

第15章 实例研究1 数据双重性

本实例研究的内容主要是用于说明数据的双重性。数据有许多种形式，因此我们需要以不同的方式来浏览。如果是扩展页原型，我们可以通过点击数据列的上部来对数据分类和查看数据。但在网络上，在没有XML以前，要想实现这种功能并不是一件简单的事情。不过有了XML，我们可以很容易地在客户端存储XML文档，并随心所欲地对数据进行重新分类和排序。

本章中的例子最初是为一家大保险公司写的。这家保险公司想要一个能让职员对表格的细节进行修改的系统（这些书面表格包括保险申请单、保险生效单等，而不是HTML表单）。一些人用表单名称查询表单，而另一些人用编号来查询表单。如果想建立一个培训需求量最小的系统，而不是强迫用户去适应操作表单的一系列规定，我们就要提供一套系统，它可以使用户以标题或编号来查阅表单。

这个解决方案中采用了下列技术：

- 用ASP把一个SQL服务器数据库变成XML。
- 用XSL动态转换XML。
- 用DHTML压缩/扩展概要。
- 用XML DOM的客户脚本添加/去除/更新结点。

15.1 商业需求

在设计过程中从用户界面开始就面临一个主要挑战显然早了些。这是一个电子商务站点，其产品是可以订购的书面表单。但问题在于这些表单具有双重性，它们同时用名字和编号作为标识符。

在网络应用中解决这个“数据双重性”问题的方法很多，可以采用一些XML之前的技术，其中一些技术比另一些技术更简单。但是，如果要想得到一个直观而简单的解决方案，我们选择XML。

15.2 系统要求

我们设定如下设计目标：

- 改进浏览方式。
- 提供用名字或编码查询产品的功能。
- 提供一个完全直观的用户界面。
- 尽可能减少服务器的访问次数。

我们认为如果能达到这些目标，这个项目就可以认为是成功的。

15.3 设计时间：让我们开始吧

最初的解决方案是建立在Microsoft技术上的一个系统，其中网络站点建立在Microsoft Site

Server。但为了本书实例说明的需要，我们使用了一个 SQL Server数据库和ASP（这个例子在Windows 9x上的PWS或Windows NT上的IIS都运行良好）。由于这项应用的目标用户平台的限制，并且都是在室内使用，所以我们可以自由选择浏览器的配置。我们选择了 Microsoft的IE 5，因为它是当前最新的浏览器。

我们想为网络管理者提供标准的双模式界面——一个“左边是内容，右边是数据”的标准表格。管理者可以在左侧的内容菜单上点击所要求的表格类型，右边显示的数据就会相应更新，显示所选表格的一些细节信息如表格编号、名称、类型、描述等。管理者可以在右边编辑表格信息，或者通过在左边点击转移到另一个表单上去。图15-1所示的屏幕拷贝显示了用户界面的样子。

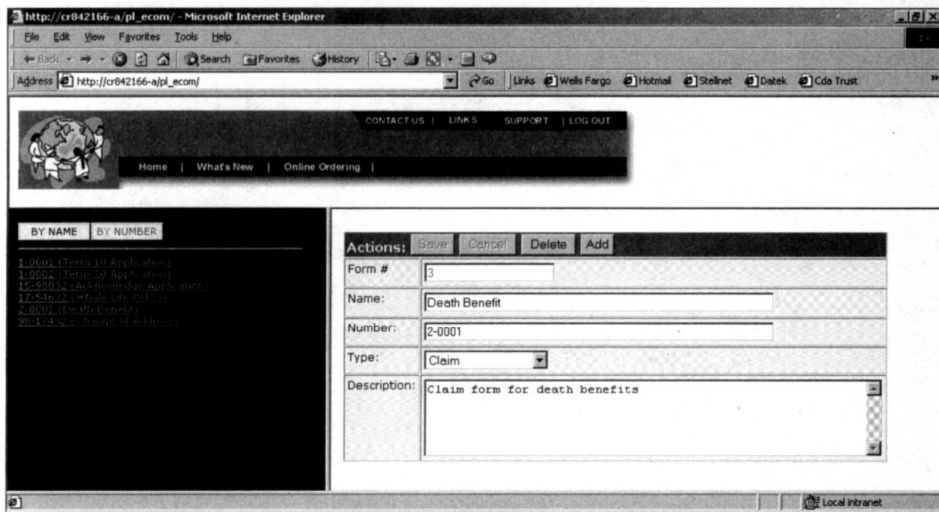


图 15-1

由于关键数据的双重性，用户可以根据名称或编号查找一个表单，因此我们需要两类不同的内容：一个是表单编号，另一个是表单名字。我们可以采用传统方法使用服务器端的ASP。在这个例子中一个ASP页面将生成“以名字”为标识的页面，另一个将生成“以编号”为标识的页面。这种方法的不足之处是需要建立和维护两个页面，并且一旦更改显示内容就要对服务器进行数据传输。

一个改进方法就是使两种显示共用一个页面，并同时向客户传送两种显示内容，但用DHTML隐藏其中一种显示。这种方法比第一种方法更简便，而且避免了对服务器的访问，但这种方法使通过网络传输的数据量增加了一倍。

为了寻找一种更好的方案，我们尝试了XML和XSL。XML和XSL可以理想地只传送一次数据，实现在客户端处理解释选项的要求。XML十分适合本例要求。

15.4 实现方法概述

在XML方案中，用户可以通过点击某个键实现一种显示内容到另一种的变换，而不需要额外的服务器访问。数据以XML格式只传给客户一次。在客户端，可以根据需要对XML数据进行再解释。

在高端，这个系统包括如下步骤：

1) ASP读取SQL Server生成的表并建立一个XML流。

2) 客户端脚本加载XML文档。

3) 客户端脚本加载并运行XSL文档。

图15-2显示了这个解决方案的各个部分以及它们之间的关系。

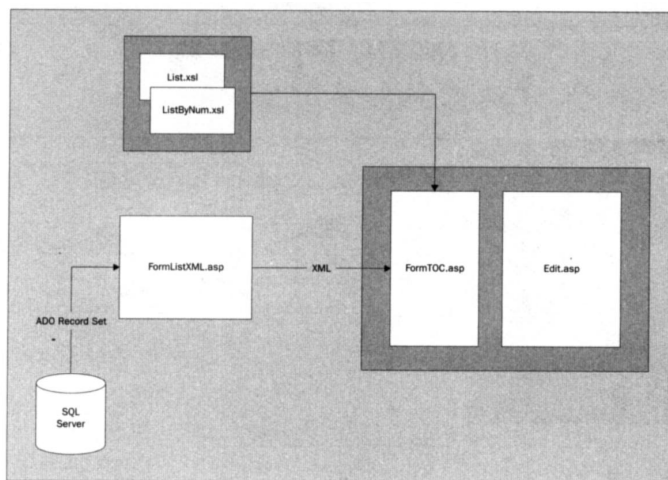


图 15-2

正如我们刚才看到的，这意味着我们在屏幕左侧将会有两种不同的显示内容。图 15-3中的屏幕拷贝显示了这两种显示内容。

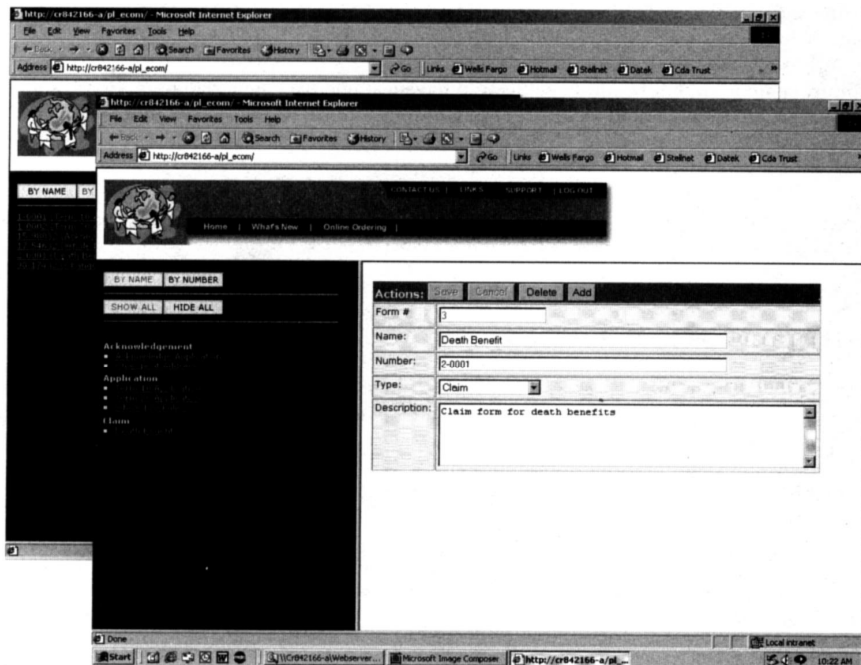


图 15-3

15.5 建立数据库

本例中建立数据库所需要的文件是本书提供的下载代码的一部分，文件格式为 SQL 脚本格式：WroxProXMLForms.sql。

要使用这个文件，你首先要用 SQL 生成一个新的数据库：WroxProXMLForms。然后，如果使用的是 SQL Server 7，就用 SQL Server Query Analyzer 运行对应新生成数据库的 WroxProXMLForms.sql。如果是 SQL Server 6.5，可以在 Tools | SQL Query Tool 菜单下实现同样的功能。再回到 Enterprise Manager 你会看到已经生成了 ecom_form_type 和 ecom_product 表格。

完成这个过程后，你可以用 ODBC Data Sources 控制面板应用程序建立一个叫 WroxProXML 的系统 DSN。

注意下载文件中的 edit.asp 同前面一个解决方案中的相应文件完全不同。在前一个解决方案中那个文件的功能是由几个 Site Server 页面来完成的。这些页面是标准化的，并含有调用各种 XML 树更新函数的命令，这些函数稍后将会在本章中提到。

15.6 提供 XML 功能的 ASP

整个过程中的第一步是 ASP 页面 FormListXML.asp 从 SQL Server 表格中读取数据，并返回一个 XML 流。从 ASP 页面返回的缺省值是 HTML，所以我们要指定其返回值为 XML。命令如下：

程序清单 15-1

```
<%Response.ContentType = "text/xml"%>
```

一旦设定返回值类型，除开 ASP 脚本标志或 ASP response.write 生成的内容以外的任何内容都被认为是 XML，需要有完整的格式。换言之，这意味着生成的文件：

- 必须在开始声明是 XML。
- 必须只有唯一的一个根元素。
- 必须有相匹配的开始和结束标志。
- 必须有正确嵌套的标志。

任何一个返回 XML 流的 ASP 页面，如 FormListXML.asp 都必须符合这些要求，否则你的浏览器（IE 5）就会报告解析错误。

我们生成的 XML 文件格式如下：

程序清单 15-2

```
<?xml version="1.0"?>
<FORMLIST TYPE="Insurance Forms">
  <FORMS TYPE="3rd Party Booklets and Article Reprints">
    <FORM>
      <FORMTITLE>NAIC - "Life Insurance Buyer's Guide"</FORMTITLE>
      <SKU>Product140</SKU>
      <FORMNUMBER>15-11297</FORMNUMBER>
    </FORM>
    <FORM>
      ...
    </FORM>
  </FORMS>
</FORMLIST>
```

```

    </FORM>
  </FORMS>
</FORMLIST>

```

在这种情况下，一个<FORMLIST>包含一个或多个<FORMS>，这是一种群组机制，可以将多个表单组成特定的“类”（例如应用表格）。在每个<FORMS>分支上，我们可以有一个或多个<FORM>分支，分支中含有实际表单的相关信息。<FORMTITLE>和<FORMNUMBER>元素是自解释性的，<SKU>是表单唯一的标识符，这个标识符是必须的，因为表单编号并不总是唯一的。

下面我们来看看 FormListXML.asp 的代码，这些代码将为我们生成 XML。像以前提到的那样，我们设定 ASP response 对象的 Content Type 属性为“test/xml”，并以此为开始。同时我们也将 Response 对象的终止设为 -1，防止它在浏览器中缓存：

程序清单 15-3

```

<%
Response.ContentType = "text/xml"
Response.Expires = -1
%>

```

接着，我们开始把 XML 写到浏览器中去。调用应用程序将会识别出 ASP 分隔符外的任何内容都是 XML，因为在 HTTP 头中的 Content Type 属性已经告诉应用程序这个特性了。接着开始写入根元素<FORMLIST>和 TYPE 属性。

程序清单 15-4

```

<?xml version="1.0"?>
<FORMLIST TYPE="Insurance Forms">

```

上述命令将会把这一行原样写回到调用应用程序，和 ASP 分隔符外的 HTML 标识符的作用一样。现在，我们要建立一个和数据库的连接，将使用 ADO。同前面生成的 System DSN 相连很简单：

程序清单 15-5

```

<?xml version="1.0"?>
<FORMLIST TYPE="Insurance Forms">
<%
' Set up connection...
Set cnEcom = Server.CreateObject("ADODB.Connection")
cnEcom.Open "DSN=WroxProXML", "sa", ""

```

在最简单的层次上，为了得到信息和生成 XML，我们将会生成一个记录集并反复向其中写入 XML 标识和内容。但是，ASP 在字符管理方面可能不太完善，所以为了最优，我们将字符管理交给 SQL Server。这种做法产生了一个有些冗长的 SQL 语句，如下所示：

程序清单 15-6

```

sSQL = "SELECT '<FORM><FORMTITLE>' + RTRIM(ep.name) + '</FORMTITLE><SKU>' + " & _
"RTRIM(ep.sku) + '</SKU><FORMNUMBER>' + RTRIM(ep.number) + " & _
"'</FORMNUMBER></FORM>', ep.form_type AS form_type_group, " & _

```

```
"eft.form_type FROM ecom_product ep, ecom_form_type eft " & _
"WHERE ep.form_type = eft.type_id ORDER BY eft.form_type"
```

```
Set rsForms = cnEcom.execute(sSQL)
```

将来的ADO和SQL Server版本将会更好地支持XML，省去这类字符管理的连接。但在这里，我们要返回每个表单的XML标识作为SQL Server数据的一部分。

这个过程将会为我们生成一个记录集，按照其中的步骤来生成 XML文档。记录集中的每一行都要含有各个表格的完整XML标识以及表格类型。用Form-type来对生成的XML进行排序和分组，同时也生成<FORMS>标识的TYPE属性。

接着，我们开始一个循环，将结果写下来，每条记录都以 </FORM>作为结束标识，并插入带有TYPE属性集的一个新的开始标识。对每一条记录都进行如下过程：

程序清单 15-7

```
thisFormType = ""
init = true
Do While Not rsForms.eof
    If thisFormType <> rsForms.fields(1) Then
        '// moving on to a new one...
        If not init Then
            '// we've already done some, we have to close the tag
            Response.Write "</FORMS>" & vbCrLf
        else
            init = false
        End if
        thisFormType = rsForms.fields(1)
        Response.Write "<FORMS TYPE=" & _
            Server.HtmlEncode(Trim(rsForms.fields(2))) & ">"
    End if
```

生成对应记录的XML后，还有一些SQL语句留下的字符，所以我们去掉这些字符：

程序清单 15-8

```
strThis = rsForms.fields(0)
strThis = replace(strThis, " ", "&nbsp;")
strThis = replace(strThis, "& ", "& ")
strThis = replace(strThis, "& ", "& ")
strThis = replace(strThis, "& ", "& ")
```

然后将记录返回给客户，在记录集中跳到下一条记录：

程序清单 15-9

```
Response.Write strThis
rsForms.MoveNext
Loop
Response.Write "</FORMS>" & vbCrLf

Set rsForms = nothing
Set cnEcom = nothing
%>
</FORMLIST>
```


图15-4是我们生成的XML文档的结构（含有一些样本数据），这是在IE 5中显示的结果。

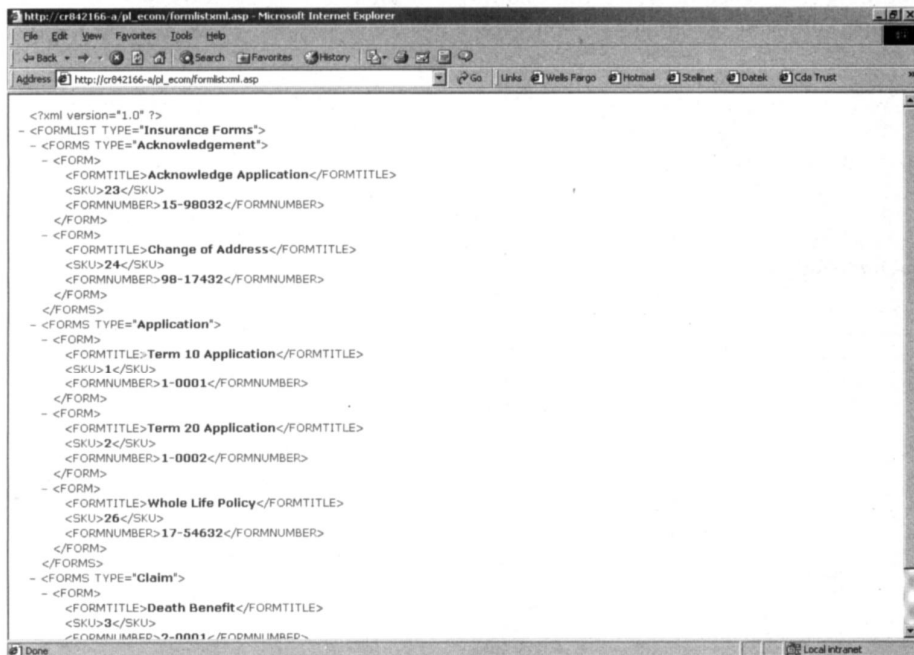


图 15-4

一个简单的开发窍门是一旦生成XML的ASP正常工作，就将浏览器产生的XML存储起来。这样你就可以生成测试数据。当对基于生成XML的ASP其他页面进行测试时需要用这些测试数据（要想生成这些数据，只要在浏览器中查看XML文档时选择File下的Save As...）。用XML Notepad或你喜欢的ASCII编辑器可以方便地编辑测试数据。使用这些从XML文档中生成的测试数据，令你能用真实数据的一个子集进行检验，从而去除了调试过程中的一个潜在错误发生源，你也不用担心复杂的SQL和底层表格，因为可以确信数据的形式是正确的。我发现这是一种非常有效的增强产品性能的技巧。

站在使用的角度上，最简单的办法是单独调试生成XML的ASP，而不是作为一个更完整的应用程序的一部分来调试。IE 5的一个很好的特性就是其内部有缺省XSL样式表，在一个XML文档没有明确指定所使用的解释样式表时可以实现XML文档的解释（如上一个屏幕拷贝所示）。它同时也具有展开/折叠功能，可以很好地实现对XML的检查，以保证你得到的东西正是你所想要的。

已经讨论过了如何从数据库生成XML，下面我们看看如何调用XML，以及如何显示管理者可以编辑的表格。

15.7 客户端页面

客户端的开始页面default.asp生成了包含两个表的一个表集，左边是FormTOC.asp页面，生

成内容表格，右边是edit.asp页面，可以编辑：

程序清单 15-10

```
<frameset FRAMEBORDER="0" rows="120,*">
  <frame SRC="banner.asp">
  <frameset name="container" FRAMEBORDER="1" cols="350,*">
    <frame name="toc" SRC="FormTOC.asp">
    <frame name="mainbody" id="mainbody" SRC="edit.asp">
  </frameset>
</frameset>
```

TOC图文框

FormTOC.asp是包含XML表单列表的页面。它包含了控制应用程序的逻辑，所以我们从讨论这个页面开始。这个页面基本上是一个“核心”，所有的工作都是通过它里面包含的脚本完成的。页面上只有两个按钮和一个<div>元素。xslresult <DIV>是一个占位符，它接收用XSL转换的XML结果：

程序清单 15-11

```
<body>

  <button id="btnByName"
    onClick="changeXSL('<%=ByNameURL%>'); setButtons('<%=ByNameURL%>');"
    style="width:80">By Name</button>
  <button id="btnByNumber"
    onClick="changeXSL('<%=ByNumberURL%>'); setButtons('<%=ByNumberURL%>');"
    style="width:80">By Number</button>

  <hr>
  <div id="xslresult">
    <!-- resulting HTML will be inserted here -->
  </div>

</body>
```

1. 显示两个视图

我们提到过表单具有双重特性（因为它们可以用名字或编号查询），所以，为了让用户在XML的两个视图之间切换，我们用两种XSL样式表之中的一种来转换它。为了进行转换，需要用MSXML建立两个DOM，一个用于存储刚用formlistxml.asp生成的XML，另一个保存转换的内容。

当事件OnLoad()终止时，init()函数马加载刚刚生成的XML，init()函数在一个叫source的XML DOM中。然后它调用函数changeXSL()，为我们想要的界面选择合适的结构。另一个DOM叫style，它保存转换过的页面结果。

转换过的XML放在<DIV>占位符之间，在这里用文档其他部分采用的CSS样式表来确定XML的显示格式。

下面是这个页面的全部代码，稍后将更详细地对其进行讨论：

程序清单 15-12

```
<%Response.Expires = -1%>

<%
```



```

Const ByNameURL = "list.xsl"
Const ByNumberURL = "listbynum.xsl"
%>

<html>
<head>
    <link REL="stylesheet" TYPE="text/css" HREF="list.css">
</head>

<script FOR="window" EVENT="onload">
    init();
</script>

<script>
var source;
var style;
var root;
var styleURL;

function init()
{
    // Do init stuff. Called by the parent frame.
    source = new ActiveXObject('Microsoft.XMLDOM');
    source.async = false;

    source.load('formlistxml.asp');

    // did the XML file load OK?
    if (source.parseError.errorCode != 0)
    {
        msg = 'Error loading data file.';
        msg += '\nDescription: ' + source.parseError.reason;
        msg += '\nSource text: ' + source.parseError.srcText;
    }

    root = source.documentElement;

    style = new ActiveXObject('Microsoft.XMLDOM');
    style.async = false;

    styleURL = ((typeof(styleURL)=='undefined') ? '<%=ByNumberURL%>' : styleURL);
    changeXSL(styleURL);
    setButtons('<%=ByNumberURL%>');
}

/*
*****
XML tree manipulation functions
*****
*/

function addBranch(sType)
{
    var newBranch = root.selectSingleNode("//FORMS").cloneNode(false);
    newBranch.setAttribute("TYPE", sType);
    root.selectSingleNode("//FORMLIST").appendChild(newBranch);
}

```

```

    return newBranch;
}

function addNode(oForm)
{
    // add a new node into the tree.
    // pick a node to use as a template...
    var newNode = root.selectSingleNode("//FORM").cloneNode(true);
    var insertPoint, newBranch;

    newNode.selectSingleNode("FORMTITLE").text = oForm.formTitle;
    newNode.selectSingleNode("FORMNUMBER").text = oForm.formNumber;
    newNode.selectSingleNode("SKU").text = oForm.sku;

    // find the node that corresponds to the parent for this form type
    insertPoint = root.selectSingleNode("//FORMS[@TYPE='" + oForm.formType + "']");

    if (insertPoint==null)
    {
        // couldn't find that form type, create a branch for it
        insertPoint = addBranch(oForm.formType);
    }
    insertPoint.appendChild(newNode);

    update();
}

function updateNode(oForm)
{
    // update an existing node
    var theNode = root.selectSingleNode('//FORM[SKU="' + oForm.sku + '"]');
    if (theNode==null)
    {
        // Should never get here. This would mean "node not found", so if
        // that happens just reload the tree
        parent.location.reload();
    }
    else
    {
        theNode.getElementsByTagName("FORMTITLE").item(0).text = oForm.formTitle;
        theNode.getElementsByTagName("FORMNUMBER").item(0).text = oForm.formNumber;

        var oldType = theNode.parentNode.attributes.item(0).text;
        var newType = oForm.formType;

        if (newType!=oldType)
        {
            // The form type has changed. We need to move it.

            // First create a clone...
            var theNewParent =
                root.selectSingleNode("//FORMS[@TYPE='" + newType + "']");
            if (theNewParent==null)
            {
                // couldn't find that form type, create a branch for it
                theNewParent = addBranch(newType);
            }

```

```

        theNewParent.appendChild(theNode.cloneNode(true));

        // Now delete the original one...
        theNode.parentNode.removeChild(theNode);
    }
    update();
}

function deleteNode(sFormno)
{
    // remove a node from the tree
    var theNode = root.selectSingleNode('//FORM[FORMNUMBER="' + sFormno + '"]');
    if (theNode==null)
    {
        // Should never get here. This would mean "node not found", so if
        // that happens just reload the tree
        document.location.reload();
    }
    else
    {
        theNode.parentNode.removeChild(theNode);
        update();
    }
}

/*
*****
XSL-related Functions
*****
*/

function update()
{
    // apply the XSL
    if (style.documentElement && source.documentElement)
    {
        document.all.item('xslresult').innerHTML = source.transformNode(style);
    }
}

function changeXSL(xsldoc)
{
    // load a new XSL
    styleURL = xsldoc;
    style.load(styleURL);
    update();

    if (xsldoc=='<%=ByNameURL%>')
    {
        hideAll('UL');
    }
}

/*
*****
Functions for the DHTML collapse/expand
*****
*/

```

```

*/

function getChildElem(eSrc,sTagName)
{
    var cChildren = eSrc.children;
    var iLen = cChildren.length;
    for (var i=0; i < iLen; i++)
    {
        if (sTagName == cChildren[i].tagName)
        {
            return cChildren[i];
        }
    }
    return false;
}

function document.onclick()
{
    // expand/collapse
    var eSrc = window.event.srcElement;
    if ("clsHasChildren" == eSrc.className && (eChild = getChildElem(eSrc,"UL")))
    {
        eChild.style.display = ("block" == eChild.style.display ? "none" : "block");
    }
}

function showAll(sTagName)
{
    var cElems = document.all.tags(sTagName);
    var iNumElems = cElems.length;
    for (var i=1; i < iNumElems; i++) cElems[i].style.display = "block";
    document.all.btnShowAll.disabled = true;
    document.all.btnHideAll.disabled = false;
}

function hideAll(sTagName)
{
    var cElems = document.all.tags(sTagName);
    var iNumElems = cElems.length;
    for (var i=1; i < iNumElems; i++) cElems[i].style.display = "none";
    document.all.btnShowAll.disabled = false;
    document.all.btnHideAll.disabled = true;
}

/*
*****
Misc Functions
*****
*/

function setButtons(state)
{
    // set the disabled state of the buttons
    // ie: if already sorted by number, disable the button to allow you to do that
    document.all.btnByName.disabled = (state=='<%=ByNumberURL%>');
    document.all.btnByNumber.disabled = (state=='<%=ByNameURL%>');
}

```

```
function navigateTo(sSKU)
{
    // navigate to a new form
    var approved = true;
    if (window.parent.mainbody.dirty)
    {
        // the data has changed. Raise a warning.
        msg = 'You have made changes which have not been saved.';
        msg += '\n\nTo abort your changes, press OK.';
        msg += '\nTo resume editing, press Cancel.\n';
        if (!confirm(msg))
        {
            approved = false;
        }
    }

    if (approved)
    {
        // it's OK to move to another record
        sURL = 'edit.asp?sku='+sSKU;
        window.parent.mainbody.location.href = sURL;
    }
}

</script>

<body>

    <button id="btnByName"
        onClick="changeXSL('<%=ByNameURL%>'); setButtons('<%=ByNameURL%>')"
        style="width:80">By Name</button>
    <button id="btnByNumber"
        onClick="changeXSL('<%=ByNumberURL%>'); setButtons('<%=ByNumberURL%>')"
        style="width:80">By Number</button>

    <hr>
    <div id="xslresult">
        <!-- resulting HTML will be inserted here -->
    </div>

</body>

</html>
```

讨论过页面后，下面讨论一下位于页面中心的各种函数。

2. 函数清单

表15-1列出了对XML文档进行操作的函数名称和作用。

表 15-1

函 数	描 述
init()	初始化代码
addNode()	向XML树上添加一个新结点
updateNode()	更新XML树上已存在的一个结点
deleteNode()	从XML树上去除一个结点
update()	应用XSL转换并更新占位符<DIV>
changeXSL()	加载一个新的XSL文档

我们也使用表15-2中列出的几种函数。

表 15-2

函 数	描 述
setButtons()	对选择by name和by number浏览方式的按钮进行状态设定(激活/休眠状态)
navigateTo()	接收一个表格编号并加载带选定表单信息的编辑屏幕

除了对XML进行解释,我们在by name界面上还提供了一个DHTML展开/折叠目录。表15-3是支持这个目录的函数。

表 15-3

函 数	描 述
showAll()	展开树的所有分支并显示所有结点
hideAll()	折叠树的所有分支并隐藏第一级结点以外的所有结点
getChildElements()	得到某个给定标识的所有子结点(如果有)
document.onclick() (事件处理器)	检查一个结点的等级,如果有子结点的话展开/折叠

我们在这里不准备讨论DHTML的展开/折叠功能,但它们包含在下载源代码中。我将只讨论在XML解释过程中起关键作用的那些函数。

3. 加载XML表单

前面已经提到过当浏览器中的onload事件结束时,会运行下列初始代码:

程序清单 15-13

```
function init()
{
    // Do init stuff. Called by the parent frame.
    source = new ActiveXObject('Microsoft.XMLDOM');
    source.async = false;

    source.load('formlistxml.asp');

    // did the XML file load OK?
    if (source.parseError.errorCode != 0)
    {
        msg = 'Error loading data file.';
        msg += '\nDescription: ' + source.parseError.reason;
        msg += '\nSource text: ' + source.parseError.srcText;
    }

    root = source.documentElement;

    style = new ActiveXObject('Microsoft.XMLDOM');
    style.async = false;

    styleURL = ((typeof(styleURL)=='undefined') ? '<%=ByNumberURL%>' : styleURL );
    changeXSL(styleURL);
    setButtons('<%=ByNumberURL%>');
}
```


我们已经看到 FormListXML.asp 是从数据库生成 XML 文档的 ASP 页面的名称。init() 函数生成了一个新的 Microsoft XML DOM 的例子，并将由 FormTOC.asp 生成的 XML 文档加载到其中。

FormTOC.asp 页面使用表 15-4 中列出的四个全局页面变量。

表 15-4

变 量	描 述
Source	与 ASP 执行结果一起加载的 XML DOM
Style	包含当前加载的 XSL 的 XML DOM
Root	一个指向 source.documentElement 的快捷键
StyleURL	查看当前用来解释 XML 的 XSL 样式表

当运行 init() 函数时，styleURL 设定为 list.xsl，即缺省样式表。在我们讨论如何应用 XSL 样式表以前，先来看看 XSL 本身。

4. 显示 HTML

用一个客户端脚本生成页面是十分有效的，但不足之处是你不能只要在浏览器中点击 View Source 就能看到 HTML 了。如果在页面生成过程中你需要看到 HTML，只要将这个按钮加到页面上去：

程序清单 15-14

```
<button onclick='alert (document.body.innerHTML);'>View Source</button>
```

15.8 “以编号浏览” XSL 样式表

在这里提到的两个样式表中，ListByNum.xsl 是比较简单的一个。它生成了一个表单的清单，按照表单的编号顺序排列，用 <xsl:foreach> 元素的 order-by 属性表示。为了方便，我们在编号后的括号里显示表格名字。注意编号是如何进入一个与 JavaScript 函数相连的 HTML 超链接中去的（见下面示例）：

程序清单 15-15

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <xsl:for-each select="//FORM" order-by="FORMNUMBER">
      <a>
        <xsl:attribute name="HREF">
          JavaScript:navigateTo('<xsl:value-of select="SKU" />');
        </xsl:attribute>
        <xsl:attribute name="TITLE">
          <xsl:value-of select="FORMTITLE" />
        </xsl:attribute>
        <xsl:value-of select="FORMNUMBER" />
        <span style="color:silver">
          (<xsl:value-of select="FORMTITLE" />)
        </span>
      </a>
      <br />
    </xsl:for-each>
  </template>
</xsl:stylesheet>
```

```
</xsl:template>  
</xsl:stylesheet>
```

生成的带JavaScript函数的链接如下（参见图 15-5）：

程序清单 15-16

```
<a HREF="JavaScript:navigateTo('sku')" TITLE="title">formnumber</a><br />
```

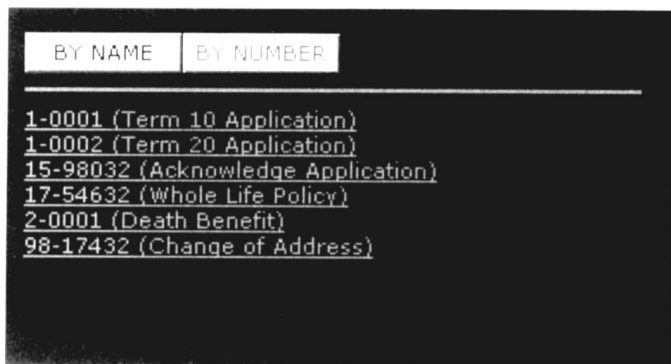


图 15-5

XSL文档在解释的HTML中建立URL，HTML将选定表的各种信息显示在编辑屏幕上。这个过程是通过一个调用函数 `navigateTo()` 完成的。其中表编号作为一个参数传递过去。两个样式表都生成了同样的调用 `navigateTo()` 的指令。

这种特性使我们在非描述性网络应用程序中会遇到一点设计上的问题，用户可能从右边的表单开始编辑图文框的信息，然后没有存储他们所做的修改就移动到另一个表单上去。为了避免这种情况出现，我们在主编辑图文框中采用了一个布尔标识变量（“dirty”）。在这个窗口中的编辑控制将会把 `onKeyDown` 或 `onChange`（取决于哪个适用于控制）中的标识设定为 `true`，以此来标明用户更改过的数据。

`sku`是表单的唯一标识符（表单编号不是唯一的），我们用它来浏览各个表单。所有的浏览过程都是由 `FormTOC.asp` 中的 `navigateTo()` 函数激活的。如同我们刚刚提到的，当XSL文档浏览XML文档时，它们建立了调用 `navigateTo()` 函数的超链接，将正在处理的 `<form>` 结点的SKU传递过去。

`navigateTo()` 函数也用来保证右侧编辑窗口的数据在用户移到另一个表单之前没有被更改，这项函数是通过检查窗口中 `dirty` 变量的值来实现的。如果数据没有被改变或用户确认想转换到其他窗口去，`navigateTo()` 函数将会根据传递给它的SKU来加载新的URL。如果数据被更改了，我们仅仅是警告用户，告诉他们没有存储修改内容。这就让用户可以选择是放弃修改还是继续，或者是接着编辑。

程序清单 15-17

```
function navigateTo(sSKU)  
{  
    // navigate to a new form  
    var approved = true;
```

```

if (window.parent.mainbody.dirty)
{
    // the data has changed. Raise a warning.
    msg = 'You have made changes which have not been saved.';
    msg += '\n\nTo abort your changes, press OK.';
    msg += '\n\nTo resume editing, press Cancel.\n';
    if (!confirm(msg))
    {
        approved = false;
    }
}

if (approved)
{
    // it's OK to move to another record
    sURL = 'edit.asp?sku='+sSKU;
    window.parent.mainbody.location.href = sURL;
}
}

```

15.9 “以名字浏览” XSL样式表

这个文件更复杂一些，因为它多了一项 form type，而by-number XSL样式表则无须考虑这一点。实际上，如果回过头来查阅在本例开始时介绍的 XML文档结构，会发现我们正是想要用这个格式来显示数据的。

在第9章关于转换XML中已经看到，可以用XSL输出标记甚至HTML标记。对于现在讨论的这些数据，我要加上用DHTML实现的折叠和展开所有的标题这个功能。这个功能只与由 list.xsl生成的by name显示界面有关，因此生成折叠/展开按钮的代码就在list.xsl中，这个模板只在XML文档的根部应用一次：

程序清单 15-18

```

<xsl:template match="/">
  <BUTTON id="btnShowAll" ONCLICK="showAll('UL')" style="width:80">
    Show All
  </BUTTON>
  <BUTTON id="btnHideAll" ONCLICK="hideAll('UL')" style="width:80">
    Hide All
  </BUTTON>
  <hr />
  <UL><xsl:apply-templates select="FORMLIST/FORMS" /></UL>
</xsl:template>

```

上述过程生成了下面的格式化HTML：

程序清单 15-19

```

<BUTTON id="btnShowAll" ONCLICK="showAll('UL')" style="width:80">
  Show All
</BUTTON>
<BUTTON id="btnHideAll" ONCLICK="hideAll('UL')" style="width:80">
  Hide All

```

```

</BUTTON>
<hr />
<UL>the list of form generated by formListxml.asp</UL>

```

下一个模板完成了大部分工作。这个模板应用于每个 <FORMS> 结点，而你只需要记住表单的类型。在解释的HTML中每种表单的类型都成为属于这种类型的一系列的“父”结点。子结点用HTML 标识符建立，从而使它们具有明显的显示标志。

同第一个样式表一样，这个样式表也建立了一个调用 navigateTo() 函数的超链接：

程序清单 15-20

```

<xsl:template match="FORMS">
  <LI CLASS="clsHasChildren">
    <xsl:value-of select="@TYPE" />
    <UL>
      <xsl:for-each select="FORM" order-by="FORMTITLE">
        <LI>
          <A>
            <xsl:attribute name="HREF">
              JavaScript:navigateTo('<xsl:value-of select="SKU" />');
            </xsl:attribute>
            <xsl:value-of select="FORMTITLE" />
          </A>
        </LI>
      </xsl:for-each>
    <xsl:if test="FORMS"><xsl:apply-templates /></xsl:if>
  </UL>
</LI>
</xsl:template>

```

15.10 激活XSL样式表

讨论过XSL文档后，下面来看看调用XSL文档的函数。

如果按下 by name 或 by number 按钮，系统将会调用 changeXSL() 函数，其中 xsl doc 参数指明应该用哪个XSL文档解释XML数据。changeXSL() 将调用在样式变量中保存的XML DOM里的load() 函数，从而加载XSL文档。如果我们要更换到 by name 视图，开始时所有的父结点都折叠在一起：

程序清单 15-21

```

function changeXSL(xsl doc)
{
  // load a new XSL
  styleURL = xsl doc;
  style.load(styleURL);
  update();

  if (xsl doc=='<%=ByNameURL%>')
  {
    hideAll('UL');
  }
}

```

changeXSL() 函数完成所有的工作，我们正是在这里进行的 XSL 转换。我们读取 XML 文档并用 XSL 转换它。生成的 HTML 字符串被赋给 xslresult <DIV>。浏览器将重新解释刚刚被更改的 <div>，这个 <div> 将显示 XML 文档中的格式化内容。

程序清单 15-22

```
function update()
{
    // apply the XSL
    if (style.documentElement && source.documentElement)
    {
        document.all.item('xslresult').innerHTML = source.transformNode(style);
    }
}
```

15.11 问题：保持树的同步

上面提出的解决方案如果仅用于浏览效果还很不错，但这是一个维护性应用程序，因此要用于编辑数据。这意味着 XML 文档中的数据可以被更改，而原有的 XML 文档被废弃了。

对待这个矛盾有两种解决方法：或者我们得到一份新的数据拷贝，让 ASP 重新生成 XML 文档，或者我们利用 XML DOM 的功能更新客户端的 XML 文档拷贝。

我选择用客户端脚本来保持客户的 XML 拷贝随时更新。我们需要考虑的改变是：

- 添加（新表）。
- 删除。
- XML 文档中一个或多个域的改变。

要记住先在客户端进行更改，然后更新服务器上的数据，最后我们需要激活客户端脚本来反映这些变化。Site Server 在应用程序中对更改的数据进行了存储后，我们在其上增加了函数调用指令，使客户端的 XML 进行更新。这个过程在 edit.asp 中完成。

这个设计中有一点风险，因为这是一个非事物系统。有可能发生了对数据的更改，但不知什么原因没有注意到这个变化。那么我们的 XML 数据与基础数据库数据就不会同步了。我认为在这里这是一个可以接受的风险，因为它发生的可能性很小，而且这个系统也不是一个具有关键使命的系统，如果非常偶然地出现了数据不同步，问题也可以通过刷新浏览器来“解决”。

为了更新 XML 文件我们需要在客户端上添加 3 个函数，每个函数对应上面列出的一种情况。add() 和 change() 函数接收一个叫 oForm 的 JavaScript 对象，这个对象是为传递表单属性而生成的（如果你对 JavaScript 不熟悉，可以将函数看作是对象，可以通过简单的赋值向对象添加方法）。每个 oForm 对象有下列属性：

- FormTitle。
- FormNumber。
- SKU。

你会注意到这些属性同我们在 XML 中用的一样。更改过的 Site Server 代码生成了 oForm 对象并将其作为一个参数传给 add() 和 change() 函数。

addNode()函数在XML树上添加一个新表单。它通过复制一个已存在的结点来完成这个过程，并用我们传递的 oForm对象中的新值来更新结点。模式匹配字符串 //FORM选择第一个<FORM>结点：

程序清单 15-23

```
function addNode(oForm)
{
    // add a new node into the tree.
    // pick a node to use as a template...
    var newNode = root.selectSingleNode("//FORM").cloneNode(true);
    var insertPoint, newBranch;

    newNode.selectSingleNode("FORMTITLE").text = oForm.formTitle;
    newNode.selectSingleNode("FORMNUMBER").text = oForm.formNumber;
    newNode.selectSingleNode("SKU").text = oForm.sku;
    ...
}
```

一旦已经有了这个结点，我们通过在表单类型中查找可以找到树中要插入新结点的位置。如果不能找到相应的位置，就需要生成一个新的分支。有可能发生的情况是：存在一些表单的类型，但在我们生成XML时还没有这种类型的表单（例如，在类型表单中有一个 Application类型，但没有表单标识为 Application类型的表单）。

当我们确定了插入点以后，将新结点作为那个点的第一个子结点。模式匹配字符串 //FORM[@TYPE = '"+oForm.formType+"']发现带 type属性的第一个<FORM>结点的值为 oForm.formType：

程序清单 15-24

```
...
// find the node that corresponds to the parent for this form type
insertPoint = root.selectSingleNode("//FORMS[@TYPE='"+ oForm.formType + "']");

if (insertPoint==null)
{
    // couldn't find that form type, create a branch for it
    insertPoint = addBranch(oForm.formType);
}
insertPoint.appendChild(newNode);

update();
}
```

addBranch()函数负责在XML树中插入一个新的分支。它从对<FORMS>结点进行复制这样一个浅层次开始（仅复制一个结点，不复制结点上的子元素），然后将我们将TYPE属性的值设定为传递来的字符串，并把这个新结点添加到根元素 FORMLIST上：

程序清单 15-25

```
function addBranch(sType)
{
    var newBranch = root.selectSingleNode("//FORMS").cloneNode(false);
    newBranch.setAttribute("TYPE", sType);
}
```



```

root.selectSingleNode("//FORMLIST").appendChild(newBranch);

return newBranch;
}

```

更新一个结点同添加一个结点很相似，但我们需要考虑和解决一个新的逻辑错误。表的类型有可能被改变，如果这样的话我们所要做的就不仅仅是更新一个已知的结点，而是要去掉旧结点并在XML树中某个新的父结点下重新生成这个结点：

程序清单 15-26

```

function updateNode(oForm)
{
    // update an existing node
    var theNode = root.selectSingleNode('//FORM[SKU="' + oForm.sku + '"]');
    if (theNode==null)
    {
        // Should never get here. This would mean "node not found", so if
        // that happens just reload the tree
        parent.location.reload();
    }
    else
    {
        theNode.getElementsByTagName("FORMTITLE").item(0).text = oForm.formTitle;
        theNode.getElementsByTagName("FORMNUMBER").item(0).text = oForm.formNumber;

        var oldType = theNode.parentNode.attributes.item(0).text;
        var newType = oForm.formType;

        if (newType!=oldType)
        {
            // The form type has changed. We need to move it.

            // First create a clone...
            var theNewParent =
                root.selectSingleNode('//FORMS[@TYPE="' + newType + '"]');

            if (theNewParent==null)
            {
                // couldn't find that form type, create a branch for it
                theNewParent = addBranch(newType);
            }

            theNewParent.appendChild(theNode.cloneNode(true));

            // Now delete the original one...
            theNode.parentNode.removeChild(theNode);
        }
        update();
    }
}

```

最简单的更新函数是删除函数。我们需要做的就是找到要删除的结点将它从树上去掉。模式字符串 `//FORM[FORMNUMBER=" " + sFormno + ""]` 表示“找到一个FORM结点，其子元素FORMNUMBER的值为sFormno”：

```
function deleteNode(sFormno)
{
    // remove a node from the tree
    var theNode = root.selectSingleNode('//FORM[FORMNUMBER="' + sFormno + '"]');
    if (theNode==null)
    {
        // Should never get here. This would mean "node not found", so if
        // that happens just reload the tree
        document.location.reload();
    }
    else
    {
        theNode.parentNode.removeChild(theNode);
        update();
    }
}
```

注意当这些函数完成它们的任务后，更新 XML树的函数也调用update() 函数，update()反过来又重新将当前的样式表用于已经更新的XML文档。

15.12 小结

在这个实例研究中，我们用 XML为终端用户提供了一个强化的功能界面，而这是用过去的技术所无法实现的。

我们面临的问题是：产生的用户界面要允许用户用两种标识符中的一个 名字或编号来浏览同一部分数据。我们将数据作为 XML送到客户端一次，而不是每次想改变浏览方式时都要与服务器进行数据交换，然后由客户端根据我们的需求用 XML生成想要的显示方式。

我们也允许用户更新他们正在浏览的数据 这些更新不仅仅在客户端生效，而且也要在服务器上起作用 我们再一次利用了客户工作站的能力，通过添加、删除和改变结点来实时更新XML文档。

总而言之，我们实现了设计目标，整个项目是成功的。如果你仔细想想，这个应用在一些情况下可能是非常有用的。毕竟，我们不太可能具有指向同一物理实体的几种不同方式，但我们可以为用户提供数据的不同显示方式。利用相似的技术我们可以允许用户重新归类数据，许多时候我们都可能需要这种操作，从本例中的对图书细节进行归类（即以题目、 ISBN或作者等分类）到提供足球明星的各种相关信息（以姓名、编号、他们在首发阵容中出现的次数等归类），每种操作都允许客户以他们想要的形式浏览数据，并且无须再一次连接网络服务器和重新下载数据。