第七篇 高级应用

第一章 XML 及其应用

XML 是标准扩展语言的简称，是未来 Web 编程的标准。在这一章中，我们将讲述 XML 在 ASP.NET 中的应用

7.1.1 制作广告条

在这个程序中，我们通过 XML 语言实现每次访问网页时，将显示不同的广告条。在本例中，我们只调用了两条广告

源文件：**advanceapp\intro.aspx**
Intro.aspx 的代码如下：

```
<html>
<center><title>广告条演示</title></center>
    <head>
        <link rel="stylesheet"href="intro.css">
    </head>
 <body>
        <center>
          <form action="intro.aspx" method="post" runat="server">
<h2>广告条演示</h2>
       <asp:adrotator AdvertisementFile="intro.xml" BorderColor="black" BorderWidth=1
runat="server"/>
      </form>
          </center>
    </body>
</html>
```

intro.xml 的 代码如下：

```
<Advertisements>
    <Ad>
        <ImageUrl>./hp1.gif</ImageUrl>
        <NavigateUrl>http://www.yesky.com</NavigateUrl>
        <AlternateText>欢迎访问！</AlternateText>
        <Keyword>Computers</Keyword>
        <Impressions>80</Impressions>
    </Ad>
```

```
<Ad>
    <ImageUrl>./hp2.gif</ImageUrl>
    <NavigateUrl>http://www.yesky.com</NavigateUrl>
    <AlternateText>欢迎访问</AlternateText>
    <Keyword>Computers</Keyword>
    <Impressions>80</Impressions>
</Ad>
</Advertisements>
```

在 intro.aspx 中，我们使用了<asp:adrotator AdvertisementFile="intro.xml" BorderColor="black" BorderWidth=1 runat="server"/>这条语句来嵌入 intro.xml 文件。运行效果如图：

当我们点刷新按狃或者按 F5 键，我们将看到另一条广告条。

在本例中，我们用到了 AdRotator 服务器控件，在 xml 文件中，我们我们可以定义其属性：如表所示：

属性	描述
ImageUrl	图象文件的绝对或相对地址
NavigateUrl	当图象被点击时，可访问相应的网页
AlternateText	当鼠标移动到图片上时，将显示提示信息
Keyword	指定广告条的分类，我们可以利用此属性来对广告条进行分类
Impressions	指定图片在表格中的大小

7.1.2 XML 和 dataset 控件结合使用

数据访问是一个应用系统的核心。公用语言运行环境（Common Language Runtime）提供了管理数据访问应用程序接口（API）的方法。而这些 API 将不论它的的数据源是什么，如 SQL Server, OLEDB, XML 等，都能提取出我们所需要的数据。我们在具体编程的时候，有 3 个对象将常常用到：Connections, Commands, and DataSets。

对象	描述
Connection	与数据源进行连接。如 SQL Server 或者一个 XML 文件。
Command	对数据进行操纵。如进行删除（delete），提取（select）,更新（update）
Dataset	显示出所需的数据

为了使我们能使用 SQL 语句，我们要先导入 System.Data 和 System.Data.SQL 这两个名字空间。

语句如下：

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SQL" %>
```

下面的表格是对几个典型的 SQL 语句的说明：

SQL 语句	示例
Select（对单个表的操作）	SELECT * from Student WHERE stuname = ‘小李’；
Select（对多个表的操作）	SELECT * from Student S, Dept D WHERE S.dept= D.dept；
Insert	INSERT into Student VALUES ('小王'，21，’男’)；
Delete	DELETE from Student WHERE name=’小王’;
Update	UPDATE Student SET age = 21 WHERE name=’小李’;

在执行查询之前，我们要先构件一个 SQLDataSetCommand 对象，在执行查询以后，我们需要把数据转移到 DataSet 中，我们可以使用下面的代码：

我们假设有一个 test 数据库，在这个数据库中有一个 student 表。

```
Dim myConnection As New SQLConnection("server=localhost;uid=sa;pwd=;database=test")
Dim myCommand As New SQLDataSetCommand("select * from student", myConnection)
Dim ds As New DataSet()
myCommand.FillDataSet(ds, "student")
```

可能读者不禁要问：为什么要用 XML 文件存储数据吗?为什么不使用数据库？

这是因为：对很多目的用途来说，用数据库太过浪费了。.要使用一个数据库，你必须安装和支持一个分离的服务器处理进程（a separate server process）,它常要求有安装和支持它的管理员（administrator）。你必须学习 SQL 语句, 并用 SQL 语句写查询，然后转换数据，再返回。而如果你用 XML 文件存储数据，将可减少额外的服务器的负荷。还有，你还找到了一个编辑数据的简单方法。你只要使用文本编辑器,而不必使用复杂的数据库工具。XML 文件很容易备份，和朋友共享，或下载到你的客户端。同样的，你可以方便地通过 ftp 上载新的数据到你的站点。

XML 还有一个更抽象的优点，即作为层次型的格式比关系型的更好。 它可以用一种很直接的方式来设计数据结构来符合你的需要。你不需要使用一个实体-关系编辑器，也不需要使你的图表（schema）标准化。 如果你有一个元素（element）包含了另一个元素，你可以直接在格式中表示它，而不需要使用表的关联。

注意，在很多应用中，依靠文件系统是不够充分的。如果更新很多，文件系统会因为同时写入而受到破坏。数据库则通常支持事务处理，可以应付所发生的请求而不至于损坏。对于复杂的查询统计要有反复、及时的更新，此时数据库表现都很优秀。当然，关系型数据库还有很多优点，包括丰富的查询语言，图表化工具，可伸缩性，存取控制等等。

在下面这样的案例中，正如大多数中小规模的、基于发布信息的站点一样，你可能涉及的大多数数据存取都是读，而不是写，数据虽然可能很大，但相对来说并没有经常的更新变化，你也不需要做很复杂的查询，即使你需要做，也将用一个独立的查询工具，那么成熟的 rdbms 的优点消失了，而面向对象型的数据模型的优点则可以得到体现。

最后,为你的数据库提供一个查询器外壳来进行 SQL 查询并将他们转化进入 xml stream 也是完全有可能的。

所以你可以选择这二种方式之一。XML 正变成一种非常健壮的，便于编程的工具，作为某个成熟的数据库的前端工具来进行存储和查询。(oracle 的 xsql servlet 即是这种技术的一个很好的例子。)

7.1.3 XML 语法

现在我们来介绍一下 XML Schemas：

随着 XML Schema 规范的逐渐普及，它成为 W3C 推荐标准的那一天已经不远了。我们来看一下这个新的定义文档类型的方法。

一、定义元素

文档类型定义的工作现在大都是由 DTDs 来完成的,现在的 DTD 在功能上有一些限制，随着 XML 应用的越来越广泛，这些限制逐渐阻碍了 XML 的发展。

DTD 的语法不同于 XML 的语法，因而需要文档编写者另外的学习一套符号语言。而软件也需要另外的一个解析器来对 DTD 文件进行解析。

在 DTD 中没有办法来定义能够从程序语言变量直接映射到 XML 数据的数据类型和数据格式。 没有一组被人所熟知的基本的元素以供文档编写者选择。DTDs 是 XML 从其前辈 SGML 那儿继承过来的（事实上，XML 本身就是一种 SGML 的简化版本），这个相对比较成熟的技术能够很快的让 XML 运行起来，并且能够让熟悉 SGML 的人对 XML 有更多的感觉。然而，很快的人们就发现，XML 需要一种更具表达能力的解决方案，而不仅仅是 DTD。

定义元素需要指定元素的名称、元素的内容模式、元素的属性、以及内嵌的子元素。在 XML Schemas 中，元素的内容模式是通过类型来定义的。一个服从于某个特定的 schema 的 XML 文档只能按照 schema 中定义的元素模式来编写，这同 DTD 的规则是一样的。元素可以是简单类型的，也可以使复杂类型的。在 Schemas 规范中定义了很多的简单类型，例如： string，integer 和 decimal。简单类型的元素不能在包含子元素和属性，而复杂类型的元素则能够嵌套子元素，并能够包含有属性。

我们来看看一个简单的定义元素的例子：

<element name="quantity" type="positive-integer"/>

<element name="amount" type="decimal"/>

我们知道在面向对象的观点中，有聚集和继承的概念，可以在已有的类中衍生出新类。在这儿 Schema 借用了这些观点，用户也可以通过聚集和继承来在老元素的基础上定义新的元素。聚集能够把一组已存在的元素组合成一个新的元素。继承则通过扩展已存在的元素来定义一个新的元素，并且这个新的元素能够代表被继承的那个元素。

比如，如果我们要从 decimal 类型中派生一个新的 value 元素，并让它有这样的内容模式：<value unit="Celsius">42</value>，我们可以这样的定义：

<element name="value">
<complexType base=&single;decimal&single; derivedBy=&single;extension&single;>
<attribute name=&single;unit&single; type=&single;string&single;/>
</complexType>
</element>

下面我们把 time 和 value 元素聚集到一个 measurement 元素中，形成这样的内容模式：

<measurement>
<time>2000-10-08 12:00:00 GMT<time/>
<value unit="Celsius">42</value>

那么最终我们的 schema 元素定义结果就应该是这样的：

<element name=&single;measurement&single; type=&single;measurement&single;/>
<complexType name=&single;measurement&single;>
<element name=&single;time&single; type=&single;time&single;/>
<element name=&single;value&single; type=&single;value&single;/>
</complexType>

和上面的 schema 等同 DTD 是：

<!ELEMENT measurement (time, value)>

```
<!ELEMENT time (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST value (unit)>
```

显然，它所能表达的意思就少了很多。

从 Java 等面向对象语言中引入的继承的特性，还包括了可以定义抽象元素来迫使实现子类元素，或者定义一个终结元素来禁止元素再被其它元素所继承。这使得从元素到 Java 或者 C++语言的类的一一映射变得更为直观可行。

二、基数表达

XML Schema 比起 DTD 来能够更方便的表达元素基数的概念。所谓基数是指一个元素在文档中出现的次数。在 DTD 中，使用的是正则表达式来表示基数的，而正则表达式的表达能力在所有的形式化文法中是最低的，这使得你在定义文档的时候会遇到很多的困难。你只能够指定一次(1)，零次或者更多 (*)，一次或者更多 (+) 这三种基数，并用这些基数序列来构成 DTD。XML Schema 中则是使用 minOccurs 和 maxOccurs 这两个属性来定义元素基数，分别用来指定元素出现的最少次数和最多次数：

```
<element ref="optionalElement" minOccurs="0"/>
<element ref="twoOrMoreElements" minOccurs="2" maxOccurs="unbounded"/>
<element ref="exactlyOneElement" />
```

minOccurs 和 maxOccurs 的缺省值都是 1，也就是说，当没有指定这两个属性的时候，元素只允许在文档中出现一次。除了这两个属性，你还可以用 choice 和 all 这两个元素。元素 choice 只允许它的所有子元素中的一个出现在文档中。而元素 all 则最为宽松，能够让其所有的子元素在文档中以任意的顺序出现任意的次数。而这些概念在 DTD 中都是难以表达的。

```
<xsd:choice>
<element ref="EitherThis"/>
<element ref="OrThat"/>
</xsd:choice>
<xsd:all>
<element ref="Ying"/>
<element ref="Yang"/>
</xsd:all/>
```

三、名域

XML Schema 中还支持名域。一个 Schema 除了可以定义 XML 文档词汇表外，还可以通过名域来定义目标名域，和其它可能会使用到的词汇名域。例如：

```
<xsd:schema targetNamespace=&single;http://www.physics.com/measurements&single;
xmlns:xsd=&single;http://www.w3.org/1999/XMLSchema&single; xmlns:units=
&single;http://www.physics.com/units&single;>
```

```
<xsd:element name=&single;units&single; type=&single;units:Units&single;/>
<xsd:element name=&single;measurement&single; type=&single;measurement&single;/>
<complexType name=&single;measurement&single;>
<element name=&single;time&single; type=&single;time&single;/>
<element name=&single;value&single; type=&single;value&single;/>
</complexType>
```

这种机制为建立复杂的名域体系提供了一种模块化而又易于扩展的方法。使得名域的作用能够真正的被体现出来。

XML Schema 提供了一个丰富而更具弹性的机制来定义 XML 文档词汇表。它使用 XML 语言本身来定义关于一个 XML 文档的元信息（meta-information），这使得 XML 的协同工作能力大大的增强了。而许多对于 DTD 功能上的改进和增强，也使得它最终必定会终结 DTD，作为 XML 的一个标准出现。在 Apache Project 和 IBM alphaworks 的主页上现在已经有很多的 Schema 的工具出现了，你如果有兴趣的话，不妨去看看。

好了，有上面的介绍，我们来看如何从 xml 文件中读取数据。DataSet 控件提供了 ReadXml 方法。同时在 XML 文件中，必须存在 schema 和我们所需要的数据（Data）。
好了，有了上面的介绍，我们先看一下 data1.aspx 文件：
源文件：**advanceapp\data1.aspx**

```
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data" %>
<html>
<head>
  <title>
    XML 演示
  </title>
</head>
<script language="VB" runat="server">
      Sub Page_Load(Src As Object, E As EventArgs)
            Dim DS As New DataSet
            Dim FS As FileStream
            Dim Reader As StreamReader
            FS = New FileStream(Server.MapPath("data1.xml"),FileMode.Open,FileAccess.Read)
            Reader = New StreamReader(FS)
            DS.ReadXml(Reader)
            FS.Close()
            Dim Source As DataView
            Source = new DataView(ds.Tables(0))
            MySpan.InnerHtml = Source.Table.TableName
            MyDataGrid.DataSource = Source
            MyDataGrid.DataBind()
      End Sub
</script>
<body>
```

```
<center>
  <h3><font face="Verdana">XML 演示 <span runat="server" id="MySpan"/></font></h3>
  <ASP:DataGrid id="MyDataGrid" runat="server"/>
</center>
</body>
</html>
```

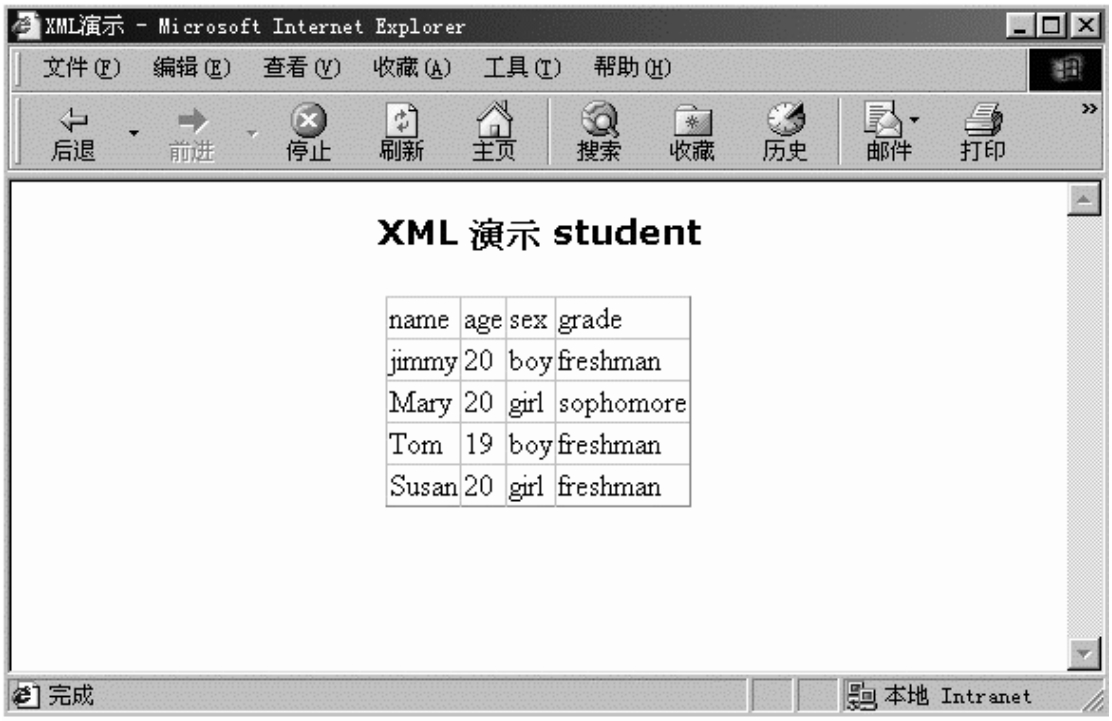
我们再来看看 xml 文件（**advanceapp\data1.aspx**）的内容：

data1.xml 的文件如下：

```
<center>
<root>
<schema id="DocumentElement" targetNamespace="" xmlns="http://www.w3.org/1999/XMLSchema"
xmlns:xdo="urn:schemas-microsoft-com:xml-xdo" xdo:DataSetName="DocumentElement">
    <element name="student">
        <complexType content="elementOnly">
            <all>
                <element name="name" type="string"></element>
                <element name="age"  type="int"></element>
                <element name="sex"  type="string"></element>
                <element name="grade" type="string"></element>
             </all>
        </complexType>
    </element>
</schema>
<DocumentElement>
    <student>
        <name>jimmy</name>
        <age>20</age>
        <sex>boy</sex>
        <grade>freshman</grade>
    </student>
    <student>
        <name>Mary</name>
        <age>20</age>
        <sex>girl</sex>
        <grade>sophomore</grade>
            </student>
    <student>
        <name>Tom</name>
        <age>19</age>
        <sex>boy</sex>
        <grade>freshman</grade>
    </student>
```

```
    <student>
        <name>Susan</name>
        <age>20</age>
        <sex>girl</sex>
        <grade>freshman</grade>
    </student>
</DocumentElement>
</root>
</center>
```

程序运行后演示如下：



当然我们也可以把 schema 和数据分开成对立的文件。在主文件中，我们要分别使用 the ReadXmlData 和 ReadXmlSchema 方法。

如：

读取 schema.xml 的内容：

```
FS = New FileStream(Server.MapPath("schema.xml"),FileMode.Open,FileAccess.Read)
Schema = new StreamReader(FS)
DS.ReadXmlSchema(Schema)
FS.Close()
```

读取 data.xml 的内容：

```
FS = New FileStream(Server.MapPath("data.xml"),FileMode.Open,FileAccess.Read)
Reader = New StreamReader(FS)
DS.ReadXmlData(Reader)
FS.Close()
```

我们上面提到的 data1.xml 分成两部分：

第一部分：生成 schema.Xml 文件：

```
<schema id="DocumentElement" targetNamespace="" xmlns="http://www.w3.org/1999/XMLSchema" xmlns:xdo="urn:schemas-microsoft-com:xml-xdo" xdo:DataSetName="DocumentElement">
    <element name="student">
        <complexType content="elementOnly">
            <all>
                <element name="name" type="string"></element>
                <element name="age"  type="int"></element>
                <element name="sex"  type="string"></element>
                <element name="grade" type="string"></element>

            </all>
        </complexType>
    </element>
</schema>
```

第二部分生成 data.xml：

```
<DocumentElement>
    <student>
        <name>jimmy</name>
        <age>20</age>
        <sex>boy</sex>
        <grade>freshman</grade>
    </student>
    <student>
        <name>Mary</name>
        <age>20</age>
        <sex>girl</sex>
        <grade>sophomore</grade>
            </student>
    <student>
        <name>Tom</name>
        <age>19</age>
        <sex>boy</sex>
        <grade>freshman</grade>
    </student>
    <student>
        <name>Susan</name>
        <age>20</age>
        <sex>girl</sex>
        <grade>freshman</grade>
    </student>
</DocumentElement>
```

7.1.4 小结

通过上面的介绍，使我们对 XML 在 ASP.NET 中有了一定的认识。在下面的章节中，我们将对 XML 进行更深入的讲解。

第二章 三层结构及其应用

7.2.1 概念及环境

ASP.NET 中的三层结果开发方法，其实其思想跟 Java 的一样。Java 中的三层架构为前端的 html、Jsp、Servlet，中间层为 JavaBean、EJB，后面为数据库服务器。而在 ASP.NET 中，前段为 html、asp、aspx 等，中间层为有.vb、.cs 等文件编译而成的.dll 控件，后面为数据库服务器。

在我们的三层架构中，我们的数据库层通过中间层来连接以及操作，前端给中间层传递参数，并接受中间层的参数。在我们的 ASP.NET 中，我们主要关注的是我们的中间层与前端的数据交互。

我们一般统称中间层为组件，组件可以用.vb 编译而成，也可以用.cs 文件编译而成。中间层一般为.dll 文件。微软的.NET 技术在这个方面比他的以前的任何版本都要来的简单，这也是它的一打好处之一。以前我们要注册一个.dll 文件，有是注册有是重启动，而在.net 上，我们的.dll 文件拿来就用，不用再考虑注册的问题。

在没有 Visual stutio.net 之前，我们用写成的.bat 文件来把.vb 和.cs 文件编译成.dll 文件，在.bat 文件里，我们写入编译的文件名称、相关联的名字空间、要编译成的文件名以及对应的命令名称，然后运行就行了。听起来很复杂，这也是很多初学者在编译第一个.dll 文件时所害怕的事情。但是做起来很简单的。下面我们举一个例子来说明.bat 文件的写法，假设我们有一个文件名为：saidy.vb 的文件，我们要把它编译成 saidy.dll 的文件，其中用到 System、System.Data、System.Data.SQL 名字空间，我们可以创建一个 dblink.bat 文件，内容如下：

```
vbc  /out:..\bin\saidy.dll  /t:library  /r:system.dll /r:system.data.dll  / r:system.data.sql.dll
     dblink.vb
```

这是编译.vb 程序的命令，如果是编译.cs 文件，则命令会是不一样，我们假定有一个 saidy.cs 的文件，按照上面的要求，我们编译如下：

```
cs  /out:..\bin\saidy.dll  /t:library  /r:system.dll /r:system.data.dll  / r:system.data.sql.dll
    dblink.cs
```

我们可以看出来，大部分是一样的。

当然，如果我们有微软公司的 vs.net 编程环境，则我们不用这么麻烦，我们可以象编译 vb 或者 vc 程序一样方便的编译.dll 文件。微软公司的 vs.net 是一个集大成者，把各种语言整合起来，在这个环境下都可以写出不同语言的程序。具体的应用我们会在专门的章节上介绍的。

7.2.2 一个基于三层架构的例子

我们通过具体的例子来说明三层架构的应用，我们建一个小项目来说明这个问题。有时为了安全性，我们通常把与数据库的连接用一个动态连接库文件封装起来，这样我们就要把写数据库连接的.vb 或者.cs 文件编译成动态连接库.dll 文件。甚至我们把对数据库的相关操作页编译成.dll 文件。

下面是我们的与数据库连接以及操作的文件 dblink.vb 的主要部分，对数据库的连接：

Dim dbl As SQLConnection

对数据库的操作，我们把它写在一个方法里面，在返回相应值：

```
Function getdata() as DataView
    Dim sComm as SQLDataSetCommand
    Dim sDS as DataSet
    Dim sStr as String
    dbl = New SQLConnection("server=localhost;uid=sa;password=;database=howff")
    sStr = "select * from color"
    sComm = new SQLDataSetCommand(sStr,dbl)
    sDS = new DataSet()
    sComm.FillDataSet(sDS,"color")
    Return sDS.Table["color"].DefaultView
End Function
```

我们第六个语句就用到上面的与数据库的连接变量，我们这个函数的功能是从表“ color “ 中选出所有的元素，并返回表结构的形式。完整的代码如下：

```
Imports System
Imports System.Data
Imports System.Data.SQL
'创建名字空间
Namespace db
'创建一个类
Public Class dblink
'建立数据库的连接
Dim dbl As SQLConnection
'方法
  Public Function getdata() As DataView
    Dim sComm As SQLDataSetCommand
    Dim sDS As DataSet
    dbl = New SQLConnection("server=localhost;uid=sa;password=;database=howff")
    Dim sStr As String
    sStr = "select * from color"
    sComm = New SQLDataSetCommand(sStr, dbl)
    '填充数据
```

```
            sDS = New DataSet()
            sComm.FillDataSet(sDS, "color")
            '返回
            Return sDS.Tables("color").DefaultView
        End Function
    End Class
End Namespace
```

我们再写一个前端掉用页面 saidy.aspx，我们首先要引入我们创建的名字空间：

```
<%@ Import Namespace="db" %>
```

在页面装入的时候，我们用此方法：

```
Sub Page_Load(Sender As Object, E As EventArgs)
    '建立一个新的对象
    Dim newdb As dblink
    newdb = new dblink()
    '数据来源
    Products.DataSource = newdb.getdata()
    '数据绑定
    Products.DataBind()
End Sub
```

下面看看我们完整的代码**(advanceapp\dblink.aspx)**：

```
<%@ Import Namespace="db" %>
<html>
<script language="VB" runat="server">
  Sub Page_Load(Sender As Object, E As EventArgs)
    '建立一个新的对象
    Dim newdb As dblink
    newdb = new dblink()
    '数据来源
    Products.DataSource = newdb.getdata()
    '数据绑定
    Products.DataBind()
  End Sub
</script>
<body style="font: 10pt verdana" bgcolor="CCCCFF">
<BR><BR><BR>
<CENTER>
  <h3>.NET->三层架构！</h3>
</CENTER>
<BR><BR>
<CENTER>
  <ASP:DataList id="Products" ShowHeader=false ShowFooter=false RepeatColumns="2"
RepeatDirection="horizontal" BorderWidth=0 runat="server">
```

```
      <template name="itemtemplate">
        <table>
          <tr>
            <td width="150" style="text-align:center; font-size:8pt; vertical-align:top;
                height:50">
              <p>
              <%# DataBinder.Eval(Container.DataItem, "id") %> <br>
              <%# DataBinder.Eval(Container.DataItem, "name", "{0:C}").ToString() %>
            </td>
          </tr>
        </table>
      </template>
    </ASP:DataList>
</CENTER>
</body>
</html>
```

我们看到，在这个页面当中，没有出现与数据库交互的语句，这样我们就很好的把数据操作封装起来了，我们的运行效果如下：

7.2.4 小结

在本章中，我们讲解一个基于三层架构的例子，这只是一个非常简单的例子，我们知道.NET 在这方面的功能是非常强大的，你可以用它来写非常复杂的组件。

第三章 使用 MSMQ

7.3.1 基本概念

MSMQ(MicroSoft Message Queue，微软消息队列)是在多个不同的应用之间实现相互通信的一种异步传输模式，相互通信的应用可以分布于同一台机器上，也可以分布于相连的网络空间中的任一位置。它的实现原理是：消息的发送者把自己想要发送的信息放入一个容器中（我们称之为 Message），然后把它保存至一个系统公用空间的消息队列(Message Queue)中；本地或者是异地的消息接收程序再从该队列中取出发给它的消息进行处理。

在消息传递机制中，有两个比较重要的概念。一个是消息，一个是队列。消息是由通信的双方所需要传递的信息，它可以是各式各样的媒体，如文本、声音、图象等等。消息最终的理解方式，为消息传递的双方事先商定，这样做的好处是，一是相当于对数据进行了简单的加密，二则采用自己定义的格式可以节省通信的传递量。消息可以含有发送和接收者的标识，这样只有指定的用户才能看到只传递给他的信息和返回是否操作成功的回执。消息也可以含有时间戳，以便于接收方对某些与时间相关的应用进行处理。消息还可以含有到期时间，它表明如果在指定时间内消息还未到达则作废，这主要应用与时间性关联较为紧密的应用。

消息队列是发送和接收消息的公用存储空间，它可以存在于内存中或者是物理文件中。消息可以以两种方式发送，即快递方式(express)和可恢复模式（recoverable），它们的区别在于，快递方式为了消息的快速传递，把消息放置于内存中，而不放于物理磁盘上，以获取较高的处理能力；可恢复模式在传送过程的每一步骤中，都把消息写入物理磁盘中，以得到较好的故障恢复能力。消息队列可以放置在发送方、接收方所在的机器上，也可以单独放置在另外一台机器上。正是由于消息队列在放置方式上的灵活性，形成了消息传送机制的可靠性。当保存消息队列的机器发生故障而重新启动以后，以可恢复模式发送的消息可以恢复到故障发生之前的状态，而以快递方式发送的消息则丢失了。另一方面，采用消息传递机制，发送方必要再担心接收方是否启动、是否发生故障等等非必要因素，只要消息成功发送出去，就可以认为处理完成，而实际上对方可能甚至未曾开机，或者实际完成交易时可能已经是第二天了。

采用 MSMQ 带来的好处是：由于是异步通信，无论是发送方还是接收方都不用等待对方返回成功消息，就可以执行余下的代码，因而大大地提高了事物处理的能力;当信息传送过程中，信息发送机制具有一定功能的故障恢复能力；MSMQ 的消息传递机制使得消息通信的双方具有不同的物理平台成为可能。

在微软的.net 平台上利用其提供的 MSMQ 功能，可以轻松创建或者删除消息队列、发送或者接收消息、甚至于对消息队列进行管理。

在.NET 产品中，提供了一个 MSMQ 类库“ System.Messaging.dll ”。它提供了两个类分别对消息对象和消息队列对象进行操作。在能够使用 MSMQ 功能之前，你必须确定你的机器上安装了 MSMQ 消息队列组件，并确保服务正在运行中。在使用 ASP.NET 编程时，应在头部使用：

```
<%@ Assembly Name=”System.Messaging”%>
<%@ Import NameSpace=”System.Messsaging”%>
```

将 MSMQ 类库引入 ASP.NET 文件

1. 对消息队列的创建

```
dim MsgQue as MessageQueue
MsgQue=New MessageQueue(MsgPath)
```

其中：MsgPath 可以为本地私有队列，如“ .\MyQueue ”，也可以为其他机器的公有队列，如“ Saidy\777$\MyQueue ”,Saidy 为另一机器名。

2. 消息的发送

```
dim MsgQue as MessageQueue
MsgQue.Send(Msg)
```

其中：Msg 为任一对象。

3. 消息的接收

消息的接收又分成同步和异步方式两种,同步接收在规定时间内从消息队列中取出收到的第一条消息，当消息队列中没有消息时，程序处于等待状态；异步接收方式则是定义了一个事件处理函数，当消息队列中第一个消息到达时立即触发该函数。

1） 同步方式

```
dim Msg as Message
dim Fmt As XmlMessageFormatter
 Fmt= CType(MsgQue.Formatter,XmlMessageFormatter)
Fmt.TargetTypeNames = new String(){"System.String"}
Msg=MsgQue.receive(New TimeSpan(0,0,3))
```

首先定义收到消息应转换成的格式，然后在指定时间内去接收消息

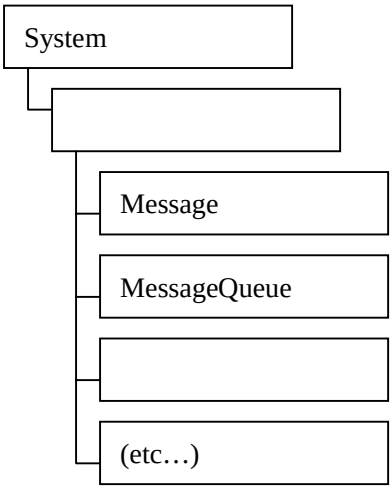
2） 异步方式

```
dim Fmt As XmlMessageFormatter
‘定义接收消息类型
Fmt = CType(MsgQue.Formatter,XmlMessageFormatter)
Fmt.TargetTypeNames = new String(){"System.String"}

‘定义消息处理函数入口
AddHandler MsgQue.ReceiveCompleted, New ReceiveCompletedEventHandler
                    (AddressOf OnReceiveCompleted)

‘定义消息处理函数
Public Shared Sub OnReceiveCompleted(s As Object, asyncResult As
ReceiveAsyncEventArgs)
    Dim MsgQue As MessageQueue = CType(s,MessageQueue)
    Dim Msg As Message = MsgQue.EndReceive(asyncResult.AsyncResult)
‘此时 Msg.Body 即为所取消息对象
    MsgQue.BeginReceive()
‘重新定义异步接收方式
```

```
End sub

    ‘启动异步接收方式
    MsgQue.BeginReceive
```

7.3.2 消息队列的命名空间体系结构

如图所示：



消息队列常用操作：

。用 create 方法创建你指定路径的消息队列，使用 delete 方法删除一个已经存在的消息队列。

。使用 exists 方法判别是否存在一个消息队列。

。使用 GetPublicQueues 方法获取消息队列网络中的一个消息队列

。使用 Peek 或者是 BeginPeek 方法查看消息队列中的消息，而不会删除它们

。使用 Receive 或者上 BeginReceive 方法从消息队列中取出一个消息，同时在消息队列中删除它。

。使用 Send 方法，送一个消息到指定的消息队列中。

7.3.3 消息队列的操作

1． 创建消息队列

。创建公共消息队列

MessageQueue.Create(“MyMachine\MyQueue”)

。创建私有消息队列

MessageQueue.Create(“MyMachine\Private$\MyPrivateQueue”)

说明：标识 Private$表示创建的是私有消息队列

2． 队列引用说明

当你创建了一个 MessageQueue 部件的一个实例以后，就应指明和哪个队列进行通信。在。Net 中有 3 种访问指定消息队列的方法：

。使用路径，消息队列的路径被机器名和队列名唯一确定，因而可以用消息队列路径来指明使用的消息队列。

。使用格式名（format name），它是由 MSMQ 在消息队列创建时或者应用程序在队列创建以后生成的唯一标识。

。使用标识名（label），它是消息队列创建时由队列管理者指定的带由描述意义的名字。它可能并不唯一。

采用路径（path）方式引用队列

消息队列类型	路径使用格式
Public queue	MachineName\QueueName
Private queue	MachineName\Private$\QueueName
Journal queue	MachineName\QueueName\Journal$
Machine journal queue	MachineName\Journal$
Machine dead letter queue	MachineName\Deadletter$
Machine transactional dead letter queue	MachineName\XactDeadletter$

。因为消息队列服务器接收到一个使用路径方式使用消息队列的操作请求时，会去解析出路径和格式名（format name）,因此它的效率上不如格式名方式使用队列。

。消息队列未连接时，只能使用格式名方式对它发送消息。

路径名的引用除了 path 属性以外，还可以由 MachineName 和 QueueName 两个属性得到。

路径引用的例子：

MessageQueue1.path=".\MyQueue"

采用格式名（format name）方式引用队列

格式名由公有私有标识串加上队列产生的 GUID，以及其他必需的标识构成。

消息队列类型	格式名的构成规则
Public queue	PUBLIC=QueueGUID
Private queue	PRIVATE=MachineGUID\QueueNumber
Journal queue	PUBLIC=QueueGUID;JOURNAL 或者 PRIVATE=MachineGUID\QueueNumber;JOURNAL
Foreign queues	DIRECT=AddressSpecification\QueueName

格式名由不由用户指定，而是在队列创建时由队列管理者自动产生。

。当你的部件作为一个 WEB sevice 或者是 WEB 调用的一部分的时候，最好采用格式名方式引用队列，因为它速度较快。

。当向一个非连接的队列发送消息时，应使用格式名方式，因为当队列不可连接时，路径解析会导致失败。

。网络拓扑结构发生变化或者消息队列重建以后，格式名会变化。

可以由消息队列对象的 FormatName 属性得到格式名。

例如：采用格式名方式引用消息队列的例子

MessageQueue1.Path = "FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75"

采用标识方式引用消息队列

标识是消息队列创建时，由消息队列创建者指定由于描述队列的文本属性。采用标识的好处在于屏蔽了低层的具体位置，对于移植和程序修改时，应用的修改很小。它的缺点也是显然的，就是不能保证标识的唯一性，当标识有冲突时，向该标识发送消息会导致一个错误的发生。标识可以由访问消息队列对象的 Label 属性得到。

总结，我们对消息队列引用的步骤大致为：

首先，产生一个 MessageQueue 对象的实例。

然后，根据引用消息队列的方式设置不同的属性。

如果为路径方式，设置它的 path 属性

如果为格式名方式，设置其 FormatName 属性

如果为标识方式，设置它的 Label 属性

3． 删除消息队列

删除一个队列使用 Delete 方法。当删除一个队列时，队列中含有的所有消息将首先被删除，然后删除该队列，它不会把消息队列中的信息发往死信队列中。删除一个队列最主要的问题是用户是否有删除该队列的足够的权限。

使用 Delete 方法的例子如下：

MessageQueue.Delete(“MyMachine\MyQueue”)

4．清除消息队列中的内容

有时我们需要把送入消息队列中而尚未发出的消息清除，或者定期需对消息发送日志队列进行清除，可以使用消息队列对象提供的 Purge 方法。它把指定的消息队列中的消息全部清空，并不再发送。

使用的例子如下：

MessageQueue1.Path="MyMachine\MyQueue"

MessageQueue1.Purge()

5．创建消息队列对象的实例

创建一个消息队列实例的步骤如下：

1． 创建一个 MessageQueue 类的实例

例如：

dim MyQue as New MessageQueue

2． 对 MessageQueue 类实例的 path 属性进行设置

例如：

MyQue.path=".\MyQueue"

3．设置你需要其他 MessageQueue 类的属性

6．消息队列配置属性

关于队列的属性：

path 属性：它可以决定引用队列的三种方式，路径引用、格式名引用、标识引用

category 属性：标识当前使用的队列的类型。Category 是队列所有者定义的 GUID 值。该 GUID 值可以有 GUID 生成工具产生或者是用户自定义的数字值。GUID 值不会唯一，这样才可以根据相同的 GUID 值，把多个消息队列划分为不同的类别（category）。

跟发送数据类型相关的属性

Formatter 属性：决定在一个队列中如何发送和接收消息的顺序，以及可以在一个消息中发送什么样的内容。

和队列交互相关的属性

DenyShareReceive 属性：决定同一时间内只有一个部件能够访问消息队列中的消息。

CanRead 和 CanWrite 属性：决定队列是否可以被读取或者是写入。

MaximumQueueSize 和 MaximumJournalSize 属性:以千字节为单位设置一个队列（日志队列）的消息最大容纳量。一旦接收的消息到达这个容量，新的消息将不再被接收。一般情况下，消息队列的最大值为消息队列管理员所设置，如果这个值没有控制的话，那么缺省的消息队列最大容量将是无限制的。

UseJournalQueue 属性：:设置是否将收到的消息拷贝到日志消息队列中去。

7．消息发送

MSMQ 消息队列中定义的消息由一个主体（body）和若干属性构成。消息的主体可以由文本、二进制构成，根据需要还可以被加密。你可以在属性窗口或者是直接对消息对象的属性进行赋值。但是，在 MSMQ 中消息的大小不能够超过４ＭＢ。

消息可以被送往公用、私有、日志、死信、交易队列。

。简单消息的发送

简单消息发送的步骤大致为：

１．建立与要发送消息的队列的连接

２．指定你要发送的消息的格式

３．提供要发送的数据

４．使用 Send 方法将消息发送出去

采用简单消息发送方式发送的数据类型可以是：对象、原数据类型、数据流或其他简单的数据类型。

例子：以发送一个整型数和字符串为例

首先创建一个连接：

Dim MessageQueue1 as new MessageQueue ("MyMachine\MyQueue")

或者

```
Dim MessageQueue1 as New MessageQueue
MessageQueue1.path=”MyMachine\MyQueue”)
```

由于是标准数据类型，消息格式可以不指定，使用缺省的

然后发送数值１

```
MessageQueue1.Send(1)
```

再发送字符串“ hello world”

```
MessageQueue1.Send("Hello world")
```

。发送复杂的消息

相对于简单的消息发送，发送较为复杂的消息对象能够给你带来对消息更多的控制的控制能力。对于复杂对象的发送，我们是通过创建消息的对象的实例，然后对它的属性实行相应设置来实现的。

发送复杂消息的步骤大致为：

１． 创建一个 MessageQueue 的实例，然后对它的 path 属性设置，以指明操作的消息队列。

２． 创建一个消息对象实例。

３． 设置消息的主体（body），然后修改不希望使用缺省值的属性。

４． 同样使用 send 方法把消息对象发送至相应的队列中。

例子：

```
‘创建 MessageQueue 实例，指明连接的队列
dim MessageQueue1 as New MessageQueue(“.\MyQueue”)
 ‘创建消息对象实例
dim MyMessage as New Message(“Hello world”)
 ‘设置消息队列属性
MyMessage.Label=”MyLabel”
 ‘发送消息
MessageQueue1.Send(MyMessage)
```

8．消息接收

消息的接收分为两种，即同步和异步。

同步方式，我们使用 receive 方法。当调用 receive 方法时，它会从指定的消息队列中选取出第一个适合要求的消息对象返回给用户，然后把它从消息队列中删除。如果调用 receive 方法的时候，消息队列中没有一条记录存在，那么方法将导致程序挂起，直到有消息到达消息队列中。为了不使这个等待过程太长，你可以在调用 Receive 方法时指定 time-out 值（毫秒为单位），以指定在相应时间到达后退出等待过程。在使用 Receive 方式时，还可以指定消息队列的 DenyShareReceive 属性，防止其他用户对队列进行操作。试想这样一种情况，当消息队列中剩一条消息的时候，2 个用户同时对它调用 peek 方法，发现都有消息，于是都放心的使用 Receive 方式去获取消息，结果必然导致其中一个用户被挂起。如果，在调用 receive 方法以前使用了 DenyShareReceive 属性拒绝其他用户的对该队列的 Receive 方法，就会避免上述现象的发生。

同步接收的例子：

```
dim MyMessageQue as MessageQueue

MyMessageQue=New MessageQueue("MyMachine\MyQueue")
'指定要连接的消息队列
dim MyMessage as New Message
MyMessage=MyMessageQue.Receive(1000)
'同步收取一条消息，超时时间为 1 秒
```

Peek 方法和 Receive 方法比较类似，只是它并不把取得的消息对象从消息队列中删除。Peek 方法只取出消息队列中的第一条消息，若要取得所有的消息可以使用 GetMessages 方法或者是 GetMessagesEnumrator 方法。Peek 方法也是同步方法，所以当消息队列中没有消息的时候，它也将被挂起，直到有消息产生。同样的，Peek 也可以指定超时时间（单位毫秒），一般最常用的方式是设为 0 或无穷大。设为 0 的意义是，对消息队列搜索看是否有消息产生，并马上返回；设为无穷大的意义是，直到有消息产生才进行处理。

例如：

```
'设置为超时时间为 0
MyMessage.TimeToBeReceived=0

 '设置超时时间为无穷大
MyMessage.TimeToBeReceived=Message.Infinite

 '以同步方式使用 Peek 方法的例子
dim MyMessageQue as New MessageQueue("MyMachine\MyQueue")

'指定连接的消息队列
'对消息队列查询首条消息，若无 1 秒钟后超时返回
dim MyMessage as Message
MyMessage=MyMessageQue.Peek(1000)
```

异步接收方式，是指接收消息时，不必理会调用的消息接收方法是否成功，方法将立即返回，并继续进行程序处理。对于异步接收方式收到的消息，有两种处理方式。一种是消息接收到以后，会发出一个事件，我们可以定义一个事件处理函数，在该函数中，对接收到的消息进行处理。另一种方法是为消息接收函数定义一个回调函数，当消息接收到以后，它会去处理带来的消息。

在 ASP.NET 中，事件处理方式中，在事件处理函数中，我们使用 BeginPeek 或者是 BeginReceive 方法从消息队列中取得一条消息。如果要处理多条消息，那么就要反复调用 BeginPeek 或者 BeginReceive 方法。

在回调处理方式中，回调函数定义了一个和 BeginPeek 或者 BeginReceive 方法绑定到一起的代理，这样即使是在消息处理当中，代理仍然可以监视是否有新的消息到来。

因为消息处理函数方式用得比较普遍，所以下面我们就重点介绍消息函数方式得使用。

在事件模型中，首先必须绑定一个消息处理函数到一个消息，它是当异步调用完成以后，你希望对它进行处理的代码入口。然后调用 BeginReceive 方法，启动异步接收模式。当接收过程完成，.net 平台会发出一个消息，表示已经从消息队列中接收到一个消息，然后调用

和该消息绑定的处理函数，以处理到达的消息。

它的步骤如下：

① 对 MessageQueue 对象的 BeginReceiveCompleted 事件绑定一个消息处理函数

② BeginReceiveCompleted 处理函数中，创建一个消息对象实例和一个消息队列 对象的实例

③ 定义如何处理收到的信息，代码框架如下：

```
Public Sub MyQueue_BeginReceiveCompleted(sender as Object, args as System.Messaging.ReceiveAsyncEventArgs)
    Dim myMessage as Message
    Dim myMessageQue as MessageQueue
    MyMessageQue = CType(sender,MessageQueue)
    myMessage = MyMessageQue.EndReceive(args.AsyncResult)
End Sub
```

④ 在主程序中，使用 BeginReceive 方法启动异步接收方式。

例如：

```
MyMessageQue.BeginReceive
```

使用 peek 方法的异步接收模式和使用 receive 方法的方式差不多，就不再多述了。

9．消息确认

为确保消息被正确发送到目的消息队列，我们还可以对消息队列进行设置，让其返回消息是否正确发送到指定队列的确认消息。确认消息有两种，一种是消息队列到达指定队列后发出的确认消息，另一种是消息被对方应用从消息队列中删除。而每一种消息又存在有两种形式，确认和否定。当消息正确到达目的队列或应用时，它会发回发送成功的确认消息。如果消息无法到达指定的队列或应用，那么它会发回发送失败的确认消息。不过收到发送失败的消息，未必是对方消息队列无法到达，它有可能是在超时时间设置过小或者是无法通过对方的验证。

从对方返回的确认消息通常不会放入一般的队列中，而是放入一个称之为管理队列的特殊队列中。确认消息也和一般的消息不一样，它不包含消息的主体，仅仅通过消息的头部就可以知道确认消息的意义。

在 ASP.NET 中，确认消息发往的管理队列，由其 AdministrationQueue 属性指定的队列所决定。

你可以把不同的确认消息发往不同的管理队列中。对管理队列的操作就和操作不同的队列是一样的，可以用 peek 方法查看，也可以用 remove 移走。

那么，如何设置一个消息，要求它返回确认信息呢？

首先设置一个消息对象的 AdministrationQueue 属性，指定确认消息返回的管理队列。

接着设置返回消息的确认类型，使用消息对象的 Acknowledge 属性。

Acknowledge 属性的值可以是：

。FullReachQueue 无论发送的消息到达或不能到达目的队列，都会返回确认消息

。FullReceive 无论发送消息能否到达对方应用程序，都会返回确认消息。

。NotAcknowledgeReachQueue 只有发送的消息未能到达目的队列，才返回确认消息。

。NotAcknowledgeReceive 只有发送的消息未能到达对方应用程序，才返回确认消息。

。None 不返回任何确认消息

最后，发送消息，并检查管理队列，看是否有确认消息产生。

10．消息日志

日志队列可以保存你操作过的消息的备份。它的好处是，一但发现前面的操作失败，可以从日志队列中重新创建出原先的消息对象，然后再进行操作。例如，向远方发送一个消息对象，然后对方返回一个失败的确认。我们可以从失败确认消息中提取出一个和开始发送的消息相关的 ID 值，然后根据提出的 ID 值从日志队列中找到发送的消息，重新创建一个消息对象，并再次发送。在.net 中，我们使用 ReceiveByCorrelationID 或 PeekByCorrelationID 方法根据 correlation ID 值取得消息对象。

在一台机器上，都会有一个全局消息队列，它保存任何从该机器发出的消息，而不论消息发送是否成功。每个消息队列也可以有自己的消息日志队列。日志队列的使用有两种方式，一种是对消息队列对象设置 UseJournalQueue 属性，它表示对该队列收到的所有消息使用日志记录方式，而对于发出的消息不做任何记录；另一种方法是对消息对象设置 UseJournalQueue 属性，所有被发送的消息将被记录到系统日志队列中去。消息日志队列有一个最大容量，称作 quota，一旦日志队列存储容量到达该值后，以后到来的本应存储的的消息将不再被存储，同时不会发出任何的出错信息。所以作为管理人员，应该定期清理日志队列，以防止上述现象的发生。消息队列只是被动的接收端，它们不可能返回确认消息，或者发送删除的消息到死信队列中，或者是进行超时处理。

例子：

设置消息队列对象的 UseJournalQueue 属性，以记录收到的消息到日志队列中
MyMessageQueue.UseJournalQueue=True

设置消息对象的 UseJournalQueue 属性，以记录收到的消息到系统日志队列中
MyMessage.UseJournalQueue=True

7.3.4 小结

通过本章的介绍，我们对 MSMQ 有了一定的了解，并且对消息队列的操作有了一定的了解。