第55章

Web 服务和 ASP.NET

本章的内容如下：

- SOAP 和 WSDL 的语法
- 如何通过 Web 服务使用 SOAP 和 WSDL
- 提供和使用 Web 服务
- Web 服务的用法。
- 使用 SOAP 标题交换数据

Web 服务是利用 SOAP(Simple Object Access Protocol，简单对象访问协议)在 HTTP 上执行远程方法调用的一种新方法。过去这个问题一直非常棘手，因为使用过任何 DCOM(分布式 COM)的人们，在实例化远程服务器上的对象、调用方法和获取结果时感到非常麻烦，并且在进行必要的配置时，需要具有很高的技巧。

SOAP 的出现使事情变得简单多了。SOAP 技术是一个基于 XML 的标准，它详细描述了怎样在 HTTP 上以可重复的方式进行方法调用。远程 SOAP 服务器能够理解这些调用并执行所有困难的工作，如实例化所需的对象、进行调用以及给客户端返回 SOAP 格式的响应等。

通过.NET Framework，可以非常容易地利用上述技术。与 ASP.NET 一样，我们可以在服务器上使用完整的 C#和.NET 技术，而且(也许是更重要的)可以从任何平台上通过 HTTP 访问服务器，从而实现的 Web 服务的简单利用。换句话说，例如，Linux 代码就可以使用.NET Web 服务，或者 Internet 启用的电冰箱。过去作者就曾经成功地把 ASP.NET Web 服务和 Macromedia Flash 组合在一起，创建启用数据的 Flash 内容。

此外，也可以使用 WSDL(Web Service Description Language, Web 服务描述语言)完整地描述 Web 服务，还可以在运行期间动态地查找 Web 服务。WSDL 使用带有 XML 架构的 XML 提供对所有方法的描述(以及对调用方法所需类型的描述)。现在各式各样的类型可用于 Web 服务，既有简单的基元类型，又有完整的 DataSet 对象，这样，完全存储在内存中的数据库就可以被编组到客户端，从而大大减少加数据库服务器上加载的数据量。

注意，本章讨论的是 ASP.NET Web 服务，而不是 WCF Web 服务，后者是近期才添加到.NET 中。ASP.NET Web 服务使用起来比较简单，足以满足大多数需要，而 Windows Communication Foundation(WCF) Web 服务包含 ASP.NET Web 服务的全部功能，还添加了额外的功能。WCF 详解第 43 章。

55.1 SOAP

如前所述，SOAP 是一个与 Web 服务交换数据的方法。有关这项技术的书有很多，尤其是微软公司决定在.NET Framework 中采用这项技术之后，SOAP 方面的书就更多了。稍微考虑一下，可以发现 SOAP 的工作原理和 HTTP 的工作原理比较相似，这非常有趣，但并不是必需的。大多数情况下，我们不必考虑与 Web 服务进行交换时所采用的格式，只要得到希望的结果就够了。

因此，本节不深入探讨 SOAP 的技术细节，而是给出一些简单的 SOAP 请求和响应，以便您对 SOAP 有一个感性的认识。

假定要用下面的签名调用 Web 服务中的方法：

```
int DoSomething(string stringParam, int intParam)
```

这条语句必需的 SOAP 标题和主体如下所示，最上面是 Web 服务的地址：

```
POST /SomeLocation/myWebService.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/DoSomething"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <DoSomething xmlns="http://tempuri.org/">
         <stringParam>string</stringParam>
         <intParam>int</intParam>
      </DoSomething>
   </soap:Body>
</soap:Envelope>
```

length 参数用于指定内容的总字节数，它的大小随着 string 和 int 参数中发送的值而变化。Host 也是变化的，它取决于 Web 服务的位置。

上面代码引用的 soap 名称空间定义用于构建消息的各种元素。通过 HTTP 发送上面的代码时，实际发送的数据将有所不同(但是相关)。例如，可以使用简单的 GET 方法调用上面的方法：

```
GET /SomeLocation/myWebService.asmx/DoSomething?stringParam= string & intParam= int
   HTTP/1.1
Host: hostname
```

这个方法的 SOAP 响应如下：

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```
    <soap:Body>
      <DoSomethingResponse xmlns="http://tempuri.org/">
       <DoSomethingResult>int</DoSomethingResult>
      </DoSomethingResponse>
    </soap:Body>
  </soap:Envelope>
```

其中 length 参数的值根据 int 参数值的变化而改变。

此外，通过 HTTP 的实际响应也比较简单，例如：

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0"?>
<int xmlns="http://tempuri.org/">int</int>
```

这是一种比较简单的 XML 格式。

如本节开始时所讨论的，有许多语法问题可以完全忽略。只有在需要考虑语法时，语法才会变得很重要。

55.2　WSDL

WSDL 可以完整地描述 Web 服务、可用的方法，以及调用这些方法的各种方式。此外，虽然过多地讨论 WSDL 的细节对我们并没有太多的好处，但对 WSDL 的总体理解却非常有用。

WSDL 是另一种与 XML 完全兼容的语法。WSDL 通过可用的方法、这些方法所使用的类型、通过各种协议(纯 SOAP、HTTP GET 等)发送给方法的请求消息和从方法中发送出的响应消息的格式，以及上面规范的各种绑定，指定 Web 服务。WDSL 由各种客户端读取，而不只是.NET，还有本章前言提及的 Macromedia Flash。

WSDL 文件中最重要的部分或许是类型定义部分。这一部分使用 XML 架构描述数据交换的格式，数据交换的格式要通过可使用的 XML 元素和元素之间的关系定义。

例如，上一节中的示例所使用的 Web 服务方法：

```
int DoSomething(string stringParam, int intParam)
```

下面是为请求所做的类型声明：

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
             ...other namespaces...>
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
              targetNamespace="http://tempuri.org/">
      <s:element name="DoSomething">
        <s:complexType>
```

```
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="stringParam"
                     type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="intParam"
                     type="s:int" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="DoSomethingResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
                     type="s:int" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
 </wsdl:types>
 ...other definitions...
</definitions>
```

这些类型都是以前我们看到的 SOAP 和 HTTP 请求及响应所必需的，并且这些类型被绑定在文件中的后期操作上。所有这些类型都使用标准的 XML 架构语法指定，例如：

```
<s:element name="DoSomethingResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
                 type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
```

这个示例指定一个 DoSomethingResponse 元素，该元素包含一个子元素 DoSomething Result，这个子元素包含了一个整数。这个整数必须出现 1 次，即它必须包含在内。

如果可以访问 Web 服务的 WSDL，就可以使用 WSDL。不久我们可以看到，对 WSDL 的使用并不困难。

上面对 SOAP 和 WSDL 进行简短的讨论，接下来讨论如何创建和使用 Web 服务。

55.3 Web 服务

Web 服务的讨论分为两个方面：

- 创建 Web 服务，这一部分主要讨论如何编写 Web 服务和如何把它们放在 Web 服务器上。
- 使用 Web 服务，这一部分主要讨论如何在客户端应用程序中使用 Web 服务。

55.3.1 提供 Web 服务

把代码直接放到.asmx 文件中或者从这些文件中引用 Web 服务类，都可以提供 Web 服务。如同 ASP.NET 页面一样，在 Visual Studio .NET 中创建 Web 服务也使用后一种方法，目的是把问题讲述

得更清楚一些。

如图 55-1 所示，使用 File | New | Web Site 命令在 C:\ProCSharp\Chapter55 目录下创建一个 Web 服务项目 PCSWebService1，此时系统会生成一组类似的文件，它们与创建 Web 应用程序项目时所生成的一组文件相似，其位置选项也相同。实际上，唯一的区别就是创建 Web 应用程序时生成的文件是 Default.aspx，而创建 Web 服务项目时生成的文件是 Service.asmx，其代码隐藏是 App_Code/ Service.cs。

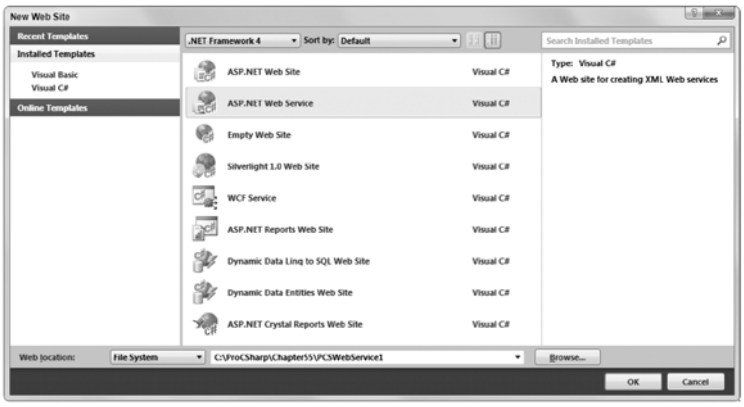

图　55-1

Service.asmx 中的代码如下所示：

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.cs" Class="Service" %>
```

它引用代码文件/App_Code/ Service.cs。下面的程序清单是生成的代码：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class Service : System.Web.Services.WebService
{
    public Service()
    {
        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

代码段 PCSWebService1\App_Code\Service.cs

这段代码包含几个标准名称空间引用，并定义 Web 服务类 Service(它在 Service.asmx 中引用)，Service 类继承自 System.Web. Services.WebService。WebService 属性指定 Web 服务的名称空间，它允许客户端应用程序区分不同 Web 服务中同名的 Web 服务方法。WebServiceBinding 属性与 Web 服务的交互操作性相关，如 WS-I Basic Profile 1.1 规范中的定义所示。简言之，这个属性可以声明，Web 服务为一个或多个 Web 方法支持标准的 WSDL 描述，或者像这个示例一样，指定一组新的 WDSL 定义。其中还有一个注释掉的 ScriptService 属性，如果取消注释，就可以使用 ASP.NET AJAX 脚本调用 Web 方法。现在可以在这个 Web 服务类上提供其他方法。

在添加可以通过 Web 服务访问的方法时，只需要把方法定义为 public，并给方法提供 WebMethod 属性。这个属性仅把方法标记为可通过 Web 服务访问。稍后将会学习返回类型和参数使用的类型，现在用下面的方法替换自动生成的 HelloWorld()方法。

```
[WebMethod]
public string CanWeFixIt()
{
   return "Yes we can!";
}
```

现在编译该项目。

要检查是否一切正常工作，可用 Ctrl+F5 组合键运行应用程序，就会进入 Web 服务的测试页面，如图 55-2 所示。

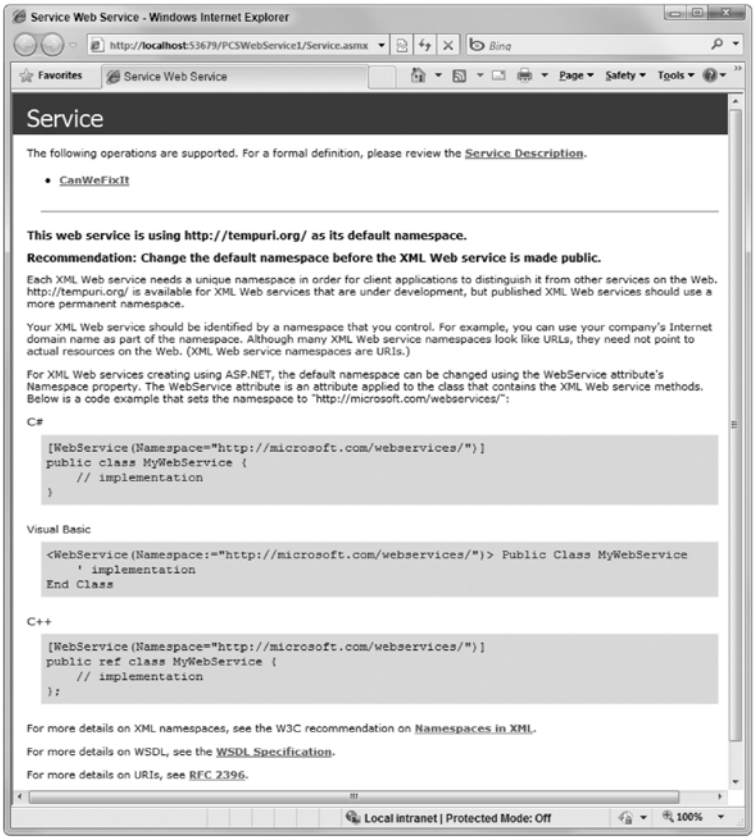

图 55-2

注意，这个测试页面默认仅可用于本地计算机的调用者，即使 Web 服务驻留在 IIS(Internet Information Services，Internet 信息服务)中，也是如此。

在浏览器中显示的大多数文本都说明把 Web 服务的名称空间设置为 http://tempuri.org/。这在开发过程中不是问题，尽管以后应修改它(如 Web 页面中的文本所示)。为此可以使用 WebService 属性，但目前可以不修改它。

单击方法名称，可以得到 SOAP 请求和响应的信息，此外，还可以得到一个示例，说明了如何通过 HTTP GET 和 HTTP POST 方法获得请求和响应。另外，也可以单击 Invoke 按钮，对方法进行测试。如果方法需要简单的参数，那么在这个窗体中也可以输入这些参数(如果方法需要较复杂的参数，这个窗体就不允许以这种方式测试方法)。这样，就可以看到方法调用所返回的 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://tempuri.org/"> Yes we can! </string>
```

这说明方法运行良好。

单击图 55-2 的浏览器屏幕上的 Service Description 链接，可以查看 Web 服务的 WSDL 描述。其中最重要的部分是关于请求和响应的元素类型的描述：

```
<wsdl:types>
  <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
    <s:element name="CanWeFixIt">
      <s:complexType />
    </s:element>
    <s:element name="CanWeFixItResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CanWeFixItResult"
                     type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
```

该描述文件比较长，除了包含服务的各种绑定之外，还可以包含请求和响应所需类型的描述。

1. 对于 Web 服务可用的类型

Web 服务可以用于交换表 55-1 中所示的类型。

表　55-1

String	Char	Byte
Boolean	Int16	Int32
Int64	UInt16	UInt32
UInt64	Single	Double

(续表)

Guid	Decimal	DateTime
XmlQualifiedName	class	struct
XmlNode	DataSet	enum

以上所有类型的数组都可以使用，因为它们都是泛型集合类型，如 List<string>。还要注意，只能编组 class 和 struct 类型的公共属性和字段。

55.3.2 使用 Web 服务

既然明白了如何创建 Web 服务，接下来就讨论如何使用它们。为此需要在代码中生成一个知道如何与给定 Web 服务进行通信的代理类。这样，代码中对 Web 服务进行的任何调用都要通过这个代理类，从表面看，这个代理类就等同于 Web 服务，代码也会认为我们有了 Web 服务的本地副本。而实际的情况是有许多 HTTP 通信工作在进行，只是对我们屏蔽了其中的细节。有两种方式可以完成这项任务：第一，可以使用 WSDL.exe 命令行工具；第二，可以使用 Visual Studio .NET 中的 Add Web Reference 菜单选项。

从命令行中使用 WSDL.exe 时，它会根据 Web 服务的 WSDL 描述生成一个包含代理类的.cs 文件。使用 Web 服务的 URL 来指定该文件，例如：

```
WSDL http://localhost:53679/PCSWebService1/Service.asmx?WSDL
```

注意，这里和随后的例子使用默认的文件系统驻留 Web 应用程序。为了使上面的 URL 起作用，Web 服务的 Visual Web Developer Web Server 必须正在运行。这仍不能保证 Web 服务的端口号(在这里是 61968)仍相同。这适合于演示，因为一般我们希望 Web 服务驻留在固定的 Web 服务器(如 IIS)上，否则就必须继续重新生成代理类。确保 Web 服务可用于测试的一种方式是在一个解决方案中包含多个 Web 站点。

这样就会在 Service.cs 文件中为上一节中的示例生成一个代理类。这个代理类将以 Web 服务命名，在这个示例中就是 Service，该代理类包含一些方法，那些方法将可以调用与服务同名的方法。在使用这个类时，只需把所生成的.cs 文件添加到项目中，并在代码中使用下面的代码行：

```
Service myService = new Service();
string result = myService.CanWeFixIt();
```

默认情况下，生成的类将放在根名称空间中，因此不需要使用 using 语句，但是，可以使用 WSDL.exe 命令行选项/n:<namespace>指定一个不同的名称空间。

虽然这项技术非常有效，但是，如果服务正处于开发或处于连续更改中，就比较麻烦。当然，为了在每次编译之前自动更新所生成的代理，这项技术可以用项目的构建选项执行，但是我们有更好的方法。

在一个新的空白网站 PCSWebClient1 中，为上一节中的示例创建客户端(在 C:\ProCSharp\Chapter55

目录下)，阐明这个更好的方法。现在创建这个项目，添加 Default.aspx 页面，并在 Default.aspx 页面中添加下面的代码：


```
<form id="form1" runat="server">
   <div>
      <asp:Label Runat="server" ID="resultLabel" />
      <br />
      <asp:Button Runat="server" ID="triggerButton" Text="Invoke CanWeFixIt()" />
   </div>
</form>
```

代码段 PCSWebClient1\Default.aspx

接下来将把单击按钮事件处理程序绑定到 Web 服务。首先需要在项目中添加对 Web 服务的引用。其方法是：右击 Solution Explorer 窗口中的新客户端项目，选择 Add Web Reference 选项。然后，在出现的窗口中输入 Web 服务文件 Service1.asmx 的 URL，或者使用本地计算机链接中的 Web 服务，自动查找它，如图 55-3 所示。

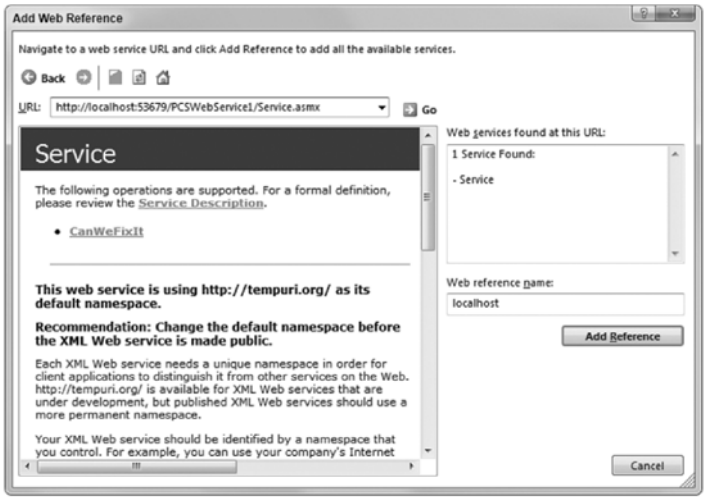

图　55-3

接着，可以使用 Add Reference 按钮添加引用。但应先把 Web 引用的默认内容从 localhost 改为 myWebService。现在单击 Add Reference 按钮，在 Solution Explorer 窗口中把 myWebService 添加到该项目的 App-WebReferences 文件夹中。如果在 Solution Explorer 窗口中查看这个文件夹时，就可以看到 Service.disco、Service.discomap 和 Service1.wsdl 文件已添加到项目中。

Web 引用名(myWebService)也是使用代理类需要引用的名称空间。在 Default.aspx.cs 的代码中添加下面的 using 语句：

```
using myWebService;
```

现在就可以在类中使用服务，而不必完全限定它的名称。

在设计视图中双击该按钮，添加下面的代码，把事件处理程序添加到窗体上的按钮中：

```
protected void triggerButton_Click(object sender, EventArgs e)
{
```

```
        Service myService = new Service();
        resultLabel.Text = myService.CanWeFixIt();
    }
```

运行应用程序并且单击该按钮，浏览器窗口将显示 CanWeFixIt()方法的执行结果。

如果使用 ASP.NET Development Server(即 Web 应用程序驻留在本地文件系统上，而没有驻留在 IIS 上)，就会产生 401:Unauthorized 错误。这是因为，默认情况下，这个服务器配置为需要 NTLM 身份验证。为了修正这个错误，可以在 PCSWebService1 的属性页面的 Start Options 页面上取消选择 NTLM Authentication 复选框，或者在调用 Web 服务方法时传递默认的证书。后一种方式需要以下代码:

```
myService.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

以后这个 Web 服务也许会更改，但是，使用这个方法，可以只右击 Server Explorer 窗口中的 App_WebReference 文件夹，并选择 Update Web/Service Reference 命令，生成一个新的代理类，以供使用。

另外，Web 服务以后可能在部署它时会移动。如果查看客户端应用程序的 web.config 中，就会看到如下代码：

可从 wrox.com 下载源代码

```
    <appSettings>
    <add key="myWebService.Service"
value="http://localhost:53679/PCSWebService1/Service.asmx"/>
    </appSettings>
```

代码段 PCSWebClient1\web.config

由于这个设置配置把 Web 请求发送到什么地方，因此必须确保它匹配 Web 服务的位置，或者使用其他方式记录这个信息。

55.4 扩充事件登记示例

既然我们已经对 Web 服务的创建和使用有了一定的了解，接下来要应用这些知识扩充第 40 章中的会议室登记应用程序。具体来说，就是从应用程序中提取数据库访问方面的内容，把它们放到一个 Web 服务中。这个 Web 服务有两个方法：

- GetData()，它返回一个 DataSet 对象，DataSet 对象包含 PCSDemoSite 数据库中的 3 个表。
- AddEvent()，因为它添加一个事件，返回受影响的行数，所以客户端应用程序可以检查数据是否发生变化。

此外，使用能够减少加载量的技术设计 Web 服务。具体来说，就是把包含会议室登记数据的 DataSet 对象存储在 Web 服务应用程序中的应用层上。这意味着对数据的多个请求将不需要进行额外的数据库请求工作。这样，只有在数据库中添加新的数据时，才会刷新应用层 DataSet 中的数据。通过其他方式对数据库的更改(如手动编辑)就不会反映在 DataSet 对象中。而且，只有知道 Web 服务是唯一可以直接访问数据的应用程序，这样我们就没有什么可担心的。

55.4.1　事件登记 Web 服务

在 Visual Studio 中，在 C:\ProCSharp\Chapter55 目录下创建一个名为 PCSWebService2 的新 Web 服务项目。首先，把第 41 章代码的 PCSDemoSite 中的数据库文件(MeetingRoomBooker.mdf)复制到 Web 服务的 App_Data 目录下。接着，需要给项目添加 Global.asax 文件，然后在它的 Application_Start() 处理程序中更改代码。把 MeetingRoomBooker 数据库中的所有数据都加载到数据集中并存储它。其中涉及的大部分代码前面已讨论过，因为前面已经把数据库加载到 DataSet 中。如本章前面所示，也可以使用存储在 web.config 文件中的连接字符串。web.config 文件的代码如下所示(连接字符串应放在单独一行上)：


```
<?xml version="1.0" ?>
 <configuration>
   ...
   <connectionStrings>
      <add name="MRBConnectionString"
        connectionString="Data Source=.\SQLExpress;Integrated
           Security=True;AttachDBFilename=|DataDirectory|MeetingRoomBooker.mdf;
           User Instance=True"
        providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

代码段 PCSWebService2\web.config

在 Global.asax 文件的 Application_Start()处理程序中，添加如下代码：


```
void Application_Start(Object sender, EventArgs e)
{
   System.Data.DataSet ds;
   System.Data.SqlClient.SqlConnection sqlConnection1;
   System.Data.SqlClient.SqlDataAdapter daAttendees;
   System.Data.SqlClient.SqlDataAdapter daRooms;
   System.Data.SqlClient.SqlDataAdapter daEvents;

   using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
   {
      sqlConnection1.ConnectionString =
         ConfigurationManager.ConnectionStrings["MRBConnectionString"]
            .ConnectionString;

      sqlConnection1.Open();

      ds = new System.Data.DataSet();
      daAttendees = new System.Data.SqlClient.SqlDataAdapter(
                     "SELECT * FROM Attendees", sqlConnection1);
      daRooms = new System.Data.SqlClient.SqlDataAdapter(
                     "SELECT * FROM Rooms", sqlConnection1);
      daEvents = new System.Data.SqlClient.SqlDataAdapter(
                     "SELECT * FROM Events", sqlConnection1);

      daAttendees.Fill(ds, "Attendees");
      daRooms.Fill(ds, "Rooms");
      daEvents.Fill(ds, "Events");
   }
```

```
        Application["ds"] = ds;
    }
```

代码段 PCSWebService2\Global.asax

这里需要注意的最重要的代码在最后一行中。通常，Application(类似 Session)对象都有一个名/值对的集合，可以在该集合中存储数据。这里在 Application 存储器中创建一个名称(ds)，它从数据库中提取 ds 的序列化值，其中 ds 包含 Attendees、Rooms 和 Events 表。这样，Web 服务对象的所有实例在任何时间都可以访问这些值。

这项技术非常适合用于只读数据，因为多个线程可以访问它，减少了在数据库上的负载。但要注意，由于 Events 表有可能发生变化，因此在 Events 表发生变化时，必须更新应用层的 DataSet 类。稍后会介绍这一内容。

接下来用一个新的服务(MRBService)替换默认的 Service 服务。为此，删除已有的 Service. asmx 和 Service.cs 文件，在项目中添加一个新的 Web 服务，命名为 MRBService(确保选择把代码放在一个单独的文件中)。接着把 GetData()方法添加到 MRBService.cs 的服务中：

可从 wrox.com 下载源代码

```
[WebMethod]
public DataSet GetData()
{
    return (DataSet)Application["ds"];
}
```

代码段 PCSWebService2\App_Code\MRBService.cs

上面代码使用与 Application_Load()方法相同的语法访问存储的 dataset，这样，就可以仅将数据强制转换为正确的类型，并返回。

注意，为了使上述代码正常工作，并便于以后添加的其他 Web 方法正常工作，可以添加下面的 using 语句：

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Services;
```

AddEvent()方法稍微有点复杂。从概念上讲，需要做下面的事情：

- 接受来自客户端的事件数据
- 使用这些数据创建 SQL INSERT 命令
- 连接到数据库并且执行 SQL 语句
- 如果添加成功，就需要刷新 Application["ds"]中的数据
- 把成功或失败的通知返回给客户端(如果有必要，客户端就会刷新它的 DataSet)

从现在开始，就要接受正确数据类型的所有字段：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                    string eventAttendees, DateTime eventDate)
{
```

```
    ...
}
```

下面声明数据库访问需要的对象，连接到数据库并执行查询，完成这些工作使用的所有代码与 PCSDemoSite 中的代码相似(记住，这里也需要连接字符串，从 web.config 中提取它)：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                    string eventAttendees, DateTime eventDate)
{
   System.Data.SqlClient.SqlConnection sqlConnection1;
   System.Data.SqlClient.SqlDataAdapter daEvents;
   DataSet ds;

   using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
   {
      sqlConnection1.ConnectionString =
         ConfigurationManager.ConnectionStrings[“MRBConnectionString”]
            .ConnectionString;

      System.Data.SqlClient.SqlCommand insertCommand =
         new System.Data.SqlClient.SqlCommand(
            "INSERT INTO [Events] (Name, Room,"
            + "AttendeeList, EventDate) VALUES (@Name, @Room, @AttendeeList,"
            + "@EventDate)", sqlConnection1);
      insertCommand.Parameters.Add("Name", SqlDbType.VarChar, 255).Value
         = eventName;
      insertCommand.Parameters.Add("Room", SqlDbType.Int, 4).Value
         = eventRoom;
      insertCommand.Parameters.Add("AttendeeList", SqlDbType.Text, 16).Value
         = eventAttendees;
      insertCommand.Parameters.Add("EventDate", SqlDbType.DateTime, 8).Value
         = eventDate;

      sqlConnection1.Open();
      int queryResult = insertCommand.ExecuteNonQuery();
   }
}
```

同前，使用 queryResult 存储受查询影响的行数。如果 queryResult 的结果是 1，则说明查询成功，然后就可以对数据库进行新的查询，以便刷新 DataSet 中的 Events 表。在执行更新时，必须锁定应用程序数据，确保在更新过程中其他线程不可以访问 Application["ds"]。使用 Application 对象的 Lock() 和 Unlock()方法，可以实现对数据的锁定和解锁：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                    string eventAttendees, DateTime eventDate)
{
     ...
     int queryResult = insertCommand.ExecuteNonQuery();
     if (queryResult == 1)
     {
        daEvents = new System.Data.SqlClient.SqlDataAdapter(
                      "SELECT * FROM Events", sqlConnection1);
```

```
            ds = (DataSet)Application["ds"];
            ds.Tables["Events"].Clear();
            daEvents.Fill(ds, "Events");
            Application.Lock();
            Application["ds"] = ds;
            Application.UnLock();
        }
    }
}
```

最后，返回 queryResult，以便让客户端知道查询是否成功：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                    string eventAttendees, DateTime eventDate)
{
   ...
   return queryResult;
}
```

至此，就完成了 Web 服务。同前，在 Web 浏览器中查看.asmx 文件，可以测试 Web 服务，这样，不用编写任何客户端代码，就可以添加记录，并查看 GetData()方法返回的 DataSet 的 XML 表示。

在继续之前，需要讨论 DataSet 对象和 Web 服务的组合使用。初看起来这似乎是交换数据的一种荒谬方式，而实际上这是一种极其有用的技术。然而，DataSet 类的用途非常广泛，该事实有隐含意义。如果查看为 GetData()方法生成的 WSDL，就会看到如下代码：

```
<s:element name="GetDataResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="GetDataResult">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>
```

可以看出，这是非常通用的代码，允许传递的 DataSet 对象包含用内联架构指定的任何数据。但是，这表示 WSDL 没有完整地描述 Web 服务。对于.NET 客户端这不是个问题，在前面的示例中传递简单的字符串时，一切都很正常，唯一的区别是我们交换了一个 DataSet 对象。但是，非.NET 客户端必须提前了解要传递的数据，或者某个等价的 DataSet 类，才能访问数据。这包括脚本客户端，例如，使用客户端 ASP.NET AJAX 代码处理已检索数据的客户端。

这个问题的解决方法是把数据重新打包为另一种格式，如结构数组。如果这么做，就可以以任意方式定制生成的 XML，XML 可以由 Web 服务的架构完整地描述。这还可以影响性能，因为传递 DataSet 对象会生成大量 XML，在大多数情况下远远超过了需要的 XML。重新打包数据导致的系统开销比在 Web 上发送数据的系统开销少得多，而且数据较少，序列化和反序列化也较快。所以如果

性能比较重要，就应避免以这种方式使用 DataSet 对象，除非要使用 DataSet 对象提供的其他功能。

但是，这里使用 DataSet 对象不成问题，而且还可以大大简化其他代码。

55.4.2　事件登记客户端

本节使用的客户端是第41章开发的PCSDemoSite网站。下面把这个应用程序命名为PCSDemoSite2，放在C:\ProCSharp\Chapter55目录下，并且使用PCSDemoSite中的代码作为起点。

对项目主要进行两个主要的修改：第一，从应用程序中删除对数据库直接访问的所有内容，并使用 Web 服务代替；第二，引入一个从 Web 服务返回的 DataSet 的应用层存储器，并且只有在必要时才对 DataSet 进行更新，这将大大降低数据库上的负载。

对于新的 Web 应用程序，首先需要添加对 PCSWebService2/MRBService.asmx 服务的 Web 引用。添加的方法在本章前面部分中已经讨论过，即在 Server Explorer 窗口中右击项目，定位.asmx 文件，调用 Web 引用 MRBService，单击 Add Reference 按钮。在此之前，可能还需要在浏览器中查看 Web 服务的.asmx 文件，以启动 ASP.NET Development Server。因为我们不再使用本地数据库，所以还可以从 App_Data 目录中删除它，从 web.config 中删除 MRBConnectionString 项。其余修改操作都在 MRB.ascx 和 MRB.ascx.cs 中进行。

开始时，删除 MRB.ascx 上的所有数据源，并删除当前所有数据绑定控件上的 DataSourceID 项。这是因为我们要在代码隐藏文件中处理数据绑定。

注意，在修改或删除 Web 服务器控件的 DataSourceID 属性时，需要确认是否要删除已定义的模板，因为不能保证控件使用的数据对于这些模板仍旧有效。因为这里要使用相同的数据，只是从另一个数据源中提取，所以应保留模板。如果确定删除了模板，最终的 HTML 布局就会变成默认布局，它看起来不漂亮，于是必须从头添加它们或重写它们。

接着，需要给 MRB.ascx.cs 添加一个属性，用于存储 Web 服务返回的 DataSet。这个属性使用 Application 状态存储器，其方式与 Web 服务中的 Global.asax 相同。代码如下：

可从 wrox.com 下载源代码

```
public DataSet MRBData
{
   get
   {
      if (Application["mrbData"] == null)
      {
         Application.Lock();
         MRBService.MRBService service = new MRBService.MRBService();
         service.Credentials = System.Net.CredentialCache.DefaultCredentials;
         Application["mrbData"] = service.GetData();
         Application.UnLock();
      }
      return Application["mrbData"] as DataSet;
   }
   set
   {
      Application.Lock();
```

```
            if (value == null && Application["mrbData"] != null)
            {
               Application.Remove("mrbData");
            }
            else
            {
               Application["mrbData"] = value;
            }
            Application.UnLock();
         }
      }
```

代码段 PCSDemoSite2\MRB\MRB.ascx.cs

注意需要锁定和解锁 Application 状态，这与 Web 服务中相同。另外，注意只有在需要时才填充 Application["mrbData"]存储器，即在它为空时填充它。这个 DataSet 对象现在可用于 PCSDemoSite2 的所有实例，也就是说多个用户可以读取数据，而无需调用 Web 服务或访问数据库。这里还设置了证书，如前所述，在使用 ASP.NET Development Server 中的 Web 服务时需要证书。如果不需要它，就可以注释掉这行代码。

要绑定 Web 页面上的控件，可以提供 DataView 属性，它映射到存储在这个属性中的数据，如下所示：

```
private DataView EventData
{
   get
   {
      return MRBData.Tables["Events"].DefaultView;
   }
}

private DataView RoomData
{
   get
   {
      return MRBData.Tables["Rooms"].DefaultView;
   }
}

private DataView AttendeeData
{
   get
   {
      return MRBData.Tables["Attendees"].DefaultView;
   }
}

private DataView EventDetailData
{
   get
   {
      if (EventList != null && EventList.SelectedValue != null)
      {
         return new DataView(MRBData.Tables["Events"], "ID=" +
           EventList.SelectedValue.ToString(), "",
```

```
                DataViewRowState.CurrentRows);
            }
            else
            {
               return null;
            }
         }
      }
```

还可以删除已有的 eventData 字段和 EventData 属性。

大多数属性都很简单，只有最后一个属性有点新意。在本例中，要筛选 Events 表中的数据，只获得一个事件，用于在细节视图 FormView 控件中显示。

既然没有使用数据源控件，就必须自己绑定数据。为此，可调用页面的 DataBind()方法，但仍需要为页面上的各种数据绑定控件设置数据源 DataView 属性。一种较好的方式是在 OnDataBinding()事件处理程序的重写版本中设置它，如下所示：

```
protected override void OnDataBinding(EventArgs e)
{
   roomList.DataSource = RoomData;
   attendeeList.DataSource = AttendeeData;
   EventList.DataSource = EventData;
   FormView1.DataSource = EventDetailData;
   base.OnDataBinding(e);
}
```

这里把 roomList、attendeeList、EventList 和 FormView1 的 DataSource 属性设置为前面定义的属性。接着给 Page_Load()方法添加 DataBind()调用：

```
void Page_Load(object sender, EventArgs e)
{
   if (!this.IsPostBack)
   {
      nameBox.Text = Context.User.Identity.Name;
      DateTime trialDate = DateTime.Now;
      calendar.SelectedDate = GetFreeDate(trialDate);
      DataBind();
   }
}
```

此外，还需要更改 submitButton_Click()方法，以使用 Web 服务的 AddData()方法。同样，其中的大部分代码保持不变，只有数据添加代码需要更改：

```
void submitButton_Click(object sender, EventArgs e)
{
   if (Page.IsValid)
   {
      ...
      try
      {
         MRBService.MRBService service = new MRBService.MRBService();
         if (service.AddEvent(eventBox.Text, int.Parse(roomList.SelectedValue),
            attendees, calendar.SelectedDate) == 1)
```

```
            {
               MRBData = null;
               DataBind();
               calendar.SelectedDate =
                  GetFreeDate(calendar.SelectedDate.AddDays(1));
            }
         }
         catch
         {
         }
      }
   }
```

事实上，这里所做的全部工作是为了简化事情，因此，当使用设计良好的 Web 服务时，通常可以忽略许多内部工作机制，而应更关注用户体验。

这段代码没有很多注释。继续使用 queryResult 很有帮助，锁定应用程序也是必需的，这些内容在前面曾经提到过。

最后要修改的地方有两处，一处是 EventList_SelectedIndexChanged()方法：

```
protected void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
   FormView1.DataSource = EventDetailData;
   EventList.DataSource = EventData;
   EventList.DataBind();
   FormView1.DataBind();
}
```

这就确保事件列表的数据源和细节视图正确地刷新。

还需要添加 EventList_SelectedIndexChanging()处理程序，它必须关联到 EventList 控件的 SelectedIndexChanging 事件上：

```
protected void EventList_SelectedIndexChanging(object sender,
   ListViewSelectEventArgs e)
{
   EventList.SelectedIndex = e.NewSelectedIndex;
}
```

不添加这个处理程序，代码就会失败，因为我们进行的修改需要这个事件的处理程序。

虽然 PCSDemoSite2 网站从外表和功能上来看与 PCSDemoSite 非常相似，但是 PCSDemoSite2 执行起来会更好一些。很容易把同一 Web 服务用到其他应用程序中，例如，如果添加更多方法，就可以在页面上显示事件，甚至可以编辑事件、与会者姓名和房间等。这样做并不会破坏 PCSDemoSite2，它仅忽略任何新创建的方法。但必须引入新的触发机制，以更新缓存在事件列表中的数据，因为在其他地方修改这些数据会使数据过时。

55.5 使用 SOAP 标题交换数据

本章要讨论的最后一个主题是使用 SOAP 标题交换信息，而不包括方法参数中的信息。讨论这个主题的原因是：这是一个用于维护用户登录的非常好的系统。我们将不详细论述如何设置服务器，

用于 SSL 连接，也不讨论可以使用 IIS 配置的各种身份验证方法，因为这些都不影响交换信息所需的 Web 服务代码。

假定有一个服务包含一个简单的身份验证方法，其签名如下所示：

```
AuthenticationToken AuthenticateUser(string userName, string password);
```

其中 AuthenticationToken 是我们定义的一个类型，它可以在后续方法调用中由用户使用，例如：

```
void DoSomething(AuthenticationToken token, OtherParamType param);
```

一旦用户登录，他就可以使用从 AuthenticateUser()方法中接收到的令牌(token)访问其他方法。尽管常常用较复杂的方式实现它，但这种技术是典型的安全 Web 系统。

使用 SOAP 标题交换标识(或任何其他数据)可以进一步简化这个过程。由于可以对方法进行限制，因此只有在方法调用中包含指定的 SOAP 标题，才调用它们。这就简化了它们的结构，如下所示：

```
void DoSomething(OtherParamType param);
```

其优点是，一旦在客户端上设置了标题，它就一直存在。在设置初始位后，就可以在后续的所有 Web 方法调用中忽略身份验证标识。

为了查看这个过程，在 C:\ProCSharp\Chapter55\目录下创建一个新的 Web 服务项目，命名为 PCSWebService3，在 App_Code 目录下添加一个新类 AuthenticationToken：


可从 wrox.com 下载源代码

```
using System;
using System.Web.Services.Protocols;

public class AuthenticationToken : SoapHeader
{
   public Guid InnerToken;
}
```

代码段 PCSWebService3\App_Code\AuthenticationToken.cs

通常使用 GUID 来标识该令牌，因为这样可以确保它是唯一的。

要声明 Web 服务可以有一个自定义 SOAP 标题，只需给服务类添加一个新类型的公共成员：


可从 wrox.com 下载源代码

```
public class Service : System.Web.Services.WebService
{
   public AuthenticationToken AuthenticationTokenHeader;
```

代码段 PCSWebService3\App_Code\Service.cs

还需要使用 System.Web.Services.Protocols.SoapHeaderAttribute 属性，标记那些需要额外 SOAP 标题才能正常工作的 Web 方法。在添加这样的方法前，应先添加一个非常简单的 Login()方法，客户端可以使用该方法获得身份验证令牌：

```
[WebMethod(true)]
public Guid Login(string userName, string password)
{
   if ((userName == "Karli") && (password == "Cheese"))
   {
      Guid currentUser = Guid.NewGuid();
      Session["currentUser"] = currentUser;
```

```
            return currentUser;
        }
        else
        {
            Session["currentUser"] = null;
            return Guid.Empty;
        }
    }
```

如果使用正确的用户名和密码，就会生成一个新的 Guid 对象，它存储在会话层的变量中，并返回给用户。如果身份验证失败，就返回一个空的 Guid 实例，该 Guid 存储在会话层中。True 参数允许会话状态用于这个 Web 方法，因为在 Web 服务中，默认情况下禁用它，而这个功能需要它。

接着是一个接受标题的方法，该方法用 SoapHeaderAttribute 属性指定：

```
[WebMethod(true)]
[SoapHeaderAttribute("AuthenticationTokenHeader",
                     Direction = SoapHeaderDirection.In)]
public string DoSomething()
{
    if (Session["currentUser"] != null &&
        AuthenticationTokenHeader != null &&
        AuthenticationTokenHeader.InnerToken
        == (Guid)Session["currentUser"])
    {
        return "Authentication OK.";
    }
    else
    {
        return "Authentication failed.";
    }
}
```

这将根据 AuthenticationTokenHeader 标题是否存在，是否不是空的 Guid，以及是否匹配存储在 Session["currentUser"]中的一个字符串(此时存在 Session 变量)，返回两个字符串中的一个。

接着创建一个简单的客户端，测试该服务。创建一个新的 PCSWebClient2 网站，给用户界面使用包含如下简单代码的 Default.aspx 页面：


```
<form id="form1" runat="server">
    <div>
        User Name:
        <asp:TextBox Runat="server" ID="userNameBox" /><br />
        Password:
        <asp:TextBox Runat="server" ID="passwordBox" /><br />
        <asp:Button Runat="server" ID="loginButton" Text="Log in" /><br />
        <asp:Label Runat="server" ID="tokenLabel" /><br />
        <asp:Button Runat="server" ID="invokeButton"
            Text="Invoke DoSomething()" /><br />
        <asp:Label Runat="server" ID="resultLabel" /><br />
    </div>
</form>
```

代码段 PCSWebClient2\Default.aspx

把 PCSWebService3 服务添加为一个 Web 引用(因为 Web 服务相对于解决方案位于本地，所以可以单击 This Solution 链接中的 Web Services，快速获得引用)，Web 服务的名称为 authenticateService，再把下面的 using 语句添加到 Default.aspx.cs 中：


可从
wrox.com
下载源代码

```
using System.Net;
using authenticateService;
```

代码段 PCSWebClient2\Default.aspx.cs

需要使用 System.Net 名称空间，因为它包含 CookieContainer 类。这个类用于存储 cookie 的引用，如果正在运行使用会话状态的 Web 服务，就需要它。这是因为 Web 服务需要跨客户端的多个调用，以某种方式检索正确的会话状态，其中在每次回发时重新创建服务的代理。检索 Web 服务使用的 cookie，以存储会话状态，在 Web 服务调用之间存储它，然后在以后的调用中使用它，就可以在 Web 服务中维护正确的会话状态。如果不这么做，Web 服务就会丢失其会话状态，此时需要客户提供登录信息。

在代码中，将使用一个受保护的成员存储 Web 引用代理，用另一个受保护的成员存储一个布尔值，表示用户是否通过了身份验证：

```
public partial class _Default : System.Web.UI.Page
{
   protected Service myService;
   protected bool authenticated;
```

Page_Load()方法首先初始化 myService 服务，并准备一个 CookieContainer 实例，由该服务使用：

```
   protected void Page_Load(object sender, EventArgs e)
   {
      myService = new Service();
      myService.Credentials = CredentialCache.DefaultCredentials;
      CookieContainer serviceCookie;
```

接着检查已存储的 CookieContainer 实例或创建一个新实例。在这两种方式中，都要把 CookieContainer 赋予 Web 服务代理，以准备在调用后检索 Web 服务的 cookie 信息。这里使用的存储器是窗体的 ViewState 集合(这是在回发之间持久化信息的一种有效方式，其工作方式类似于在应用程序或会话层中存储信息)：

```
      if (ViewState["serviceCookie"] == null)
      {
         serviceCookie = new CookieContainer();
      }
      else
      {
         serviceCookie = (CookieContainer)ViewState["serviceCookie"];
      }
      myService.CookieContainer = serviceCookie;
```

然后 Page_Load()方法确定是否有一个已存储的标题，并相应地把该标题赋予代理(以这种方式赋予标题是把要发送的数据作为 SOAP 标题必要的唯一一步)。这样，调用的任何事件处理程序(例如，Web 方法调用按钮的事件处理程序)就不需要担心赋予标题的问题了——这一步已经完成了：

```
        AuthenticationToken header = new AuthenticationToken();
        if (ViewState["AuthenticationTokenHeader"] != null)
        {
            header.InnerToken = (Guid)ViewState["AuthenticationTokenHeader"];
        }
        else
        {
            header.InnerToken = Guid.Empty;
        }
        myService.AuthenticationTokenValue = header;
    }
```

接着在设计器中双击 Login 按钮，为它添加一个事件处理程序：

```
    protected void loginButton_Click(object sender, EventArgs e)
    {
        Guid authenticationTokenHeader = myService.Login(userNameBox.Text,
                                                         passwordBox.Text);
        tokenLabel.Text = authenticationTokenHeader.ToString();
        if (ViewState["AuthenticationTokenHeader"] != null)
        {
            ViewState.Remove("AuthenticationTokenHeader");
        }
            ViewState.Add("AuthenticationTokenHeader", authenticationTokenHeader);
            if (ViewState["serviceCookie"] != null)
            {
                ViewState.Remove("serviceCookie");
            }
            ViewState.Add("serviceCookie", myService.CookieContainer);
        }
```

这个处理程序使用在两个文本框中输入的任何数据调用 Login()方法，显示返回的 Guid，并把该 Guid 存储在 ViewState 集合中。它还更新已存储在 ViewState 集合中的 CookieContainer 存储器，以备重用。

最后，必须以相同的方式为 Invoke DoSomething()按钮添加一个处理程序：

```
    protected void invokeButton_Click(object sender, EventArgs e)
    {
        resultLabel.Text = myService.DoSomething();
        if (ViewState["serviceCookie"] != null)
        {
            ViewState.Remove("serviceCookie");
        }
        ViewState.Add("serviceCookie", myService.CookieContainer);
    }
```

这个处理程序仅输出 DoSomething()方法返回的文本，并更新 CookieContainer 存储器，这与 loginButton_Click()方法相同。

在运行这个应用程序时，可以直接单击 Invoke DoSomething()按钮，因为 Page_Load()方法已经指定一个备用的标题(如果没有指定标题，就会抛出一个异常，因为已经指定这个方法必须有一个标题)。这会使 DoSomething()方法返回一条失败消息，如图 55-4 所示。

如果试着用除 Karli 和 Cheese 之外的任何用户名和密码登录，就会得到相同的结果。另一方面，如果使用这些证书进行登录，接着调用 DoSomething()方法，就会获得成功的消息，如图 55-5 所示。

此外，还可以看到 Guid 用于验证有效性的字符串表示。

当然，使用这种技术通过 SOAP 标题交换数据的应用程序可能会复杂得多。它们需要以比使用会话存储器更灵活的方式存储登录令牌，或许存储在数据库中。为了完整起见，也可以在经过某段时间时给这些令牌执行自己的过期机制，给用户提供注销选项，即删除令牌。尽管会话状态在指定的时间(默认为 20 分钟)过后过期，但还可以使用更复杂、更强大的机制。甚至可以根据用户使用的 IP 地址验证令牌，获得更高的安全性。但此处的关键点是用户的用户名和密码只发送一次，然后使用 SOAP 标题简化以后的方法调用。

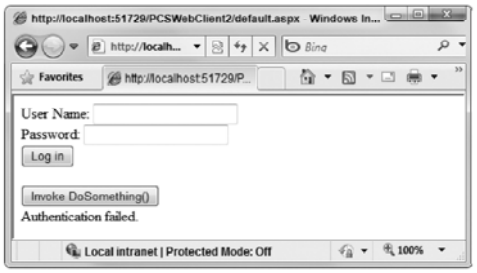

图　55-4

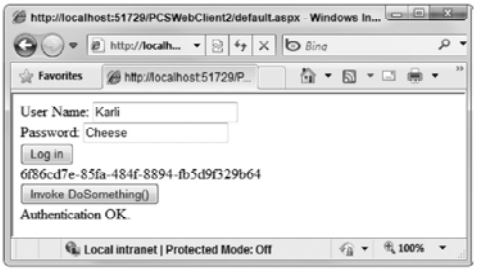

图　55-5

55.6　小结

本章讨论了如何应用 C#和 Visual Studio .NET 开发平台创建和使用 Web 服务。虽然它们易学易用，但是用途却非常大。

目前已经有许多 Web 服务可以从任何平台上访问。这是因为使用了 SOAP 协议，由于 SOAP 协议不限制使用.NET，因此从任何平台上都可以访问 Web 服务。

本章开发的主要示例阐明了如何轻松地创建.NET 分布式应用程序。这里假定用户使用一个服务器测试示例，但是这并不是说 Web 服务不应与客户端完全分离。如果需要额外的数据层，则它可以位于数据库中一个单独的服务器上。

对于大型应用程序，数据缓存技术是另一项非常重要的技术，因为可能会出现数千个用户同时连接的情况。

从最后一个示例可看出，通过 SOAP 标题交换数据是另一种可以用在应用程序中的有效技术。这个示例交换的是注册令牌，更复杂的数据也可以通过这种方式交换。例如，它可以用于 Web 服务的简单“密码保护”，而无需借助于施加更复杂的安全性。

Web 服务的使用者并不一定是 Web 应用程序，因为也可以从 Windows 窗体或 WPF 应用程序(在公司的内联网中使用这些应用程序当然看起来更好一些)中使用 Web 服务。另外，在 ASP.NET 中，Web 服务可以使用简单的事件驱动系统，异步调用其他 Web 服务。这非常适合于 Windows 窗体应用程序，它意味着，在复杂的 Web 服务执行复杂的工作，应用程序仍可以保持响应。

最后，值得一提的是 Web 服务是最新 WCF 技术(参见第 43 章)中的一个被分离出来的子集。但这并不意味着应总是使用 WCF 服务，而不使用 Web 服务——Web 服务实现起来要简单得多，而简单是一个巨大的优势。