第 13 章　ASP.NET 内置对象,应用程序配置和缓存

Web 应用程序在传统的意义上来说是无状态的，Web 应用不能像 Win Form 那样维持客户端状态，所以在 Web 应用中，通常需要使用内置对象进行客户端状态的保存。这些内置对象能够为 Web 应用程序的开发提供设置，配置以及检索等功能。

13.1　ASP.NET 内置对象

在 ASP 的开发中，这些内置对象已经存在，这些内置对象包括 Response、Request、Application 等，虽然 ASP 是一个可以称得上是“过时的”技术，但是在 ASP.NET 开发人员中依旧可以使用这些对象。这些对象不仅能够获取页面传递的参数，某些对象还可以保存用户的信息，如 Cookie、Session 等。

13.1.1　Request 传递请求对象

Request 对象是 HttpRequest 类的一个实例，Request 对象用于读取客户端在 Web 请求期间发送的 HTTP 值。Request 对象常用的属性如下所示。

- QueryString：获取 HTTP 查询字符串变量的集合。
- Path：获取当前请求的虚拟路径。
- UserHostAddress：获取远程客户端 IP 主机的地址。
- Browser：获取有关正在请求的客户端的浏览器功能的信息。

1．QueryString：请求参数

QueryString 属性是用来获取 HTTP 查询字符串变量的集合，通过 QueryString 属性能够获取页面传递的参数。在超链接中，往往需要从一个页面跳转到另外一个页面，跳转的页面需要获取 HTTP 的值来进行相应的操作，例如新闻页面的 news.aspx?id=1。为了获取传递过来的 id 的值，则可以使用 Request 的 QueryString 属性，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(Request.QueryString["id"]))          //如果传递的 ID 值不为空
    {
        Label1.Text = Request.QueryString["id"];                   //将传递的值赋予标签中
    }
    else
    {
        Label1.Text = "没有传递的值";                               //提示没有传递的值
    }
    if (!String.IsNullOrEmpty(Request.QueryString["type"]))        //如果传递的 TYPE 值不为空
    {
        Label2.Text = Request.QueryString["type"];                 //获取传递的 TYPE 值
    }
    else
```

```
        {
            Label2.Text = "没有传递的值";                                    //无值时进行相应的编码
        }
    }
```

上述代码使用 Request 的 QueryString 属性来接受传递的 HTTP 的值，当通过访问页面路径为“http://localhost:29867/Default.aspx”时，默认传递的参数为空，因为其路径中没有对参数的访问。而当访问的页面路径为“http://localhost:29867/Default.aspx?id=1&type=QueryString&action=get”时，就可以从路径中看出该地址传递了三个参数，这三个参数和值分别为 id=1、type=QueryString 以及 action=get。

2．Path：获取路径

通过使用 Path 的方法可以获取当前请求的虚拟路径，示例代码如下所示。

```
Label3.Text = Request.Path.ToString();                                  //获取请求路径
```

当在应用程序开发中使用 Request.Path.ToString()时，就能够获取当前正在被请求的文件的虚拟路径的值，当需要对相应的文件进行操作时，可以使用 Request.Path 的信息进行判断。

3．UserHostAddress：获取 IP 记录

通过使用 UserHostAddress 的方法，可以获取远程客户端 IP 主机的地址，示例代码如下所示。

```
Label4.Text = Request.UserHostAddress;                                  //获取客户端 IP
```

在客户端主机 IP 统计和判断中，可以使用 Request.UserHostAddress 进行 IP 统计和判断。在有些系统中，需要对来访的 IP 进行筛选，使用 Request.UserHostAddress 就能够轻松的判断用户 IP 并进行筛选操作。

4．Browser：获取浏览器信息

通过使用 Browser 的方法，可以判断正在浏览网站的客户端的浏览器的版本，以及浏览器的一些信息，示例代码如下所示。

```
Label5.Text = Request.Browser.Type.ToString();                          //获取浏览器信息
```

这些属性能够获取服务器和客户端的相应信息，也可以通过“?”号进行 HTTP 的值的传递和获取，上述代码运行结果如图 13-1 所示。

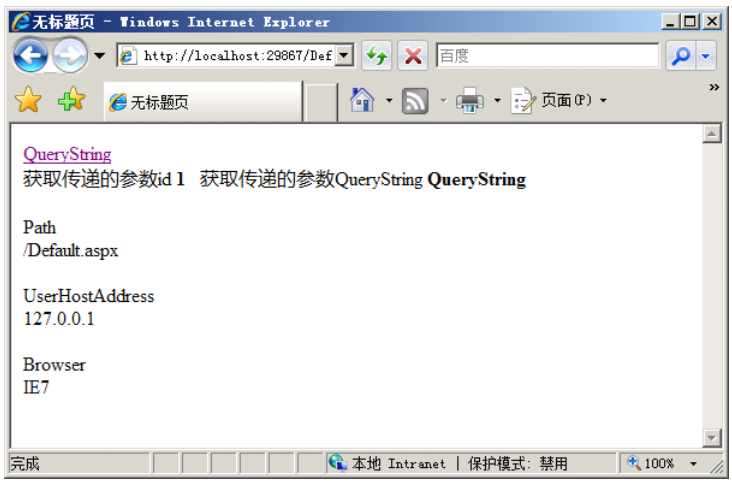

图 13-1　Request 对象

Request 不仅包括这些常用的属性，还包括其他属性，例如用于获取当前目录在服务器虚拟主机中的绝对路径（如 ApplicationPath）。另外，开发人员也可是使用 Request 中的 Form 属性进行页面中窗体的值集合的获取。

13.1.2 Response 请求响应对象

Response 对象是 HttpResponse 类的一个实例。HttpResponse 类用户封装页面操作的 HTTP 响应信息。Response 对象的常用属性如下所示。

- BufferOutput：获取或设置一个值，该值指示是否缓冲输出，并在完成处理整个页面之后将其发送。
- Cache：获取 Web 页面的缓存策略。
- Charset：获取或设置输出流的 HTTP 字符集类型。
- IsClientConnected：获取一个值，通过该值指示客户端是否仍连接在服务器上。
- ContentEncoding：获取或设置输出流的 HTTP 字符集。
- TrySkipIisCustomErrors：获取或设置一个值，指定是否支持 IIS 7.0 自定义错误输出。

1. Response 常用属性

BufferOutput 的默认属性为 True。当页面被加载时，要输出到客户端的数据都暂时存储在服务器的缓冲期内并等待页面所有事件程序，以及所有的页面对象全部被浏览器解释完毕后，才将所有在缓冲区中的数据发送到客户端浏览器，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("缓冲区清除前..");                                    //输出缓冲区清除
}
```

上述代码在 cs 文件中重写了 Page_Load 事件，该事件用于中向浏览器输出一行字符串“缓冲区清除前”。在 ASPX 页面中，可以为页面增加代码以判断缓冲区的执行时间，示例代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <div>
    <% Response.Write("缓冲区被清除"); %>                                //输出字符串
    </div>
    </form>
</body>
```

上述代码在页面中插入了一段代码，并输出字符串“缓冲区被清除”。在运行该页面时，数据已经存放在缓冲区中。然后 IIS 才开始读取 HTML 组件的部分，读取完毕后才将结果送至客户端浏览器，所以在运行结果中可以发现，“缓冲期清除前”是在“缓冲区被清除”字符串之前出现，如图 13-2 所示。

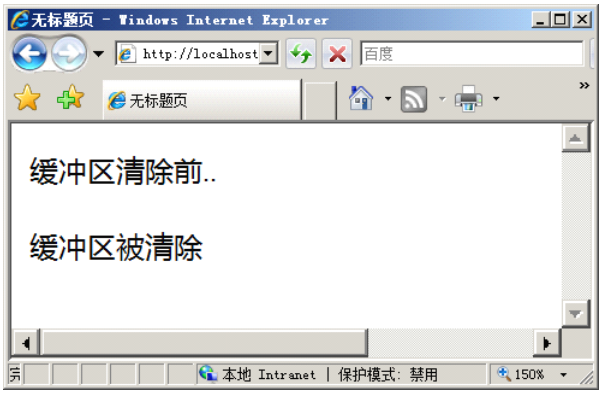

图 13-2　BufferOutput

因为 BufferOutput 属性默认为 true，所以上述代码并无法看到明显的区别，当在浏览器输出前清除缓冲区时，则可以看出区别。示例代码如下所示。

```
Response.Write("缓冲区清除前..");
Response.Clear();                                    //清除缓冲区
```

当使用 Response 的 Clear 方法时，缓冲区就被显式的清除了。在运行后，“缓冲区清除前”字符串被清除，并不会呈现给浏览器。当需要屏蔽 Clear 方法对缓冲区的数据清除，则可以指定 BufferOutput 的属性为 False，示例代码如下所示。

```
Response.BufferOutput = false;                       //设置缓冲区属性
Response.Write("缓冲区清除前..");                    //设置清除前字符
Response.Clear();                                    //清除缓冲区
```

使用上述代码将指定 BufferOutput 的属性为 False，在运行时缓冲区数据不会被 Clear 方法清除。

2. Response 常用方法

Response 方法可以输出 HTML 流到客户端，其中包括发送信息到客户端和客户端 URL 重定向，不仅如此，Response 还可以设置 Cookie 的值以保存客户端信息。Response 的常用方法如下所示：

- Write：向客户端发送指定的 HTTP 流。
- End：停止页面的执行并输出相应的结果。
- Clear：清除页面缓冲区中的数据。
- Flush：将页面缓冲区中的数据立即显示。
- Redirect：客户端浏览器的 URL 地址重定向。

在 Response 的常用方法中，Write 方法是最常用的方法，Write 能够向客户端发送指定的 HTTP 流，并呈现给客户端浏览器，示例代码如下所示。

```
Response.Write("<div style=\"font-size:18px;\">这是一串<span style=\"color:red\">HTML</span>流</div>");
```

上述代码则会向浏览器输出一串 HTML 流并被浏览器解析，如图 13-3 所示。

图 13-3　Response.Write 方法

当希望在 Response 对象运行时，能够中途进行停止时，则可以使用 End 方法对页面的执行过程进行停止，示例代码如下所示。

```
for (int i=0; i < 100; i++)                          //循环 100 次
{
    if (i < 10)                                      //判断 i<10
    {
        Response.Write("当前输出了第" + i + "行<hr/>");   //i<10 则输出 i
    }
    else                                             //否则停止输出
    {
        Response.End();                              //使用了 End 方法停止执行
    }
}
```

上述代码循环输出 HTML 流“当前输出了第 X 行”，当输出到 10 行时，则停止输出，如图 13-4

所示。

图 13-4　Response.End 方法

Redirect 方法通常使用于页面跳转，示例代码如下所示。

```
Response.Redirect("http://www.shangducms.com");                    //页面跳转
```

执行上述代码，将会跳转到相应的 URL。

13.1.3　Application 状态对象

Application 对象是 HttpApplication 类的实例，将在客户端第一期从某个特定的 ASP.NET 应用程序虚拟目录中请求任何 URL 资源时创建。对于 Web 应用上的每个 ASP.NET 应用程序都要创建一个单独的实例。然后通过内部 Application 对象公开对每个实例进行引用。

1．Application 对象的特性

对于 Application 对象有如下特性：

- 数据可以在 Application 对象之内进行数据共享，一个 Application 对象可以覆盖多个用户。
- Application 对象可以用 Internet Service Manager 来设置而获得不同的属性。
- 单独的 Application 对象可以隔离出来并运行在内存之中。
- 可以停止一个 Application 对象而不会影响到其他 Application 对象。

Application 对象常用的属性有：

- AllKey：获取 HttpApplicationState 集合中的访问键。
- Count：获取 HttpApplicationState 集合中的对象数。

其中 Application 对象的常用方法有：

- Add：新增一个 Application 对象变量。
- Clear：清除全部的 Application 对象变量。
- Get：通过索引关键字或变量名称得到变量的值。

- GetKey：通过索引关键字获取变量名称。
- Lock：锁定全部的 Application 对象变量。
- UnLock：解锁全部的 Application 对象变量。
- Remove：使用变量名称移除一个 Application 对象变量。
- RemoveAll：移除所有的 Application 对象变量。
- Set：使用变量名更新一个 Application 对象变量。

2．Application 对象的使用

通过使用 Application 对象的方法，能够对 Application 对象进行操作，使用 Add 方法能够创建 Application 对象，示例代码如下所示。

```
Application.Add("App", "MyValue");                    //增加 Application 对象
Application.Add("App1", "MyValue1");                  //增加 Application 对象
Application.Add("App2", "MyValue2");                  //增加 Application 对象
```

若需要使用 Application 对象，可以通过索引 Application 对象的变量名进行访问，示例代码如下所示：

```
Response.Write(Application["App1"].ToString());       //输出 Application 对象
```

上述代码直接通过使用变量名来获取 Application 对象的值。通过 Application 对象的 Get 方法也能够获取 Application 对象的值，示例代码如下所示。

```
for (int i = 0; i < Application.Count; i++)           //遍历 Application 对象
{
    Response.Write(Application.Get(i).ToString());    //输出 Application 对象
}
```

Application 对象通常可以用来统计在线人数，在页面加载后可以通过配置文件使用 Application 对象的 Add 方法进行 Application 对象的创建，当用户离开页面时，可以使用 Application 对象的 Remove 方法进行 Application 对象的移除。当 Web 应用不希望用户在客户端修改已经存在的 Application 对象时，可以使用 Lock 对象进行锁定，当执行完毕相应的代码块后，可以解锁。示例代码如下所示。

```
Application.Lock();                                   //锁定 Application 对象
Application["App"] = "MyValue3";                      //Application 对象赋值
Application.UnLock();                                 //解锁 Application 对象
```

上述代码当用户进行页面访问时，其客户端的 Application 对象被锁定，所以用户的客户端不能够进行 Application 对象的更改。在锁定后，也可以使用 UnLock 方法进行解锁操作。

13.1.4 Session 状态对象

Session 对象是 HttpSessionState 的一个实例，Session 是用来存储跨页程序的变量或对象，功能基本同 Application 对象一样。但是 Session 对象的特性与 Application 对象不同。Session 对象变量只针对单一网页的使用者，这也就是说各个机器之间的 Session 的对象不尽相同。

例如用户 A 和用户 B，当用户 A 访问该 Web 应用时，应用程序可以显式的为该用户增加一个 Session 值，同时用户 B 访问该 Web 应用时，应用程序同样可以为用户 B 增加一个 Session 值。但是与 Application 不同的是，用户 A 无法存取用户 B 的 Session 值，用户 B 也无法存取用户 A 的 Session 值。Application 对象终止于 IIS 服务停止，但是 Session 对象变量终止于联机机器离线时，也就是说当网页使用者关闭浏览器或者网页使用者在页面进行的操作时间超过系统规定时，Session 对象将会自动注销。

1．Session 对象的特性

Session 对象常用的属性有：

- IsNewSession：如果用户访问页面时是创建新会话，则此属性将返回 true，否则将返回 false。
- TimeOut：传回或设置 Session 对象变量的有效时间，如果在有效时间内有没有任何客户端动作，则会自动注销。

注意：如果不设置 TimeOut 属性，则系统默认的超时时间为 20 分钟。

Session 对象常用的方法有：

- Add：创建一个 Session 对象。
- Abandon：该方法用来结束当前会话并清除对话中的所有信息，如果用户重新访问页面，则可以创建新会话。
- Clear：此方法将清除全部的 Session 对象变量，但不结束会话。

注意：Session 对象可以不需要 Add 方法进行创建，直接使用 Session[“变量名”]=变量值的语法也可以进行 Session 对象的创建。

2．Session 对象的使用

Session 对象可以使用于安全性相比之下较高的场合，例如后台登录。在后台登录的制作过程中，管理员拥有一定的操作时间，而如果管理员在这段时间不进行任何操作的话，为了保证安全性，后台将自动注销，如果管理员需要再次进行操作，则需要再次登录。在管理员登录时，如果登录成功，则需要给管理员一个 Session 对象，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Session["admin"] = "guojing";                                    //新增 Session 对象
    Response.Redirect("Session.aspx");                               //页面跳转
}
```

当管理员单击注销按钮时，则会注销 Session 对象并提示再次登录，示例代码如下所示。

```
protected void Button2_Click(object sender, EventArgs e)
{
    Session.Clear();                                                 //删除所有 Session 对象
    Response.Redirect("Session.aspx");
}
```

在 Page_Load 方法中，可以判断是否已经存在 Session 对象，如果存在 Session 对象，则说明管理员当前的权限是正常的，而如果不存在 Session 对象，则说明当前管理员的权限可能是错误的，或者是非法用户正在访问该页面，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] != null)                                    //如果 Session[“admin”]不为空
    {
        if (String.IsNullOrEmpty(Session["admin"].ToString()))       //则判断是否为空字符串
        {
            Button1.Visible = true;                                  //显式登录控件
            Button2.Visible = false;                                 //隐藏注销控件
            //Response.Redirect("admin_login.aspx");                 //跳转到登录页面
        }
        else
        {
            Button1.Visible = false;                                 //显式注销控件
            Button2.Visible = true;                                  //隐藏注销控件
```

```
            }
        }
    }
```

上述代码当管理员没有登录时，则会出现登录按钮，如果登录了，存在 Session 对象，则登录按钮被隐藏，只显示注销按钮。其 HTML 代码如下所示。

```
    <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="登录" />
    <asp:Button ID="Button2" runat="server" onclick="Button2_Click" Text="注销" />
```

上述代码运行后如图 13-5 和图 13-6 所示。

图 13-5　登录前

图 13-6　登录后

当再次单击【注销】按钮时则会清空 Session 对象，再次返回登录窗口时会呈现同图 13-5 所示。

13.1.5　Server 服务对象

Server 对象是 HttpServerUtility 的一个实例，该对象提供对服务器上的方法和属性进行访问。

1．Server 对象的常用属性

Server 对象的常用属性如下所示。

- MachineName：获取远程服务器的名称。
- ScriptTimeout：获取和设置请求超时。

通过 Server 对象能够获取远程服务器的信息，示例代码如下所示。

```
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write(Server.MachineName);                    //输出服务器信息
    }
```

上述代码运行后将会输出服务器名称，本例输出为“WIN-YXDGNGPG621”，这个输出结果根据服务器的名称不同而不同。

2．Server 对象的常用方法

Server 对象的常用方法如下所示。

- CreatObject：创建 COM 对象的一个服务器实例。
- Execute：使用另一个页面执行当前请求。
- Transfer：终止当前页面的执行，并为当前请求开始执行新页面。
- HtmlDecode：对已被编码的消除 Html 无效字符的字符串进行解码。
- HtmlEncode：对要在浏览器中显示的字符串进行编码。
- MapPath：返回与 Web 服务器上的执行虚拟路径相对应的物理文件路径。
- UrlDecode：对字符串进行解码，该字符串为了进行 HTTP 传输而进行编码并在 URL 中发送到服务器。

❑ UrlEncode：编码字符串，以便通过 URL 从 Web 服务器到客户端浏览器的字符串传输。

在 ASP.NET 中，默认编码是 UTF-8，所以在使用 Session 和 Cookie 对象保存中文字符或者其他字符集时经常会出现乱码，为了避免乱码的出现，可以使用 HtmlDecode 和 HtmlEncode 方法进行编码和解码。HTML 页面代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <p>
        HtmlDecode:
        <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
    </p>
    <p>
        HtmlEncode:
        <asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
    </p>
    </form>
</body>
```

上述代码使用了两个文本标签控件用来保存并呈现编码后和解码后的字符串，在 CS 页面可以对字符串进行编码和解码操作，示例代码如下所示。

```
string str = "<p>(*^__^*) 嘻嘻……</p>";                    //声明字符串
Label1.Text = Server.HtmlEncode(str);                      //字符串编码
Label2.Text = Server.HtmlDecode(Label1.Text);              //字符串解码
```

上述代码将 str 字符串进行编码并存放在 Label1 标签中，Label2 标签将读取 Label1 标签中的字符串再进行解码，运行后如图 13-7 所示。

图 13-7　HtmlEncode 和 HtmlDecode

在使用了 HtmlEncode 方法后，编码后的 HTML 标注会被转换成相应的字符，如符号“<”会被转换成字符“<”。在进行解码时，相应的字符会被转换回来，并呈现在客户端浏览器中。当需要让浏览器能够接受 HTML 字符时，URL 地址栏中对页面的参数的传递不能够包括空格，换行等符号，如果需要使用该符号，可以使用 UrlEncode 方法和 UrlDecode 方法进行变量的编码解码，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string str = Server.UrlEncode("错误信息 \n 操作异常");          //使用 UrlEncode 进行编码
    Response.Redirect("Server.aspx?str=" + str);                  //页面跳转
}
```

在 Page_Load 方法中可以接收该字符串，示例代码如下所示。

```
if (Request.QueryString["str"] != "")
{
```

```
        Label3.Text = Server.UrlDecode(Request.QueryString["str"]);        //使用 UrlDecode 进行解码
    }
```

当长字符串跳转和密封的信息在页面中进行发送和传递时，可以使用 UrlEncode 方法和 UrlDecode 方法进行变量的编码解码，以提高应用程序的安全性。

3．Server.MapPath 方法

在创建文件，删除文件或者读取文件类型的数据库时（如 Access 和 SQLite），都需要指定文件的路径并显式的提供物理路径执行文件的操作，如 D:\Program Files。但是这样做却暴露了物理路径，如果有非法用户进行非法操作，很容易就显示了物理路径，这样就造成了安全问题。

而如果在使用文件和创建文件时，如果非要为创建文件的保存地址设置一个物理路径，这样非常不便并且用户体验也不好。当用户需要上传文件时，用户不可能知道也不应该知道服务器路径。如果使用 MapPath 方法能够实现。MapPath 方法以“/”开头，则返回 Web 应用程序的根目录所在的路径，若 MapPath 方法以“../”开头，则会从当前目录开始寻找上级目录，如图 13-8 所示，而其实际服务器路径如图 13-9 所示。

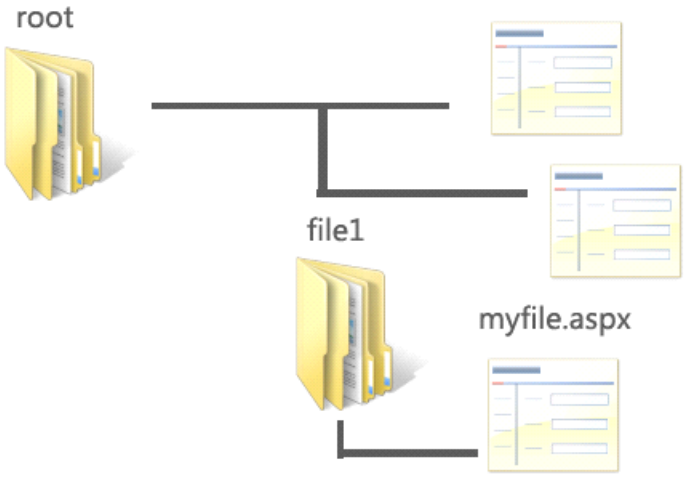

图 13-8　MapPath 示意图

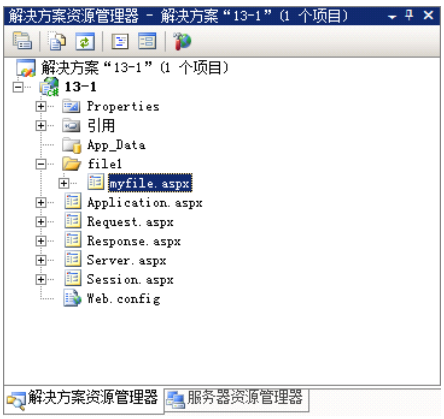

图 13-9　服务器路径

在图 13-8 所示，其中论坛根目录为 root，在根目录下有一个文件夹为 file1，在 file1 中的文件可以使用 MapPath 访问根目录中文件的方法有 Server.MapPath(“../文件名称”)或 Server.MapPath(“/文件名称”)，示例代码如下所示。

```
string FilePath = Server.MapPath("../Default.aspx");                    //设置路径
string FileRootPath = Server.MapPath("/Default.aspx");                  //设置路径
```

Server.MapPath 其实返回的是物理路径，但是通过 MapPath 的封装，通过代码无法看见真实的物理路径，若需要知道真实的物理路径，只需输出 Server.MapPath 即可，示例代码如下所示。

```
Response.Write(Server.MapPath("../Default.aspx"));                      //输出路径
```

上述代码输出结果为 D:\ASP.NET 3.5\源代码\第 13 章\13-1\13-1\Default.aspx，该结果针对不同的物理路径而不同。

13.1.6　Cookie 状态对象

Session 对象能够保存用户信息，但是 Session 对象并不能够持久的保存用户信息，当用户在限定时

间内没有任何操作时，用户的 Session 对象将被注销和清除，在持久化保存用户信息时，Session 对象并不适用。

1．Cookie 对象

使用 Cookie 对象能够持久化的保存用户信息，相比于 Session 对象和 Application 对象而言，Cookie 对象保存在客户端，而 Session 对象和 Application 对象保存在服务器端，所以 Cookie 对象能够长期保存。Web 应用程序可以通过获取客户端的 Cookie 的值来判断用户的身份来进行认证。

ASP.NET 内包含两个内部的 Cookie 集合。通过 HttpRequest 的 Cookies 集合来进行访问，Cookie 不是 Page 类的子类，所以使用方法和 Session 和 Application 不同。相比于 Session 和 Application 而言，Cookie 的优点如下所示。

- 可以配置到期的规则：Cookie 可以在浏览器会话结束后立即到期，也可以在客户端中无限保存。
- 简单：Cookie 是一种基于文本的轻量级结构，包括简单的键值对。
- 数据持久性：Cookie 能够在客户端上长期进行数据保存。
- 无需任何服务器资源：Cookie 无需任何服务器资源，存储在本地客户端中。

虽然 Cookie 包括若干优点，这些优点能够弥补 Session 对象和 Application 对象的不足，但是 Cookie 对象同样有缺点，Cookie 的缺点如下所示。

- 大小限制：Cookie 包括大小限制，并不能无限保存 Cookie 文件。
- 不确定性：如果客户端配置禁用 Cookie 配置，则 Web 应用中使用的 Cookie 将被限制，客户端将无法保存 Cookie。
- 安全风险：现在有很多的软件能够伪装 Cookie，这意味着保存在本地的 Cookie 并不安全，Cookie 能够通过程序修改为伪造，这会导致 Web 应用在认证用户权限时会出现错误。

Cookie 是一个轻量级的内置对象，Cookie 并不能将复杂和庞大的文本进行存储，在进行相应的信息或状态的存储时，应该考虑 Cookie 的大小限制和不确定性。

2．Cookie 对象的属性

Cookie 对象的属性如下所示：

- Name：获取或设置 Cookie 的名称。
- Value：获取或设置 Cookie 的 Value。
- Expires：获取或设置 Cookie 的过期的日期和事件。
- Version：获取或设置 Cookie 的符合 HTTP 维护状态的版本。

3．Cookie 对象的方法

Cookie 对象的方法如下所示：

- Add：增加 Cookie 变量。
- Clear：清除 Cookie 集合内的变量。
- Get：通过变量名称或索引得到 Cookie 的变量值。
- Remove：通过 Cookie 变量名称或索引删除 Cookie 对象。

4．创建 Cookie 对象

通过 Add 方法能够创建一个 Cookie 对象，并通过 Expires 属性设置 Cookie 对象在客户端中所持续的时间，示例代码如下所示。

```
HttpCookie MyCookie = new HttpCookie("MyCookie ");
MyCookie.Value = Server.HtmlEncode(“我的 Cookie 应用程序”);  //设置 Cookie 的值
MyCookie.Expires = DateTime.Now.AddDays(5);                  //设置 Cookie 过期时间
```

```
Response.AppendCookie(MyCookie);                    //新增 Cookie
```

上述代码创建了一个名称为MyCookie的Cookies，上述代码通过使用Response对象的AppendCookie方法进行Cookie对象的创建，与之相同，可以使用Add方法进行创建，示例代码如下所示。

```
Response.Cookies.Add(MyCookie);
```

上述代码同样能够创建一个Cookie对象，当创建了Cookie对象后，将会在客户端的Cookies目录下建立文本文件，文本文件的内容如下所示。

```
MyCookie
MyCookie
```

注意：Cookies目录在Windows下是隐藏目录，并不能直接对Cookies文件夹进行访问，在该文件夹中可能存在多个Cookie文本文件，这是由于在一些网站中进行登录保存了Cookies的原因。

5．获取Cookie对象

Web应用在客户端浏览器创建Cookie对象之后，就可以通过Cookie的方法读取客户端中保存的Cookies信息，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        HttpCookie MyCookie = new HttpCookie("MyCookie ");                          //创建 Cookie 对象
        MyCookie.Value = Server.HtmlEncode("我的 Cookie 应用程序");                  //Cookie 赋值
        MyCookie.Expires = DateTime.Now.AddDays(5);                                 //Cookie 持续时间
        Response.AppendCookie(MyCookie);                                            //添加 Cookie
        Response.Write("Cookies 创建成功");                                          //输出成功
        Response.Write("<hr/>获取 Cookie 的值<hr/>");
        HttpCookie GetCookie = Request.Cookies["MyCookie"];                         //获取 Cookie
        Response.Write("Cookies 的值:" + GetCookie.Value.ToString() + "<br/>");     //输出 Cookie 值
        Response.Write("Cookies 的过期时间:" + GetCookie.Expires.ToString() + "<br/>");
    }
    catch
    {
        Response.Write("Cookies 创建失败");                                          //抛出异常
    }
}
```

上述代码创建一个Cookie对象之后立即获取刚才创建的Cookie对象的值和过期时间。通过Request.Cookies方法可以通过Cookie对象的名称或者索引获取Cookie的值。

在一些网站或论坛中，经常使用到Cookie，当用户浏览并登录在网站后，如果用户浏览完毕并退出网站时，Web应用可以通过Cookie方法对用户信息进行保存。当用户再次登录时，可以直接获取客户端的Cookie的值而无需用户再次进行登录操作。

13.1.7 Cache缓存对象

Cache对象通过HttpContext对象的属性或Page对象的Cache属性来提供。Cache对于每个应用程序域均创建该类的实例，只要相应的应用程序域是激活状态，则该实例则为有效状态。

1．Cache对象的属性

Cache对象的属性如下所示：

- Count：获取存储在缓存中的Cache对象的项数。

- Item：获取或设置指定外键的缓存项。

2．Cache 对象的方法

Cache 对象的方法如下所示。

- Add：将指定的项添加到 Cache 对象，该对象具有依赖项，过期和优先级策略，以及一个委托。
- Get：从 Cache 对象检索指定项。
- Remove：从应用程序的 Cache 对象移除指定项。
- Insert：向 Cache 对象插入一个新项。

3．Cache 对象的使用

Cache 对象可以使用 Get 方法从相应的 Cache 对象中获取 Cache 对象的值，Get 方法能够通过 Cache 对象的名称和索引来获取 Cache 对象的值，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        Cache.Get("Label1.Text");                          //获取 Cache 对象的值
    }
    catch                                                  //捕获异常，同 try 使用
    {
        Label2.Text = "获取 Cache 的值失败!";               //输出错误异常信息
    }
}
```

通过 Cache 的 Count 属性能够获取现有的 Cache 对象的项数，示例代码如下所示。

```
Response.Write("Cache 对象的项数有" + Cache.Count.ToString());          //输出 Cache 项数
```

13.1.8 Global.asax 配置

Global.asax 配置文件也称作 ASP.NET 应用程序文件，该文件是可选文件。该文件包含用于相应 ASP.NET 或 HttpModule 引发的应用程序级别事件的代码。Global.asax 配置文件主流在基于 ASP.NET 应用程序的根目录中，在应用程序运行时，首先编译器会分析 Global.asax 配置文件并将其编译到一个动态生成的.NET Framework 类，该类是从 HttpApplication 基类派生的。Global.asax 配置文件不能够通过 URL 进行访问，以保证配置文件的安全性。

1．创建 Global.asax 配置文件

Global.asax 配置文件通常处理高级的应用程序事件，如 Application_Start、Application_End、Session_Start 等，Global.asax 配置文件通常不为个别页面或事件进行请求相应。创建 Global.asax 配置文件可以通过新建【全局应用程序类】文件来创建，如图 13-10 所示。

图 13-10　创建 Global.asax 配置文件

创建完成 Global.asax 配置文件，系统会自动创建一系列代码，开发人员只需要向相应的代码块中添加事务处理程序即可。

2．应用域开始（Application_Start）和应用域结束（Application_End）事件

在 Global.asax 配置文件中，Application_Start 事件会在 Application 对象被创建时触发，通常 Application_Start 对象能够对应用程序进行全局配置。在统计在线人数时，通过重写 Application_Start 方法可以实现实时在线人数统计，示例代码如下所示。

```
protected void Application_Start(object sender, EventArgs e)
{
    Application.Lock();                                          //锁定 Application 对象
    Application["start"] = "Application 对象被创建";              //创建 Application 对象
    Application.UnLock();                                        //解锁 Application 对象
}
```

当用户使用 Web 应用时，就会触发 Application_Start 方法，而与之相反的是，Application_End 事件在 Application 对象结束时被触发，示例代码如下所示。

```
protected void Application_End(object sender, EventArgs e)
{
    Application.Lock();                                          //锁定 Application 对象
    Application["end"] = "Application 对象被销毁";                //清除 Application 对象
    Application.UnLock();                                        //解锁 Application 对象
}
```

当用户离开当前的 Web 应用时，就会触发 Application_End 方法，开发人员能够在 Application_End 方法中清理相应的用户数据。

3．应用域错误（Application_Error）事件

Application_Error 事件在应用程序发送错误信息时被触发，通过重写该程序，可以控制 Web 应用程序的错误信息或状态，示例代码如下所示。

```
protected void Application_Error(object sender, EventArgs e)
{
    Application.Lock();                                          //锁定 Application 对象
    Application["error"] = "一个错误已经发生";                    //错误发生
    Application.UnLock();                                        //解锁 Application 对象
}
```

4．Session 开始（Session_Start）和 Session 结束（Session_End）事件

Session_Start 事件在 Session 对象开始时被触发。通过 Session_Start 事件可以统计应用程序当前访问的人数，同时也可以进行一些与用户配置相关的初始化工作，示例代码如下所示。

```
protected void Session_Start(object sender, EventArgs e)
{
    Session["count"] = 1;                                    //Session 开始执行
}
```

与之相反的是 Session_End 事件，当 Session 对象结束时则会触发该事件，当使用 Session 对象统计在线人数时，可以通过 Session_End 事件减少在线人数的统计数字，同时也可以对用户配置进行相关的清理工作，示例代码如下所示。

```
protected void Session_End(object sender, EventArgs e)
{
    Session["count"] = null;                                 //设置 Session 为 null
    Session.Clear();                                         //清除 Session 对象
}
```

上述代码当用户离开页面或者 Session 对象生命周期结束时被触发，在 Session_End 中可以清除用户信息进行相应的统计操作。

注意：Session 对象和 Application 对象都能够进行应用程序中在线人数或应用程序统计的统计和计算。在选择对象时，可以按照应用要求（特别是对象生命周期的要求）选择不同的内置对象。

13.2 ASP.NET 应用程序配置

ASP.NET 包含一个重要的特性，它为开发人员提供了一个非常方便的系统配置文件，就是常用的 Web.config 和 Machine.config。配置文件能够存用户或应用程序的储配置信息，让开发人员能够快速的建立 Web 应用环境，以及扩展 Web 应用配置。

13.2.1 ASP.NET 应用程序配置

ASP.NET 为开发人员提供了强大的灵活的配置系统，配置系统通常通过文件的形式存在于 Web 应用根目录下。这些配置文件通常包括两类，分别是 Web.config 和 Machine.config。Machine.config 是服务器配置文件。服务器配置信息通常存储在该文件中，该文件一般存储在系统目录中的“systemroot\Microsoft.NET\Framework\VersionNumber\CONFIG”目录下。一台服务器只有一个 Machine.config 文件，该文件描述了所有 ASP.NET Web 应用程序所需要的默认配置。

Web.config 是应用程序配置文件，该文件从 Machine.config 文件集成一部分基本配置，并且 Web.config 能够作为服务器上所有 ASP.NET 应用程序配置的跟踪配置文件。每个 ASP.NET 应用程序根目录都包含 Web.config 文件，所以对于每个应用程序的配置都只需要重写 Web.config 文件中的相应配置节即可。

在 ASP.NET 应用程序运行后，Web.config 配置文件按照层次结构为传入的每个 URL 请求计算惟一的配置设置集合。这些配置只会计算一次便缓存在服务器上。如果开发人员针对 Web.config 配置文件进行了更改，则很有可能造成应用程序重启。值得注意的是，应用程序的重启会造成 Session 等应用程序对象的丢失，而不会造成服务器的重启。

注意：如果针对 Web.config 文件中某些配置节（如 processModel 配置节）进行了更改，则可能需要重启 IIS 才能够让所做的应用程序配置立即生效。

13.2.2 Web.config 配置文件

ASP.NET 应用程序的配置信息都存放于 Web.config 配置文件中，Web.config 配置文件是基于 XML 格式的文件类型，由于 XML 文件的可伸缩性，使得 ASP.NET 应用配置变得灵活、高效、容易实现。同时，ASP.NET 不允许外部用户直接通过 URL 请求访问 Web.config，以提高应用程序的安全性。

1．Web.config 配置文件的优点

Web.config 配置文件使得 ASP.NET 应用程序的配置变得灵活、高效和容易实现，同时 Web.config 配置文件还为 ASP.NET 应用提供了可扩展的配置，使得应用程序能够自定义配置，不仅如此，Web.config 配置文件还包括以下优点。

- 配置设置易读性：由于 Web.config 配置文件是基于 XML 文件类型，所有的配置信息都存放在 XML 文本文件中，可以使用文本编辑器或者 XML 编辑器直接修改和设置相应配置节，相比之下，也可以使用记事本进行快速配置而无需担心文件类型。
- 更新的即时性：在 Web.config 配置文件中某些配置节被更改后，无需重启 Web 应用程序就可以自动更新 ASP.NET 应用程序配置。但是在更改有些特定的配置节时，Web 应用程序会自动保存设置并重启。
- 本地服务器访问：在更改了 Web.config 配置文件后，ASP.NET 应用程序可以自动探测到 Web.config 配置文件中的变化，然后创建一个新的应用程序实例。当浏览者访问 ASP.NET 应用时，会被重定向到新的应用程序。
- 安全性：由于 Web.config 配置文件通常存储的是 ASP.NET 应用程序的配置，所以 Web.config 配置文件具有较高的安全性，一般的外部用户无法访问和下载 Web.config 配置文件。当外部用户尝试访问 Web.config 配置文件时，会导致访问错误。
- 可扩展性：Web.config 配置文件具有很强的扩展性，通过 Web.config 配置文件，开发人员能够自定义配置节，在应用程序中自行使用。
- 保密性：开发人员可以对 Web.config 配置文件进行加密操作而不会影响到配置文件中的配置信息。虽然 Web.config 配置文件具有安全性，但是通过下载工具依旧可以进行文件下载，对 Web.config 配置文件进行加密，可以提高应用程序配置的安全性。

使用 Web.config 配置文件进行应用程序配置，极大的加强了应用程序的扩展性和灵活性，对于配置文件的更改也能够立即的应用于 ASP.NET 应用程序中。

2．Web.config 配置文件的结构

Web.config 配置文件是基于 XML 文件类型的文件，所以 Web.config 文件同样包含 XML 结构中的树形结构。在 ASP.NET 应用程序中，所有的配置信息都存储在 Web.config 文件中的“<configuration>”配置节中。在此配置节中，包括配置节处理应用程序声明，以及配置节设置两个部分，其中，对处理应用程序的声明存储在 configSections 配置节内，示例代码如下所示。

```
<configSections>
  <sectionGroup
      name="system.web.extensions"
      type="System.Web.Configuration.SystemWebExtensionsSectionGroup,
      System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <sectionGroup
```

```
            name="scripting"
            type="System.Web.Configuration.ScriptingSectionGroup,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
            <section
            name="scriptResourceHandler"
            type="System.Web.Configuration.ScriptingScriptResourceHandlerSection,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
            requirePermission="false" allowDefinition="MachineToApplication"/>
            <sectionGroup
            name="webServices"
            type="System.Web.Configuration.ScriptingWebServicesSectionGroup,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
            </sectionGroup>
          </sectionGroup>
        </sectionGroup>
      </configSections>
```

配置节设置区域中的每个配置节都有一个应用程序声明。节处理程序是用来实现ConfigurationSection 接口的.NET Framework 类。节处理程序生命中包括了配置设置接的名称，以及用来处理该配置节中的应用程序的类名。

配置节设置区域位于配置节处理程序声明区域之后。对配置节的设置还包括子配置节的是配置，这些子配置节同父配置节一起描述一个应用程序的配置，通常情况下这些同父配置节由同一个配置节进行管理，示例代码如下所示。

```
        <pages>
          <controls>
            <add tagPrefix="asp" namespace="System.Web.UI"
            assembly="System.Web.Extensions,
            Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
            <add tagPrefix="asp" namespace="System.Web.UI.WebControls"
            assembly="System.Web.Extensions,
            Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
          </controls>
        </pages>
```

虽然 Web.config 配置文件是基于 XML 文件格式的，但是在 Web.config 配置文件中并不能随意的自行添加配置节或者修改配置节的位置，例如 pages 配置节就不能存放在 configSections 配置节之中。在创建 Web 应用程序时，系统通常会自行创建一个 Web.config 配置文件在文件中，系统通常已经规定好了 Web.config 配置文件的结构。

13.2.3 ASP.NET 基本配置节

在 Web.config 配置文件中包括很多的配置节，这些配置节都用来规定 ASP.NET 应用程序的相应属性。

1．<configuration>：根配置节

所有 Web.config 的根配置节都存储与<configuration>标记中，在它内部封装了其他的配置节，示例代码如下所示。

```
<configuration>
            <syste.web>
```

```
        ……
</configuration>
```

2．<configSections>：处理声明配置节

该配置节主要用于自定义的配置节处理程序声明，该配置节由多个“<section>”配置节组成，示例代码如下所示。

```
<configSections>
  <sectionGroup
  name="system.web.extensions"
  type="System.Web.Configuration.SystemWebExtensionsSectionGroup,
  System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <sectionGroup name="scripting"
  type="System.Web.Configuration.ScriptingSectionGroup,
  System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
      <section name="scriptResourceHandler"
  type="System.Web.Configuration.ScriptingScriptResourceHandlerSection,
  System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
  PublicKeyToken=31BF3856AD364E35"
  requirePermission="false" allowDefinition="MachineToApplication"/>
    </sectionGroup>
  </sectionGroup>
</configSections>
```

其中<section>配置节包括 name 和 type 两种属性，name 属性指定配置数据配置节的名称，而 type 属性指定与 name 属性相关的配置处理程序类。

3．<appSettings>：用户自定义配置节

“<appSettings>”配置节为开发人员提供 ASP.NET 应用程序的扩展配置，通过使用“<appSettings>”配置节能够自定义配置文件，示例代码如下所示。

```
<appSettings>
    <add key="Name" value="Guojing"/>                                    //增加自定义配置节
    <add key="E-mail" value="soundbbg@live.cn"/>
</appSettings>
```

上述代码添加了两个自定义配置节，这两个自定义配置节分别为 Name 和 E-mail，用于定义该 Web 应用程序的开发者的信息，以便在其他页面中使用该配置节。

若需要在页面中使用该配置节，可以使用 ConfigurationSettings.appSettings(“key 的名称”)方法来获取自定义配置节中的配置值，示例代码如下所示。

```
    protected void Page_Load(object sender, EventArgs e)
    {
        TextBox1.Text = ConfigurationSettings.AppSettings["name"].ToString();      //获取自定义配置节
    }
```

“<appSettings>”配置节包括两个属性，分别为 Key 和 Value。Key 属性指定自定义属性的关键字，以方便在应用程序中使用该配置节，而 Value 属性则定义该自定义属性的值。

4．<customErrors>：用户错误配置节

该配置节能够指定当出现错误时，系统自动跳转到一个错误发生的页面，同时也能够为应用程序配置是否支持自定义错误。“<customErrors>”配置节包括两种属性，这两种属性分别为 mode 和 defaultRedirect。其中 mode 包括 3 种状态，这三种状态分别为 On、Off 和 RemoteOnly。On 表示启动自定义错误；Off 表示不启动自定义错误；RemoteOnly 表示给远程用户显示自定义错误。另外：

defaultRedirect 属性则配置了当应用程序发生错误时跳转的页面。

“<customErrors>”配置节还包括子配置节“<error>”，该标记用于特定状态的自定义错误页面，子标记“<error>”包括两个属性，分别为 statusCode 和 redirect，其中 statusCode 用于捕捉发生错误的状态码，而 redirect 指定发生该错误后跳转的页面，该配置节配置代码如下所示。

```
<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
    <error statusCode="403" redirect="NoAccess.htm" />
    <error statusCode="404" redirect="FileNotFound.htm" />
</customErrors>
```

上述代码则在 Web.config 文件中配置了相应的 customErrors 信息。当出现 404 错误时，系统会自行跳转到 FileNotFound.htm 页面以提示 404 错误，开发人员能够编写 FileNotFound.htm 页面进行用户提示 。

5．<globalization>：全局编码配置节

“<globalization>”用于配置应用程序的编码类型，ASP.NET 应用程序将使用该编码类型分析 ASPX 等页面，常用的编码类型包括：

- UFT-8：Unicode UTF-8 字节编码技术，ASP.NET 应用程序默认编码。
- UTF-16：Unicode UTF-16 字节编码技术。
- ASCII：标准的 ASCII 编码规范。
- Gb2312：中文字符 Gb2312 编码规范。

在配置“<globalization>”配置节时，其编码类型可以参考上述编码类型，如果不指定编码类型，则 ASP.NET 应用程序默认编码为 UTF-8，示例代码如下所示。

```
<globalization fileEncoding="UTF-8" requestEncoding="UTF-8" responseEncoding="UTF-8"/>
```

6．<sessionState>：Session 状态配置节

<sessionState>配置节用于完成 ASP.NET 应用程序中会话状态的设置，<sessionState>配置节包括以下 5 种属性：

- mode：指定会话状态的存储位置，一共有 Off、Inproc、StateServer 和 SqlServer 集中设置，Off 表示禁用该设置，Inproc 表示在本地保存会话状态，StateServer 表示在服务器上保存会话状态，SqlServer 表示在 SQL Server 保存会话设置。
- stateConnectionString：用来指定远程存储会话状态的服务器名和端口号。
- sqlConnectionString：用来连接 SQL Server 的连接字符串，当在 mode 属性中设置 SqlServer 时，则需要使用到该属性。
- Cookieless：指定是否使用客户端 cookie 保存会话状态。
- Timeout：指定在用户无操作时超时的时间，默认情况为 20 分钟。

<sessionState>配置节配置示例代码如下所示。

```
<sessionState mode="InProc" timeout="25" cookieless="false"></sessionState>
```

ASP.NET 不仅包括这些基本的配置节，还包括其他高级的配置节，高级的配置节通常用于指定界面布局样式，如母版页、默认皮肤、以及伪静态等高级功能。

13.3　ASP.NET 缓存功能

通常 Web 应用程序会处理大量的交互，在这些大量的交互中必然会造成频繁的数据处理。当 Web 应用程序中数据处理过于频繁时，会造成 Web 应用程序假死的状态，不仅如此，大量的重复请求还可能造成 Web 应用程序性能低下，这里就需要使用缓存减轻服务器压力。

13.3.1 缓存概述

为了防止不必要的数据处理，ASP.NET 允许开发人员将页面或数据进行缓存处理，当用户再次访问页面时，如果存在缓存则直接从缓存中获取用户信息或页面信息然后直接显示在客户端浏览器。这种缓存方式能够减少频繁的数据处理，减轻服务器压力。

1．应用程序缓存

应用程序缓存通过使用编程实现键/值对将任意数据存储在内存空间中。使用应用程序缓存与使用应用程序状态的方法类似。但是，与应用程序状态不相同，应用程序缓存中的数据是容易丢失的，应用程序缓存并不是在整个运行过程中都存在，应用程序缓存有一定的超时时间，当时间超过缓存设定的时间，缓存会自动注销。

2．页面输出缓存

在用户访问一个页面时，通常需要进行数据操作，例如网站的首页或其他信息页面，常常需要检索数据库中的数据并显示到页面中。如果每个用户的每次访问都需要进行一次数据操作的话，则当大量用户访问并进行数据操时将会导致性能降低甚至造成死锁。

为了避免这种情况的出现，可以使用页面输出缓存。页面输出缓存允许开发人员将整页进行数据缓存。页面输出缓存对于那些不经常更改的但需要处理大量数据和事件的页面特别适用。

3．缓存依赖

可以将缓存中的某一项的生存期配置为依赖于其他应用程序配置节，如某个文件或者数据库。当缓存依赖项被更改时，ASP.NET 将从缓存中移除对该项的数据缓存。例如网站静态页面，如果该页面数据需要进行缓存，可以保存为一个 HTML 文件，当页面被请求时，页面首先会判断是否有缓存依赖，如果存在，则执行相应的方法进行数据显示，当 HTML 文件被更改或被移除后，页面再次被请求，则会创建一个缓存依赖。

13.3.2 页面输出缓存

当用户访问页面时，整个页面将会被服务器保存在内存中，这样就对页面进行了缓存。当用户再次访问该页，页面不会再次执行数据操作，页面首先会检查服务器中是否存在缓存，如果缓存存在，则直接从缓存中获取页面信息，如果页面不存在，则创建缓存。

页面输出缓存适用于那些数据量较多，而不会进行过多的事件操作的页面，如果一个页面需要执行大量的事件更新，以及数据更新，则并不能使用页面输出缓存。使用@OutputCatch 指令能够声明页面输出缓存，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="none" %>
```

上述代码使用@OutputCatch 指令声明了页面缓存，该页面将被缓存 120 秒。@OutputCatch 指令包括 10 个属性，通过这些属性能够分别为页面的不同情况进行缓存设置，常用的属性如下所示：

- CacheProfile：获取或设置 OutputCacheProfile 名称。
- Duration：获取或设置缓存项需要保留在缓存中的时间。
- VaryByHeader：获取或设置用于改变缓存项的一组都好分隔的 HTTP 标头名称。
- Location：获取或设置一个值，该值确定缓存项的位置，包括 Any、Clint、Downstream、None、Server 和 ServerAndClient。默认值为 Any。
- VaryByControl：获取或设置一簇分好分隔的控件标识符，这些标识符包含在当前页或用户控件

内，用于改变当前的缓存项。

- NoStore：获取或设置一个值，该值确定是否设置了“Http Cache-Control：no-store”指令。
- VaryByCustom：获取输出缓存用来改变缓存项的自定义字符串列表。
- Enabled：获取或设置一个值，该值指示是否对当前内容启用了输出缓存。
- VaryByParam：获取查询字符串或窗体 POST 参数的列表。

通过设置相应的属性，可以为页面设置相应的缓存，当需要为 Default.aspx 设置缓存项时，可以使用 VaryByParam 属性进行设置，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="none" %>
```

上述代码使用了 Duration 属性和 VarByParam 属性设置了当前页的缓存属性。为一个页面进行整体的缓存设置往往是没有必要的，常常还会造成困扰，例如 Default.aspx?id=1 和 Default.aspx?id=100 在缓存时可能呈现的页面是相同的，这往往不是开发人员所希望的。通过配置 VarByParam 属性能够指定缓存参数，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="id" %>
```

上述代码则通过参数 id 进行缓存，当 id 项不同时，ASP.NET 所进行的页面缓存也不尽相同。这样保证了 Default.aspx?id=1 和 Default.aspx?id=100 在缓存时所显示的页面并不一致。VarByHeader 和 VarByCustom 主要用于根据访问页面的客户端对页面的外观或内容进行自定义。在 ASP.NET 中，一个页面可能需要为 PC 用户和 MOBILE 用户呈现输出，因此可以通过客户端的版本不同来缓存不同的数据，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="none" VaryByCustom="browser" %>
```

上述代码则为每个浏览器单独设置了缓存条目。

13.3.3 页面部分缓存

整页缓存在很多情况是不可行的，例如在留言本程序中。当用户第一次访问该页面，该页面就会被缓存在服务器内存中，当用户进行评论并进行刷新，当前页面还在缓存的生存周期中，页面不会立即显示刚才用户发表的评论，这种情况可能造成用户困扰。在这种情况下，就应该针对页面中不同的部分进行缓存，如对用户评论框进行缓存或数据绑定控件进行和 HTML 控件进行缓存，这样也可以减少数据操作次数。示例代码如下所示

```
<%@ OutputCache Duration="120" VaryByParam="*"%>
```

上述代码将缓存用户控件 120 秒，并且将针对查询字符串的每个变动进行缓存。

```
<%@ OutputCache Duration="120" VaryByParam="none" VaryByCustom="browser" %>
```

上述代码针对浏览器设置缓存条目并缓存 120 秒，当浏览器不同时，则会分别创建独立的缓存条目。

```
<%@ OutputCache Duration="120"
VaryByParam="none" VaryByCustom="browser" VaryByControl="TextBox1"%>
```

上述代码将服务器控件 TextBox1 的每个不同的值进行缓存处理。页面部分缓存是指输出存在页面的某些部分，而不是缓存整个页面的内容。页面部分缓存包括 3 种方法：

- 使用@OutputCatch 指令声明方式为用户控件设置缓存功能。
- 在代码隐藏文件中使用 PartialCachingAttribute 类设置用户控件缓存。
- 使用 ControlCachePolicy 类以编程的方式指定用户缓存。

页面部分缓存与页面缓存在很多方面都很相似，其属性基本相同，页面部分缓存主要使用@ OutputCache 对象的属性进行缓存设置。

当对页面进行页面缓存时，通常需要针对不同的 POST 参数进行缓存，如果不针对参数单独的进行缓存，则形成应用程序错误，而这种错误不是逻辑上的错误，如 http://localhost/Default.aspx?user=guojing

所呈现的页面和 http://localhost/Default.aspx?user=wujunmin 会给用户呈现相同的页面，就会导致用户困扰。使用 VaryByParam 属性可以解决这个问题，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="name"%>
```

当存在多个 POST 参数时，可以用都好分隔，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="name"%>
```

当需要为每个执行的变动进行缓存时，可以使用*进行缓存设置，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="*"%>
```

页面部分缓存通常应用于用户控件而不是 Web 窗体，除了 Location 属性，支持所有 OutputCache 属性。用户控件还支持名为 VaryByControl 的 OutputCatch 属性，该属性将根据用户控件的成员的值改变该控件的缓存。如果一个用户控件不随应用程序中的页面而改变，则需要使用 Shared=”true”参数。

13.3.4 应用程序数据缓存

应用程序缓存是由 Cache 类实现的，每个应用程序对象可以专用一个缓存实例。缓存生存期依赖于应用程序的生存期，如果应用程序重启，则会重新创建 Cache 对象。

1. 创建 Cache 对象

应用程序数据缓存是由 System.Web.Caching.Cache 类实现。该类提供了简单的字典接口。通过这个接口可以设置缓存的有效期、依赖项以及优先级等特性。Cache 对象的属性如下所示：

- Count：获取存储在缓存中的 Cache 对象的项数。
- Item：获取或设置指定外键的缓存项。

Cache 对象的方法如下所示。

- Add：将指定的项添加到 Cache 对象，该对象具有依赖项，过期和优先级策略，以及一个委托。
- Get：从 Cache 对象检索指定项。
- Remove：从应用程序的 Cache 对象移除指定项。
- Insert：向 Cache 对象插入一个新项。

Cache 方法最简单的赋值方法就是使用一个键并为其赋值，示例代码如下所示。

```
Cache["key"] = "value";                                    //创建缓存项
```

增加缓存同样可以使用 Cache 对象的 Add 方法向缓存添加项。Add 方法能够设置与 Insert 方法相同的所有选项。使用 Add 方法将返回添加到缓存中的对象，如果在缓存中已经存在该项，则 Add 方法不会替换该项，所以该操作不会产生异常。

使用 Add 方法需要为 Add 方法传递参数，这些参数包括依赖项、过期时间和缓存的优先级策略等。若只需要实现创建 Cache 方法，推荐使用 Cache 对象的 Insert 方法，示例代码如下所示。

```
Cache.Insert("key", "value");                              //插入缓存项
```

上述代码将在缓存中存储项，但是上述代码不带任何的依赖项，所以该缓存不会到期，除非缓存引擎为了给其他的缓存数据提供空间而将其删除，如果需要创建包含依赖项的缓存，则需要使用 Add 方法 。

2. 创建带依赖项的 Cache 对象

创建 Cache 对象可以通过依赖项创建对象，使用 Cache.Insert 方法创建缓存不带任何依赖项，所以该缓存不会到期。如果需要创建带依赖项的 Cache 对象，可以使用 Cache.Insert 方法的重载函数，示例代码如下所示。

```
Cache.Insert("key","xml file value",
new System.Web.Caching.CacheDependency(Server.MapPath("XmlData.xml")));      //插入缓存项
```

上述代码将“xml file value”字符创插入缓存，无需在以后的请求中从文件读取缓存信息。

CacheDependency 的作用是确保缓存在文件更改后立即到期，从而能够从文件中提取最新的数据进行数据缓存，如果缓存数据来自若干个文件，还可以指定一个文件名数组，示例代码如下所示。

```
string[] dependencies = { "CacheItem1", "CacheItem2", "CacheItem3" };          //指定数组
Cache.Insert("key","xml file value",
new System.Web.Caching.CacheDependency(null,dependencies));                    //插入缓存项
```

创建带依赖项的 Cache 对象时，可以同时创建多个依赖项。该 Cache 对象向缓存中名为 CacheItem1 等数组配置节的另一项添加依赖项，同时也向 XmlData.xml 文件添加文件依赖项，示例代码如下所示。

```
System.Web.Caching.CacheDependency
mydep1 = new System.Web.Caching.CacheDependency(Server.MapPath("XmlData.xml")); //创建缓存
string[] Mydependencies = { "CacheItem1", "CacheItem2", "CacheItem3" };        //创建数组
System.Web.Caching.CacheDependency
mydep2 = new System.Web.Caching.CacheDependency(null, Mydependencies);         //创建依赖
System.Web.Caching.AggregateCacheDependency
aggDep = new System.Web.Caching.AggregateCacheDependency();                    //创建依赖
aggDep.Add(mydep1);                                                            //增加依赖
aggDep.Add(mydep2);
Cache.Insert("Cache", "CacheItem", aggDep);                                    //插入缓存
```

上述代码创建了 Cache 对象，并向 CacheItem1 等数组配置节的另一项添加依赖项，同时也向 XmlData.xml 文件添加文件依赖项。

13.3.5 检索应用程序数据缓存对象

要从缓存中检索数据，应指定存储缓存项的键。由于缓存中所存储的信息容易丢失，即该缓存信息有可能被 ASP.NET 移除，因此开发人员首先应该确定该项是否已经存在与缓存，如果不存在缓存，则首先应该创建一个缓存项，然后再检索该项。示例代码如下所示。

```
string cache = Cache["key"].ToString();                                        //获取缓存
if (cache != null)                                                             //判断缓存
{
    Response.Write(cache);                                                     //输出缓存
}
else
{
    Cache["key"] = "value";                                                    //创建缓存
    Response.Write(Cache["key"].ToString());                                   //输出缓存
}
```

上述代码从缓存项“key”中获取缓存的值，如果缓存的值为空置，则创建一个缓存。使用 Cache 类的 Get 方法也可以获取被缓存的数据对象，如果通过 Get 方法返回缓存中的数据对象，则返回的类型为 object 类型，示例代码如下所示。

```
string cache = Cache["key"].ToString();                                        //获取缓存
if (cache != null)                                                             //判断缓存
{
    Response.Write(cache);                                                     //输出缓存
}
else
{
    Cache["key"] = "value";                                                    //创建缓存
    Response.Write(Cache.Get("key").ToString());                               //Get 缓存
```

```
}
```

Cache 类的 Get 方法可以获取被缓存的数据对象，如果使用 Cache 类的 GetEnumerator 方法则返回一个 IDictionaryEnumerator 对象，该对象可以用于循环访问包含在缓存中的键值设置及其值的枚举类型。示例代码如下所示。

```
IDictionaryEnumerator cacheEnum = Cache.GetEnumerator();                    //定义枚举
while (cacheEnum.MoveNext())                                                //遍历枚举
{
    cache = Server.HtmlEncode(cacheEnum.Current.ToString());                //输出缓存
    Response.Write(cache);
}
```

上述代码则通过使用 Cache 类的 GetEnumerator 方法则返回一个 IDictionaryEnumerator 对象给 cacheEnum 变量，并通过 cacheEnum 变量的 MoveNext 方法进行缓存遍历。

13.4 小结

本章讲解了 ASP.NET 内置对象，包括如何创建 ASP.NET 内置对象和使用 ASP.NET 内置对象。Web 应用程序从本质上来讲是无状态的，为了维持客户端的状态，必须使用 ASP.NET 内置对象进行客户端状态维护，这些状态包括 Session、Cookies 等。本章还包括：

- ASP.NET 内置对象：包括 Session、Cookies 等内置对象。
- ASP.NET 应用程序配置：包括 ASP.NET 应用程序配置。
- Web.config 配置文件：讲解了 Web.config 配置文件中常用的属性。
- 应用程序数据缓存：讲解了应用程序数据缓存。
- 检索应用程序数据缓存对象：讲解了从缓存中获取应用程序数据。

本章还讲解了缓存中的 Cache 对象和 Web.config 配置节，开发人员能够快速的构建 ASP.NET 应用环境并进行性能优化。