第 14 章　ASP.NET XML 和 Web Service

在上一章中讲到的 Web.config 配置文件就是基于 XML 文件格式的，XML（Extensible Markup Language，可扩展标记语句）是一种描述数据和数据结构的语言，XML 文本可以保存在任何存储文本中，这就让 XML 具有了可扩展性、跨平台型以及传输与存储方面的优点。

14.1　XML 简介

标记语言（Markup Language）特指一系列约定好的标记来对电子文档进行标记，以实现对电子文档的语义、结构以及格式的定义。在 ASP.NET 开发中，最常用的标记语言就是 HTML，HTML 标记语言定义了 HTML 文档的语义、结构以及格式，以便在不同的浏览器中所呈现的内容是一致的。XML 标记语言与 SGML 和 HTML 都属于标记语言，标记语言的发展如图 14-1 所示。

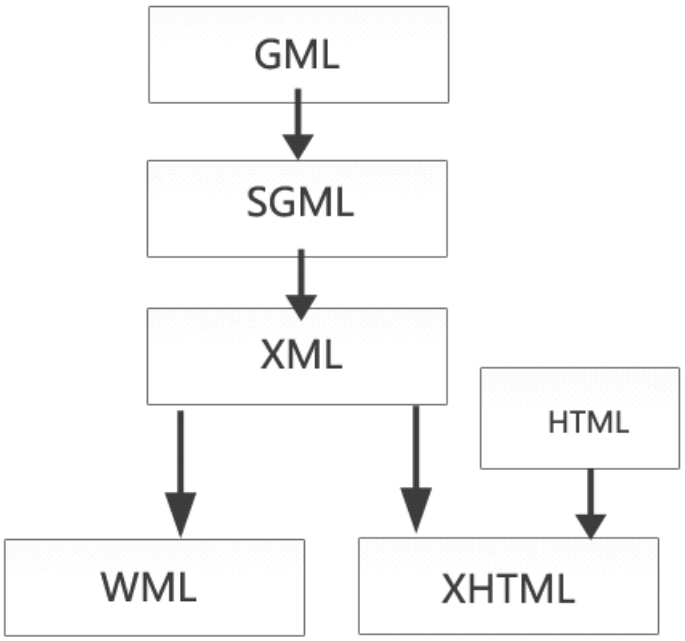

图 14-1　标记语言的发展史

为了更加方便的适应互联网的需求，1996 年开始创建 XML 标记语言。XML 标记语言不仅具备了 SGML 标记语言强大的扩展性，同样也具备 HTML 标记语言的易用性。不仅如此，ASP.NET 还将 XML 作为应用程序数据存储和传输的重要方法。

在当今互联网中，Web 应用已经成为一种分布式组件技术。传统的 Web 应用技术解决的问题是如何让人使用 Web 提供的应用，而当今的 Web 应用技术是要解决如何让应用程序使用 Web 应用。由于 Web 应用能够跨平台、跨语言的为应用程序提供服务，所以 Web 应用和 XML 应用的前景是非常广阔的。

注意:这里所说的 Web 应用的跨平台是针对浏览器而言,Windows 能够浏览一个 Web 应用,而 Linux 同样可以浏览一个 Web 应用。

14.2 读写 XML

XML 和 HTML 都是基于 SGML（Standard Generalized Markup Language，标准通用标记语言）的，但是 XML 和 HTML 却有着很大的区别，这些区别不仅仅在于格式上的区别，还在于使用性、可扩展性等等。

14.2.1 XML 与 HTML

XML 标记语言和 HTML 标记语言有着极大的不同，在应用程序开发中，XML 标记语言能够适应于大部分的应用程序环境和开发需求。这些需求是 HTML 标记语言无法做到的，XML 标记语言和 HTML 标记语言的具体区别如下所示。

- HTML 标记是固定的，并且是没有层次的，在 HTML 文档中，用户无法自行创建标签，例如<book>这样的标签浏览器很可能解析不了，HTML 中标记的作用是描述数据的显示方式，这种方式只能交付给浏览器进行处理，而 HTML 文档中的标记都是独立存在的，没有层次。
- XML 的标记不是固定的且是有层次的，在 XML 文档中，用户可以自行创建标签，例如<day>这样的标签，XML 标记不能够描述网页的外观和内容，XML 只能够描述内容的数据结构和层次，在浏览器中浏览 XML 文档，也可以发现 XML 标记是有层次的。

在 Visual Studio 2008 中，.NET Framework 提供了 System.XML 命名空间，该命名空间提供了一组可扩展类使得开发人员能够轻松的读、写、以及编辑 XML 文本。

14.2.2 创建 XML 文档

使用 Visual Studio 2008 能够创建 XML 文档，创建和使用 XML 文档无需 XML 语法分析器来专门负责分析语法，在.NET Framework 中已经集成了可扩展类。右击现有项，单击【添加新项】选项，选择 XML 文件，如图 14-2 所示。

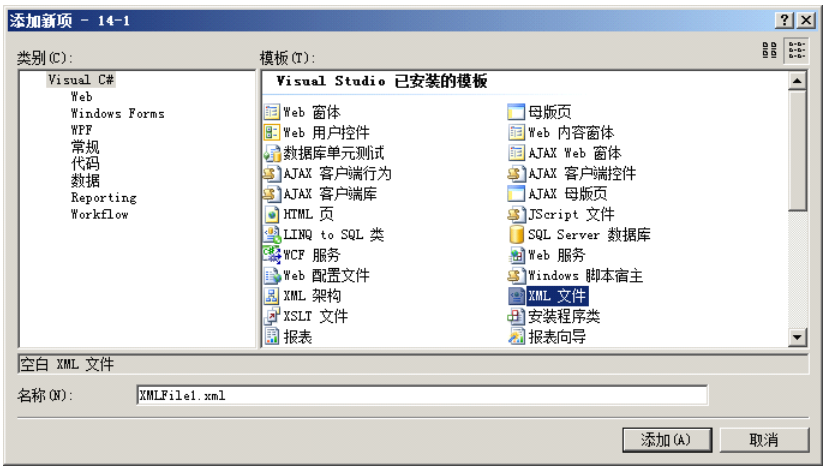

图 14-2 创建 XML 文档

创建完成后，就需要向 XML 文档中编写 XML 标记，以下是一个完整的 XML 文档示例。

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
    <ShopInformation area="Asia">
        <Shop place="Wuhan">
            <Name>武汉电脑城</Name>
            <Phone>123456789</Phone>
            <Seller>J.Dan</Seller>
            <Seller>Bill Gates</Seller>
        </Shop>
        <Shop place="ShangHai">
            <Name>武汉电脑城</Name>
            <Phone>123456789</Phone>
            <Seller>Bill Gates</Seller>
        </Shop>
    </ShopInformation>
    <ShopInformation area="USA">
        <Shop place="S">
            <Name>PC STORE</Name>
            <Phone>123456789</Phone>
            <Seller>J.Dan</Seller>
            <Seller>Bill Gates</Seller>
        </Shop>
        <Shop place="S.K">
            <Name>Windows Mobile Store</Name>
            <Phone>123456789</Phone>
            <Seller>Bill Gates</Seller>
        </Shop>
    </ShopInformation>
</Root>
```

上述 XML 文档使用了自定义标记对商城进行了描述，包括商城所在地、商城名称、电话号码以及负责人等。编写 XML 文档时，开发人员能够自定义标签进行文档描述，但是在 XML 文档的头部必须进行 XML 文档声明，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
```

上述代码在 XML 文档头部进行了声明，表示该文档是一个 XML 文档，并且说明该文档的版本为 1.0 的 XML 文档，该文档还可以包含一个 encoding 属性，指明文档中的编码类型。声明该文档是一个 XML 文档后，则需要在 XML 文档中编写根标记，这个标记可以是开发人员自定义标记名称，在这里被命名为 Root，示例代码如下。

```
<Root>
<!-- 根标记内的所有内容 -->
</Root>
```

上述代码创建了一个根标记，在这里命名为 Root。在 XML 文档中，所有的标记都应该被包含在一个根标记中，这样不仅方便描述也方便查阅。XML 文档中的根标记不能够重复使用，如果重复使用则会提示异常。

在根标记内，应该编写需要描述的信息的标记。在这里，描述一个商城需要的一些属性，包括商城所在的州、所在地以及商城的主营类型等，通过 XML 标记语言可以自行创建标记来描述，示例代码如下所示。

```
<ShopInformation area="USA">                                    //地区描述
    <Shop place="S">                                            //位置描述
        <Name>PC STORE</Name>                                   //商城名称
        <Phone>123456789</Phone>                                //商城电话
        <Seller>J.Dan</Seller>                                  //商城销售人员
        <Seller>Bill Gates</Seller>
    </Shop>
    <Shop place="S.K">
        <Name>Windows Mobile Store</Name>
        <Phone>123456789</Phone>
        <Seller>Bill Gates</Seller>
    </Shop>
</ShopInformation>
```

上述代码对商城的信息进行了描述，这些标签的意义如下所示：

- ShopInformation：商城信息，包括 area 属性来描述所在州或板块，这里说明了是在 USA 地区。
- Shop：商城在该板块的所在州、省市等信息。
- Name：商城的名称。
- Phone：商城的联系电话。
- Seller：商城的销售人员。

这些标签都是用户自定义的，XML 文档允许开发人员自定义标签并，另外，XML 文档也不局限所要描述的对象格式。例如当上述代码也可以编写另外一种样式时，同样能够被 XML 所识别，示例代码如下所示。

```
<ShopInformation>
    <Area name="Usa">                                           //另一种地区表示方式
        <Shop>
            <Name>PC STORE</Name>
            <Phone>123456789</Phone>
            <Seller>J.Dan</Seller>
            <Seller>Bill Gates</Seller>
            <Place>S.K</Place>                                  //地区直接放在描述中
        </Shop>
        <Shop>
            <Name>Windows Mobile Store</Name>
            <Phone>123456789</Phone>
            <Seller>Bill Gates</Seller>
            <Place>S</Place>
        </Shop>
    </Area>
</ShopInformation>
```

技巧：良好的缩进能够让 XML 文档更加方便阅读，同时 XML 文档是大小写敏感的，对于 XML 标记，标记头和标记尾的大小写规则必须匹配。

14.2.3 XML 控件

在 ASP.NET 中提供了针对 XML 读写的控件 XML 控件，XML 控件可以很好的解决 XML 文档的显示问题，如果需要浏览 XML 文档的数据，则只需要编写 XML 控件中的 DocumentSource 属性即可，示

例代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <div>
    <asp:Xml ID="xml1" runat="server" DocumentSource="~/XMLFile1.xml"></asp:Xml>        //使用 XML 控件
    </div>
    </form>
</body>
```

运行后如图 14-3 所示。

图 14-3　XML 控件

运行后会发现 XML 文档的内容都显示出来了，但是却没有层次感，因为 XML 控件并没有把记录分开，而是连续的呈现 XML 文档的内容。如果需要按照规范或开发人员的意愿呈现给浏览器，则必须使用 XSL 样式表。

14.2.4　XML 文件读取类（XmlTextReader）

XmlTextReader 类属于 System.Xml 命名空间，XmlTextReader 类提供对 XML 数据的快速、单项、无缓冲的数据读取功能，因为 XmlTextReader 类是基于流的，所以使用 XmlTextReader 类读取 XML 内容只能够从前向后读取，而不能逆向读取。

因为 XmlTextReader 类的流形式，节约了读取 XML 文档的时间，也大量的节约了读取 XML 所需花费的内存空间，当需要读取 XML 节点时，只需要使用 XmlTextReader 类的 Read()方法即可，示例代码如下所示。

```
XmlTextReader rd = new XmlTextReader(Server.MapPath("XMLFile1.xml"));          //构造函数
while (rd.Read())                                                              //遍历节点
{
    Response.Write("Node Type is : " + rd.NodeType +" <br/>");             //输出 Node
    Response.Write("Name is : " + rd.Name + " <br/>");                      //输出 Name
    Response.Write("Value is :" + rd.Value + " <br/>");                    //输出 Value
    Response.Write("<hr/>");
}
```

上述代码使用 XmlTextReader 类的构造函数创建了 XmlTextReader 对象，并通过使用 XmlTextReader 类的 Read()方法进行 XmlTextReader 对象的遍历。遍历 XML 文档后，需要使用 Close 方法进行 XmlTextReader 对象的关闭操作，这一点是非常重要的，如果不使用 XmlTextReader 类的 Close 方法，则相应的 XML 文件正在被进程使用，只有使用了 Close 方法才能将相应的文件关闭掉。示例代码如下

所示。

```
            rd.Close();                                                    //关闭 Reader
```

XmlTextReader 类遍历 XML 文件运行结果如图 14-4 所示。

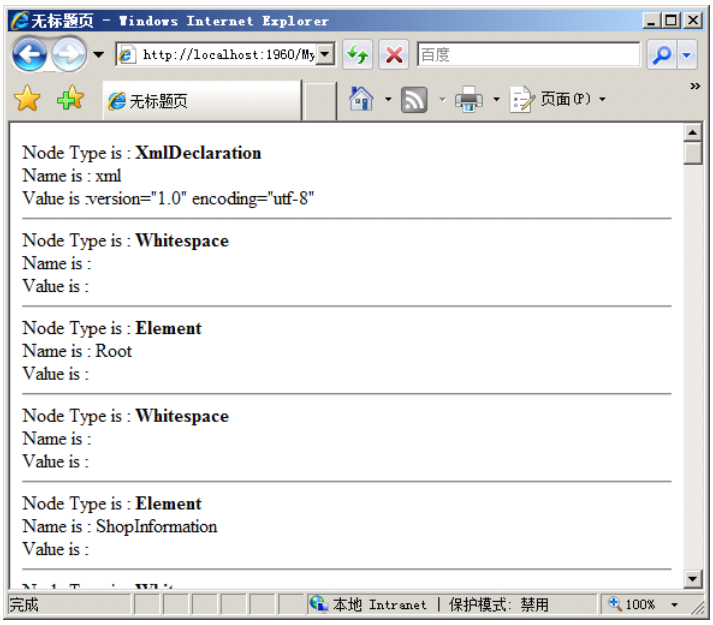

图 14-4　XmlTextReader 类遍历 XML 文件

在使用 XmlTextReader 类读取 XML 文件中相应的节点时，XmlTextReader 类的 NodeType 会检查节点的类型，而 XmlTextReader 类的 Name 和 Value 会分别检查节点的名称和值，相应的 XML 代码如下所示。

```
<Shop place="Wuhan">
    <Name>武汉电脑城</Name>
    <Phone>123456789</Phone>
    <Seller>J.Dan</Seller>
    <Seller>Bill Gates</Seller>
</Shop>
```

上述代码中，使用 XmlTextReader 类进行读取，则 Shop 节点的 NodeType 为 Element，Name 的值为 Shop，Value 的值为空。XML 文档中不止以上几种节点类型，XmlNodeType 也包括其他节点类型，这些类型如下所示。

- Attribut：XML 元素的属性。
- CDATA：用于转义文本块，避免将文本块识别为标记。
- Comment：XML 文档的注释。
- Document：作为文档树的根的文档对象，可供每个 XML 文档进行访问。
- DocumentType：XML 文档类型的声明。
- Element：XML 元素。
- EndElement：当 XmlTextReader 达到元素末尾时返回。
- Entity：实体声明。
- Text：元素的文本内容。
- WhiteSpace：标记间的空白。
- XmlDeclaration：XML 节点声明，它是文档中的第一个节点。

在 XML 文档中，空白标记和根节点的节点类型是不相同的，XmlTextReader 类读取 XML 文件并遍历节点类型，根节点和空白节点遍历后结果如下所示。

```
Node Type is:XmlDeclaration
Nameis:xml
Value is:version="1.0" encoding="utf-8"
Node Type is:Whitespace
Nameis:
Value is:
```

其中根节点的节点类型为 XmlDeclaration，Value 值为 version="1.0" encoding="utf-8"。

14.2.5 XML 文件编写类（XmlTextWriter）

XmlTextWriter 类属于 System.Xml 命名空间，同 XmlTextReader 类相同的是，XmlTextWriter 类同样提供没有缓存，直向前的方式进行 XML 文件操作，但是与 XmlTextReader 类操作相反，XmlTextWriter 类执行的是写操作。XmlTextWriter 类的构造函数包括三种重载形式，分别为一个字符串、一个流对象和一个 TextWriter 对象。通过使用 XmlTextWriter 类可以动态的创建 XML 文档，示例代码如下所示。

```
XmlTextWriter wr = new XmlTextWriter("newXml.xml", null);                    //读取 XML
try
{
    wr.Formatting = Formatting.Indented;                                     //格式化输出
    wr.WriteStartDocument();                                                 //开始编写文档
    wr.WriteStartElement("ShopInformation");                                 //编写节点
    wr.WriteStartElement("Shop");                                            //编写节点
    wr.WriteAttributeString("place", "北京");                                //编写节点
    wr.WriteElementString("Name", "中关村");                                 //编写节点
    wr.WriteElementString("Phone", "123456");                                //编写节点
    wr.WriteElementString("Seller", "Guojing");                              //编写节点
    wr.WriteEndElement();                                                    //结束节点编写
    wr.WriteEndElement();                                                    //结束节点编写
    Response.Write("操作成功");
}
catch
{
    Response.Write("操作失败");
}
```

上述代码创建了一个 XmlTextWriter 对象并通过 XmlTextWriter 对象编写 XML 文档，在使用 XmlTextWriter 类构造函数时，可以指定编码类型，或使用默认的编码类型，若使用默认的编码类型，参数传递 null 即可，默认编码类型将为 UTF-8，示例代码如下所示。

```
XmlTextWriter wr = new XmlTextWriter("newXml.xml", null);                    //创建写对象
```

使用了 XmlTextWriter 类创建对象后，则需要使用 XmlTextWriter 对象的 Formatting 方法指定输出的格式，示例代码如下所示。

```
wr.Formatting = Formatting.Indented;                                         //格式化输出
```

指定了输出格式之后，则需要开始为 XML 文档创建节点，在创建节点前，首先需要声明 XML 文档，则必须输出<?xml version="1.0" encoding="utf-8" ?>声明，声明 1.0 版本的 xml 文档代码如下所示。

```
wr.WriteStartDocument();                                                     //开始编写节点
```

声明文档后就可以使用 WriteStartElement 进行节点的创建，创建节点代码如下所示。

```
wr.WriteStartElement("Shop");                                    //开始编写节点
```

上述代码创建了 Shop 节点，如果需要为该节点创建 place="北京"属性则需要使用 WriteAttributeString 方法进行创建，示例代码如下所示。

```
wr.WriteAttributeString("place", "北京");                        //编写属性
```

创建了父节点之后，可以通过 WriteElementString 方法创建子节点，示例代码如下所示。

```
wr.WriteElementString("Name", "中关村");                         //创建子节点
```

节点全部创建完成后，需要使用 WriteEndElement 方法进行尾节点的编写，示例代码如下所示。

```
wr.WriteEndElement();                                            //结束节点编写
```

一个 XML 文档就编写完毕了，编写完成并不能自动的更新 XML 文档，还需要使用 Flush 方法进行数据更新，更新完毕后还需要关闭 XmlTextWriter 对象示例代码如下所示。

```
wr.Flush();                                                      //更新文件
wr.Close();                                                      //结束写对象
```

使用 Flush 方法就能够将 XML 数据保存在文件中，运行后 XML 文档结构如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<ShopInformation>
    <Shop place="北京">
        <Name>中关村</Name>
        <Phone>123456</Phone>
        <Seller>Guojing</Seller>
    </Shop>
</ShopInformation>
```

14.2.6 XML 文本文档类（XmlDocument）

XML 文档在内存中是以 DOM 为表现形式的，DOM（Document Object Model）是对象化模型，DOM 是以树的节点形式来标识 XML 数据，14.2.2 中的 XML 文档读入 DOM 结构中，则在内存中的构造图如图 14-5 所示。

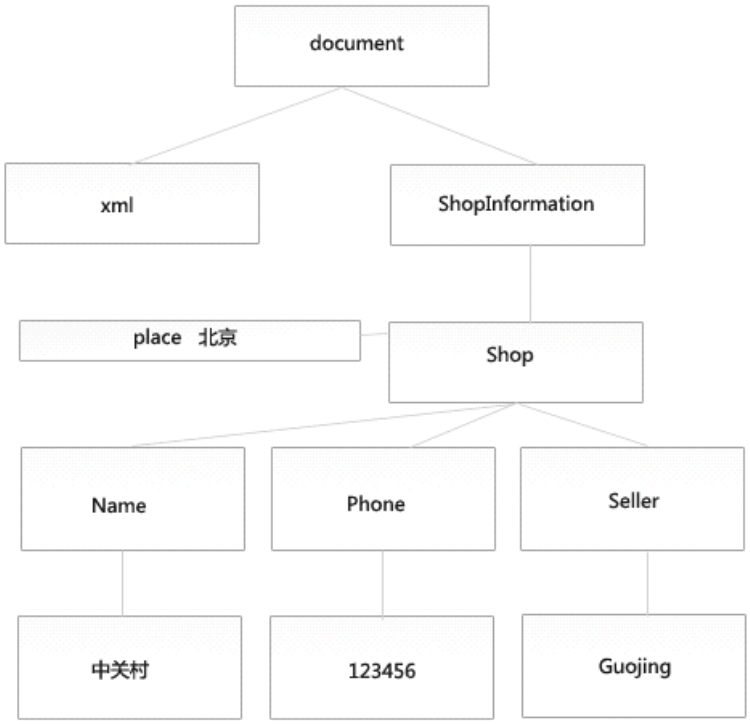

图 14-5　XML 文档构造

XmlDocument 类同样也属于命名空间 System.Xml，XmlDocument 类可以实现第一、第二级的 W3C DOM。它使用 DOM 以一个层次结构树的形式将整个 XML 数据加载到内存中，从而能够使开发人员能够对内存中的任意节点进行访问、插入、更新和删除。由于 XmlDocument 类，简化开发人员对 XML 文档进行访问、插入和删除等操作。

XmlDocument 类继承自 System.Xml.XmlNode，该抽象类表示单个节点并具有基本的属性和方法来操作节点。利用 XmlDocument 对象的 DocumentElement 属性能够表示单个节点并进行操作。XmlDocument 对象的 DocumentElement 返回一个指向文档元素的索引，可以通过读取给定的节点的 HanChildNodes 属性判断是否包括子节点。另外，使用 XmlDocument 对象的 HasChildNodes 和 ChildNodes 属性可以读取和遍历 XML 文件，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();                    //创建 XmlDocument 对象
    doc.Load(Server.MapPath("newXml.xml"));                 //载入文件
    Response.Write("读取中..<hr/>");
    XmlNode node = doc.DocumentElement;                     //读取节点
    output(node);                                           //使用输出函数
    Response.Write("读取完毕..<hr/>");                       //输出 HTML 字串
}
protected void output(XmlNode node)
{
    if (node != null)                                       //如果节点不等于空
    {
        format(node);                                   //格式化输出
```

```
        }
        if (node.HasChildNodes)                                    //判断是否包括子节点
        {
            node = node.FirstChild;                                //获取子节点
            while (node != null)
            {
                output(node);                                      //使用递归
                node = node.NextSibling;                           //遍历节点值和信息
            }
        }
    }
    protected void format(XmlNode node)
    {
        if (!node.HasChildNodes)                                   //判断是否包括子节点
        {
            Response.Write("node name is" + node.Name);            //输出节点
            Response.Write("node value is" + node.Value);          //输出节点
            Response.Write("<br/>");
        }
        else
        {
            Response.Write(node.Name);
            if (XmlNodeType.Element == node.NodeType)              //遍历节点
            {
                XmlNamedNodeMap map = node.Attributes;             //遍历节点
                foreach (XmlNode att in map)
                {
                    Response.Write("attrnode name is" + att.Name); //格式化输出节点
                    Response.Write("attrnode value is" + att.Value); //格式化输出节点
                    Response.Write("<br/>");
                }
            }
        }
    }
```

上述代码通过使用 XmlDocument 类遍历节点，使用 XmlDocument 类遍历节点，首先需要创建一个 XmlDocument 对象，并使用 Load 方法加载一个 XML 文档，示例代码如下所示。

```
        XmlDocument doc = new XmlDocument();                       //创建 XmlDocument 对象
        doc.Load(Server.MapPath("newXml.xml"));                    //载入 XML 文件
```

创建对象之后，则需要使用递归的方法遍历显示每个节点。在遍历节点的过程中，需要对每个节点进行是否有子节点的判断，如果包括子节点，则先输出子节点，如果没有子节点则继续输出根节点。XmlDocument 对象也可以向 XML 文档中添加一个新的元素，示例代码如下所示。

```
        XmlDocument doc = new XmlDocument();                       //创建 XmlDocument 对象
        doc.Load(Server.MapPath("newXml.xml"));                    //载入 XML 文件
        XmlNode node = doc.DocumentElement;                        //创建节点对象
        node.RemoveChild(node.FirstChild);                         //移除根节点
```

上述代码使用了 XmlDocument 对象的 Load 方法载入 XML 文档，当需要插入 XML 数据时，则先需要移除根节点，移除根节点之后就能够开始添加节点，示例代码如下所示。

```
        XmlNode Shop = doc.CreateElement("Shop");                  //创建节点 Shop
        XmlNode shop1 = doc.CreateElement("Name");                 //创建节点 Name
```

```
XmlNode shop2 = doc.CreateElement("Phone");                    //创建节点 Phone
XmlNode shop3 = doc.CreateElement("Seller");                   //创建节点 Seller
XmlNode NameText = doc.CreateTextNode("NameText");             //创建节点文本
XmlNode PhoneText = doc.CreateTextNode("PhoneText");           //创建节点文本
XmlNode SellerText = doc.CreateTextNode("SellerText");         //创建节点文本
shop1.AppendChild(NameText);                                   //添加文本
shop2.AppendChild(PhoneText);                                  //添加文本
shop3.AppendChild(SellerText);                                 //添加文本
Shop.AppendChild(shop1);                                       //添加 Shop 子节点
Shop.AppendChild(shop2);                                       //添加 Shop 子节点
Shop.AppendChild(shop3);                                       //添加 Shop 子节点
node.AppendChild(Shop);                                        //添加 Shop 节点
```

上述代码分别为节点添加子节点，并为子节点添加文本，添加完成后，需要使用 XmlDocument 对象的 Save 方法进行保存，示例代码如下所示。

```
doc.Save("newXml.xml");
```

使用 XmlDocument 对象的 Save 方法即可将 XML 内容保存在 XML 文档中。使用 XmlDocument 对象不仅能够读取，新增 XML 文档，还支持修改、删除等操作，例如使用 PrependChild 和 InsertBefore，InsertAfter 等方法进行新增和删除节点和子节点操作。

14.3 XML 串行化

使用 XML 串行化能够方便 XML 的存储或传输，能够把一个对象的公共域和属性保存为一种串行格式的过程，反串行化则是使用串行的状态信息将对象从串行 XML 状态还原为初始状态的过程。.NET Framework 提供了命名空间来简化开发人员进行 XML 串行化。

14.3.1 XmlSerializer 串行化类

.NET Framework 提供了 System.Runtime.Serialization 和 System.Xml.Serialization 以提供串行化功能。而 System.Xml.Serialization 提供了一个 XmlSerializer 类，该类可以将一个对象串行和反串行化为 XML 格式。XmlSerializer 类虽然能够执行串行化和反串行化，但是串行化和反串行化有一个最大的区别，就是串行化调用 Serialize 方法，而反串行化调用 Deserialize 方法。

如果需要将一个对象进行 XML 串行化，可以在类前添加[Serializable()]标识，声明一个定制串行化属性，如果需要将整个类都能够支持串行化，则必须添加该属性。XML 串行化还包括以下属性：

- [XmlRoot]：用来识别作为 XML 文件根元素的类或结构，可以用此标记把一个元素的名称设置为根元素。
- [XmlElement]：共有的属性或字段可以作为一个元素被串行化到 XML 结构中。
- [XmlAttribute]：共有的属性或字段可以作为一个属性被串行化到 XML 结构中。
- [XmlIgnore]：共有的属性或字段不包括在串行化的 XML 结构中。
- [XmlArray]：共有的属性或字段可以作为一个元素数组被串行到 XML 结构中。
- [XmlArrayItem]：用来识别可以放到一个串行化数组中的类型。

14.3.2 基本串行化

基本串行化是指让.NET Framework 自动的对对象进行串行化操作。使用基本串行化进行操作，只要求对象拥有类属性[Serializable()]即可。如果类中的某些属性或字段不需要进行串行化操作，则使用[NonSerializable()]属性即可，示例代码如下所示。

```
using System.Xml.Serialization;
namespace Ser
{
    [Serializable()]                                                    //设置串行化属性
    class MySer
    {
        private string Shop { get; set; }                               //设置 Shop 属性
        private string Name { get; set; }                               //设置 Name 属性
        private string Phone { get; set; }                              //设置 Phone 属性
        private string Seller { get; set; }                             //设置 Seller 属性
        [NonSerialized()]                                               //设置非串行化
        private string Age { get; set; }                                //设置 Age 属性
    }
}
```

上述代码使用了[Serializable()]属性声明一个类，则该类的成员都能够被串行化，而 Age 属性使用了[NonSerializable()]属性声明了该属性不应被串行化。使用[NonSerializable()]能够精确的控制串行化。若需要将类进行 XML 串行，则可以通过使用[XmlAttribute]等属性进行 XML 串行化操作，示例代码如下所示。

```
using System.Xml.Serialization;
namespace Ser
{
    [Serializable()]                                                    //设置串行化
    [XmlRoot("SerXML")]
    class MySer
    {
        [XmlElement("Shop")]                                            //设置节点属性
        private string Shop { get; set; }                               //设置 Shop 属性
        [XmlElement("Name")]                                            //设置节点属性
        private string Name { get; set; }                               //设置 Name 属性
        [XmlElement("Phone")]                                           //设置节点属性
        private string Phone { get; set; }                              //设置 Phone 属性
        [XmlElement("Seller")]                                          //设置节点属性
        private string Seller { get; set; }                             //设置 Seller 属性
        [NonSerialized()]                                               //设置非串行化
        [XmlElement("Age")]                                             //设置节点属性
        private string Age { get; set; }                                //设置 Age 属性
    }
}
```

上述代码运行后的结果形式呈现为 XML 格式，其格式如下所示。

```
<Shop>this.Shop</Shop>
<Name>this.Name</Name>
<Phone>this.Phone</Phone>
<Seller>this.Seller</Seller>
```

```
//<Age></Age>
```

在串行化结果输出时，并没有输出 Age 标签，是因为在 Age 属性前定义了[NonSerializable()]以控制该字段不会被串行化输出。

14.4 XML 样式表 XSL

XSL 是 XML 的样式表语言，这种定义很像 HTML 中的 CSS。XSL 转换就是 XSLT，XSLT 是 XSL 标准中的重要组成部分，它可以把一个 XML 文档的数据以不同结构或格式转换为另一个文档，通过使用 XSL 能够将 XML 进行格式化输出。

14.4.1 XSL 简介

与 HTML 样式相比，XML 样式要更加复杂，HTML 中标记的含义都是固定的，而 XML 允许开发人员能够自行创建标签，所以用样式控制 XML 会比在 HTML 控制更加复杂。

1．XSL 与 HTML

HTML 样式控制通常是使用 CSS 层叠样式表进行控制，而 XML 中的样式控制通常使用 XSL 文档进行控制，HTML 中的 CSS 和 XML 中的 XSL 有以下特征：

- CSS：用于 HTML 样式控制，由于 HTML 标签事先是规定好的，所以浏览器知道如何显示 HTML 样式，如在 HTML 文档中，<table></table>标签能够以表格的形式呈现在客户端浏览器。
- XSL：用于 XML 样式控制，由于 XML 的标签不是事先规定好的，所以 XML 文档中的标记并不能被浏览器理解，如 XML 文档中的<table></table>并不会被浏览器解释成表格。

相比于 CSS 而言，XSL 能够作为 XML 文件的样式表而存在，而 XSL 又不仅仅需要提供样式控制，还需要提供 XML 文档的方法等，XSL 包括 3 部分，这 3 部分分别为：

- 转换 XML 文档的方法。
- 定义 XML 部分和模式的方法。
- 格式化 XML 文档的方法。

2．XSLT

XSLT 翻译为可扩展样式语言转换，XSL 包含三种语言，其中最重要的是 XSLT。XSL 转换实际上就是 XSLT，在 Visual Studio 2008 中，可以直接创建 XSLT 文件对 XML 文件进行样式控制，定义部分和模板方法。

XSLT 可以将一个 XML 文档的数据以不同的结构或格式转换为另一个文档格式，如 HTML。为了让 XML 文件能够格式化输出到浏览器并能够进行样式控制，XSLT 能够对 XML 进行样式控制，通过编写模板，以及简易的编程控制就能够读取 XML 中节点的数据并重新组织，当用户访问 XML 文件时，能够同 HTML 一样被浏览器解释并呈现到客户端浏览器。

3．XSLT 与 XSL

XSLT 是 XSL 中最重要的语言，也是 XSL 中最重要的组成部分，简而言之，XSL 通过 XSLT 将一个 XML 源中的数据重新组织并呈现成另一种 XML 样式。

14.4.2 使用 XSLT

使用 XSL 对 XML 进行样式控制和格式化 XML 文档，首先需要创建一个 XML 文档，这里 XML 文档代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
    <ShopInformation area="Asia">
        <Shop place="Shanghai" value="Wuhan">
            <Name>上海电脑</Name>
            <Phone>123456789</Phone>
            <Seller>J.Dan</Seller>
            <Seller>Bill Gates</Seller>
        </Shop>
        <Shop place="Wuhan">
            <Name>广埠屯</Name>
            <Phone>123456789</Phone>
            <Seller>Bill Gates</Seller>
        </Shop>
    </ShopInformation>
</Root>
```

创建完成 XML 文档后则需要创建 XSL 文档，如图 14-6 所示。

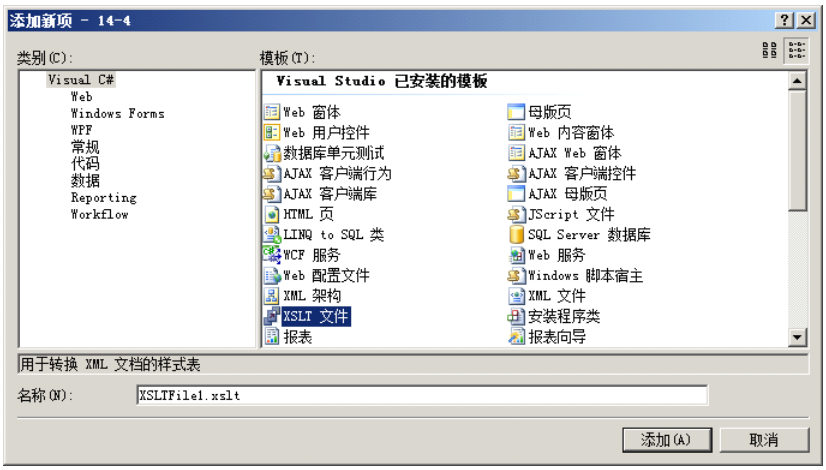

图 14-6　创建 XSLT 文件

创建 XSLT 文件后，系统会自动生成代码，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

上述代码将 XSLT 文件的输出方法设置为 XML，为了能够方便对 XML 页面进行样式控制，可以将

输出方法设置为 HTML，XSLT 文件示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl">
    <xsl:output method="html" indent="yes"/>
    <xsl:template match="/Root/ShopInformation">
        <head>
            <title>
                一个 XSLT 样例
            </title>
        </head>
        <body>
            <div style="border:1px solid #ccc; padding:5px 5px 5px 5px;font-size;14px;">一个 XSLT 样例</div>
            <div style="padding:5px 5px 5px 5px;font-size;12px;">
                <xsl:value-of select="Shop"/>
            </div>
        </body>
    </xsl:template>
</xsl:stylesheet>
```

上述代码使用了 XSLT 文件对 XML 文件进行样式控制，首先需要声明 XSLT 文件，因为 XSLT 文件同样是基于 XML 的，所以在文件头部必须进行声明，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
```

同样 XSLT 也需要进行声明，示例代码如下所示。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl"
>
```

声明 XSLT 文件后，则可以编写 XSLT 文件的输出属性，创建时默认为 XML，如果需要通过 XSLT 文件进行 XML 样式控制，则需要更改为 HTML，示例代码如下所示。

```
    <xsl:output method="html" indent="yes"/>
```

编写 XSLT 文件的输出属性后，就可以编写 XSL 的模板，模板可以自定义标签也可以使用 HTML 标签。在编写模板时，需要指定模板规则的作用点，通过配置 match 属性可以配置模板规则的作用点，示例代码如下所示。

```
    <xsl:template match="/Root/ShopInformation">
```

从 XML 文件可以看出，根节点为 Root，根节点 Root 下有一个 ShopInformation 节点，为了显示 Shop 节点的数据，则需要在模板规则的作用点的 match 属性设置路径。模板中，在需要呈现 XML 文档中相应节点的值可以使用<xsl:value-of>元素进行呈现，<xsl:value-of>元素将拷贝 XML 文档中相应的节点的值到该元素，并替换后呈现给浏览器，示例代码如下所示。

```
                <xsl:value-of select="Shop"/>
```

上述代码首先会通过模板路径找到相应节点，在这里使用 select 属性声明所要找到的节点名称，如<xsl:value-of select="Shop"/>，找到 Shop 节点后会将 Shop 节点的值替换<xsl:value-of select="Shop"/>，呈现给浏览器。在 XML 文档中，需要声明外部 XSLT 文件才能在访问 XML 页面时正确的解释标签，示例代码如下所示。

```
<?xml-stylesheet type="text/xsl" href="XSLTFile1.xslt"?>
```

直接在浏览器中浏览 XSLT 文件，则可以看到 XSLT 的结构树，如图 14-7 所示。XSLT 文件制作了 XML 页面呈现时所需要的样式，从另一个角度来说，当用户在 XML 页面中定义了标签后，浏览器并不能解释这个标签，而可以通过 XSLT 文件告知浏览器如何解释自定义标签并呈现到页面，XML 文件在

浏览器中运行结果如图 14-8 所示。

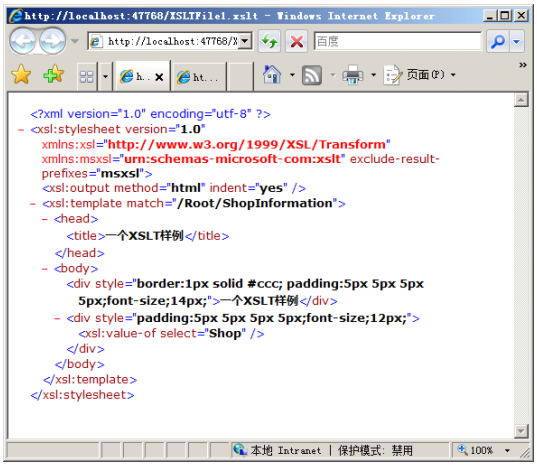

图 14-7　XSLT 结构树

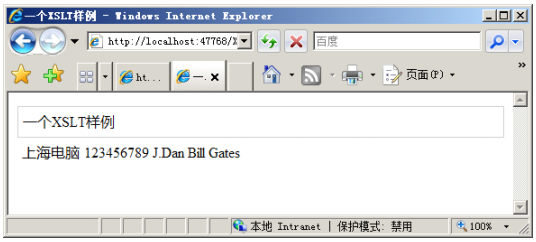

图 14-8　XML 文件格式化输出

注意：IE 5.0 以下版本的浏览器很可能无法浏览结构树，如果需要在浏览器中直接浏览 XSLT 文件或 XML 文件，需要 IE 5.0 以上版本。

14.5　Web 服务（Web Service）

Web Service 是 Web 服务器上的一些组件，客户端应用程序可通过 Web 发出 HTTP 请求来调用这些服务。通过 ASP.NET 开发人员可以创建自定义的 Web Service 或使用内置的应用程序服务，并从任何客户端应用程序调用这些服务。

14.5.1　什么是 Web 服务

Web 服务（Web Service）可以被看作是服务器上的一个应用单元，它通过标准的 XML 数据格式和通用的 Web 协议为其他应用程序提供信息。Web Service 为其他应用程序提供接口从而能够实现特定的任务，其他应用程序可以使用 Web Service 提供的接口实现信息交换。

Web Service 的设计是为了解决不同平台，不同语言的技术层的差异，使用 Web Service 无论使用何种平台，何种语言都能够使用 Web Service 提供的接口，各种不同平台的应用程序也可以通过 Web Service 进行信息交互。

例如，当 Web 应用程序需要制作登录操作时，可以在 Web 页面进行登录操作设计，当 Web 应用逐渐壮大，当 Web 应用的某些应用可以发布到用户的操作系统时，就可以编写相应的应用程序来进行操作，如使用 QQ 类型的软件进行网站登录。但是这样做无疑产生了安全隐患，如果将服务器的用户名和地址等代码发布到本地，这样一些非法人员很可能能够通过反编译获取软件的信息，从而进行用户信息的盗取，而使用 Web Service，本地应用程序可以调用 Web 应用中相应的方法来实现本地登录功能，而这些方法是存在于 Web Service 中。Web Service 还具有以下特性：

- 实现了松耦合：应用程序与 Web Service 执行交互前，应用程序与 Web Service 之间的连接是断开的，当应用程序需要调用 Web Service 的方法时，应用程序与 Web Service 之间建立了连接，当应用程序实现了相应的功能后，应用程序与 Web Service 之间的连接断开。应用程序与 Web

应用之间的连接是动态建立的，实现了系统的松耦合。

- 跨平台性：Web Service 是基于 XML 格式并切基于通用的 Web 协议而存在的，对于不同的平台，只要能够支持编写和解释 XML 格式文件就能够实现不同平台之间应用程序的相互通信。
- 语言无关性：无论是用何种语言实现 Web Service，因为 Web Service 基于 XML 格式，只要该语言最后对于对象的表现形式和描述是基于 XML 的，不同的语言之间也可以共享信息。
- 描述性：Web Service 使用 WSDL 作为自身的描述语言，WSDL 具有解释服务的功能，WSDL 还能够帮助其他应用程序访问 Web Service。
- 可发现性：应用程序可以通过 Web Service 提供的注册中心查找和定位所需的 Web Service。

Web Service 也是使用和制作分布式所需的条件，使用 Web Service 能够让不同的应用程序之间进行交互操作，这样极大的简化了开发人员的平台的移植难度。

14.5.2 Web 服务体系结构

要讲到 Web Service 体系结构就不得不提到 SOA，SOA（Serveice-Oriented Architecture，面向服务的体系结构）是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。

在 SOA 中，接口采用中立的方式定义，接口只声明开发人员如何继承和实现该接口，接口的声明应该是中立的、不依赖于平台、语言而实现的。接口相当于如何规定开发人员规范的进行 Web Service 中功能的实现。SOA 具有以下特点。

- SOA 服务具有平台独立的自我描述 XML 文档。Web 服务描述语言（WSDL， Web Services Description Language）是用于描述服务的标准语言。
- SOA 服务用消息进行通信，该消息通常使用 XML Schema 来定义（也叫做 XSD， XML Schema Definition）。

Web Service 体系结构则采用了 SOA 模型，Web Service 模型包含三个角色，这三个角色包括服务提供者、服务请求者和服务注册中心，如图 14-9 所示。

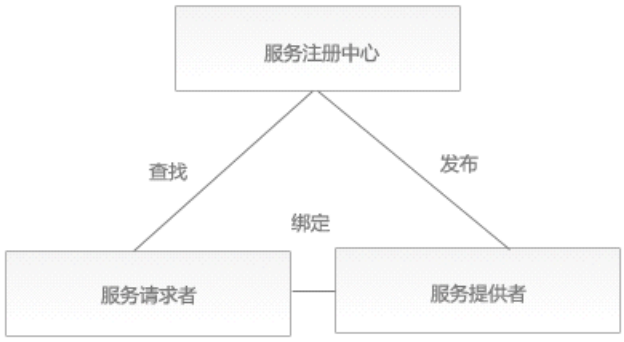

图 14-9 Web Service 体系结构

其中，服务提供者也可以称为服务的拥有者，它通过提供服务接口使 Web Service 在网络上是可用的。服务接口是可以被其他应用程序访问和使用的软件组件，如果服务提供者创建了服务接口，服务提供者会向服务注册中心发布服务，以注册服务描述。相对于 Web Service 而言，服务提供者可以看作访问服务的托管平台。

服务请求者也称为 Web Service 的使用者，服务请求者可以通过服务注册中心查找服务提供者，当

请求者通过服务器中心查找到提供者之后，就会绑定到服务接口上，与服务提供者进行通信。相对于 Web Service 而言，服务请求者是寻找和调用提供者提供的接口的应用程序。

服务注册中心提供请求者和提供者进行信息通信，当服务提供者提供服务接口后，服务注册中心则会接受提供者发出的请求，从而注册提供者。而服务请求者对注册中心进行服务请求后，注册中心能够查找到提供者并绑定到请求者。

14.5.3 Web 服务协议栈

在 Web Service 体系结构中，为了保证体系结构中的每个角色都能够正确和执行 Web Service 体系结构中的发布、查找和绑定操作，Web Service 体系必须为每一层标准技术提供 Web Service 协议栈。Web Service 协议栈如图 14-10 所示。

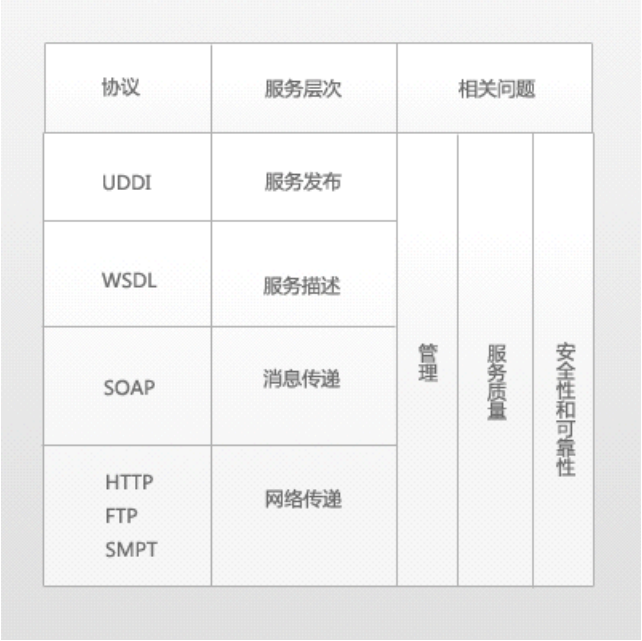

图 14-10　Web Service 协议栈示意图

在 Web Service 协议栈中，最为底层的是网络传输层，Web 服务协议是 Web Service 协议栈的基础。用户需要通过 Web 服务协议来调用服务接口。网络传输层可以使用多种协议，包括 HTTP、FTP 以及 SMTP。

在网络传输层上一层的则是消息传递层，消息传递层使用 SOAP 作为消息传递协议，以实现服务提供者，服务注册中心和服务请求者之间进行信息交换。

在消息传底层之上的是服务描述层，服务描述层使用 WSDL 作为消息协议，WSDL 使用 XML 语言来描述网络服务，在前面的章节中也讲到，WSDL 具有自我描述性，它能够提供 Web 服务的一些特定信息。服务描述层包括了 WSDL 文档，这些文档包括功能、接口、结构等定义和描述。

在服务描述层之上的是服务发布层，该层使用 UDDI 协议作为服务的发布/集成协议。UDDI 提供了 Web 服务的注册库，用于存储 Web 服务的描述信息。服务发布层能够为提供者提供发布 Web 服务的功能，也能够为服务请求者提供查询，绑定的功能。

当 Web Service 中触发了事件，如服务提供者发布服务接口、服务请求者请求服务等，服务提供者首先使用 WSDL 描述自己的服务接口，通过使用 UDDI 在服务器发布层向服务注册中心发布服务接口。

服务注册中心则会返回 WSDL 文档。当服务请求者对服务注册中心执行服务请求，请求着通过 WSDL 文档的描述绑定相应的服务接口。

14.6　简单 Web Service 示例

在了解了 Web Service 基本的概念和协议栈的运行过程后，可以使用 Visual Studio 2008 进行 Web Service 应用程序的创建。单击菜单栏上的【文件】选项，在下拉菜单中选择【新建项目】选项，在新建项目窗口中选择【ASP.NET Web 服务应用程序】选项进行相应的应用程序创建，如图 14-11 所示。

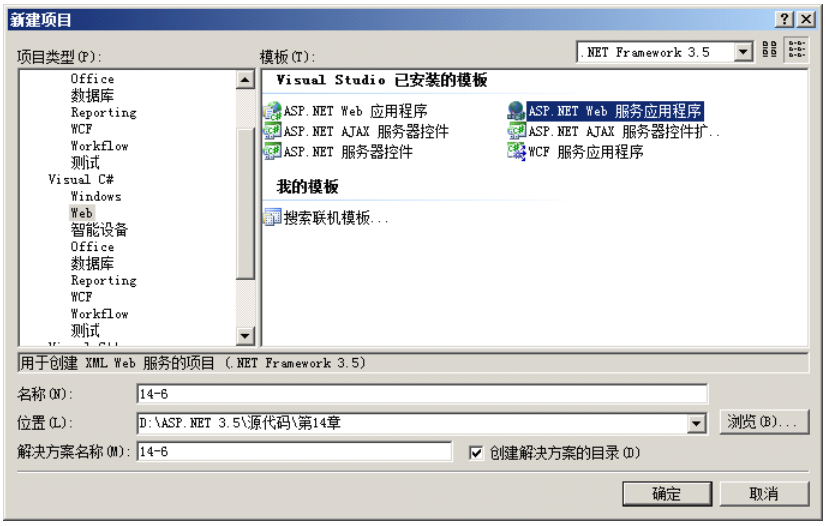

图 14-11　创建 ASP.NET Web 服务应用程序

单击确定，系统则默认创建一个“Hello World”Web Service 应用程序，示例代码如下所示。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Services;                                    //使用 WebServer 命名空间
using System.Web.Services.Protocols;                          //使用 WebServer 协议命名空间
using System.Xml.Linq;
namespace _14_6
{
    /// <summary>
    /// Service1 的摘要说明
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [ToolboxItem(false)]
    // 若要允许使用 ASP.NET AJAX 从脚本中调用此 Web 服务，请取消对下行的注释。
    // [System.Web.Script.Services.ScriptService]
    public class Service1 : System.Web.Services.WebService
    {
```

```
        [WebMethod]                                                    //声明为 Web 方法
        public string HelloWorld()                                     //创建 Web 方法
        {
            return "Hello World";
        }
    }
}
```

在上述代码中，系统引入了默认命名空间，这些空间为 Web Service 应用程序提供基础保障，这些命名空间声明代码如下所示。

```
using System.Web;
using System.Web.Services;                                     //使用 WebServer 命名空间
using System.Web.Services.Protocols;                           //使用 WebServer 协议命名空间
```

运行该 Web Service 应用程序，运行结果如图 14-12 所示。

图 14-12　Web Service 应用程序

在运行 Web Service 应用程序后，Web Service 应用程序将呈现一个页面。该页面显示了 Web Service 应用程序的名称，名称下面列举了 Web Service 应用程序中的方法。当开发人员增加方法时，Web Service 应用程序方法列表则会自动增加。创建 Web Service 应用程序方法代码如下所示。

```
        [WebMethod]                                                    //声明为 Web 方法
        public string PostMyTopic()                                    //创建 Web 方法
        {
            return "Your Topic has been posted";                       //方法返回值
        }
```

保存并运行后，Web Service 应用程序方法列表则会自动增加，如图 14-13 所示。单击该方法，Web Service 应用程序会跳转到另一个页面，该页面提供了方法的调用测试，以及 SOAP 各个版本请求和相应的示例，如图 14-14 所示。

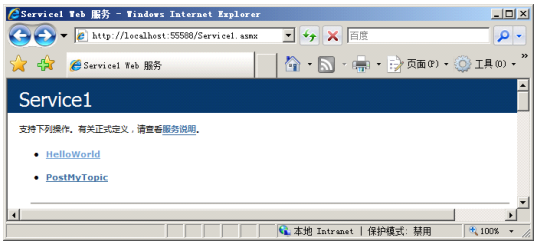

图 14-13　Web Service 应用程序方法列表

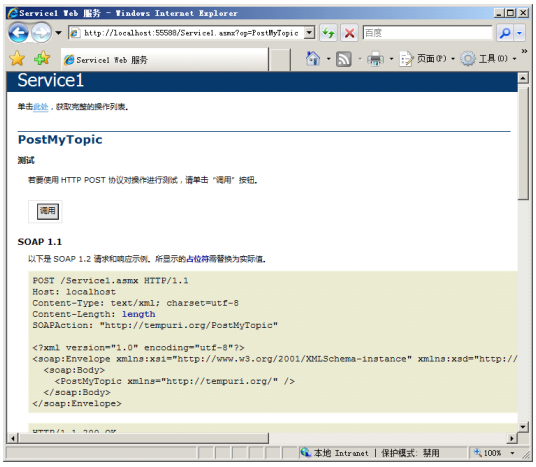

图 14-14　测试方法

单击【调用】按钮，则浏览器会通过 HTTP-POST 协议向 Web 服务递交请求信息，方法被执行完毕后，返回 XML 格式的结果，如图 14-15 所示。

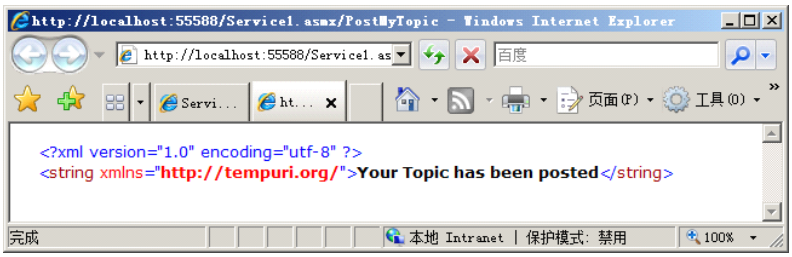

图 14-15　返回结果

14.7　自定义 Web Service

在创建 Web Service 应用程序后，系统会自动创建 Web Service 应用程序并生成相关代码，通过修改自动生成的代码，能够快速创建和自定义 Web Service 应用程序，自定义 Web Service 应用程序能够让不同的应用程序引用 Web Service 提供的框架进行逻辑编程。

14.7.1　创建自定义的 Web Service

通过创建自定义 Web Service 能够进行应用程序开发，Web Service 同样支持带参数传递的方法，并能够在 Web Service 中进行数据查询等操作，保证了代码的安全性。创建一个 Web Service，并编写相应的查询方法，示例代码如下所示。

```
[WebMethod]
public string Search(string title)
{
    try
    {
        SqlConnection
        con = new SqlConnection("server='(local)';database='mytable';uid='sa';pwd='sa'");
```

```
            con.Open();                                                    //打开数据库连接
            string strsql = "select * from mynews where title like '%" + title + "%'";   //查询语句
            SqlDataAdapter da = new SqlDataAdapter(strsql, con);           //创建适配器
            DataSet ds = new DataSet();                                    //创建数据集
            int i = da.Fill(ds, "mytable");                                //填充数据集
            string result = "";                                            //初始化空字符串
            for (int j = 0; j < i; j++)                                    //遍历循环数据集
            {
                result += ds.Tables["mytable"].Rows[j]["title"].ToString() + "\n";//输出结果，生成字符串
            }
            return result;                                                 //返回结果
        }
        catch
        {
            string result = "没有任何结果";
            return result;
        }
    }
```

上述代码通过创建了一个 Web Service 方法进行新闻查询，通过新闻的标题 title 字段查询数据库中索引类似信息，运行 Web Service 应用程序后，其界面如图 14-16 所示。

单击 Search 按钮进行 Web Service 应用程序测试，对于需要传递参数的方法，测试过程中可以输入方法进行测试，如图 14-17 所示。

图 14-16　自定义 Web Service 应用程序

图 14-17　输入参数

在文本框中输入 title，单击【调用】按钮，则会向方法中传递参数并执行方法，执行方法后将会返回 string 类型的值，如图 14-18 所示。

图 14-18　查询结果

在使用 Web Service 应用程序返回数据集时，可以直接返回 DataSet 对象，Web Service 应用程序执

行后将会将 DataSet 对象转换为 XML 格式并返回 XML 格式的执行结果，Search 代码更改如下所示。

```
[WebMethod]
 public DataSet Search(string title)
 {
      SqlConnection con = new SqlConnection("server='(local)';database='mytable';uid='sa';pwd='sa'");
      con.Open();                                                                    //打开数据连接
      string strsql = "select * from mynews where title like '%" + title + "%'";    //生成 SQL 语句
      SqlDataAdapter da = new SqlDataAdapter(strsql, con);                           //创建适配器
      DataSet ds = new DataSet();                                                    //创建数据集
      int i = da.Fill(ds, "mytable");                                                //填充数据集
      return ds;                                                                     //返回数据集
 }
```

上述代码查询后直接返回 ds 记录集，当输入查询字串“t”后，运行结果如图 14-19 所示。

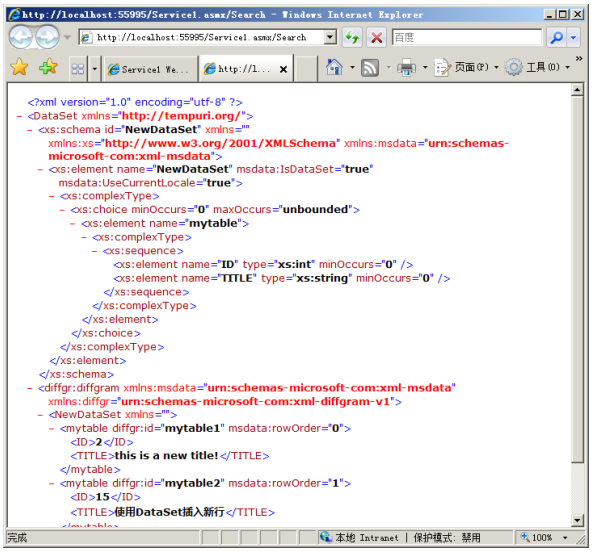

图 14-19　返回 DataSet 记录集

14.7.2　使用自定义的 Web Service

当 Web 应用程序需要使用 Web Service 应用程序并调用其方法时，只需要添加服务引用即可。右击 Web 应用程序，选择【添加服务引用】选项，在弹出的添加服务引用窗口中单击【发现】按钮查找服务，如图 14-20 所示。

选择相应的服务引用后并更改命名空间，再单击【确定】按钮确认添加，则服务引用添加成功，在解决方案管理器中则会出现相应的服务引用，如图 14-21 所示。

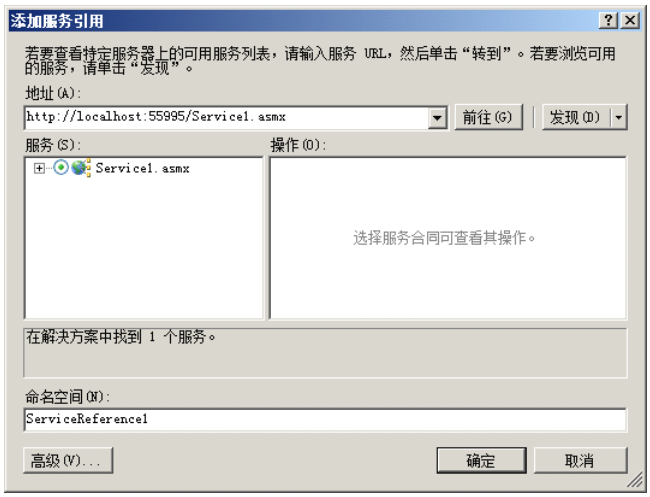

图 14-20　添加服务引用

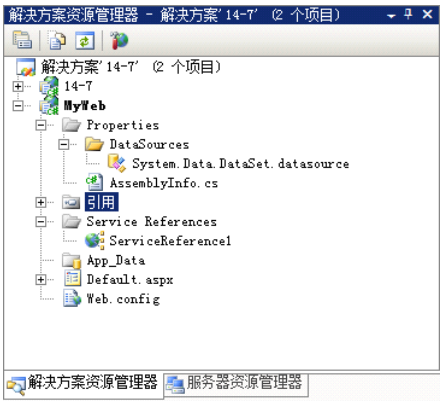

图 14-21　服务引用添加完成

添加了服务引用之后，可以通过 Web 窗体使用和调用 Web Service 应用程序中的方法，Web 窗体中 HTML 代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <div>
        Search :
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="开始搜索" onclick="Button1_Click"/>
        <br />
        <asp:GridView ID="GridView1" runat="server">
        </asp:GridView>
    </div>
    </form>
</body>
```

创建了 Web 应用后则需要为按钮单击事件编写代码，示例代码如下所示。

```
        protected void Button1_Click(object sender, EventArgs e)
        {
            ServiceReference1.Service1SoapClient
            cs = new MyWeb.ServiceReference1.Service1SoapClient()        ;//使用服务引用
            DataSet ds = new DataSet();                                  //创建数据集
            ds=cs.Search(TextBox1.Text);                                 //执行 Web Service 方法
            GridView1.DataSource = ds.Tables[0].DefaultView              ;//绑定控件
            GridView1.DataBind();                                        //填充数据
        }
```

上述代码使用了服务引用，通过服务引用进行方法的实现，运行后如图 14-22 和图 14-23 所示。

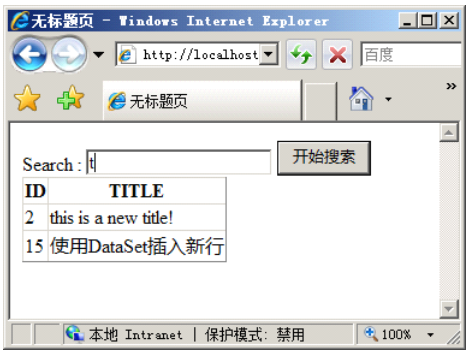

图 14-22　使用服务引用

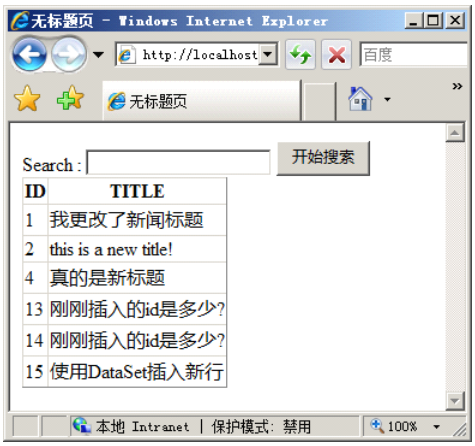

图 14-23　执行搜索

通过使用 Web Service 应用程序，从本地代码来看，隐藏了对数据的连接字串和查询操作代码，从应用程序角度来看，Web Service 应用程序保证了应用程序的封闭性和安全性。

14.8　小结

本章讲解了 XML 文件基础，以及 Web Service 基础，XML 作为.NET 平台下微软强推的一种标记语言技术，其作用是不言而喻的。在 SQL Server 以及微软的其他应用软件中，也能够经常看到 XML 的影子，并且 SQL Server 2005 已经开始尝试支持 XML 数据类型，这说明 XML 在当今世界中的运用越来越广阔，也说明在未来的应用中，XML 技术包含着广大的前景。通过讲解 Web Service 的基本概念，包括什么是 Web Service，以及 Web Service 协议栈。同时，本章还包括：

- 创建 XML 文档：包括如何创建 XML 文档。
- XML 控件：演示了如何使用 XML 控件呈现 XML 数据。
- XmlTextReader 类：讲解了 XmlTextReader 类操作 XML 文档。
- XmlTextWriter 类：讲解了 XmlTextWriter 类操作 XML 文档。
- XmlDocument 类：讲解了 XmlDocument 类操作 XML 文档。
- XML 串行化：讲解了 XML 串行化基本功能。
- XSL 简介：介绍了 XSL 与 HTML 的异同。
- 使用 XSLT：介绍了 XSLT 语法并通过 XSLT 控制 XML 样式。
- 什么是 Web Service：讲解了 Web Service 基本概念。
- Web Service 协议栈：讲解了 Web Service 协议栈的基本概念，以及 Web Service 是如何运作的。

本章还简单的讲解了 XML 串行化功能和通过程序创建 Web Service 示例，虽然串行化和 Web Service 只做了基本的讲解，但是熟练的掌握这些基础能够为今后的分布式开发打下良好的基础。