第 11 章　用户控件和自定义控件

在 ASP.NET 中，系统自带的服务器控件为应用程序开发提供了诸多便利。在应用程序开发中，许多功能都需要重复使用，而如果在应用程序开发中重复的编写类似的代码是非常没有必要的。ASP.NET 让开发人员可以自行开发用户控件和自定义控件以提升代码的复用性，本章即将讲解用户控件和自定义控件的开发和使用。

11.1　用户控件

在 ASP 编程中，开发人员经常使用 Include 方式包含其他文件从而简化编程过程。而在 ASP.NET 中，控件能够提高应用程序中代码的复用性，不仅 ASP.NET 提供了服务器控件，ASP.NET 还支持用户自定义控件，从而提高了代码的复用性。

11.1.1　什么是用户控件

用户控件使开发人员能够根据应用程序的需求，方便的定义和编写控件。开发所使用的编程技术将与编写 Web 窗体的技术相同，只要开发人员对控件进行修改，就可以将使用该控件的页面的所有控件都进行更改。为了确保用户控件不会被修改、下载，被当成一个独立的 Web 窗体来运行，用户控件的后缀名为.ascx，当用户访问页面时，用户控件是不能被用户直接访问的。

注意：虽然.ascx 文件会阻止用户的直接访问，但是一些常用的下载工具还是能够下载.ascx 文件。

11.1.2　编写一个简单的控件

用户控件是以.ascx 为后缀名的，在 Visual Studio 2008 中，可以通过【添加新项】选项创建一个用户控件，如图 11-1 所示。

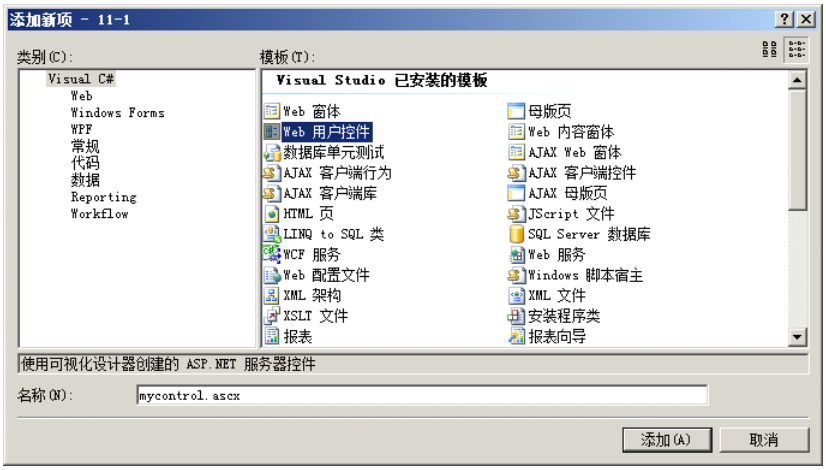

图 11-1　创建用户控件

用户控件创建完毕后，会生成一个.ascx 页面。.ascx 页面结构同.aspx 页面基本没有什么区别。在解决方案管理器中可以打开.aspx 页面和.ascx 页面进行对比，其结构并没有太大的变化，如图 11-2 和图 11-3 所示。

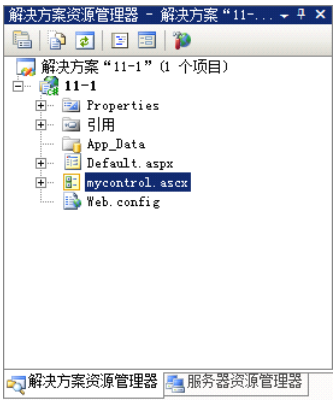

图 11-2　创建一个用户控件

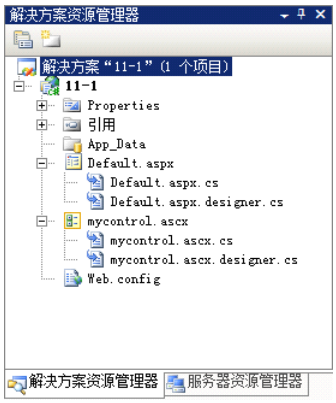

图 11-3　用户控件的结构

用户控件中并没有“<html><body>”等标记，因为.ascx 页面作为控件被引用到其他页面，引用的页面（如.aspx 页面）其中已经包含<body><html>等标记。而如果控件中使用这样的标记，可能会造成页面布局混乱。用户控件创建完成后，.ascx 页面代码如下所示。

```
<%@ Control Language="C#" AutoEventWireup="true"
 CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
```

其中没有任何的“<body><html>”等标记，而.ascx.cs 页面代码基本同.aspx 相同，示例代码如下所示。

```
using System;                                                    //使用系统命名空间
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;                                                //使用 Web 命名空间
using System.Web.Security;
using System.Web.UI;                                             //使用 UI 命名控件
using System.Web.UI.HtmlControls;                                //使用 Html 控件命名空间
using System.Web.UI.WebControls;                                 //使用 Web 控件命名空间
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;                                           //使用 LINQ 命名空间
namespace _11_1
{
    public partial class mycontrol : System.Web.UI.UserControl   //从控件类派生
    {
        protected void Page_Load(object sender, EventArgs e)     //页面加载方法
        {
        }
    }
}
```

用户控件能够提高复用性，前面介绍的服务器控件，从很多情况下来说都可以看作是用户控件的一种。当网站需要登录框时，不可能在每个需要登录的地方都重新编写一个登录框，最好的方法是每个页面都能够引用一个登录框。当需要对登录框进行修改时，可以一次性的将所有的页面都修改完毕，而不

需要对每个页面都修改登录框。

要达到这种目的，使用用户控件是最好不错的了。.ascx 页面允许开发人员拖动服务器控件，并编写相应的样式来实现用户控件，同时用户控件也能够支持事件、方法、委托等高级编程。编写一个用户登录窗口，可以通过几个 TextBox 控件和 Button 控件来实现，示例代码如下所示。

```
<%@ Control Language="C#"
AutoEventWireup="true" CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
<div style="border:1px solid #ccc; width:300px; background:#f0f0f0; padding:5px 5px 5px 5px; font-size:12px;">
    用户登录<br /><br />
    用户名 : <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br /><br />
    密码: <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox><br /><br />
    <asp:Button ID="Button1" runat="server" Text="登录" />
    <asp:HyperLink ID="HyperLink1" runat="server">还没有注册?</asp:HyperLink>
</div>
```

上述代码创建了一个登录框界面。当用户进行网站访问时，网站希望用户能够注册和登录到网站从而提高网站的用户粘度、提升访问量。所以设置登录窗口是非常必要的，界面布局如图 11-4 所示。

图 11-4　编写用户登录界面

当界面布局完毕后，就需要为用户控件编写事件。当用户单击【登录】按钮时，就需要进行事件操作。同 Web 窗体一样，双击按钮同样会自动生成事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = "登录成功";                                    //显示登录信息
}
```

当单击【登录】按钮时，系统提示登录成功，当然这里只是一个简单的用户控件。如果要实现复杂的用户控件的登录窗口，还需要对用户登录进行验证、查询和判断等功能。当用户控件制作完毕后，就可以在其他页面引用用户控件，示例代码如下所示。

```
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/mycontrol.ascx" %>        //声明控件引用
```

在这段代码中，有几个属性是必须编写的，这些属性的功能如下所示：

- TagPrefix：定义控件位置的命名控件。有了命名空间的制约，就可以在同一个页面中使用不同功能的同名控件。
- TagName：指向所用的控件的名字。
- Src：用户控件的文件路径，可以为相对路径或绝对路径，但不能使用物理路径。

了解了相关属性，就能够在其他页面中引用该控件了，示例代码如下所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_1._Default" %>
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/mycontrol.ascx" %>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>用户控件</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <Sample:Login runat="server" id="Login1"></Sample:Login>
    </div>
    </form>
</body>
</html>
```

上述代码声明了用户控件，并使用了用户控件，使用用户控件代码如下所示。

```
    <Sample:Login runat="server" id="Login1"></Sample:Login>                    //使用用户控件
```

从上述代码可以看出，用户控件的格式为 TagPrefix:TagName，当声明了用户控件后，就可以使用 TagPrefix:TagName 的方式使用用户控件。这样一个用户控件就使用完毕了，如图 11-5 所示。

图 11-5　使用用户控件

运行 Default.aspx 页面，虽然在 Default.aspx 页面中没有使用制作和编写任何控件，以及代码，但是却已经运行了登录框，这说明用户控件已经被运行了，如图 11-6 所示。

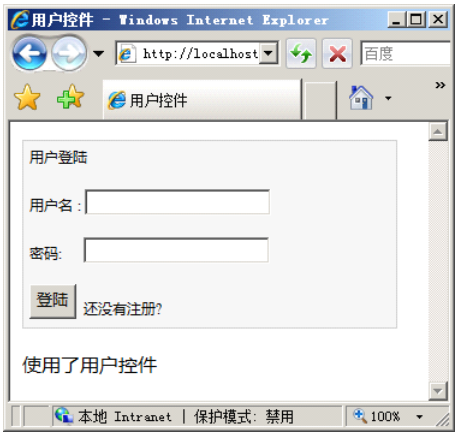

图 11-6　运行用户控件

当需要对登录框进行修改，而无需对页面进行修改时，只需要修改相应的用户控件即可。当多个页

面进行同样的用户控件的使用时，若需要对多个页面的控件进行样式或逻辑的更改只需要修改相应的控件，而不需要进行繁冗的多个页面的修正。

11.1.3 将 Web 窗体转换成用户控件

在编写用户控件时，会发现 Web 窗体的结构和用户控件的结构基本相同。如果开发人员已经开发了 Web 窗体，并在今后的需求中决定能够在应用程序全局中能够访问此 Web 窗体，那么就可以将 Web 窗体改成用户控件。如果需要将 Web 窗体更改为用户控件，首先需要对比 Web 窗体和用户控件的区别:

- Web 窗体中有<body><html><head>等标记，而用户控件没有。
- Web 窗体和用户控件所声明的方法不同。

在了解以上区别后，就可以很容易的将 Web 窗体转换成用户控件。首先，只需要删除<body><html><head>等标记即可。在删除标记后，好需要对两种窗体的声明方式进行更改，对于 Web 窗体，其标记方式如下代码所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_1._Default" %>
```

而对于用户控件，声明代码如下所示。

```
<%@ Control Language="C#"
AutoEventWireup="true" CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
```

在将 Web 窗体更改为用户控件时，只需要将 Page Language 更改为 Control Language 即可。这样就完成了 Web 窗体向用户控件的转换过程。

注意：有的时候，标记中还包括 ClassName 属性，当包含 ClassName 属性时，还需要修改相应的 ClassName 属性。

11.2 自定义控件

用户控件能够执行很多操作。并实现一些功能，但是在复杂的环境下，用户控件并不能够达到开发人员的要求，是因为用户控件大部分都是使用现有的控件进行组装，编写事件来达到目的。于是，ASP.NET 允许开发人员编写自定义控件实现复杂的功能。

11.2.1 实现自定义控件

自定义控件与用户控件不同，自定义控件需要定义一个直接或间接从 Control 类派生的类，并重写 Render 方法。在.NET 框架中，System.Web.UI.Control 与 System.Web.UI.WebControls.WebControl 两个类是服务器控件的基类，并且定义了所有服务器控件共有的属性、方法和事件，其中最为重要的就是包括了控制控件执行生命周期的方法和事件，以及 ID 等共有属性。

实现自定义控件，必须创建一个自定义控件，自定义控件将会编译成 DLL 文件。创建自定义控件如图 11-7 所示。

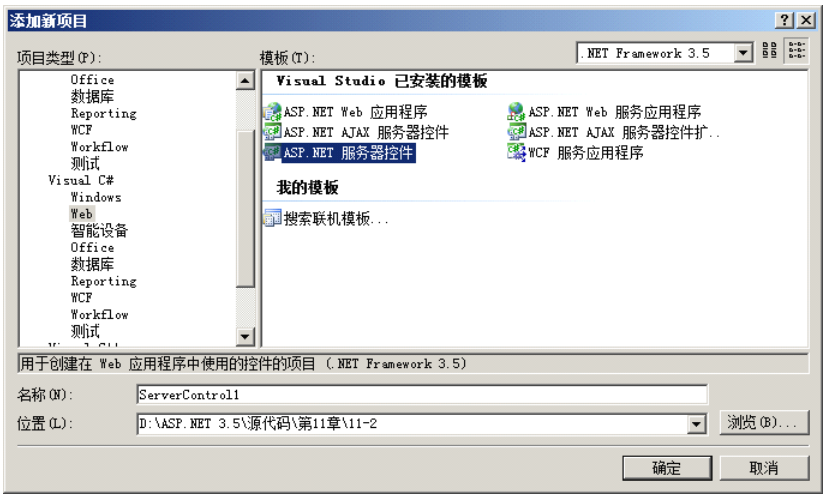

图 11-7　创建自定义控件

自定义控件创建完成后，会自动生成一个类，并在类中生成相应的方法，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;                                    //使用 UI 命名空间以便继承
namespace ServerControl1
{
    [DefaultProperty("Text")]                                       //声明属性
    [ToolboxData("<{0}:ServerControl1 runat=server></{0}:ServerControl1>")]   //设置控件格式
    public class ServerControl1 : WebControl
    {
        [Bindable(true)]                                            //设置是否支持绑定
        [Category("Appearance")]                                    //设置类别
        [DefaultValue("")]                                          //设置默认值
        [Localizable(true)]                                         //设置是否支持本地化操作
        public string Text                                          //定义 Text 属性
        {
            get                                                     //获取属性
            {
                String s = (String)ViewState["Text"];               //获取属性的值
                return ((s == null) ? "[" + this.ID + "]" : s);     //返回默认的属性的值
            }
            set                                                     //设置属性
            {
                ViewState["Text"] = value;
            }
        }
        protected override void RenderContents(HtmlTextWriter output)          //页面呈现
        {
            output.Write(Text);
```

```
        }
    }
}
```

开发人员可以在源代码中编写和添加属性。当需要呈现给 HTML 页面输出时，只需要重写 Render 方法即可，示例代码如下所示。

```
protected override void RenderContents(HtmlTextWriter output)
{
    output.Write("定义的 Text 属性的值为:" + Text);                    //输出为页面呈现
}
```

在使用服务器控件时，会发现控件有很多的属性，例如 SqlConnection 属性、Color 属性等，如图 11-8 所示。

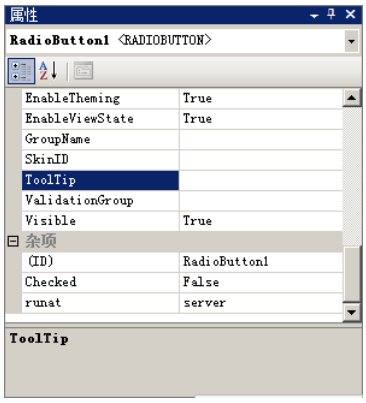

图 11-8　控件的属性

为了实现服务器控件的智能属性配置，开发人员能够在源代码中编写属性，示例代码如下所示。

```
public string GuoJingString                                    //编写属性
{
    get { return (String)ViewState["GuoJingString"]; }         //获取属性
    set { ViewState["GuoJingString"] = value; }                //设置属性
}
```

当自定义控件编写完毕后，需要在需要使用该控件的项目中添加引用。右击现有项目，选择【添加引用】选项，如果是同在一个解决方案下，则只要选择【项目】选项卡即可。而如果不在同一解决方案，则需要选择【浏览】选项卡浏览相应的 DLL 文件，如图 11-9 和图 11-10 所示。

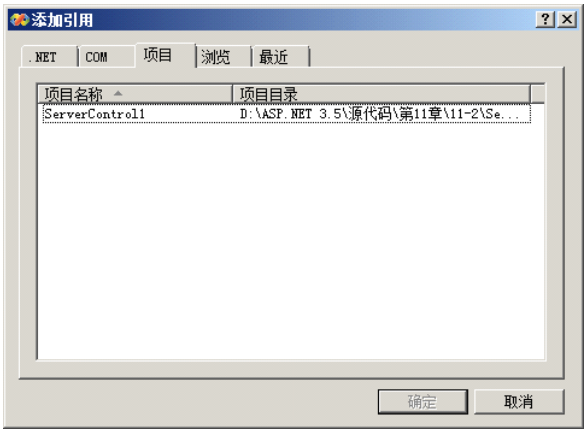

图 11-9　添加项目引用

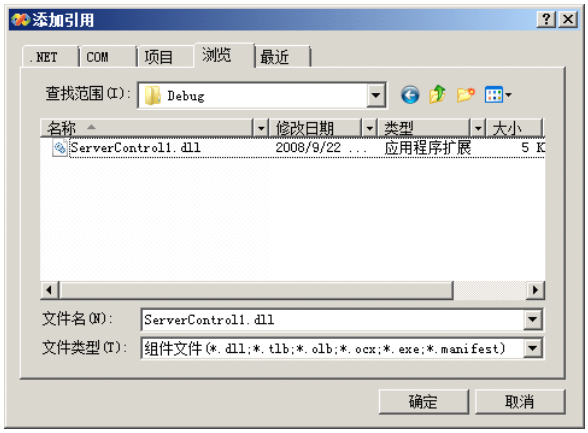

图 11-10　浏览 DLL

单击【确定】按钮完成引用的添加后，就可以在页面中使用此自定义控件。若需要在页面中需要使用此自定义控件，同样同用户控件一样需要在头部声明自定义控件，示例代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="ServerControl1" Assembly="ServerControl1" %>
```

上述代码向页面注册了自定义控件，自定义注册完毕后，就能够在页面中使用该控件。同时，在工具栏中也会呈现自定义控件，如图 11-11 所示。自定义控件呈现在工具箱之后，就可以直接拖动自定义控件到页面，并且配置相应的属性，如图 11-12 所示。

图 11-11　呈现自定义控件

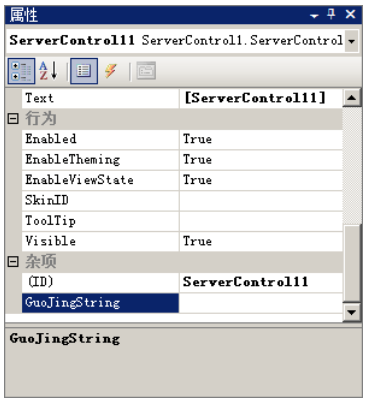

图 11-12　配置自定义属性

正如图 11-12 所示，开发人员能够在自定义控件中编写属性，这些属性可以是共有属性也可以是用户自定义的属性，用户可以拖动自定义控件使用于自己的应用程序中并通过属性进行自定义控件的配置。用户拖动自定义页面到控件后，页面会生成相应的自定义控件的 HTML 代码，示例代码如下所示。

```
<form id="form1" runat="server">
    <div>
        <MyControl:ServerControl1 ID="ServerControl11" runat="server" />
    </div>
</form>
```

上述代码就在页面中使用了自定义控件。在 ASP.NET 服务器控件中，很多的控件都是通过自定义控件来实现的，开发人员能够开发相应的自定义控件并在不同的应用中使用而无需重复开发。

11.2.2　复合自定义控件

单单一个简单的控件并不能实现太多的效果，在实际开发中，可能需要更多的功能，这种复杂功能控件最常见的就是 SqlDataSource 控件。SqlDataSource 控件是数据源控件，通过 SqlDataSource 控件能够配置数据源，并且实现分页、插入、删除等功能。复合自定义控件就类似这样一个功能复杂的控件。编写复合自定义控件有以下几种方式：

- 创建用户控件，并使用用户控件封装的用户界面实现复合控件。
- 开发一个编译控件，封装一个按钮控件和文本框控件，通过重写 Render 方法呈现。
- 从现有的控件中派生出新控件。
- 从基本控件类之一派生来创建自定义控件。

通过编写复合控件，能够让控件开发更加灵活，控件的使用人员也能够更加方便的配置控件，例如，重写登录控件，前台页面制作人员使用该控件时，可以为控件配置验证等功能，方便前台人员配置和使用。如图 11-13 所示。

图 11-13　登录控件

为了实现登录控件，就必须在自定义控件中添加相应的服务器控件，在登录控件中，需要两个 TextBox 来让用户输入用户名和密码，填写完成后，必须单击登录按钮实现登录事件。在类中创建 TextBox 和 Button 代码如下所示。

```
public class ServerControl1 : WebControl
{
                                                                    //创建服务器控件
    public TextBox NameTextBox = new TextBox();                     //创建 TextBox 控件
    public TextBox PasswordTextBox = new TextBox();                 //创建密码控件
    public Button LoginButton = new Button();                       //创建 Button 控件
    …
    …
}
```

上述代码创建了两个 TextBox 控件和一个 Button 控件。其中，NameTextBox 让用户能够输入用户名，而 PasswordTextBox 能够让用户输入密码。当用户单击 LoginButton 时，就需要实现登录操作，在这里就需要声明一个事件，示例代码如下所示。

```
    public event EventHandler LoginClick;                           //声明事件
```

完成对控件和事件的声明，就需要进行属性的编写。在登录控件中，希望在前台开发人员在开发过程中，能够轻易的配置属性进行使用，从而提高代码的复用性。在图 11-13 所表示控件中，开发人员希望控件的使用人员能够配置背景颜色、边框粗细、内置距离、登录说明和跳转连接等。在代码中，可以分别为这些属性进行配置，示例代码如下所示。

```
    [Bindable(true)]                                                //设置是否支持绑定
    [Category("Appearance")]                                        //设置类别
    [DefaultValue("")]                                              //设置默认值
    [Localizable(true)]                                             //设置是否支持本地化操作
    public string LoignBackGroundColor                              //设置背景属性
    {
        get { return (String)ViewState["LoignBackGroundColor"]; }   //获取背景
        set { ViewState["LoignBackGroundColor"] = value; }          //设置背景
    }
```

上述代码定义了一个属性，在属性定义前，可以对属性进行描述，如代码中 Bindable、Category 等，这些常用的描述意义如下所示：

- Bindable：是否用于绑定。
- Category：属性或事件显示在一个设置为“按分类顺序”的模式，如果不指定，则会显示在杂项中。

- DefaultValue：指定属性的默认值。
- Localizable：指定属性是否本地化。

编辑相应属性，在属性配置中就能够做相应的配置，如图 11-14 所示。

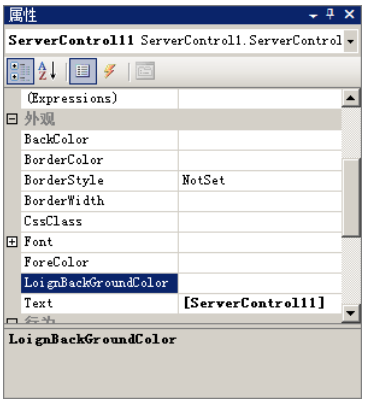

图 11-14　自定义属性

在代码中，将 Category 属性设置为 Appearance，这个属性就会在【外观】选项卡中出现。配置完成 LoginBackGroundColor 后，就可以为其他的属性做相应的配置，示例代码如下所示。

```
[Bindable(true)]                                          //设置是否支持绑定
[Category("Appearance")]                                  //设置类别
[DefaultValue("")]                                        //设置默认值
[Localizable(true)]                                       //设置是否支持本地化操作
public string LoignBackGroundColor                        //设置背景颜色
{
    get { return (String)ViewState["LoignBackGroundColor"]; }   //获取属性的值
    set { ViewState["LoignBackGroundColor"] = value; }          //设置属性默认值
}
//登录边框粗细
[Bindable(true)]                                          //设置是否支持绑定
[Category("Appearance")]                                  //设置类别
[DefaultValue("")]                                        //设置默认值
[Localizable(true)]                                       //设置是否支持本地化操作
public string LoginBorderWidth                            //设置边框粗细
{
    get { return (String)ViewState["LoginBorderWidth"]; }   //获取边框属性的值
    set { ViewState["LoginBorderWidth"] = value; }          //设置边框默认值
}
//登录的内置距离
[Bindable(true)]                                          //设置是否支持绑定
[Category("Appearance")]                                  //设置类别
[DefaultValue("")]                                        //设置默认值
[Localizable(true)]                                       //设置是否支持本地化操作
public string LoginPadding                                //设置内置距离
{
    get { return (String)ViewState["LoginPadding"]; }       //获取内置距离的值
    set { ViewState["LoginPadding"] = value; }              //设置默认值
}
//登录说明
```

```
[Bindable(true)]                                                    //设置是否支持绑定
[Category("Appearance")]                                            //设置类别
[DefaultValue("")]                                                  //设置默认值
[Localizable(true)]                                                 //设置是否支持本地化操作
public string LoginInformation                                      //设置登录信息
{
    get { return (String)ViewState["LoginInformation"]; }           //获取登录信息的值
    set { ViewState["LoginInformation"] = value; }                  //设置默认登录信息值
}
//登录跳转 URL
[Bindable(true)]                                                    //设置是否支持绑定
[Category("Appearance")]                                            //设置类别
[DefaultValue("")]                                                  //设置默认值
[Localizable(true)]                                                 //设置是否支持本地化操作
public string ResignURL                                             //设置登录跳转 URL
{
    get { return (String)ViewState["ResignURL"]; }                  //获取 URL 的值
    set { ViewState["ResignURL"] = value; }                         //设置 URL 默认值
}
```

编写完成属性后，就可以通过重写 Render 方法呈现不同的 HTML，示例代码如下所示。

```
protected override void RenderContents(HtmlTextWriter output)      //编写页面输出
{
    output.RenderBeginTag(HtmlTextWriterTag.Div);                   //创建 Div 标签
    output.RenderBeginTag(HtmlTextWriterTag.Tr);                    //创建 Tr 标签
    NameTextBox.RenderControl(output);                              //添加控件
    output.RenderBeginTag(HtmlTextWriterTag.Td);                    //创建 Td 标签
    output.RenderBeginTag(HtmlTextWriterTag.Br);                    //创建 Br 标签
    output.RenderBeginTag(HtmlTextWriterTag.Tr);                    //创建 Tr 标签
    PasswordTextBox.RenderControl(output);                          //添加控件
    output.RenderBeginTag(HtmlTextWriterTag.Td);                    //输出 Td 标签
}
```

上述代码使用了 HtmlTextWriter 类，HtmlTextWriter 类能够动态的创建 HTML 标签。上述代码中使用了 HtmlTextWriter 类的对象的 RenderBeginTag 方法创建相应的 HTML 标记。重写 Render 方法以呈现不同的 HTML 后，用户就能够看到登录界面，当用户单击【登录】按钮后，应该执行登录事件，这里应该是个事件冒泡，编写按钮提交事件代码如下所示。

```
public void Submit_Click(object sender, EventArgs e)
{
    EventArgs arg = new EventArgs();                                //编写按钮事件方法
    if (LoginClick != null)                                         //判断事件冒泡是否为空
    {
        LoginClick(LoginButton, arg);                               //触发事件
    }
}
```

编写按钮事件后，整个自定义控件就制作完毕了。相比之下，自定义控件的制作并不是那么难，反而自定义控件能够实现更多的效果，并呈现不同的样式，并且允许界面开发人员能够通过相应的配置呈现不同的样式。

11.3 用户控件和自定义控件的异同

对比用户控件和自定义控件，很多人或认为用户控件更加容易开发，而自定义控件的门槛较高，不方便应用程序的开发。其实不然，用户控件更适合创建内部的应用程序特定的控件，例如用户登录控件会在该项目中经常使用，所以创建用户控件能够极快的提高应用程序开发。而自定义控件通常应用到更适合创建通用的可再分发的控件，例如常用的开源 HTML 编辑器 Fckeditor 就可以说是一个优秀的自定义控件。

通常用户控件在一个项目中经常使用，而自定义控件用来在通用的程序中使用，当网站应用程序开发中，导航控件如果用用户控件实现，是非常方便的。但是通过自定义控件实现，可能并不能适合所有的应用场合，当需要适应其他场合时，可能需要重新开发和编译。具体的讲，用户控件和自定义控件可以从以下几个方面来说明它们的区别：

1．使用率

在选择使用用户控件和自定义控件时，可以首先考虑使用率。如果开发的应用程序只是需要小范围的使用，则可以考虑用户控件，而如果开发的自定义控件能够在大部分的应用程序中被应用，则可以考虑自定义控件。

2．创建技术

用户控件和自定义控件的创建技术是不相同的，并且用户控件和自定义控件创建的难度也不相同，用户控件是以.ascx 形式声明并创建的，开发过程也比较简单，并且有设计器提供设计支持，而自定义控件是从 System.Web.UI.Control 派生而来的，开发过程稍微复杂，也没有设计器提供设计支持。

3．生成方式

用户控件和自定义控件生成的方式不同，用户控件是以.ascx 的形式呈现，而自定义控件是以 DLL 的形式呈现，通过添加引用，自定义控件能够在【工具箱】中显式，能够像服务器控件一样拖动到页面，并且能够通过编程开发增加自定义属性。而用户控件无法在工具箱显示，也不能够像自定义控件那样增加自定义属性。

4．性能

虽然用户控件和自定义控件编写的过程不同，也遵循不同的创建模型，但是用户控件和自定义控件都是从 System.Web.UI.Control 直接或间接的派生的，在性能上没有很大的差别，主要是因为当用户控件在页面中第一次使用时，将作为普通的服务器控件被解析并编译进配件，第二次使用时，就和其他编译型控件一样。

11.4 用户控件示例

在用户控件一节中，介绍了如何创建和使用用户控件。创建用户控件能够为应用程序开发起到非常好的作用，并且提高代码的复用性，ASP.NET 允许开发人员创建用户控件和自定义控件，并在 Visual Studio 2008 中为开发人员提供了原生的开发环境，本节将一步步的进行用户控件的开发。

11.4.1 ASP.NET 登录控件

在应用程序开发过程中，登录是必不可少的，例如当用户初次访问该页面时，可以选择登录，也可以选择注册，但是需要有一个登录框作为指导，否则用户无法登录到该网站。当用户再次回访时，登录控件也能够让用户快速的进行登录访问。

作为登录控件，不仅需要对用户的身份进行判断，还需要对用户输入的字符串进行判断，例如，“’”号是 SQL 数据库中的关键字，如果非法用户利用了 SQL 关键字中的“’”号进行登录，则会出现错误，系统会提示异常，暴露数据库，这样是非常不安全的，所以登录控件还需要对输入的字符串进行判断，判断完成后，如果不是非法用户，则再继续对身份进行验证。

11.4.2 ASP.NET 登录控件的开发

ASP.NET 登录控件开发起来并不难，主要是需要理清几个基本概念：

- 用户是否已经注册过，注册过就可以直接登录。
- 用户如果没注册过，则需要跳转到注册页面。
- 如果用户输入的是非法字符或是没有任何输入，则先不对身份进行验证，先对输入框进行验证。
- 用户的验证是通过用户名和密码一起验证的。

当理清了以上思路之后，就能够进行 ASP.NET 登录控件的开发了。首先，在现有项目中添加一个新项并在弹出窗口中选择【Web 用户控件】项目，如图 11-15 所示。

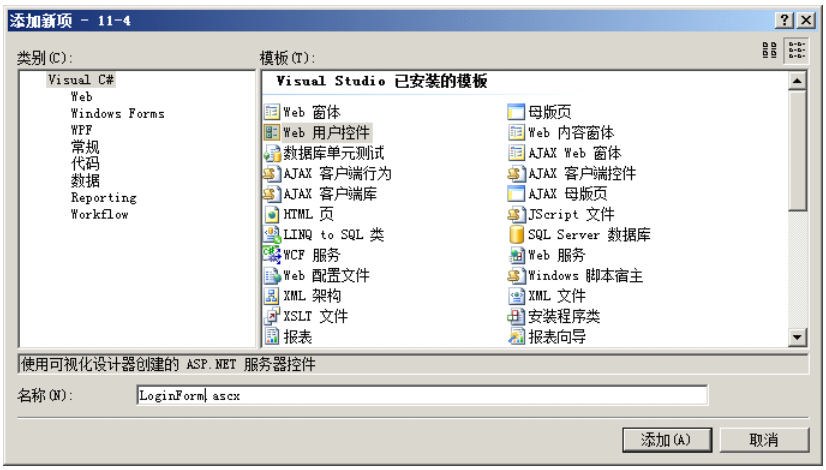

图 11-15　添加新项

创建一个用户控件，并命名为 LoginForm.ascx，方便在今后的开发中进行识别。

注意：良好的页面命名习惯也是一种良好的开发习惯，当页面增多时，可以通过页面命名的含义快速的寻找到相应的页面，当用户控件增多时，良好的命名也可以帮助快速寻找到相应的控件。

当创建完成一个 LoginForm 的用户控件后，就需要对用户控件进行页面布局，布局步骤如下所示。

- 拖动两个 TextBox，一个作为用户名输入框，一个作为密码输入框。
- 将密码输入框的 TextMode 属性设置为 Password。
- 拖动一个 Button 按钮，当用户单击 Button 按钮时，进行登录操作。
- 拖动一个 LinkButton 按钮，当用户单击 LinkButton 按钮时，跳转到登录页面。

当大概理清了控件的布局后，就可以针对控件的功能进行布局，如图 11-16 所示。

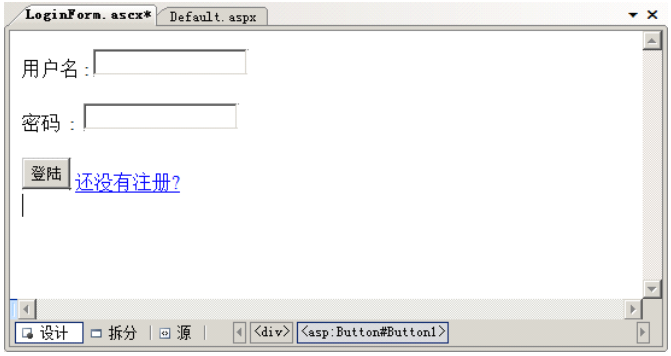

图 11-16　登录框控件初步布局

初步的对登录控件进行布局，发现这样的布局并不美观。对于访问者而言，看到一个并不美观的登录控件，很可能就没有想要登录或注册的想法。一个好的用户登录控件的布局，能够提升访问者的兴趣，于是就需要对控件进行布局更改，这里就需要借助于表格等布局工具，单击菜单栏中的【表】选项，单击下拉菜单中的【插入表】选项，系统会弹出默认的表格窗口。在这里，需要 3 行 2 列进行布局，可以在对话框中选择行数 3、列数 2 进行配置，如图 11-17 和图 11-18 所示。

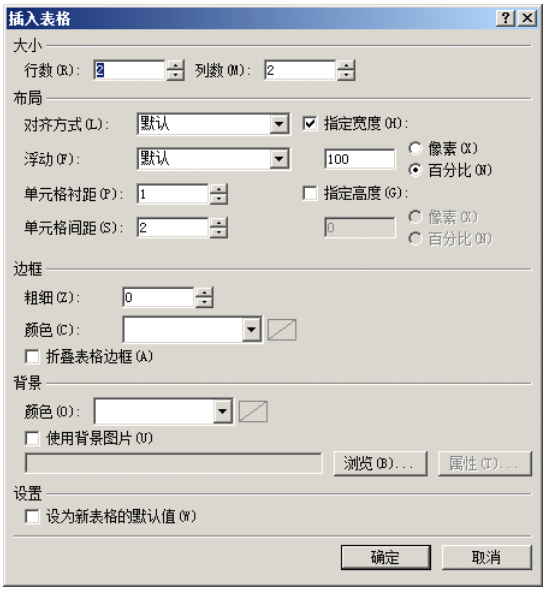

图 11-17　表格默认值

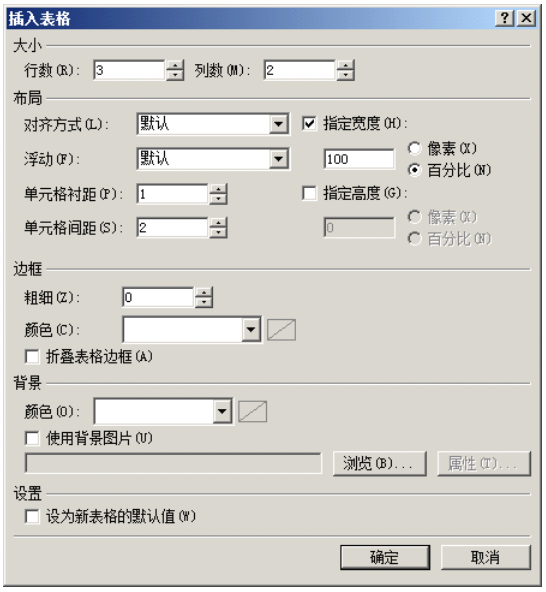

图 11-18　配置表格属性

技巧：配置表格属性后，可以勾选“设为新表格的默认值”，当下一次创建表格时，就无需再次配置，当大量的需要同样格式的表格时，可优先考虑此方案。

在编写好表格后，只需要拖动表格进行用户控件的布局即可，布局后 HTML 代码如下所示。

```
<style type="text/css">
    .style1
    {
        width: 100%;
         font-size:12px;
    }
</style>
<div style="border:1px solid #ccc; background:#f0f0f0; font-size:12px;">
    <table class="style1">
```

```
        <tr>
            <td>
            用户名 ：
            </td>
            <td>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                密码  :</td>
            <td>
            <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
            <asp:Button ID="Button1" runat="server" Text="登录" />
            </td>
            <td>
            <asp:LinkButton ID="LinkButton1" runat="server">还没有注册?
            </asp:LinkButton>
            </td>
        </tr>
    </table>
</div>
```

上述代码所呈现的样式如图 11-19 所示。

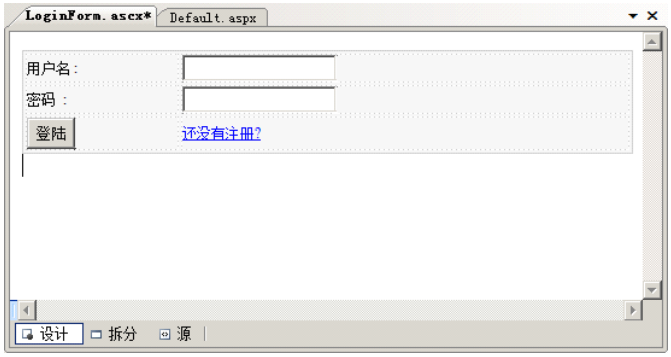

图 11-19　更改样式后的登录控件

当登录控件的样式初步制作完毕后，就需要增加一些验证控件，并编写登录框的事件。增加验证控件和增加单击事件后的 HTML 代码如下所示。

```
<style type="text/css">
    .style1
    {
        width: 100%;
        font-size:12px;
    }
</style>
<div style="border:1px solid #ccc; background:#f0f0f0; font-size:12px;">
```

```
    <table class="style1">
        <tr>
            <td>
            用户名 :
            </td>
            <td>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
                ControlToValidate="TextBox1"
                ErrorMessage="用户名不能为空">
            </asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td>
                密码  :</td>
            <td>
            <asp:TextBox ID="TextBox2" runat="server" TextMode="Password"></asp:TextBox>
            <asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server"
                 ControlToValidate="TextBox2"
                 ErrorMessage="密码不能为空">
            </asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td>
            <asp:Button ID="Button1" runat="server" Text="登录" onclick="Button1_Click" />
            </td>
            <td>
            <asp:LinkButton ID="LinkButton1" runat="server" PostBackUrl="resign.aspx">
                还没有注册?
            </asp:LinkButton>
            <asp:Label ID="Label1" runat="server" style="color: #FF3300"></asp:Label>
            </td>
        </tr>
    </table>
</div>
```

HTML 代码确定后，可以编写登录事件，示例代码如下所示。

```
        protected void Button1_Click(object sender, EventArgs e)
        {
            if (TextBox1.Text != "test" && TextBox2.Text != "test")                    //如果用户名不匹配
            {
                Label1.Text = "登录失败";                                              //提示登录失败
            }
            else
            {
                Label1.Text = "登录成功";                                              //否则登录成功
            }
        }
```

上述代码判断如果用户名如果不等于 test 并且密码不等于 test 的话，则提示登录失败，否则登录成

功。

技巧：在这里可以使用 ADO.NET 对数据库中的用户表进行操作，通过 SELECT 语句查询相应的用户，如果查询出的值返回值大于 0，则说明有这个用户，否则没有这个用户，判断“登录失败”。

11.4.3 ASP.NET 登录控件的使用

编写完成 ASP.NET 登录控件后，就可以使用登录控件进行登录页面的制作，在使用登录控件前，必须通过使用 Register 关键字向页面注册该用户控件，示例代码如下所示。

```
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/LoginForm.ascx" %>
```

上述代码向页面注册了该控件。当页面被执行时，会通过 TagPrefix 以及 TagName 判断 ASP.NET 标签，并解析成相应的 ASP.NET 控件以呈现给页面，然后页面呈现给用户。当需要使用该控件时，只需要在页面中编写引用代码，示例代码如下所示。

```
<Sample:Login runat="server" id="Login1"></Sample:Login>
```

上述代码就在相应的位置显示了用户控件，如图 11-20 所示。

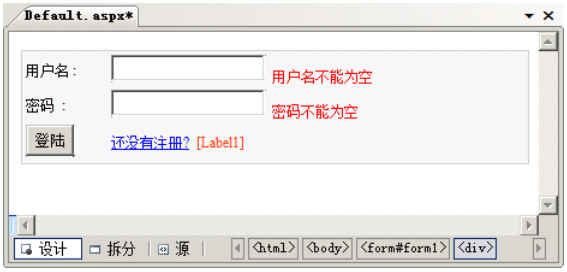

图 11-20　使用用户控件

对使用用户控件的页面进行页面布局，不会影响到用户控件的布局，对用户控件的布局，同样不会影响到页面的布局。相反的是，使用用户控件的页面无需考虑事件，也无需在该页面编写任何 C#代码，这让页面变得非常的简洁，而事件的操作可以交付给用户控件，示例代码如下所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_4._Default" %>
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/LoginForm.ascx" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <Sample:Login runat="server" id="Login1"></Sample:Login>
    </div>
    </form>
</body>
</html>
```

从上述代码可以看出，使用用户控件，并没有任何冗余的代码，这让页面代码显得非常的整洁和整齐。虽然从表面看上去只有 HTML 代码，但是并没有影响页面中程序的实现。运行后如图 11-21 所示。

图 11-21　使用用户控件

虽然在页面中并没有实现控件的布局和事件的处理，但是在页面依旧可以呈现相应的控件布局和事件处理。自定义控件的好处就在于能够将复杂的样式或事件存储在一个控件中，以便在不同的页面中进行使用。

11.5　自定义控件实例

虽然用户控件能够尽快的上手并运用在开发中，但是自定义控件的编写能够实现更多的效果。如分页效果在大部分的数据索引中，都需要使用分页。如果存在这么一个分页控件，只需要指定需要分页的表，那么可以自动分页，就能够更加方便应用程序开发了。

11.5.1　ASP.NET 分页控件

ASP.NET 能够编写自定义控件，并将自定义控件编译为 DLL 文件以保证在任何其他的项目中能够使用自定义控件。在 ASP.NET Web 应用程序开发中，对于数据的索引，通常情况下是不可能全部将数据索引到一个页面的，所以在显示数据时，就需要进行分页操作。

当用户打开一个页面时，如果将全部的数据一起显示在页面，不仅页面臃肿难看，并且用户很难找到自己需要的信息。对于页面数据的整理和索引，能够让用户更加方便的找到自己需要的信息。不仅如此，如果一次全部的将数据呈现到 HTML 页面，势必会造成 HTML 页面数据的冗长，在运行页面时，也会增加服务器的压力，让 Web 应用程序变得非常的缓慢。

1. 属性设置

使用 ASP.NET 分页控件，能够让数据分开显示，让用户能够自行选择，并且能够自行选择页码，查看相应的数据，创建一个 MyPager 自定义控件，用来执行分页操作，如图 11-22 所示。

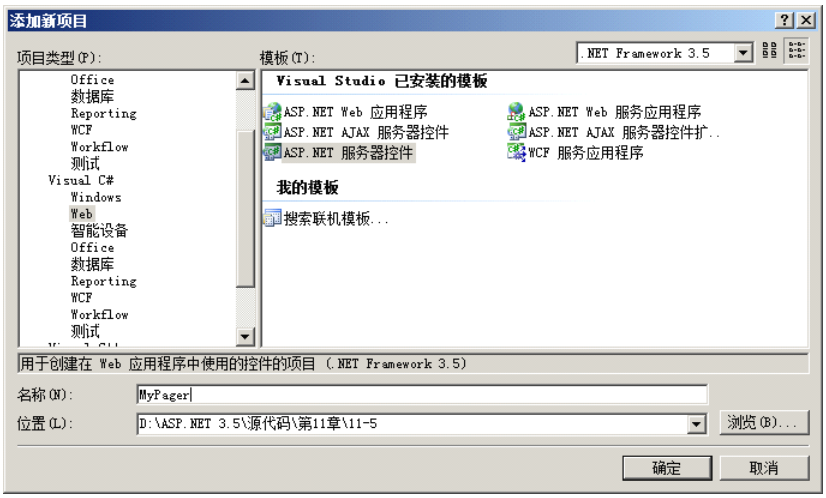

图 11-22　创建 MyPager 控件

创建完成后，就需要确定一些基本的属性，这些属性能够方便控件的使用者进行相应的配置，能够尽快的使用控件并完成编程目的。对于分页控件，通常需要确定的属性如下所示：

- PageSize：用户希望一个页面呈现多少数据。
- Server：数据库服务器的地址。
- Database：数据库服务器的数据库。
- Pwd：数据库服务器有效的密码。
- Uid：数据库服务器有效的用户名。
- Table：需要执行分页的表，如果不指定 SqlCommand，则自动生成语句。
- SqlCommand：如果不指定表，则执行 SqlCommand。
- IndexPage ：一开始的索引页面。
- PageName：当前页面的名称，用于跳转。

2．数据属性配置

在基本确定了以上属性后，就可以编写相应代码，首先需要为数据库连接和数据库的 SQL 语句的功能实现编写相应的属性，示例代码如下所示。

```
[DefaultProperty("Text")]                                    //默认属性
[ToolboxData("<{0}:MyPager runat=server></{0}:MyPager>")]    //控件呈现代码
public class MyPager : WebControl
{
    [Bindable(true)]                                         //设置是否支持绑定
    [Category("Appearance")]                                 //设置类别
    [DefaultValue("")]                                       //设置默认值
    [Localizable(true)]                                      //设置是否支持本地化操作
    public string Text                                       //Text 文本属性
    {
        get                                                  //获取属性
        {
            String s = (String)ViewState["Text"];            //获取文本属性
            return ((s == null) ? "[" + this.ID + "]" : s);  //设置文本属性默认值
        }
        set                                                  //设置属性
```

```
        {
            ViewState["Text"] = value;
        }
    }
    [Bindable(true)]                                    //设置是否支持绑定
    [Category("Data")]                                  //设置类别 Data
    [DefaultValue(10)]                                  //设置默认值
    [Localizable(true)]                                 //设置是否支持本地化操作
    public int PageSize                                 //分页数属性
    {
        get
        {
            try
            {
                Int32 s = (Int32)ViewState["PageSize"]; //获取分页值
                return ((s.ToString() == null) ? 10 : s); //设置默认值
            }
            catch                                       //如果用户输入异常
            {
                return 10;                              //返回分页数
            }
        }
        set
        {
            ViewState["PageSize"] = value;              //设置分页
        }
    }
```

上述属性设置了分页属性，当开发人员使用该控件进行分页设置时，该控件会根据不同的分页属性进行分页设置并进行分页操作。在执行分页前，还需要进行数据库的连接，这就需要编写数据连接属性，示例代码如下所示。

```
    [Bindable(true)]                                    //设置是否支持绑定
    [Category("Data")]                                  //设置类别 Data
    [DefaultValue("")]                                  //设置默认值
    [Localizable(true)]                                 //设置是否支持本地化操作
    public string Server                                //服务器地址
    {
        get
        {
            String s = (String)ViewState["Server"];     //设置服务器地址
            return ((s == null) ? ("(local)") : s);     //默认服务器地址
        }
        set
        {
            ViewState["Server"] = value;                //设置默认值
        }
    }
    [Bindable(true)]                                    //设置是否支持绑定
    [Category("Data")]                                  //设置类别 Data
    [DefaultValue("")]                                  //设置默认值
```

```
[Localizable(true)]                                    //设置是否支持本地化操作
public string DataBase                                 //数据库属性
{
    get
    {
        String s = (String)ViewState["DataBase"];      //设置数据库属性
        return ((s == null) ? ("none") : s);           //设置默认值
    }
    set
    {
        ViewState["DataBase"] = value;                 //设置默认值
    }
}
[Bindable(true)]                                       //设置是否支持绑定
[Category("Data")]                                     //设置类别 Data
[DefaultValue("")]                                     //设置默认值
[Localizable(true)]                                    //设置是否支持本地化操作
public string Uid                                      //数据库用户名
{
    get
    {
        String s = (String)ViewState["Uid"];           //设置 UID 属性
        return ((s == null) ? ("uid") : s);            //设置默认值
    }
    set
    {
        ViewState["Uid"] = value;                      //设置默认值
    }
}
[Bindable(true)]                                       //设置是否支持绑定
[Category("Data")]                                     //设置类别 Data
[DefaultValue("")]                                     //设置默认值
[Localizable(true)]                                    //设置是否支持本地化操作
public string Pwd                                      //数据库密码
{
    get
    {
        String s = (String)ViewState["Pwd"];           //设置密码属性
        return ((s == null) ? ("none") : s);           //设置默认值
    }
    set
    {
        ViewState["Pwd"] = value;                      //设置默认值
    }
}
[Bindable(true)]                                       //设置是否支持绑定
[Category("Data")]                                     //设置类别 Data
[DefaultValue("")]                                     //设置默认值
[Localizable(true)]                                    //设置是否支持本地化操作
public string Table                                    //分页的表
```

```
{
    get
    {
        String s = (String)ViewState["Table"];                    //设置分页的表
        return ((s == null) ? ("none") : s);                      //设置默认值
    }
    set
    {
        ViewState["Table"] = value;                               //设置默认值
    }
}
[Bindable(true)]                                                  //设置是否支持绑定
[Category("Data")]                                                //设置类别 Data
[DefaultValue("")]                                                //设置默认值
[Localizable(true)]                                               //设置是否支持本地化操作
public string SqlCommand                                          //Sql 语句
{
    get
    {
        String s = (String)ViewState["SqlCommand"];               //设置 SQL 语句
        return ((s == null) ? ("none") : s);                      //设置默认值
    }
    set
    {
        ViewState["SqlCommand"] = value;                          //设置默认值
    }
}
```

上述代码为数据库连接进行了属性配置，这些属性分别包括数据库连接字串、数据库服务器 IP、数据库用户名、数据库密码等，这些属性用于配置不同的数据库以呈现不同的数据。

3．页面属性配置

在编写了数据属性后，还需要编写相应的页面属性以便能够对页面进行控制，这些属性包括页面名称、索引等，示例代码如下所示。

```
[Bindable(true)]                                                  //设置是否支持绑定
[Category("Data")]                                                //设置类别 Data
[DefaultValue("")]                                                //设置默认值
[Localizable(true)]                                               //设置是否支持本地化操作
public string IndexPage                                           //索引页面
{
    get
    {
        String s = (String)ViewState["IndexPage"];                //获取当前页面配置属性
        return ((s == null) ? ("none") : s);                      //返回默认值
    }
    set
    {
        ViewState["IndexPage"] = value;                           //设置默认配置属性
    }
}
```

```
[Bindable(true)]                                                    //设置是否支持绑定
[Category("Data")]                                                  //设置类别 Data
[DefaultValue(“”)]                                                  //设置默认值
[Localizable(true)]                                                 //设置是否支持本地化操作
public string PageName                                              //页面名称
{
    get
    {
        String s = (String)ViewState["PageName"];                   //获取页面名称
        return ((s == null) ? ("none") : s);                        //设置默认值
    }
    set
    {
        ViewState["PageName"] = value;                              //返回默认值
    }
}
```

在编写完成相应的属性后，就需要重写 Render 方法来执行 HTML 流输出，示例代码如下所示。

```
protected override void RenderContents(HtmlTextWriter output)
{
    string html = "";
    string str = "server='" + Server + "';database='" + DataBase + "';uid='" + Uid + "';pwd='" + Pwd + "'";
    string strsql = "";
    SqlConnection con = new SqlConnection(str);                     //创建连接对象
    try
    {
        con.Open();                                                 //打开数据连接
        if (SqlCommand == "none" || SqlCommand == "")               //如果不自定义 Table 则自动生成
        {
            strsql = "select count(*) as mycount from " + Table + "";  //生成 SQL 语句
        }
        else
        {
            strsql = SqlCommand;                                    //设置默认 SQL 语句
        }
        SqlDataAdapter da = new SqlDataAdapter(strsql, con);        //创建适配器
        DataSet ds = new DataSet();                                 //创建数据集
        int count = da.Fill(ds, "count");                           //填充数据集
        int page = 0;                                               //数据表中的行数
        int pageCount = 0;                                          //分页数
        if (count > 0)//获取数据表中的行数
        {
            page = Convert.ToInt32(ds.Tables["count"].Rows[0]["mycount"].ToString());
        }
        if (page % PageSize > 0)                                    //开始分页
        {
            pageCount = (page / PageSize) + 1;                      //分页计算
        }
        else
        {
            pageCount = page / PageSize;
```

```
            }
            html += "<table><tr>";
            for (int i = 0; i < pageCount; i++)
            {
                if (IndexPage != i.ToString())                    //如果查看的是当前页面，则高亮显示
                {
                    html += "<td style=\"padding:5px 5px 5px 5px;background:#f0f0f0;border:1px
                    dashed #ccc;\">";                             //呈现相应的 HTML
                }
                else
                {
                    html += "<td style=\"padding:5px 5px 5px 5px;background:Gray;border:1px
                    dashed #ccc;\">";                             //呈现相应的 HTML
                }
                html += "<a href=\"" + PageName + "?page=" + i + "\">" + i + "</a>";
                html += "</td>";                                  //完成 HTML
            }
            html += "</tr></table>";                              //完成 HTML
        }
        catch(Exception ee)                                       //出现错误抛出异常
        {
            html = ee.ToString();                                 //输出异常
        }
        finally
        {
            output.Write(html);                                   //页面呈现
            con.Close();
        }
    }
```

上述代码会查询相应的数据库，例如 Table。当查询到了相应的数据库中数据的行数之后，再与 PageSize 属性进行对比，并执行分页查询，查询完成后将通过查询的条目数循环遍历输出 HTML，并将 HTML 呈现在页面中。

11.5.2 ASP.NET 分页控件的使用

自定义控件能够使用在各种其他的应用程序开发中。而任何其他的应用程序如果需要使用分页控件，则需要首先添加引用。右击当前项目，选择【添加引用】选项，单击【浏览】选项卡，浏览到项目的 bin 目录下，选择相应的 DLL 文件即可，如图 11-23 所示。

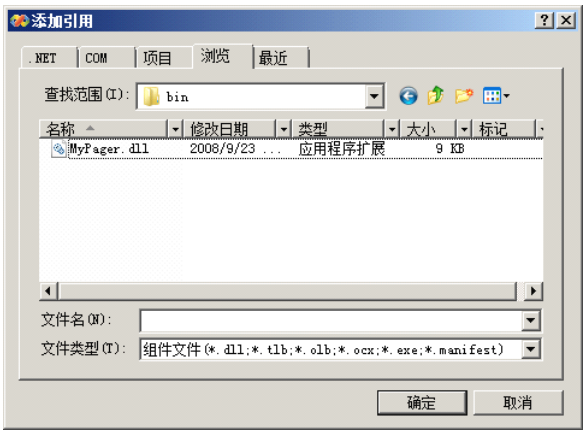

图 11-23　添加引用

添加引用后，同样需要在需要使用的页面进行注册，示例代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="MyPager" Assembly="MyPager" %>
```

添加注册后，在工具栏中就会显示自定义控件，拖动自定义控件到页面中就可以使用自定义控件并配置相应的属性，系统自动生成的 HTML 代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="MyPager" Assembly="MyPager" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <MyControl:MyPager ID="MyPager1" runat="server" DataBase="mytable"
            IndexPage="1" PageName="default.aspx" PageSize="1" Pwd="Bbg0123456#"
            Table="mynews" Uid="sa" />
    </div>
    </form>
</body>
</html>
```

为了能够使分页控件能够自动进行分页，则需要对属性进行相应的配置，如图 11-24 所示。配置完毕后，控件就能够实现自动分页，如图 11-25 所示。

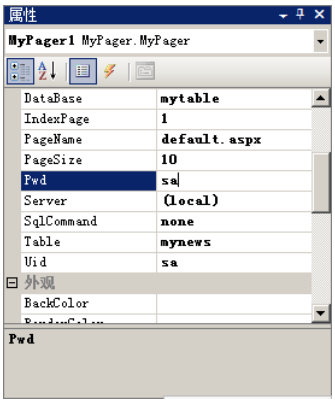

图 11-24　配置属性

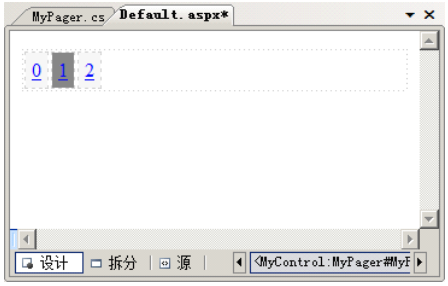

图 11-25　分页控件

通过修改相应的属性，能够为不同的表，甚至不同的数据库进行分页操作，运行结果如图 11-26 所示。

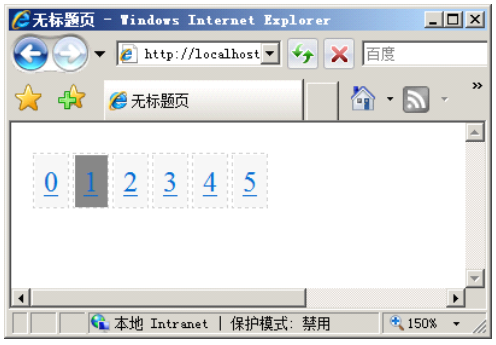

图 11-26　运行分页控件

当配置 PageSize 属性为 1 时，系统则会按每页 1 个数据来进行分页，同样当配置 PageSize 属性为 10 时，则系统会按照每页 10 个数据来进行分页。

注意：在编写分页控件时，这里需要配置服务器信息，这样做是非常不安全的，这里的属性配置可以放在 Web.config 文件中，这样配置就更加的安全，具体可参考 SqlDataSource 的做法。

11.6　小结

本节在服务器控件的基础上，着重讲解了用户控件和自定义控件。使用用户控件和自定义控件的优势就在于，用户控件和自定义控件都能够非常简单的完成并且能够达到开发的需求，而无需重复的进行代码编写。

在传统的开发概念中，用户控件和自定义控件都比较复杂，而通过本章就能够了解到用户控件和自定义控件的开发并没有想象中的复杂，用户控件和自定义控件能够适应更多的应用场合，这些控件能够重复使用和自定义，极大的方便了应用程序的开发。本章还包括：

- 用户控件：包括什么是用户控件和如何创建用户控件。
- 将 Web 窗体转换成用户控件。
- 复合自定义控件。
- 用户控件和自定义控件的异同。

本章分别通过实例创建和使用了用户控件和自定义控件，能够轻松的了解用户控件和自定义控件的异同和编程模型，对以后的开发有很大的帮助。