联架构，就选择 <code>IgnoreSchema</code>
DiffGram	读取 <code>DiffGram</code> ，并把变化应用到 <code>DataSet</code> 上
Fragment	读取包含 XDR 架构片断的文档，如 SQL Server 创建的类型
IgnoreSchema	忽略任何找到的内联架构，把数据读入当前的 <code>DataSet</code> 架构中，如果数据与 <code>DataSet</code> 架构不匹配，就删除它
InferSchema	忽略任何内联架构，根据 XML 文档中的数据创建架构。如果 <code>DataSet</code> 中已经有一个架构，就使用该架构，根据需要，可以用其他列或表来扩展它。如果存在一列，但其数据类型不符，就会抛出一个异常
ReadSchema	读取内联架构，并加载数据。不重写 <code>DataSet</code> 中的架构，但如果内联架构中的表已经存在于 <code>DataSet</code> 中，就抛出一个异常

这里也有 `ReadXmlSchema()` 方法，它读取独立的架构，并创建相应的表、列和关系。如果架构没有和数据建立关联，就可以使用 `ReadXmlSchema()` 方法。该方法也有 4 个重载版本：其参数分别是包含文件和路径名的字符串、基于 `Stream` 的对象、基于 `TextReader` 的对象和基于 `XmlReader` 的对象。

要说明如何正确地创建数据表，可以迭代表和列，并在文本框中显示名称。可以把它与最初的 Northwind 数据库比较一下，可以发现所有内容保持不变。最后一个 `foreach` 循环执行这个任务。图 33-6 显示了输出结果。

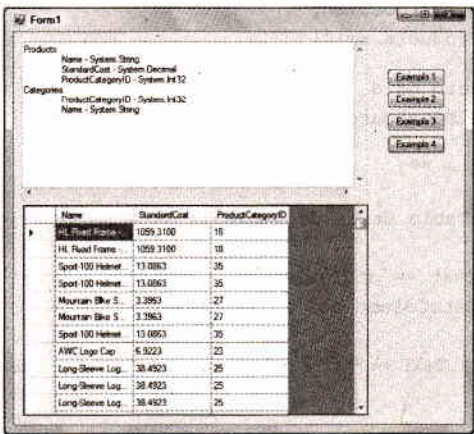


图 33-6

从列表框中可以看出，数据表中包含的所有列都有正确的名称和数据类型。

还要注意，前两个示例都没有在数据库传入或传出任何数据，所以没有定义 `SqlDataAdapter` 或 `SqlConnection`。这说明了 `System.Xml` 名称空间和 ADO.NET 的灵活性：可以用多种格式查看相同的数据。如果需要进行转换，以 HTML 格式显示数据，或者如果需要把数据绑定到网格上，就应获取这些数据，用一个方法调用，把它们以需要的格式显示出来。

33.8 在 XML 中序列化对象

序列化是把一个对象持久化到磁盘中的过程。应用程序的另一部分，甚至另一个应用程序都可以反序列化对象，使它的状态与序列化之前相同。.NET Framework 为此提供了两种方式。

本节将介绍 `System.Xml.Serialization` 名称空间。它包含的类可用于把对象序列化为 XML 文档或流。这表示把对象的公共属性和公共字段将转换为 XML 元素和/或属性。

`System.Xml.Serialization` 名称空间中最重要的类是 `XmlSerializer`。要序列化对象，首先需要实例化一个 `XmlSerializer` 对象，指定要序列化的对象类型，然后实例化一个流/写入器对象，以把文件写入流/文档中。最后一步是在 `XmlSerializer` 上调用 `Serialize()` 方法，给它传递流/写入器对象和要序列化的对象。

被序列化的数据可以为基元类型的数据、字段、数组，以及 `XmlElement` 和 `XmlAttribute` 对象格式的内嵌 XML。

为了从 XML 文档中反序列化对象，应执行上述过程的逆过程。即创建一个流/读取器对象和一个 `XmlSerializer` 对象，然后给 `Deserialize()` 方法传递该流/读取器对象。这个方法返回反序列化的对象，尽管它需要强制转换为正确的类型。



XML 序列化程序不能转换私有数据，只能转换公共数据，它也不能序列化对象图表。但是，这并不是一个严格的限制。对类进行仔细设计，就很容易避免这个问题。如果需要序列化公共数据和私有数据，以及包含许多嵌套对象的对象图形，就可以使用 `System.Runtime.Serialization.Formatters.Binary` 名称空间。

使用 `System.Xml.Serialization` 类可以完成的其他工作如下所示:

- 确定数据应是一个特性还是元素
- 指定名称空间
- 改变特性名或元素名

对象和 XML 文档之间的链接是给类添加注释的自定义 C# 特性, 这些属性可以告诉序列化程序如何输出数据。 .NET Framework 包含一个工具 `xsd.exe`, 它可以帮助创建这些特性。 `xsd.exe` 可以完成如下任务:

- 从 XDR 架构文件中生成 XML 架构
- 从 XML 文件中生成 XML 架构
- 从 XSD 架构文件中生成 `DataSet` 类
- 生成运行库类, 运行库类包含 `XmlSerialization` 类的自定义属性
- 从已经开发出来的类中生成 XSD 文件
- 限制在代码中创建的元素
- 确定生成代码的编程语言(C#、VB 或 JScript.NET)
- 在编译好的程序集中从类型中创建架构

参见 Framework 文档, 了解 `xsd.exe` 命令行选项的详细内容。

尽管 `xsd.exe` 具备这些功能, 但不一定用它为序列化创建类。这个过程很简单。下面介绍一个简单的应用程序, 它序列化一个类。示例代码的起始部分比较简单, 新建一个 `Product` 对象 `pd`, 并给它填充一些数据:



可从
wrox.com
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    //new products object
    Product pd = new Product();
    //set some properties
    pd.ProductID = 200;
    pd.CategoryID = 100;
    pd.Discontinued = false;
    pd.ProductName = "Serialize Objects";
    pd.QuantityPerUnit = "6";
    pd.ReorderLevel = 1;
    pd.SupplierID = 1;
    pd.UnitPrice = 1000;
    pd.UnitsInStock = 10;
    pd.UnitsOnOrder = 0;
}
```

代码段 frmSerial.cs

`XmlSerializer` 类的 `Serialize()` 方法实际上执行序列化, 它有 9 个重载版本。一个必需的参数是要写入数据的流, 它可以是 `Stream`、`TextWriter` 或 `XmlWriter` 参数。在本例中, 创建一个基于 `TextWriter` 的对象 `tr`。接着创建基于 `XmlSerializer` 类的对象 `sr`。 `XmlSerializer` 类需要知道正在序列化的对象的类型信息, 所以对要序列化的类型使用 `typeof` 关键字。在创建 `sr` 对象后, 调用 `Serialize()` 方法, 其参数是 `tr` (基于 `Stream` 的对象) 和要序列化的对象, 在本例中是 `pd`。确保完成后关闭该数据流。



可从
wrox.com
下载源代码

```
//new TextWriter and XmlSerializer
TextWriter tr = new StreamWriter("serialprod.xml");
XmlSerializer sr = new XmlSerializer(typeof(Product));
//serialize object
sr.Serialize(tr, pd);
tr.Close();
webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "serialprod.xml");
```

代码段 frmSerial.cs

下面介绍 **Products** 类，即要序列化的类。这个类与以前编写的任何其他类的唯一区别是给它添加了 C# 特性。这些特性中的 **XmlRootAttribute** 类和 **XmlElementAttribute** 类继承自 **System.Attribute** 类。不要把这些特性与 XML 文档中的特性相混淆。C# 属性仅是一些声明信息，在运行期间可以由 CLR 检索到(详见第 8 章)。在本例中，添加一些描述如何序列化对象的特性：



可从
wrox.com
下载源代码

```
//class that will be serialized.
//attributes determine how object is serialized
[System.Xml.Serialization.XmlRootAttribute()]
public class Product {
    private int prodId;
    private string prodName;
    private int suppId;
    private int catId;
    private string qtyPerUnit;
    private Decimal unitPrice;
    private short unitsInStock;
    private short unitsOnOrder;
    private short reorderLvl;
    private bool discount;
    private int disc;
    //added the Discount attribute
    [XmlAttributeAttribute(AttributeName="Discount")]
    public int Discount {
        get {return disc;}
        set {disc=value;}
    }
    [XmlElementAttribute()]
    public int ProductID {
        get {return prodId;}
        set {prodId=value;}
    }
    [XmlElementAttribute()]
    public string ProductName {
        get {return prodName;}
        set {prodName=value;}
    }
    [XmlElementAttribute()]
    public int SupplierID {
        get {return suppId;}
        set {suppId=value;}
    }
    [XmlElementAttribute()]
    public int CategoryID {
```

```

        get {return catId;}
        set {catId=value;}
    }
    [XmlElementAttribute()]
    public string QuantityPerUnit {
        get {return qtyPerUnit;}
        set {qtyPerUnit=value;}
    }
    [XmlElementAttribute()]
    public Decimal UnitPrice {
        get {return unitPrice;}
        set {unitPrice=value;}
    }
    [XmlElementAttribute()]
    public short UnitsInStock {
        get {return unitsInStock;}
        set {unitsInStock=value;}
    }
    [XmlElementAttribute()]
    public short UnitsOnOrder {
        get {return unitsOnOrder;}
        set {unitsOnOrder=value;}
    }
    [XmlElementAttribute()]
    public short ReorderLevel {
        get {return reorderLvl;}
        set {reorderLvl=value;}
    }
    [XmlElementAttribute()]
    public bool Discontinued {
        get {return discount;}
        set {discount=value;}
    }
    public override string ToString()
    {
        StringBuilder outText = new StringBuilder();
        outText.Append(prodId);
        outText.Append(" ");
        outText.Append(prodName);
        outText.Append(" ");
        outText.Append(unitPrice);
        return outText.ToString();
    }
}

```

代码段 frmSerial.cs

在 `Products` 类定义上面的特性中调用的 `XmlRootAttribute()` 方法把这个类标识为根元素(在 XML 文件中, 由于序列化生成)。包含 `XmlElementAttribute()` 方法的特性标识该特性下面的成员表示一个 XML 元素。

注意重写了 `ToString()` 方法, 它提供了运行反序列化示例时显示在消息框中的字符串。

查看一下刚才在序列化过程中创建的 XML 文档, 就会发现它与前面创建的其他 XML 文档非常类似。这就是本练习的目的。下面就是这个文档:

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  Discount="0">
  <ProductID>200</ProductID>
  <ProductName>Serialize Objects</ProductName>
  <SupplierID>1</SupplierID>
  <CategoryID>100</CategoryID>
  <QuantityPerUnit>6</QuantityPerUnit>
  <UnitPrice>1000</UnitPrice>
  <UnitsInStock>10</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>1</ReorderLevel>
  <Discontinued>>false</Discontinued>
</Products>
```

这里没有任何不寻常的地方。可以以使用 XML 文档的任何方式来使用这个文档。可以对它进行转换,并以 HTML 格式显示它,使用 ADO.NET 将其加载到 DataSet 中,用它加载 XmlDocument,或者像在该示例中那样,对它进行反序列化,并创建一个对象,该对象的状态与序列化前 pd 的状态一样(这就是第二个按钮的作用)。

接着添加另一个按钮事件处理程序,以反序列化一个基于 Products 的新对象 newPd。这次使用 FileStream 对象读取 XML:

```
private void button2_Click(object sender, EventArgs e)
{
    //create a reference to product type
    Product newPd;
    //new filestream to open serialized object
    FileStream f = new FileStream("serialprod.xml", FileMode.Open);
```

同样,传递 Product 的类型信息,新建一个 XmlSerializer 类。然后就可以调用 Deserialize()方法。注意在创建 newPd 对象时,仍需要进行显式的类型强制转换。此时 newPd 与 pd 的状态完全一样:



可从
wrox.com
下载源代码

```
//new serializer
XmlSerializer newSr = new XmlSerializer(typeof(Product));
//deserialize the object
newPd = (Product)newSr.Deserialize(f);
f.Close();
MessageBox.Show(newPd.ToString());
}
```

代码段 frmSerial.cs

消息框应显示产品 ID、产品名称和刚才反序列化的对象的单价。这是因为使用了在 Product 类中实现的 ToString()方法。

如果有派生的类和可能返回一个数组的属性,则也可以使用 XmlSerializer 类。下面介绍一个解决这些问题的复杂示例。

首先定义 3 个新类 Product、BookProduct (派生于 Product)和 Inventory (它包含其他两个类)。注意又重写了 ToString()方法,这次要列出 Inventory 类中的项:



```
public class BookProduct: Product
{
    private string isbnNum;
    public BookProduct() {}
    public string ISBN
    {
        get {return isbnNum;}
        set {isbnNum=value;}
    }
}

public class Inventory
{
    private Product[] stuff;
    public Inventory() {}
    //need to have an attribute entry for each data type
    [XmlAttribute("Prod",typeof(Product)),
    XmlArrayItem("Book",typeof(BookProduct))]
    public Product[] InventoryItems
    {
        get {return stuff;}
        set {stuff=value;}
    }
    public override string ToString()
    {
        StringBuilder outText = new StringBuilder();
        foreach (Product prod in stuff)
        {
            outText.Append(prod.ProductName);
            outText.Append("\r\n");
        }
        return outText.ToString();
    }
}
```

代码段 frmSerial.cs

在此我们只对 `Inventory` 类感兴趣。如果序列化这个类，就需要插入一个特性，该特性为每个要添加到数组中的类型包含一个 `XmlArrayItem` 构造函数。注意，`XmlArrayItem` 是由 `XmlAttribute` 类表示的.NET 特性的名称。

这些构造函数的第一个参数是在序列化过程中创建的 XML 文档中的元素名。如果不使用 `ElementName` 参数，元素名就会与对象类型名相同(在本例中，就是 `Product` 和 `BookProduct`)。必须指定的第二个参数是对象的类型。

如果属性返回一个对象数组或基元类型的数组，则还要使用 `XmlArrayAttribute` 类。因为要在数组中返回不同的类型，所以使用 `XmlArrayItemAttribute` 类，它允许进行更高级别的控制。

在 `button4_Click()` 事件处理程序中，新建一个 `Product` 对象和一个 `BookProduct` 对象(`newProd` 和 `newBook`)。给每个对象的各种属性添加数据，再把这些对象添加到一个 `Product` 数组中。接下来新建一个 `Inventory` 对象，并把这个数组作为参数，然后序列化 `Inventory` 对象，以便在以后重新创建它：



```
private void button4_Click(object sender, EventArgs e)
{
    //create the XmlAttributes object
    XmlAttributes attrs = new XmlAttributes();
    //add the types of the objects that will be serialized
    attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
    attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));
    XmlAttributeOverrides attrOver = new XmlAttributeOverrides();
    //add to the attributes collection
    attrOver.Add(typeof(Inventory), "InventoryItems", attrs);
    //create the Product and Book objects
    Product newProd = new Product();
    BookProduct newBook = new BookProduct();
    newProd.ProductID = 100;
    newProd.ProductName = "Product Thing";
    newProd.SupplierID = 10;
    newBook.ProductID = 101;
    newBook.ProductName = "How to Use Your New Product Thing";
    newBook.SupplierID = 10;
    newBook.ISBN = "123456789";
    Product[] addProd = { newProd, newBook };
    Inventory inv = new Inventory();
    inv.InventoryItems = addProd;
    StreamWriter tr = new StreamWriter("inventory.xml");
    XmlSerializer sr = new XmlSerializer(typeof(Inventory), attrOver);
    sr.Serialize(tr, inv);
    tr.Close();
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "inventory.xml");
}
```

代码段 frmSerial.cs

XML 文档如下所示:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
  </Product>
  <Book Discount="0">
    <ProductID>101</ProductID>
    <ProductName>How to Use Your New Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
```

```

    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
    <ISBN>123456789</ISBN>
  </Book>
</Inventory>

```

button2_Click()事件处理程序实现 **Inventory** 对象的反序列化。注意在新建的 **newInv** 对象中, 我们迭代了数组, 以说明其数据保持不变:

```

private void button2_Click(object sender, System.EventArgs e)
{
    Inventory newInv;
    FileStream f=new FileStream("order.xml", FileMode.Open);
    XmlSerializer newSr=new XmlSerializer(typeof(Inventory));
    newInv=(Inventory)newSr.Deserialize(f);
    foreach(Product prod in newInv.InventoryItems)
        listBox1.Items.Add(prod.ProductName);
    f.Close();
}

```

不能访问源代码的序列化

这些代码都很好地发挥了作用, 但如果不能访问已经序列化的类型的源代码, 该怎么办? 如果没有源代码, 就不能添加属性。此时可以采用另一种方式。可以使用 **XmlAttribute** 类和 **XmlAttributeOverrides** 类, 这两个类可以完成刚才的任务, 但不需要添加属性。下面的代码说明了这两个类的工作方式。

对于这个示例, 假定 **Inventory**、**Product** 和派生的 **BookProduct** 类在一个单独的 DLL 中, 而且没有源代码。**Product** 和 **BookProduct** 类与前面的示例相同, 但应注意 **Inventory** 类中没有添加属性:



```

public class Inventory
{
    private Product[] stuff;
    public Inventory() {}
    public Product[] InventoryItems
    {
        get {return stuff;}
        set {stuff=value;}
    }
}

```

代码段 frmSerial.cs

下面处理 **button1_Click()**事件处理程序中的序列化:

```

private void button1_Click(object sender, System.EventArgs e)
{

```

序列化过程的第一步是创建一个 **XmlAttribute** 对象, 为每个要重写的数据类型创建一个 **XmlElementAttribute** 对象:

```

XmlAttribute attrs=new XmlAttributes();
attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));

```

从中可以看出,我们给 `XmlAttributes` 类的 `XmlElement` 集合添加了新的 `XmlElementAttribute` 对象。`XmlAttributes` 类的属性对应于可以应用的特性,前面示例中的 `XmlArray` 和 `XmlArrayItems` 仅是其中的几个属性而已。现在我们有有一个 `XmlAttributes` 对象,并在 `XmlElement` 集合中添加了两个基于 `XmlElementAttribute` 的对象。

接着创建 `XmlAttributeOverrides` 对象:

```
XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
attrOver.Add(typeof(Inventory),"InventoryItems",attrs);
```

这个类的 `Add()` 方法有两个重载版本。第一个重载版本的参数是要重写的对象的类型信息和前面创建的 `XmlAttributes` 对象,本例使用第二个重载版本,其参数也是一个字符串值,该字符串值是重写对象的成员。在本例中,要重写 `Inventory` 类中的 `InventoryItems` 成员。

创建 `XmlSerializer` 对象时,把 `XmlAttributeOverrides` 对象添加为参数。现在 `XmlSerializer` 类知道我们要重写的类和需要为这些类型返回的内容。



可从
wrox.com
下载源代码

```
//create the Product and Book objects
Product newProd=new Product();
BookProduct newBook=new BookProduct();
newProd.ProductID=100;
newProd.ProductName="Product Thing";
newProd.SupplierID=10;
newBook.ProductID=101;
newBook.ProductName="How to Use Your New Product Thing";
newBook.SupplierID=10;
newBook.ISBN="123456789";
Product[] addProd={newProd,newBook};

Inventory inv=new Inventory();
inv.InventoryItems=addProd;
TextWriter tr=new StreamWriter("inventory.xml");
XmlSerializer sr=new XmlSerializer(typeof(Inventory),attrOver);
sr.Serialize(tr,inv);
tr.Close();
}
```

代码段 frmSerial.cs

如果执行 `Serialize()` 方法,就会得到如下 XML 输出:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>>false</Discontinued>
  </Product>
```

```

<Book Discount="0">
  <ProductID>101</ProductID>
  <ProductName>How to Use Your New Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>0</ReorderLevel>
  <Discontinued>false</Discontinued>
  <ISBN>123456789</ISBN>
</Book>
</Inventory>

```

可以看出, 得到的 XML 与前面的示例完全相同。为了反序列化对象, 重新创建基于 **Inventory** 的对象, 需要创建在序列化对象时创建所有相同的 **XmlAttribute**s、**XmlElementAttribute** 和 **XmlAttributeOverrides** 对象。之后就可以读取 XML, 像以前那样重新创建 **Inventory** 对象。下面的代码反序列化 **Inventory** 对象:

```

private void button2_Click(object sender, System.EventArgs e)
{
    //create the new XmlAttributes collection
    XmlAttributes attrs=new XmlAttributes();
    //add the type information to the elements collection
    attrs.XmlElements.Add(new XmlElementAttribute("Book",typeof(BookProduct)));
    attrs.XmlElements.Add(new XmlElementAttribute("Product",typeof(Product)));

    XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
    //add to the Attributes collection
    attrOver.Add(typeof(Inventory),"InventoryItems",attrs);

    //need a new Inventory object to deserialize to
    Inventory newInv;

    //deserialize and load data into the listbox from deserialized object
    FileStream f=new FileStream("..\..\..\inventory.xml",FileMode.Open);
    XmlSerializer newSr=new XmlSerializer(typeof(Inventory),attrOver);

    newInv=(Inventory)newSr.Deserialize(f);
    if(newInv!=null)
    {
        foreach(Product prod in newInv.InventoryItems)
        {
            listBox1.Items.Add(prod.ProductName);
        }
    }
    f.Close();
}

```

注意, 前几行代码与序列化对象所用的代码相同。

System.Xml.XmlSerialize 名称空间提供了一个功能非常强大的工具集, 可以把对象序列化到 XML 中。把对象序列化和反序列化到 XML 中替代了把对象保存为二进制格式, 因此可以通过 XML 对对象进行其他处理。这将大大增强设计的灵活性。

33.9 LINQ to XML 和.NET

把 LINQ 引入 .NET Framework 中时, 其重点是便于访问要在应用程序中使用的数据。由于在应用程序存储空间中, 一个主要的数据存储器是 XML, 因此很自然地演变为创建 LINQ to XML。

在 LINQ to XML 发布之前, 通过 System.Xml 使用 XML 并不是很容易实现。引入 System.Xml.Linq 名称空间后, 就可以利用一系列功能在代码中方便地处理 XML。

在应用程序代码中创建 XML 时, 许多开发人员以前都使用 XmlDocument 对象。这个对象可以创建 XML 文档, 以层次结构的方式追加元素、特性和其他项。通过 LINQ to XML 和引入的 System.Xml.Linq 名称空间, 会发现有一些新对象使 XML 文档的创建更加简单。

33.10 使用不同的 XML 对象

除了 .NET 4 包含的 LINQ 查询功能之外, .NET Framework 提供的 XML 对象也非常好, 它们甚至可以独立于 LINQ。在该版本中可以使用 XML 对象取代 DOM 的直接处理。在 System.Xml.Linq 名称空间中, 有一系列 LINQ to XML 帮助对象, 简化了内存中的 XML 文档的处理。

下面几节介绍这个名称空间中可用的新对象。



本章的许多示例都使用了 Hamlet.xml 文件。这个 XML 文件在 <http://metalab.unc.edu/bosak/xml/eg/shaks200.zip> 上, 以 XML 文件格式包含莎士比亚的所有戏剧。

33.10.1 XDocument 对象

Xdocument 对象替代了 .NET 3.5 之前的 XmlDocument 对象, 它更容易处理 XML 文档。XDocument 对象还和这个名称空间中的其他新对象一起使用, 如 XNamespace、XComment、XElement 和 XAttribute 对象。

XDocument 对象的一个更重要的成员是 Load() 方法:

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

这个操作会把 Hamlet.xml 文件的内容加载为内存中的一个 XDocument 对象。还可以给 Load() 方法传递一个 TextReader 或 XmlReader 对象。现在就可以以编程方式处理 XML 了:



可从
wrox.com
下载源代码

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
Console.WriteLine(xdoc.Root.Name.ToString());
Console.WriteLine(xdoc.Root.HasAttributes.ToString());
```

代码下载 ConsoleApplication1.sln

输出的结果如下:

```
PLAY
False
```

另一个重要的成员是 `Save()` 方法, 它类似于 `Load()` 方法, 可以保存到一个物理磁盘位置, 或一个 `TextWriter` 或 `XmlWriter` 对象中:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
xdoc.Save(@"C:\CopyOfHamlet.xml");
```

33.10.2 XElement 对象

一个常用的对象是 `XElement`。使用这个对象可以轻松创建包含单个元素的对象, 该对象可以是 XML 文档本身, 甚至可以只是 XML 片段。例如, 下面的例子写入一个 XML 元素及其相应的值:

```
XElement xe = new XElement("Company", "Lipper");
Console.WriteLine(xe.ToString());
```

在创建 `XElement` 对象时, 可以定义该元素的名称和元素中使用的值。在这个例子中, 元素的名称是 `<Company>`, `<Company>` 元素的值是 `Lipper`。在引用 `System.Xml.Linq` 名称空间的控制台应用程序中运行它, 得到的结果如下:

```
<Company>Lipper</Company>
```

还可以使用多个 `XElement` 对象创建比较完整的 XML 文档, 如下例所示:



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

            Console.WriteLine(xe.ToString());

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

代码下载 `ConsoleApplication1.sln`

运行这个应用程序, 得到的结果如图 33-7 所示。

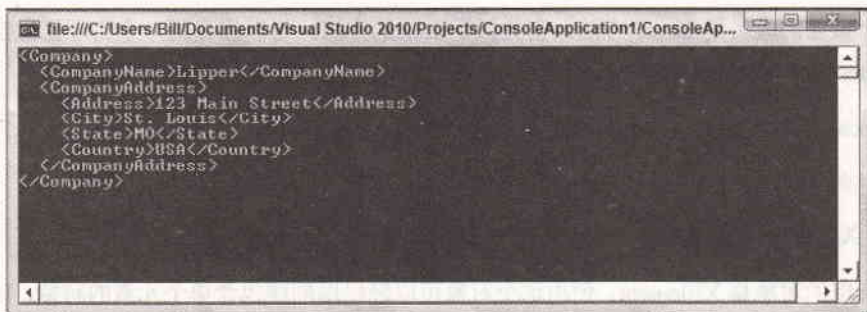


图 33-7

33.10.3 XNamespace 对象

XNamespace 对象表示 XML 名称空间，很容易应用于文档中的元素。例如，在前面的例子中，很容易给根元素应用一个名称空间：



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 ns = "http://www.lipperweb.com/ns/1";
            XElement xe = new XElement(ns + "Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));
            Console.WriteLine(xe.ToString());
            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.sln

在这个例子中，创建了一个 XNamespace 对象，具体方法是给它赋予 <http://www.lipperweb.com/ns/1> 的值。之后，就可以在根元素 <company> 中通过实例化 XElement 对象来使用它。

```
XElement xe = new XElement(ns + "Company", // .
```

这会生成如图 33-8 所示的结果。