领悟 Web 设计模式

本文发表在《程序春秋》2004 年 1 期

摘要

本文介绍了在.NET 框架下应用 Web 设计模式改进 WebForm 程序设计的一些基本方法及要点。

关键字

设计模式，ASP.NET，WebForm，MVC，Page Controller，Front Controller，Page Cache

目录

引言

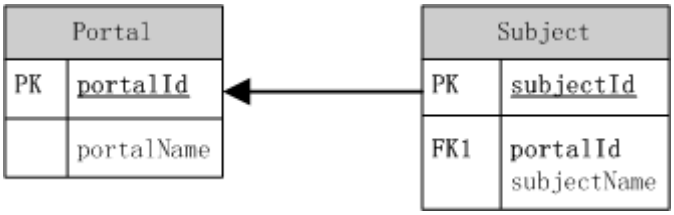
记得微软刚刚推出 ASP.NET 时，给人的震撼是开发 Web 程序不再是编写传统的网页，而像是在构造应用程序，因而微软称之为 WebForm。但是两年后的今天，有相当多的开发人员仍然延用写脚本程序的思路构建一个又一个的 WebForm，而没有发挥出 ASP.NET 的优势，就此本文希望通过实例能够启发读者一些新的思路。

由于篇幅有限，本文不可能通过一个复杂的 Web 应用来向读者展示结合设计模式的 WebForm，但是如果仅仅是一个小程序的确没有使用模式的必要。为了便于理解，希望您能把它想象成是一个大型系统中的小模块（如果代码是大型系统的一部分那么使用模式就变得非常重要）。

在本文的末尾给出了所有源程序的下载地址。

经典的 WebForm 架构

首先来看一个简单的应用，数据库设计如下图，Portal 是 Subject 的父表，通过 portalId 进行一对多关联，程序需要根据 portalId 显示不同的 Subject 列表。



按照我们编写 WebForm 一般的习惯，首先在页面上拖放一个 DropDownList、一个 DataGrid、一个 Button 控件：

界面（webForm.aspx）：

```
<form id="webForm" method="post" runat="server">
    <asp:DropDownList id="dropDownList" runat="server"></asp:DropDownList>
    <asp:Button id="button" runat="server" Text="Button"></asp:Button>
```

```
    <asp:DataGrid id="dataGrid" runat="server"></asp:DataGrid>
</form>
```

然后利用VS.NET代码隐藏功能编写的核心代码如下：

后置代码（webForm.aspx.cs）：

```
//页面初始化事件
private void Page_Load(object sender, System.EventArgs e)
{
    if ( ! IsPostBack )
    {
        string SQL_SELECT_PORTAL = "SELECT * FROM PORTAL";
        //使用using确保释放数据库连接
        //连接字符串存放在Web.Config文件中便于修改
        using( SqlConnection conn = new
SqlConnection( ConfigurationSettings.AppSettings["ConnectionString"] ) )
        {
            SqlDataAdapter dataAdapter = new SqlDataAdapter( SQL_SELECT_PORTAL, conn );
            DataSet dataSet = new DataSet();
            dataAdapter.Fill( dataSet );
            //设置下拉列表的数据源与文本域、值域
            dropDownList.DataSource = dataSet;
            dropDownList.DataTextField = "portalName";
            dropDownList.DataValueField = "portalId";
            dropDownList.DataBind();
        }
    }
}
//Button的Click事件
private void button_Click(object sender, System.EventArgs e)
{
    string SQL_SELECT_SUBJECT = "SELECT * FROM SUBJECT WHERE portalId = {0}";
    using( SqlConnection conn = new
SqlConnection( ConfigurationSettings.AppSettings["ConnectionString"] ) )
    {
        //用下拉列表选择的值替换掉SQL语句中的待定字符{0}
        SqlDataAdapter dataAdapter = new
SqlDataAdapter( string.Format( SQL_SELECT_SUBJECT, dropDownList.SelectedValue ), conn );
        DataSet dataSet = new DataSet();
        dataAdapter.Fill( dataSet );
        dataGrid.DataSource = dataSet;
        dataGrid.DataBind();
    }
}
```

国内 Button

subjectId	portalId	subjectName
1	1	要闻
2	1	时事

执行结果如图所示，程序将根据下拉列表框选择的值绑定 DataGrid，非常典型的一个 WebForm 架构，体现出 ASP.NET 事件驱动的思想，实现了界面与代码的分离。但是仔细看看可以从中发现几个问题：

- 对数据库操作的代码重复，重复代码是软件开发中绝对的“坏味道”，往往由于某些原因当你修改了一处代码，却忘记要更改另外一处相同的代码，从而给程序留下了 Bug 的隐患。
- 后置代码完全依赖于界面，在 WebForm 下界面的变化远远大于数据存储结构和访问的变化，当界面改变时您将不得不修改代码以适应新的页面，有可能将会重写整个后置代码。
- 后置代码不仅处理用户的输入而且还负责了数据的处理，如果需求发生变更，比如需要改变数据的处理方式，那么你将几乎重写整个后置代码。

一个优秀的设计需要每一个模块，每一种方法只专注于做一件事，这样的结构才清晰，易修改，毕竟项目的需求总是在不断变更的，“唯一不变的就是变化本身”，好的程序一定要为变化作出准备，避免“牵一发而动全身”，所以一定要想办法解决上述问题，下面让我们来看看设计模式。

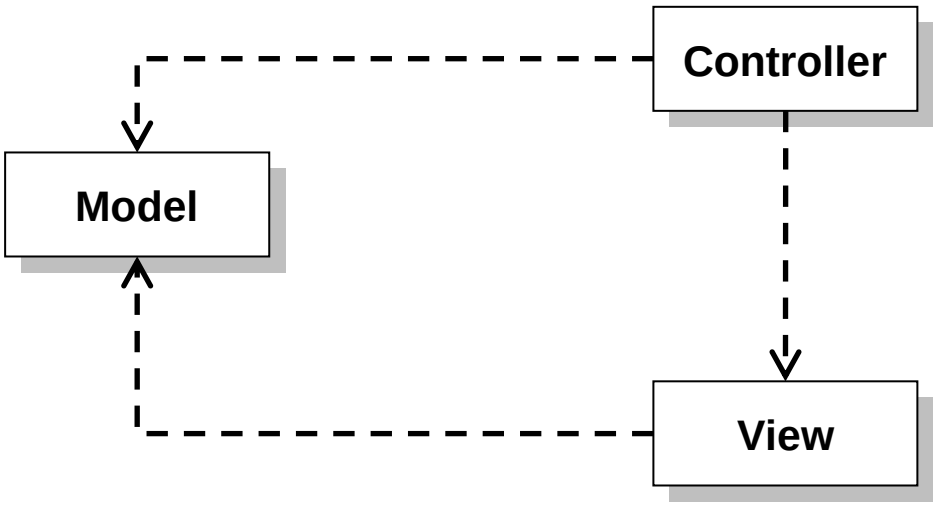
设计模式

设计模式描述了一个不断重复出现的问题以及对该问题的核心解决方案，它是成功的构架、设计及实施方案，是经验的总结。设计模式的概念最早来自于西方建筑学，但最成功的案例首推中国古代的“三十六计”。

MVC 模式下的 WebForm

MVC 模式是一个用于将用户界面逻辑与业务逻辑分离开来的基础设计模式，它将数据处理、界面以及用户的行为控制分为：Model－View－Controller。

- Model：负责当前应用的数据获取与变更及相关的业务逻辑
- View：负责显示信息
- Controller：负责收集转化用户的输入



View 和 Controller 都依赖于 Model，但是 Model 既不依赖于 View，也不依赖于 Controller，这是分离的主要优点之一，这样 Model 可以单独的建立和测试以便于代码复用，View 和 Controller 只需要 Model

提供数据，它们不会知道、也不会关心数据是存储在 SQL Server 还是 Oracle 数据库中或者别的什么地方。

根据 MVC 模式的思想，可以将上面例子的后置代码拆分为 Model 和 Controller，用专门的一个类来处理数据，后置代码作为 Controller 仅仅负责转化用户的输入，修改后的代码为：

Model（SQLHelper.cs）： 封装所有对数据库的操作。

```
private static string SQL_SELECT_PORTAL = "SELECT * FROM PORTAL";
private static string SQL_SELECT_SUBJECT = "SELECT * FROM SUBJECT WHERE portalId = {0}";
private static string SQL_CONNECTION_STRING =
ConfigurationSettings.AppSettings["ConnectionString"];
public static DataSet GetPortal()
{
    return GetDataSet( SQL_SELECT_PORTAL );
}
public static DataSet GetSubject( string portalId )
{
    return GetDataSet( string.Format( SQL_SELECT_SUBJECT, portalId ) );
}
public static DataSet GetDataSet( string sql )
{
    using( SqlConnection conn = new SqlConnection( SQL_CONNECTION_STRING ) )
    {
        SqlDataAdapter dataAdapter = new SqlDataAdapter( sql, conn );
        DataSet dataSet = new DataSet();
        dataAdapter.Fill( dataSet );
        return dataSet;
    }
}
```

Controller（webForm.aspx.cs）： 负责转化用户的输入

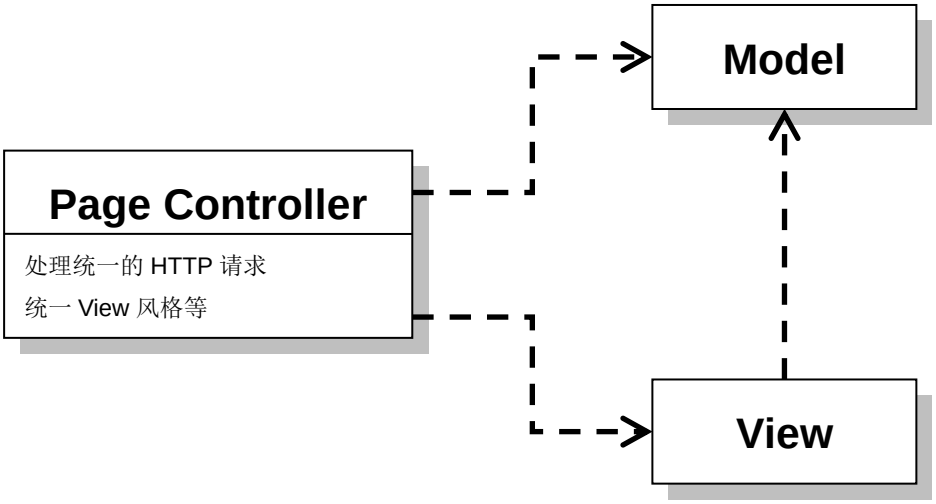
```
private void Page_Load(object sender, System.EventArgs e)
{
    if ( ! IsPostBack )
    {
        //调用Model的方法获得数据源
        dropDownList.DataSource = SQLHelper.GetPortal();
        dropDownList.DataTextField = "portalName";
        dropDownList.DataValueField = "portalId";
        dropDownList.DataBind();
    }
}
private void button_Click(object sender, System.EventArgs e)
{
    dataGrid.DataSource = SQLHelper.GetSubject( dropDownList.SelectedValue );
    dataGrid.DataBind();
}
```

修改后的代码非常清晰，M-V-C各司其制，对任意模块的改写都不会引起其他模块的变更，类似于MFC中Doc/View结构。但是如果相同结构的程序很多，而我们又需要做一些统一的控制，如用户身份的判断，统一的界面风格等；或者您还希望Controller与Model分离的更彻底，在Controller中不涉及到Model层的代码。此时仅仅靠MVC模式就显得有点力不从心，那么就请看看下面的Page Controller模式。

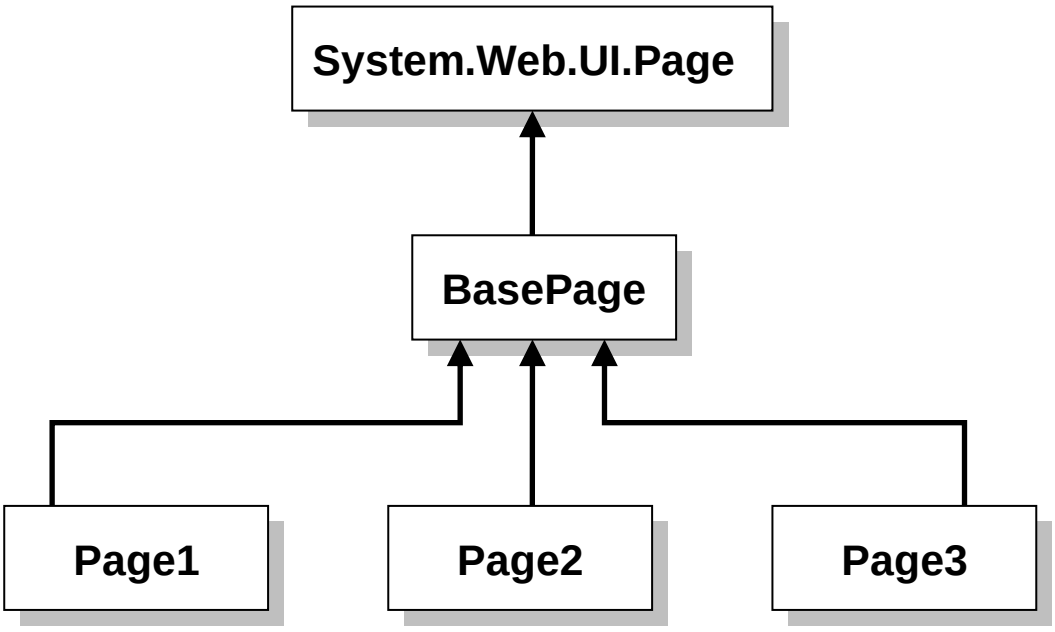
Page Controller 模式下的 WebForm

MVC 模式主要关注Model与View之间的分离，而对于Controller的关注较少（在上面的MVC模式中我们仅仅只把Model和Controller分离开，并未对Controller进行更多的处理），但在基于WebForm的应用程序中，View和Controller本来就是分隔的（显示是在客户端浏览器中进行），而Controller是服务器端应用程序；同时不同用户操作可能会导致不同的Controller策略，应用程序必须根据上一页面以及用户触发的事件来执行不同的操作；还有大多数WebForm都需要统一的界面风格，如果不对此处理将可能产生重复代码，因此有必要对Controller进行更为仔细的划分。

Page Controller模式在MVC模式的基础上使用一个公共的页基类来统一处理诸如Http请求，界面风格等，如图：



传统的WebForm一般继承自System.Web.UI.Page类，而Page Controller的实现思想是所有的WebForm继承自定义页面基类，如图：



利用自定义页面基类，我们可以统一的接收页面请求、提取所有相关数据、调用对Model的所有更新以及向View转发请求，轻松实现统一的页面风格，而由它所派生的Controller的逻辑将变得更简单，更具体。

下面看一下Page Controller的具体实现：

Page Controller（BasePage.cs）：

```
public class BasePage : System.Web.UI.Page
{
    private string _title;
    public string Title//页面标题，由子类负责指定
    {
        get
        {
            return _title;
        }
        set
        {
            _title = value;
        }
    }
    public DataSet GetPortalDataSource()
    {
        return SQLHelper.GetPortal();
    }
    public DataSet GetSubjectDataSource( string portalId )
    {
        return SQLHelper.GetSubject( portalId );
    }
    protected override void Render( HtmlTextWriter writer )
    {
        writer.Write( "<html><head><title>" + Title + "</title></head><body>" );//统一的页面头
        base.Render( writer );//子页面的输出
        writer.Write( @"<a href=""http://www.asp.net"">ASP.NET</a></body></html>" );//统一的页面尾
    }
}
```

现在它封装了Model的功能，实现了统一的页面标题和页尾，子类只须直接调用：

修改后的Controller（webForm.aspx.cs）：

```
public class webForm : BasePage//继承页面基类
{
    private void Page_Load(object sender, System.EventArgs e)
    {
        Title = "Hello, World!";//指定页面标题
        if ( ! IsPostBack )
```

```
            {
                dropDownList.DataSource = GetPortalDataSource();//调用基类的方法
                dropDownList.DataTextField = "portalName";
                dropDownList.DataValueField = "portalId";
                dropDownList.DataBind();
            }
        }
        private void button_Click(object sender, System.EventArgs e)
        {
            dataGrid.DataSource = GetSubjectDataSource( dropDownList.SelectedValue );
            dataGrid.DataBind();
        }
    }
```

从上可以看出BagePage Controller接管了大部分原来Controller的工作，使Controller变得更简单，更容易修改（为了便于讲解我没有把控件放在BasePage中，但是您完全可以那样做），但是随着应用复杂度的上升，用户需求的变化，我们很容易会将不同的页面类型分组成不同的基类，造成过深的继承树；又例如对于一个购物车程序，需要预定义好页面路径；对于向导程序来说路径是动态的（事先并不知道用户的选择）。

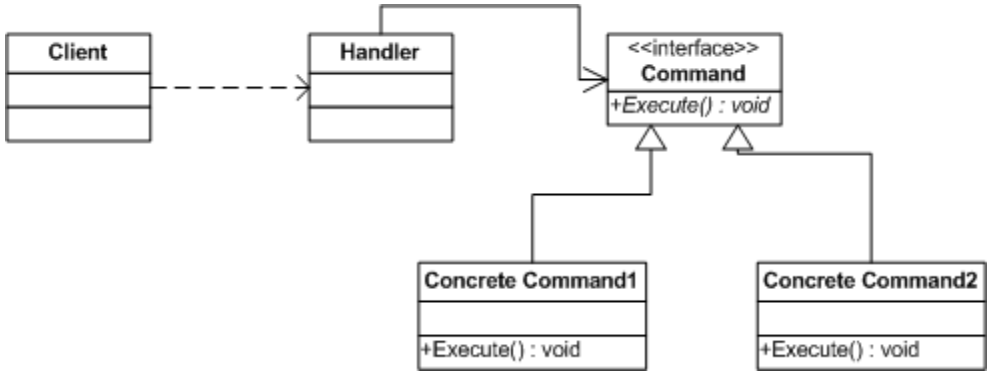
面对以上这些应用来说仅仅使用Page Controller还是不够的，接下来再看看Front Controller模式。

Front Controller模式下的WebForm

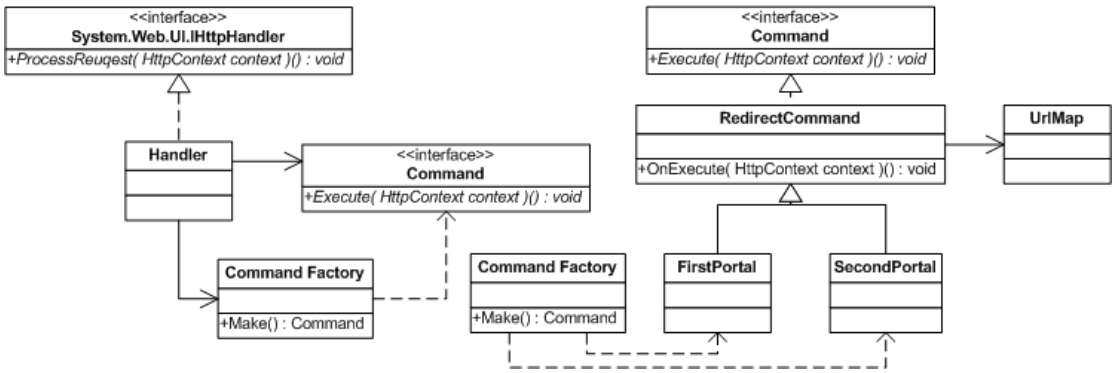
Page Controller的实现需要在基类中为页面的公共部分创建代码，但是随着时间的推移，需求会发生较大的改变，有时不得不增加非公用的代码，这样基类就会不断增大，您可能会创建更深的继承层次结构以删除条件逻辑，这样一来我们很难对它进行重构，因此需要更进一步对Page Controller进行研究。

Front Controller通过对所有请求的控制并传输解决了在Page Controller中存在的分散化处理的问题，它分为Handler和Command树两个部分，Handler处理所有公共的逻辑，接收HTTP Post或Get请求以及相关的参数并根据输入的参数选择正确的命令对象，然后将控制权传递到Command对象，由其完成后面的操作，在这里我们将使用到Command模式。

Command模式通过将请求本身变成一个对象可向未指定的应用对象提出请求，这个对象可被存储并像其他的对象一样被传递，此模式的关键是一个抽象的Command类，它定义了一个执行操作的接口，最简单的形式是一个抽象的Execute操作，具体的Command子类将接收者作为其一个实例变量，并实现Execute操作，指定接收者采取的动作，而接收者具有执行该请求所需的具体信息。



因为Front Controller模式要比上面两个模式复杂一些，我们再来看看例子的类图：



关于Handler的原理请查阅MSDN，在这就不多讲了，我们来看看Front Controller模式的具体实现：

首先在Web.Config里定义：

```
<!-- 指定对Dummy开头的aspx文件交由Handler处理 -->
<httpHandlers>
    <add verb="*" path="/WebPatterns/FrontController/Dummy*.aspx"
type="WebPatterns.FrontController.Handler,WebPatterns"/>
</httpHandlers>
<!-- 指定名为FrontControllerMap的页面映射块，交由UrlMap类处理，程序将根据key找到对应的url
作为最终的执行路径，您在这可以定义多个key与url的键值对 -->
<configSections>
    <section name="FrontControllerMap" type="WebPatterns.FrontController.UrlMap,
WebPatterns"></section>
</configSections>
<FrontControllerMap>
    <entries>
        <entry key="/WebPatterns/FrontController/DummyWebForm.aspx"
url="/WebPatterns/FrontController/ActWebForm.aspx" />
        。。。
    </entries>
</FrontControllerMap>
```

修改webForm.aspx.cs：

```
private void button_Click( object sender, System.EventArgs e )
{
    Response.Redirect( "DummyWebForm.aspx?requestParm=" + dropDownList.SelectedValue );
}
```

当程序执行到这里时将会根据Web.Config里的定义触发类Handler的ProcessRequest事件：

Handler.cs：

```
public class Handler : IHttpHandler
{
    public void ProcessRequest( HttpContext context )
    {
        Command command = CommandFactory.Make( context.Request.Params );
        command.Execute( context );
    }
    public bool IsReusable
```

```
    {
        get
        {
            return true;
        }
    }
}
```

而它又会调用类CommandFactory的Make方法来处理接收到的参数并返回一个Command对象，紧接着它又会调用该Command对象的Execute方法把处理后参数提交到具体处理的页面。

```
public class CommandFactory
{
    public static Command Make( NameValueCollection parms )
    {
        string requestParm = parms["requestParm"];
        Command command = null;
        //根据输入参数得到不同的Command对象
        switch ( requestParm )
        {
            case "1" :
                command = new FirstPortal();
                break;
            case "2" :
                command = new SecondPortal();
                break;
            default :
                command = new FirstPortal();
                break;
        }
        return command;
    }
}
public interface Command
{
    void Execute( HttpContext context );
}
public abstract class RedirectCommand : Command
{
    //获得Web.Config中定义的key和url键值对，UrlMap类详见下载包中的代码
    private UrlMap map = UrlMap.SoleInstance;
    protected abstract void OnExecute( HttpContext context );
    public void Execute( HttpContext context )
    {
        OnExecute( context );
        //根据key和url键值对提交到具体处理的页面
```

```
            string url = String.Format( "{0}?{1}",
map.Map[ context.Request.Url.AbsolutePath ], context.Request.Url.Query );
            context.Server.Transfer( url );
        }
}
public class FirstPortal : RedirectCommand
{
        protected override void OnExecute( HttpContext context )
        {
            //在输入参数中加入项portalId以便页面处理
            context.Items["portalId"] = "1";
        }
}
public class SecondPortal : RedirectCommand
{
        protected override void OnExecute(HttpContext context)
        {
            context.Items["portalId"] = "2";
        }
}
```

最后在ActWebForm.aspx.cs中：

```
dataGrid.DataSource =
GetSubjectDataSource( HttpContext.Current.Items["portalId"].ToString() );
dataGrid.DataBind();
```

上面的例子展示了如何通过Front Controller集中和处理所有的请求，它使用CommandFactory来确定要执行的具体操作，无论执行什么方法和对象，Handler只调用Command对象的Execute方法，您可以在不修改 Handler的情况下添加额外的命令。它允许让用户看不到实际的页面，当用户输入一个URL时，然后系统将根据web.config文件将它映射到特定的URL，这可以让程序员有更大的灵活性，还可以获得Page Controller实现中所没有的一个间接操作层。

对于相当复杂的Web应用我们才会采用Front Controller模式，它通常需要将页面内置的Controller替换为自定义的Handler，在Front Controllrer模式下我们甚至可以不需要页面，不过由于它本身实现比较复杂，可能会给业务逻辑的实现带来一些困扰。

以上两个Controller模式都是处理比较复杂的WebForm应用，相对于直接处理用户输入的应用来讲复杂度大大提高，性能也必然有所降低，为此我们最后来看一个可以大幅度提高程序性能的模式：Page Cache模式。

Page Cache模式下的WebForm

几乎所有的WebForm面临的都是访问很频繁，改动却很少的应用，对WebForm的访问者来说有相当多的内容是重复的，因此我们可以试着把WebForm或者某些相同的内容保存在服务器内存中一段时间以加快程序的响应速度。

这个模式实现起来很简单，只需在页面上加入：<%@ OutputCache Duration="60" VaryByParam="none" %>，这表示该页面会在60秒以后过期，也就是说在这60秒以内所有的来访者看到该页面的内容都是一样的，但是响应速度大大提高，就象静态的HTML页面一样。

也许您只是想保存部分的内容而不是想保存整个页面，那么我们回到MVC模式中的SQLHelper.cs，我对它进行了少许修改：

```
public static DataSet GetPortal()
{
    DataSet dataSet;
    if ( HttpContext.Current.Cache["SELECT_PORTAL_CACHE"] != null )
    {
        //如果数据存在于缓存中则直接取出
        dataSet = ( DataSet ) HttpContext.Current.Cache["SELECT_PORTAL_CACHE"];
    }
    else
    {
        //否则从数据库中取出并插入到缓存中，设定绝对过期时间为3分钟
        dataSet = GetDataSet( SQL_SELECT_PORTAL );
        HttpContext.Current.Cache.Insert( "SELECT_PORTAL_CACHE", dataSet, null,
DateTime.Now.AddMinutes( 3 ), TimeSpan.Zero );
    }
    return dataSet;
}
```

在这里把SELECT_PORTAL_CACHE作为Cache的键，把GetDataSet(SQL_SELECT_PORTAL)取出的内容作为Cache的值。这样除了程序第1次调用时会进行数据库操作外，在Cache过期时间内都不会进行数据库操作，同样大大提高了程序的响应能力。

小结

自从.NET框架引入设计模式以后在很大程度上提高了其在企业级应用方面的实力，可以毫不夸张的说在企业级应用方面.NET已经赶上了Java的步伐并大有后来居上之势，本文通过一个实例的讲解向读者展示了在.NET框架下实现Web设计模式所需的一些基本知识，希望能起到一点抛砖引玉的作用。

参考资源

http://msdn.microsoft.com/architecture/patterns/Esp/default.aspx

下载地址

作者信息

袁剑

Microsoft ASP.NET MVP

http://blog.joycode.com/gOODiDEA

2003年12月6日