ASP.NET Framework

深度历险

作者：uestc95@263.net

时间：2002/05/01

(探讨技术问题可以联系我：Uestc95@263.net MSN：Uestc95@hotmail.com)

写在前面的话

这篇电子文档本来是作为一本打算出版的技术书籍的其中一个章节撰写的，各位在阅读的时候会注意到这一点（带有章节编号），但是由于工作繁忙和其他的一些原因，这本技术书籍未能出版。其实出版与否不重要，重要的是如果能对大家在学习 ASP.NET 的过程中有所帮助就足够了。

最初这份电子文档是在www.dev-club.com论坛上面陆续发表的，但是由于论坛的条件限制，无法将完整的文章展现在大家面前。在长时间的忙碌工作之后，终于有时间将这篇电子文档整理出来配上插图以及 C#代码，方便大家阅读。

如果这篇电子文档能对使得你对 ASP.NET 技术有更深层次的理解，我就会高兴不已了，:-)

如果你想要索取相关的 C#代码，可以发送电子邮件到：uestc95@263.net ，并在邮件主题注明：“ 索取《ASP.NET 深度历险》源代码 ”。

目前从事的工作的.NET Framework 技术架构和组件开发的研究，如果有志同道合的朋友，也希望我们能多多探讨。

第四章 ASP.NET Framework深度历险

本章内容主要是深入的探讨有关 ASP.NET 架构(ASP.NET Framework)方面的技术细节和核心机制。ASP.NET 技术一种强大的技术，其相对于以往的 ASP 技术而言，它已经不是简单的进步可以形容的了，而是一场彻底的 Web 开发技术革命。而我们只有透彻的了解才能真正的驾驭它。

如果不深入的了解一些 ASP.NET Framework 内部的一些机制，我们即便是口口声声说自己使用 ASP.NET 技术来构建 Web 应用程序其实也和以前的 ASP 没有什么两样的，一方面 ASP.NET 技术为了我们的快速开发而封装了所有的技术细节，但是另一方面，也间接的使我们成了简单的 Coding man，只是机械的去用 ASP.NET 提供出来的种种便利的控件或者事件接口。如果想成为真正的 ASP.NET 开发高手，透彻的深入理解 ASP.NET Framework 底层技术细节是不可避免的，也是必须要去做的，否则你只能成为浮于表面的 Coding man。
在这里本章内容不打算简单介绍 ASP.NET 的入门知识，我们也不会浪费时间去做这些。

ASP.NET 除了名字和古老的 ASP 有些相同外，已经是完完全全的改变了，虽然你仍能在 ASP.NET 中发现你熟悉的 Session,Application 等等对象，但是不要尝试将他们同远古的 ASP 时代的 Session 等等画上等号，他们的实现机制和技术细节都已经是完全的不同了。

让我们来慢慢的深入到 ASP.NET Framework 的核心内部，看看它是如何实现的，看看她是如何能承担起下一代 Web 开发技术平台这个美誉的。

4.1、ASP.NET Framework 深度历险 – 一个 HTTP 请求的生命周期

在基于 WEB 的应用程序开发当中，其核心就是在客户端机器以及服务器端机器之间通过 HTTP 协议相互传递需要的信息，因此，了解一个 WEB 服务器是如何处理 HTTP 请求的就显得格外的重要了。深入的理解一个 HTTP 请求的生命周期对于今后的 WEB 应用程序开发也是格外重要的。在本节，我们就一同来了解一下一个 HTTP 请求的生命周期吧。

4.1.1 回顾 ASP 时代的 HTTP 请求处理过程

在这一小节内，我们来跟随考古学家共同了解一下古老的 ASP 运行机制：

在以前的 ASP 时代，当你请求一个*.asp 文件的时候，这个 HTTP 请求首先会被一个名为 inetinfo.exe 进程所截获，这个 inetinfo.exe 进程实际上就是 WWW 服务进程，在截获这个 HTTP 请求之后它会将这个请求转交给 asp.dll 进程，asp.dll 进程就会解释执行这个 asp 叶面，然后将解释后的数据流返回给客户端浏览器。

我们可以通过下面的图来清楚的了解古老的 ASP 时代，服务器进程是如何处理来自客户端的 HTTP 请求的：

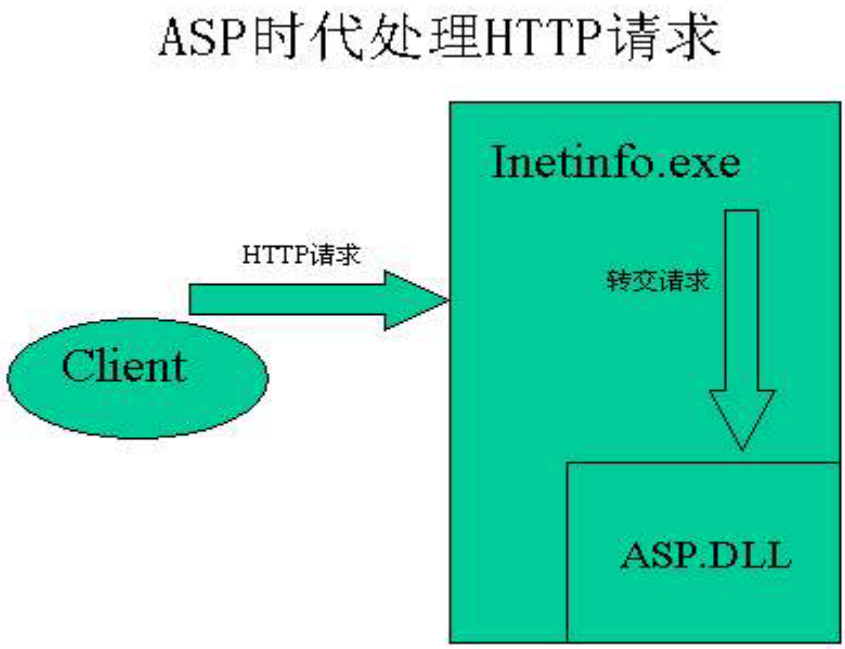

图 4.1.1 ASP 时代处理 HTTP 请求

其实 ASP.DLL 是一个依附在 IIS 的 ISAPI 文件，它负责了对诸如 ASP 文件 ASA 文件等等的解释执行，我们可以在 IIS 配置选项看到它是如何配置的：

图 4.1.2　ISAPI 配置

通过图 4.1.2，我们可以看到红色标记的部分，它清楚的告诉我们 IIS 会将对.asp 以及.asa 等文件的 HTTP 请求转交给 asp.dll 这个 ISAPI 文件来处理。

在古老的 ASP 时代，针对这些 asp 等文件的 HTTP 请求被 asp.dll 所截获，从而被解释执行。真是由于解释执行，每一次处理这些文件都是从头来过进行一番痛苦的解释才能执行使得 IIS 在处理大流量的 HTTP 请求的时候并不是得心应手，这也是 ASP 时代最大的弊端。

4.1.2 ASP.NET 时代的 HTTP 请求处理方式

看完了远古时代 ASP 的 HTTP 请求处理方法，转过头来我们看看如今的 ASP.NET Framework 是如何处理一个 HTTP 请求的。

当客户端向 WEB 服务器请求一个*.aspx 文件的时候，同 ASP 类似，这个 HTTP 请求也会被 inetinfo.exe 进程（因为它就是 WWW 服务）截获，它判断文件的后缀之后，将这个请求转交给 ASPNET_ISAPI.dll，而 ASPNET_ISAPI.dll 则会通过一个被称为 Http PipeLine 的管道，将这个 HTTP 请求发送给 ASPNET_WP.exe 进程，当这个 HTTP 请求进入 ASPNET_WP.exe 进程之后，ASP.NET Framework 就会通过 HttpRuntime 来处理这个 HTTP 请求，处理完毕将结果返回客户端。

接下来我们同样来看看一个描述 ASP.NET Framework 是如何处理 HTTP 请求的图示：

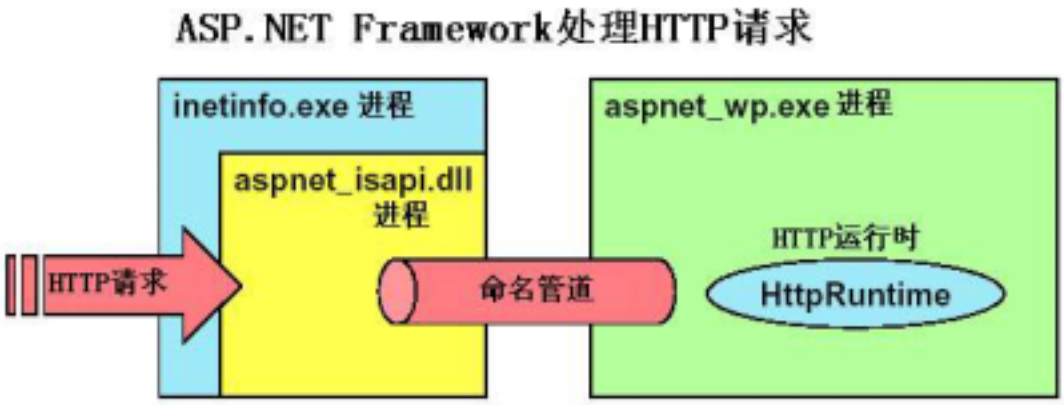

图 4.1.3 ASP.NET Framework 处理 HTTP 请求

通过上面的图我们可以清楚的了解到 ASP.NET Framework 是如何接收、处理一个 HTTP 请求的完整过程。

不过,也许有些读者在看完上面的图示之后就会有这样的疑问了：ASP.NET Framework 处理一个 HTTP 请求的流程和以前的 ASP 时代好像并没有太大的改进啊。不要着急，在 ASP.NET Framework 中我们甚至能够了解到 HttpRuntime 的细节。接下来我们继续在 ASP.NET Framework 的世界深入历险下去。
继续深入的第一步当然是要到 HttpRuntime 的世界瞧一瞧了。

4.1.3 深入 ASP.NET Framework HTTP 运行时

在本节我们会一同看看 HTTP 运行时内部的一些东西。

当一个 HTTP 请求被送入 HttpRuntime 之后，这个 HTTP 请求会继续被送入到一个被称之为 HttpApplication Factory 的一个容器当中，而这个容器会给出一个 HttpApplication 实例来处理传递进来的 HTTP 请求，而后这个 HTTP 请求会依次进入如下几个容器中：

HttpModule → HttpHandler Factory → HttpHandler

当系统内部的 HttpHandler 的 ProcessResquest 方法处理完毕之后，整个 Http Request 就被处理完成了，客户端也就得到相应的东东了。

ASP.NET Framework 处理 HTTP 请求的流程示意图如下：

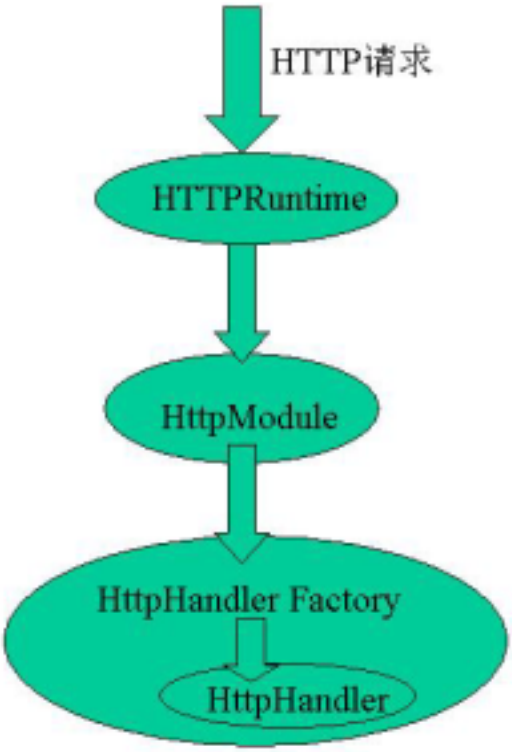

图 4.1.4

现在我们可以整理一下完整的一个 HTTP 请求在 ASP.NET Framework 下是如何被处理的：

HttpRequest → inetinfo.exe → ASPNET_ISAPI.dll → Http Pipeline → ASPNET_WP.exe → HttpRuntime → HttpApplication Factory → HttpApplication → HttpModule → HttpHandler Factory → HttpHandler → HttpHandler.ProcessRequest()

读者或许会问，我知道这个HTTP请求处理流程有什么用处呢？当然有用了，比如如果你想要中途截获一个 Http Request 并且做些自己的处理，该如何做呢？

通过仔细的对上面图示的观察，读者应当知道能够在什么地方截获到这个 HTTP 请求。对，就是在 HTTPRuntime 运行时内部来做到这一点的，确切的说，是在 HttpModule 这个容器中做到这一点的。既然已经进入了神奇的 HttpRuntime 内部世界，我们当然要多看几眼了，接下来我们会继续我们的历险步伐，向 HttpModule 容器进军！

4.2、ASP.NET Framework 深度历险 – HttpModule 是如何工作的？

我们已经知道 HttpModule 容器是一个 HTTP 请求的必经之路，而我们在此小节的任务就是在 HttpModule 的内部深入看个究竟。

4.2.1 HttpModule 在 ASP.NET Framework 中的位置

我们上回说到，一个来自于客户端的 HTTP 请求被截获后经过层层转交（怎么都在踢皮球？呵呵）到达了 HttpModule 这个“请求监听器”。HttpModule 就类似于安插在 ASPNET_WP.EXE 进程中的一个窃听器，稍微有些常识的人都会很自然的想象得到窃听器是用来做什么的，而我们的 HttpModule 可以说是作窃听器的绝好人选了，但是需要明确的是，HttpModule 绝对不是简单的监听器，它可以做到更多的东西，比如它可以对截获的请求增加一些内容等等。

那么 HttpModule 在整个 ASP.NET Framework 中的位置处在哪里呢？下面我们通过图示来看看：

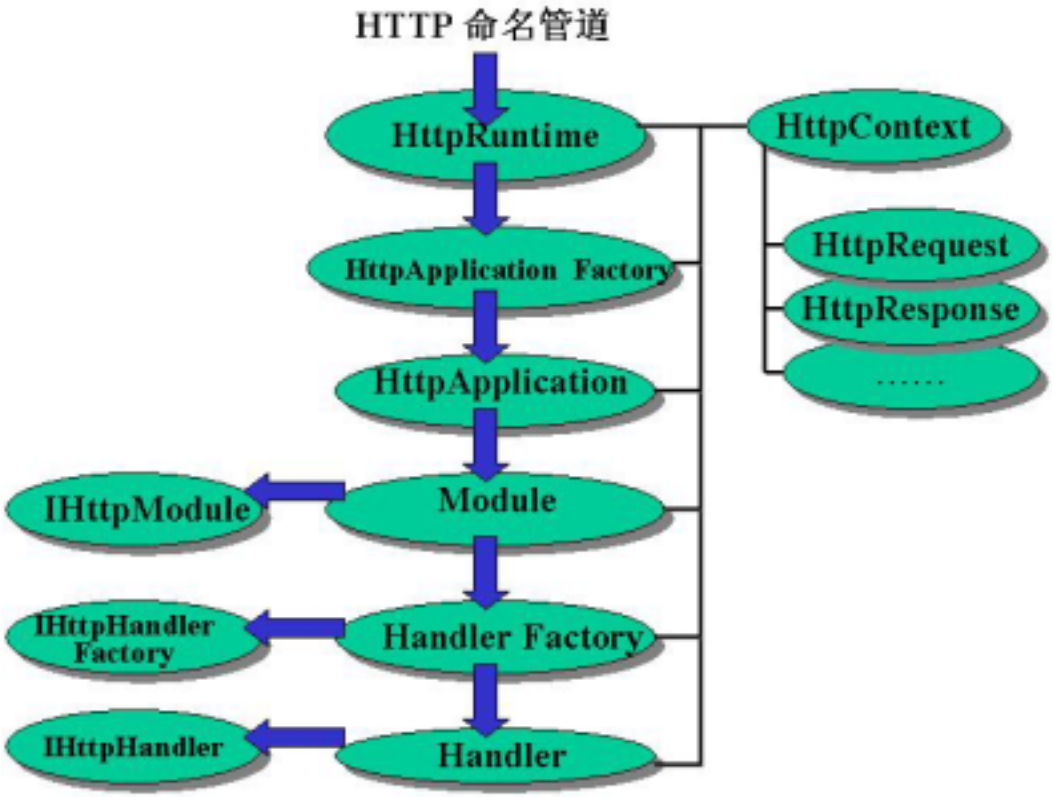

图 4.2.1 HttpModule 的位置

另外需要明白的是,当一个 HTTP 请求到达 HttpModule 的时候,整个 ASP.NET Framework 系统还并没有对这个 HTTP 请求做任何的真正处理，也就是说此时对于 HTTP 请求来讲，HttpModule 只是它路过的一个地方而以。但是正是因为 HttpModule 是一个 HTTP 请求的“必经之路”，所以我们可以在这个 HTTP 请求传递到真正的请求处理中心（HttpHandler）之前附加一些我们需要的信息在这个 HTTP 请求信息之上，或者针对我们截获的这个 HTTP 请求信息作一些额外的工作，或者在某些情况下干脆终止满足一些条件的 HTTP 请求，从而可以起到一个 Filter 过滤器的作用，而不仅仅是一个窃听器了。
通过查阅 MSDN（不要去相信.NET SDK 自带的那个 QuickStarts Web 文档，正式版本中竟然在很多地方没有更新这个文档，很多东西在正式版本是无效的），你会发现系统 HttpModule 实现了一个叫做 IHttpModule 的接口，很自然的就应当想到，只要我们自己的类能够实现 IHttpModule 接口，不就可以完全替代系统的 HttpModule 了吗？完全正确。

在我们开始自己的 HttpModule 类之前，我先来告诉你系统中的那个 HttpModule 是什么样子的，ASP.NET 系统中默认的 HttpModule 有以下几个：

```
System.Web.Caching.OutputCacheModule
System.Web.SessionState.SessionStateModule
System.Web.Security.WindowsAuthenticationModule
System.Web.Security.FormsAuthenticationModule
System.Web.Security.PassportAuthenticationModule
System.Web.Security.UrlAuthorizationModule
System.Web.Security.FileAuthorizationModule
```

这些系统默认的 HttpModule 是在文件 machine.config 中配置，这个文件位于你安装的.NET 框架所在的目录中，比如在你的系统文件目录中的 C:\WINNT\Microsoft.NET\Framework\v1.0.3705\CONFIG\machine.config。

在我们开发 ASP.NET 应用程序的时候会频繁的使用到一个 web.config 配置文件，那么这个 machine.config 和我们常见的 web.config 有什么关系呢？原来在 ASP.NET Framework 启动处理一个 Http Request 的时候，她会依次加载 machine.config 以及你请求页面所在目录的 web.config 文件,里面的配置是有<remove>标签的，什么意思不说也知道了吧。如果你在 machine.config 中配置了一个自己的 HttpModule，你仍然可以在离你最近 web.config 文件中“remove”掉这个映射关系。

4.2.2 构建我们自己的 HttpModule

在上一小节中，我们谈到了系统默认的各个 HttpModule 均继承实现了一个叫做 IhttpModule 的接口。我们先来看看这个接口的真实面目吧：

语法：public interface IHttpModule
需求：
名称空间: System.Web
平台: Windows 2000, Windows XP Professional,
Windows .NET Server family
装配件: System.Web (in System.Web.dll)
公共成员方法：
void Dispose();
参数：无
返回值：void
作用：销毁不再被 module 所使用的资源。
void Init(HttpApplication context);
参数：HttpApplication 类型的实例
返回值：void
作用：初始化一个 module，为捕获 HTTP 请求做出一些准备。

了解了接口 IhttpModule 的方法，我们也就知道了如何去实现它了。

接下来，我们来开始我们自己的 HttpModule 构建历程吧。

1) 打开 VS.NET 新建一个“ Class Library ”项目，将它命名为 MyHttpModule。
2) 引用 System.Web.dll 文件

在代码区域敲入：

```
using System;
using System.Web;

namespace MyHttpModuleTest
{
    /// <summary>
    /// 说明：用来实现自定义 HttpModule 的类
    /// 作者：uestc95
    /// 联系：uestc95@263.net
    /// </summary>
    public class MyHttpModule:IHttpModule
```

```
    {
        /// <summary>
        /// 说明：构造器方法
        /// 作者：uestc95
        /// 联系：uestc95@263.net
        /// </summary>
        public MyHttpModule()
        {

        }

        /// <summary>
        /// 说明：实现 IHttpModule 接口的 Init 方法
        /// 作者：uestc95
        /// 联系：uestc95@263.net
        /// </summary>
        /// <param name="application">HttpApplication 类型的参数
</param>
        public void Init(HttpApplication application)
        {
            application.BeginRequest +=new
EventHandler(this.Application_BeginRequest);
            application.EndRequest +=new
EventHandler(this.Application_EndRequest);
        }

        /// <summary>
        /// 说明：自己定义的用来做点事情的私有方法
        /// 作者：uestc95
        /// 联系：uestc95@263.net
        /// </summary>
        /// <param name="obj">传递进来的对象参数</param>
        /// <param name="e">事件参数</param>
        private void Application_BeginRequest(Object obj,EventArgs e)
        {
   //声明 HttpApplication
       HttpApplication application=(HttpApplication)obj;
            HttpContext context=application.Context;
            HttpResponse response=context.Response;
```

```
            HttpRequest request=context.Request;

            response.Write("我来自 Application_BeginRequest,:)");

        }

        /// <summary>
        /// 说明：自己定义的用来做点事情的私有方法
        /// 作者：uestc95
        /// 联系：uestc95@263.net
        /// </summary>
        /// <param name="obj">传递进来的对象参数</param>
        /// <param name="e">事件参数</param>
        private void Application_EndRequest(Object obj,EventArgs e)
        {
            HttpApplication application=(HttpApplication)obj;
            HttpContext context=application.Context;
            HttpResponse response=context.Response;
            HttpRequest request=context.Request;

            response.Write("我来自 Application_EndRequest,:)") ;

        }

        /// <summary>
        /// 说明：实现 IHttpModule 接口的 Dispose 方法
        /// 作者：uestc95
        /// 联系：uestc95@263.net
        /// </summary>
        public void Dispose(){}
    }
}
```

3）在 VS.NET 中编译之后，你会得到 MyHttpModule.dll 这个文件。

4）接下来我们的工作就是如何让 ASPNET_WP.exe 进程将 http request 交给我们自己写的这个 HttpModule 呢？方法就是配置 web.config 文件。

在 web.config 文件中增加如下几句话：

```
    <httpModules>
        <add name="test"
```

```
type="MyHttpModuleTest.MyHttpModule,MyHttpModule"/>
        </httpModules>
```

注意要区分大小写，因为 web.config 作为一个 XML 文件是大小写敏感的。“ type=MyHttpModuleTest.MyHttpModule,MyHttpModule ”告诉我们，系统将会将 HTTP 请求交给位于 MyHttpModule.dll 文件中的 MyHttpModuleTest.MyHttpModule 类去处理。而这个 DLL 文件系统将会自动到 \bin 子目录或者系统全局程序集缓冲区（GAC）搜寻。我们可以将我们刚才得到的 DLL 文件放在 bin 子目录中，至于后者，你可以通过.NET SDK 正式版自带的 Config 工具做到，我们不详细说了。

好了，我们的用来截获 HTTP 请求的自定义 HttpModule 就完成并且装配完成了，你可以试着在你的 web 项目中建立一个新的 WebForm，运行看看呢？：）

最后，我们假设一个使用这个 HttpModule 的场合。A 站点提供免费的 ASP.NET 虚拟空间给大家，但是 A 站点的管理者并不想提供免费的午餐，他想要在每一个页面被浏览的时候自动弹出自己公司的广告，我总不能时刻监视所有用户的所有页面吧，并且想要在每一个页面手动添加一段 JS 代码，工作量之大是不可想象的，也是非常不现实的。那么好了，只要我们的 HttpModule 一旦被挂接完成，这一切都将是轻而易举的事情了，只要我们在每一个 HTTP 请求被我们捕获的时候，给他在 HTTP 请求信息上面附加上一些我们自己的写的 JavaScript 代码就好了！

我们上面提到在 Init()方法中使用了两个事件 BeginRequest 和 EndRequest，这两个事件分别是 Init()中可以处理的所有事件的最开始事件和最终事件，在他们中间还有一些其它的事件可以被我们利用，可以查阅 MSDN，这一点我们会在下面详细的讲解。

在下一节开始之前，我们请各位读者考虑这样一个问题：在 HttpModule 中可以正常使用 Response,Request,Server,Application 对象吗？请读者自行试着写一些代码看看行不行。还有一个问题就是，在 HttpModule 中能操作 Session 对象吗？如果可以或者不可以，请仔细思考一下原因何在，在下一节我们会详细的探讨这个问题。

4.2.3 深入 HttpModule 历险

在这一小节中，我们会详细的探讨有关 HttpModule 的运行机制，从而使得各位读者能够透彻的了解 HttpModule 是如何控制 HTTP 请求的。

我们在上一节曾经提及当一个 HTTP 请求被 ASP.NET Framework 捕获之后会依次交给 HttpModule 以及 HttpHandler 来处理，但是需要明确的是，不能理解为 HttpModule 和 HttpHandler 是完全独立的。实际上是，在 HTTP 请求在 HttpModule 传递的过程中会在某个事件内将控制权交给 HttpHandler 的，而真正的处理在 HttpHandler 中执行完成之后，HttpHandler 会再次将控制权交还给

HttpModule。也就是说 HttpModule 在某个请求经过她的时候会在恰当时候同 HttpHandler 进行通信，在何时，如何通信呢？这就是下面提到的了。

我们在上一小节提到在 HttpModule 容器中最开始的事件是 BeginRequest，最终的事件是 EndRequest。你如果仔细看上次给出的源程序的话，应当发现在方法 Init()中参数我们传递的是一个 HttpApplication 类型，而我们曾经提及的两个事件正是这个传递进来的 HttpApplication 的事件之一。

HttpApplication 还有其它众多的事件，分别如下：

application.BeginRequest 事件
application.EndRequest 事件
application.PreRequestHandlerExecute 事件
application.PostRequestHandlerExecute 事件
application.ReleaseRequestState 事件
application.AcquireRequestState 事件
application.AuthenticateRequest 事件
application.AuthorizeRequest 事件
application.ResolveRequestCache 事件
application.UpdateRequestCache 事件
application.PreSendRequestHeaders 事件
application.PreSendRequestContent 事件

接下来我们来看看这些事件的含义解释：

HttpApplication 事件名称	什么时间触发
BeginRequest 事件	处理 HTTP 请求开始之前触发
AuthenticateRequest 事件	验证客户端时候触发
AuthenticateRequest 事件	执行存取的时候触发
ResolveRequestCache 事件	从缓存中得到相应时候触发
AcquireRequestState 事件	加载初始化 Session 时候触发
PreRequestHandlerExecute 事件	在 HTTP 请求送入 HttpHandler 之前触发
PostRequestHandlerExecute 事件	在 HTTP 请求送入 HttpHandler 之后触发
ReleaseRequestState 事件	存储 Session 状态时候触发
UpdateRequestCache 事件	更新缓存信息的时候触发
EndRequest 事件	在 HTTP 请求处理完成时候触发
PreSendRequestHeaders 事件	在向客户端发送 Header 之前触发
PreSendRequestContent 事件	在向客户端发送内容之前触发

需要注意的是，在事件 EndRequest 之后还会继续执行 application.PreSendRequestHeaders 事件以及 application.PreSendRequestContent 事件,而这两个事件大家想必应当从名称上面看得出来事做什么用途的了吧。是的，一旦触发了这两个事件，就表明整个对一个 HTTP 请求的处理已经执行完成了，在这两个事件中是开始向客户端传送处理完成的数据流了。细心的读者看到这里，会有一个疑问：怎么没见到进入 HttpHandler 容器就处理完成了？不是提到过 HttpHandler 容器才是真正处理 HTTP 请求的吗？

其实一开始我们就提到了，在一个 HTTP 请求在 HttpModule 容器的传递过程中，会在某一个时刻（确切的说应当是事件）中将这个 HTTP 请求传递给 HttpHandler 容器的。这个事件就是 ResolveRequestCache，在这个事件之后，HttpModule 容器会建立一个 HttpHandler 的入口实例（做好准备了，:）），但是此时并没有将 HTTP 请求的控制权交出，而是继续触发 AcquireRequestState 事件以及 PreRequestHandlerExecute 事件（如果你实现了的话）。看到了吗，最后一个事件的前缀是 Pre，这表明下一步就要进入 HttpHandler 容器了，事实上的确如此，正如我们猜想的那样，在 PreRequestHandlerExecute 事件之后，HttpModule 容器就会将控制权暂时交给 HttpHandler 容器，以便进行真正的 HTTP 请求处理工作。

而在 HttpHandler 容器内部会执行 ProcessRequest 方法来处理 HTTP 请求。在容器 HttpHandler 处理完毕整个 HTTP 请求之后，会将控制权交还给 HttpModule,HttpModule 则会继续对处理完毕的 HTTP 请求信息流（此时的 HTTP 请求信息流已经是被处理过后了的信息流了）进行层层的转交动作，直到返回到客户端为止。

怎么样,是不是有些混乱？下面是一个完整的HttpModule的生命周期示意步骤：

↓

HttpModule.AcquireRequestState()

↓

HttpModule.PreRequestHandlerExecute()

↓

进入 HttpHandler 处理 HttpRequest

↓

HttpHandler.ProcessRequest()

↓

返回到 HttpModule 接着处理（HttpHandler 生命周期结束，Session 失效）

↓

HttpModule.PostRequestHandlerExecute()

↓

HttpModule.ReleaseRequestState()

↓

HttpModule.UpdateRequestCache()

↓

HttpModule.EndRequest()

↓

HttpModule.PreSendRequestHeaders()

↓

HttpModule.PreSendRequestContent()

↓

将处理后的数据返回客户端

↓

整个 Http Request 处理结束

为了更加形象的描述整个 HttpModule 的 HTTP 请求处理周期，接下来我们来看看 HTTP 请求在整个 HttpModule 中的生命周期图：

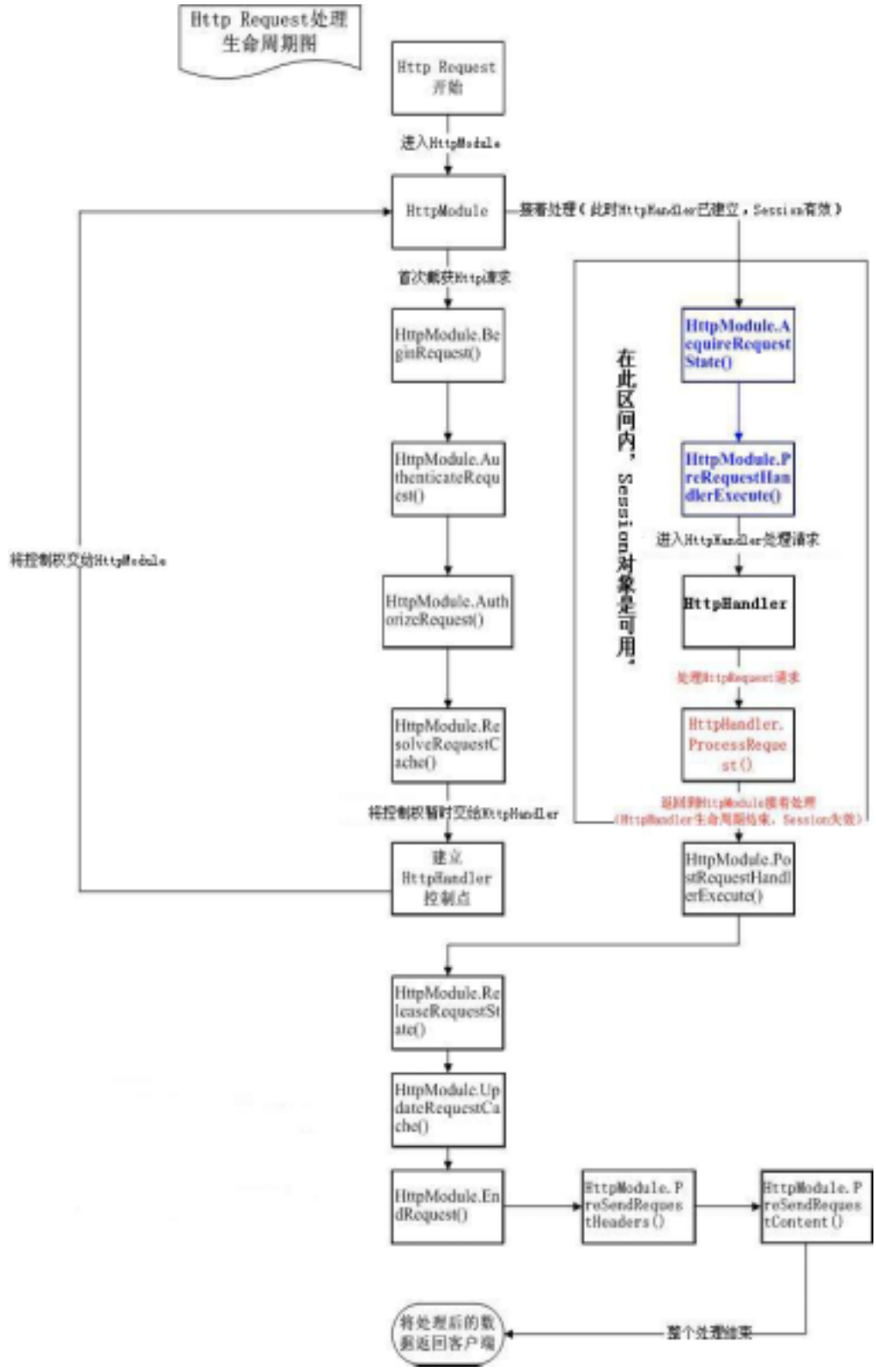
Http Request处理
生命周期图
Http Request
开始
进入HttpModule
HttpModule
接着处理（此时HttpHandler已建立，Session有效）
首次截获Http请求
HttpModule.BeginRequest()
HttpModule.AuthenticateRequest()
HttpModule.AuthorizeRequest()
HttpModule.ResolveRequestCache()
将控制权交给HttpHandler
建立HttpHandler控制点
将控制权交给HttpModule
HttpModule.AcquireRequestState()
HttpModule.PreRequestHandlerExecute()
进入HttpHandler处理请求
HttpHandler
处理HttpRequest请求
HttpHandler.ProcessRequest()
返回到HttpModule接着处理
（HttpHandler生命周期结束，Session失效）
HttpModule.PostRequestHandlerExecute()
在此区间内，Session对象是可用
HttpModule.ReleaseRequestState()
HttpModule.UpdateRequestCache()
HttpModule.EndRequest()
HttpModule.PreSendRequestHeaders()
HttpModule.PreSendRequestContent()
整个处理结束
将处理后的数据返回客户端

图 4.2.2 HttpModule 生命周期图

想必读者应当可以从上面的 HttpModule 生命周期图中找到了上一小节我们提出的那个问题：Session 对象在何种情况下可以正常的使用！为了验证上面的流程，我们可以用下面的这个自己的 HttpModuel 来验证一下就知道了。

注意我们下面给出的是类的内容，代码框架还是上一小节我们给出的那个，自己加上就好了：

```
        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new
EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new
EventHandler(this.Application_EndRequest));
            application.PreRequestHandlerExecute +=(new
EventHandler(this.Application_PreRequestHandlerExecute));
            application.PostRequestHandlerExecute  +=(new
EventHandler(this.Application_PostRequestHandlerExecute));
            application.ReleaseRequestState  +=(new
EventHandler(this.Application_ReleaseRequestState));
            application.AcquireRequestState +=(new
EventHandler(this.Application_AcquireRequestState));
            application.AuthenticateRequest +=(new
EventHandler(this.Application_AuthenticateRequest));
            application.AuthorizeRequest +=(new
EventHandler(this.Application_AuthorizeRequest));
            application.ResolveRequestCache +=(new
EventHandler(this.Application_ResolveRequestCache));
            application.PreSendRequestHeaders +=(new
EventHandler(this.Application_PreSendRequestHeaders));
            application.PreSendRequestContent +=(new
EventHandler(this.Application_PreSendRequestContent));
        }

        private void Application_PreRequestHandlerExecute(Object
source, EventArgs e)
```

```
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_PreRequestHandlerExe
cute<br>");
        }

        private void Application_BeginRequest(Object source, EventArgs
e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_BeginRequest<br>");
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_EndRequest<br>");

        }

        private void Application_PostRequestHandlerExecute(Object
source,EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_PostRequestHandlerEx
ecute<br>");

        }

        private void Application_ReleaseRequestState(Object source,
EventArgs e)
```

```
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_ReleaseRequestState<
br>");

        }

        private void Application_UpdateRequestCache(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_UpdateRequestCache<b
r>");

        }

        private void Application_AuthenticateRequest(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_AuthenticateRequest<
br>");

        }

        private void Application_AuthorizeRequest(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_AuthorizeRequest<br>
");
```

```
        }

        private void Application_ResolveRequestCache(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_ResolveRequestCache<
br>");

        }

        private void Application_AcquireRequestState(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_AcquireRequestState<
br>");

        }

        private void Application_PreSendRequestHeaders(Object source,
EventArgs e)
        {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_PreSendRequestHeader
s<br>");

        }

        private void Application_PreSendRequestContent(Object source,
EventArgs e)
        {
```

```
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("Application_PreSendRequestConten
t<br>");

        }
        public void Dispose()
        {
        }
```

将上面的代码编译成为一个类库文件，再按照上一小节给出的方法将它挂接到 ASP.NET Framework 系统,你运行此演示代码就可以看到和我们给出的 HttpModule 生命周期图是完全吻合的。

现在我们来思考这样一个问题：如果我同时编写了多个自定义的 HttpModule，并且也都装配了他们，ASP.NET Framework 会怎样加载这多个自定义的 HttpModule 呢？下面我们就一同通过一个完整的例子来演示看看：

我们在下面的代码中会建立两个 HttpModule 类库 DLL 文件，并且同时装配他们在 web.config 中。

1）、在 VS.NET 中建立一个名为 IhttpModule 的类库项目

2）、引入名称空间文件 System.Web.dll

3）、在代码区域输入如下的代码：

```
using System;
using System.Web;
using System.Collections;

public class HelloWorldModule : IHttpModule
{
    public String ModuleName
    {
              get { return "HelloWorldModule"; }
    }

    public void Init(HttpApplication application)
    {
              application.BeginRequest += (new
EventHandler(this.Application_BeginRequest));
```

```
            application.EndRequest += (new
EventHandler(this.Application_EndRequest));
            application.PreRequestHandlerExecute +=(new
EventHandler(this.Application_PreRequestHandlerExecute));
            application.PostRequestHandlerExecute  +=(new
EventHandler(this.Application_PostRequestHandlerExecute));
            application.ReleaseRequestState  +=(new
EventHandler(this.Application_ReleaseRequestState));
            application.AcquireRequestState +=(new
EventHandler(this.Application_AcquireRequestState));
            application.AuthenticateRequest +=(new
EventHandler(this.Application_AuthenticateRequest));
            application.AuthorizeRequest +=(new
EventHandler(this.Application_AuthorizeRequest));
            application.ResolveRequestCache +=(new
EventHandler(this.Application_ResolveRequestCache));
            application.PreSendRequestHeaders +=(new
EventHandler(this.Application_PreSendRequestHeaders));
            application.PreSendRequestContent +=(new
EventHandler(this.Application_PreSendRequestContent));

    }

    //Your BeginRequest event handler.
    private void Application_BeginRequest(Object source, EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("HelloWorldModule: Beginning of
Request<br>");
    }

    //Your EndRequest event handler.
    private void Application_EndRequest(Object source, EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;
```

```
            context.Response.Write("HelloWorldModule: End of
Request<br>");

    }

    private void Application_PostRequestHandlerExecute(Object
source,EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;


    context.Response.Write("HelloWorldModule:Application_PostRequest
HandlerExecute:<br>");
    }

    private void Application_ReleaseRequestState(Object source,
EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule:Application_ReleaseRequ
estState :<br>");
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="source"></param>
    /// <param name="e"></param>
    private void Application_PreRequestHandlerExecute(Object source,
EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;
            HttpResponse Response=context.Response;
            HttpRequest Request=context.Request;
```

```
    context.Response.Write("HelloWorldModule:Application_PreRequestH
andlerExecute :<br>");


    }
    private void Application_UpdateRequestCache(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule:Application_UpdateReque
stCach :<br>");
    }

    private void Application_AuthenticateRequest(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

context.Response.Write("HelloWorldModule:Application_AuthenticateReq
uest<br>");
    }

    private void Application_AuthorizeRequest(Object source, EventArgs
e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule:Application_AuthorizeRe
quest<br>");
    }

    private void Application_ResolveRequestCache(Object source,
EventArgs e)
```

```
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule:Application_ResolveRequ
estCache<br>");
    }

    private void Application_AcquireRequestState(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule:Application_AcquireRequ
estState<br>");
    }

    private void Application_PreSendRequestHeaders(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;




    context.Response.Write("HelloWorldModule:Application_PreSendRequ
estHeaders<br>");



    }

    private void Application_PreSendRequestContent(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;
```

```
    context.Response.Write("HelloWorldModule:Application_PreSendRequ
estContent<br>");
    }
    public void Dispose()
    {
    }
}
```

3）、编译这个项目之后会得到一个叫做 IhttpModule.dll 的文件

接下来我们再来共同建立另外一个很类似的自定义 HttpModule 类库项目
1）、在 VS.NET 中新建一个名为 IhttpModule2 的项目
2）、引入 System.Web.dll 文件
3）、在代码区域输入：

```
using System;
using System.Web;
using System.Collections;

public class HelloWorldModule2 : IHttpModule
{
    public String ModuleName
    {
              get { return "HelloWorldModule"; }
    }

    public void Init(HttpApplication application)
    {
              application.BeginRequest += (new
EventHandler(this.Application_BeginRequest));
              application.EndRequest += (new
EventHandler(this.Application_EndRequest));
              application.PreRequestHandlerExecute +=(new
EventHandler(this.Application_PreRequestHandlerExecute));
              application.PostRequestHandlerExecute  +=(new
EventHandler(this.Application_PostRequestHandlerExecute));
              application.ReleaseRequestState  +=(new
EventHandler(this.Application_ReleaseRequestState));
```

```
            application.AcquireRequestState +=(new
EventHandler(this.Application_AcquireRequestState));
            application.AuthenticateRequest +=(new
EventHandler(this.Application_AuthenticateRequest));
            application.AuthorizeRequest +=(new
EventHandler(this.Application_AuthorizeRequest));
            application.ResolveRequestCache +=(new
EventHandler(this.Application_ResolveRequestCache));
            application.PreSendRequestHeaders +=(new
EventHandler(this.Application_PreSendRequestHeaders));
            application.PreSendRequestContent +=(new
EventHandler(this.Application_PreSendRequestContent));

    }

    //Your BeginRequest event handler.
    private void Application_BeginRequest(Object source, EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;

            context.Response.Write("HelloWorldModule2: Beginning of
Reques<br>");
    }

    //Your EndRequest event handler.
    private void Application_EndRequest(Object source, EventArgs e)
    {
            HttpApplication application = (HttpApplication)source;
            HttpContext context = application.Context;


            context.Response.Write("HelloWorldModule2: End of
Request<br>");

    }

    private void Application_PostRequestHandlerExecute(Object
source,EventArgs e)
```

```
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;


    context.Response.Write("HelloWorldModule2:Application_PostReques
tHandlerExecute:<br>");
    }

    private void Application_ReleaseRequestState(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_ReleaseReq
uestState :<br>");
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="source"></param>
    /// <param name="e"></param>
    private void Application_PreRequestHandlerExecute(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;
              HttpResponse Response=context.Response;
              HttpRequest Request=context.Request;


    context.Response.Write("HelloWorldModule2:Application_PreRequest
HandlerExecute :<br>");


    }
```

```
    private void Application_UpdateRequestCache(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_UpdateRequ
estCach :<br>");
    }

    private void Application_AuthenticateRequest(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_Authentica
teRequest<br>");
    }

    private void Application_AuthorizeRequest(Object source, EventArgs
e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_AuthorizeR
equest<br>");
    }

    private void Application_ResolveRequestCache(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_ResolveReq
uestCache<br>");
    }
```

```
    private void Application_AcquireRequestState(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_AcquireReq
uestState<br>");
    }

    private void Application_PreSendRequestHeaders(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;




    context.Response.Write("HelloWorldModule2:Application_PreSendReq
uestHeaders<br>");



    }

    private void Application_PreSendRequestContent(Object source,
EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
              HttpContext context = application.Context;

    context.Response.Write("HelloWorldModule2:Application_PreSendReq
uestContent<br>");
    }
    public void Dispose()
    {
    }
}
```

编译这个自定义的 HttpModule 类库项目之后，会得到一个名为 IhttpModule2.dll 的文件。

下面我们需要来建立一个 Web 应用程序项目来测试这两个自定义的 HttpModule 文件的效果如何。

1）、在 VS.NET 中建立一个 Web 应用程序项目名字叫做：4.2.3

2）、在文件 web.config 中增加如下语句：

```
<httpModules>
    <add name="test" type="HelloWorldModule,IHttpModule"/>
    <add name="test2" type="HelloWorldModule2,IHttpModule2"/>
</httpModules>
```

以便通知 ASP.NET Framework。

3）、将上面我们生成的两个自定义 HttpModule 文件拷贝到此 Web 应用程序的 bin/子目录中。

4）、直接运行这个 Web 应用程序，我们会得到如下的结果：

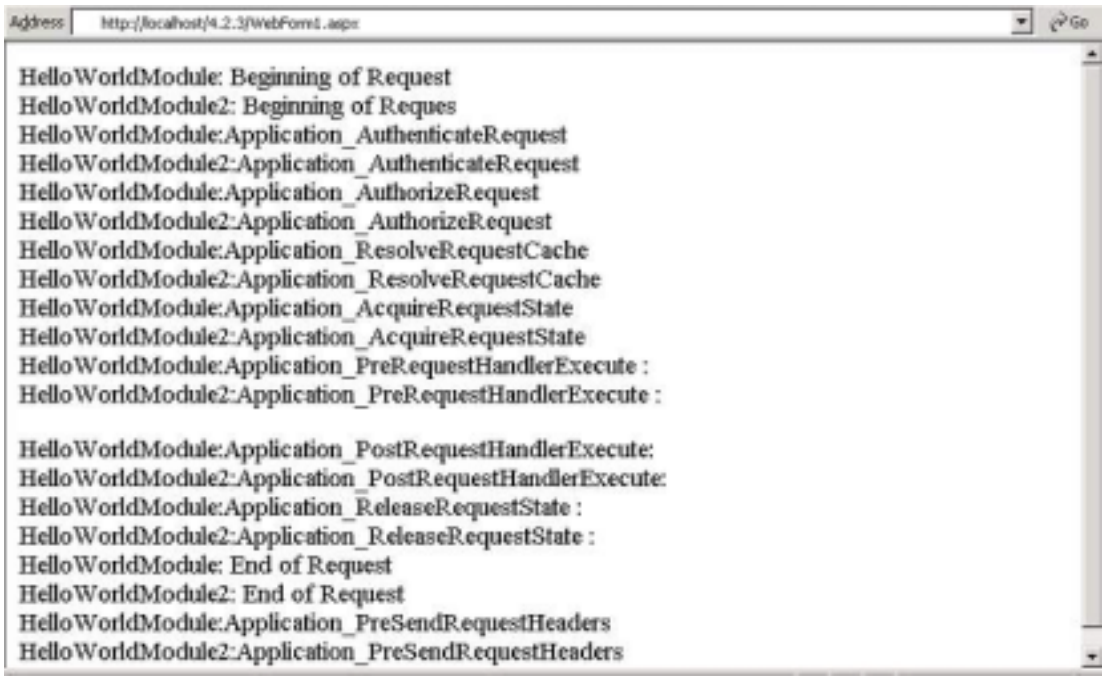

图 4.2.3 多个自定义的 HttpModule

从上面的运行结果我们可以看到，我们在 web.config 文件中引入自定义 HttpModule 的顺序就决定了多个自定义 HttpModule 在处理一个 HTTP 请求的接管顺序。而系统默认那几个 HttpModule 是最先被 ASP.NET Framework 所加载上去的。

最后，我们来这样一个问题：在前面小节中我们曾经提到过我们可以利用 HttpModule 来实现当满足某一条件的时候终止此次的 HTTP 请求,那么如何做到在我们的 HttpModule 中终止一个 HTTP 请求呢？方法就是调用 HttpApplication.CompleteRequest()方法。并且我们知道在一个 HttpModule 中首先触发的事件将是 BeginRequest，这里也就理所当然的成为了我们终止一个 HTTP 请求的地方了。

我们可以更改一下 BeginRequest 事件处理代码如下：

```
    private void Application_BeginRequest(Object source, EventArgs e)
    {
              HttpApplication application = (HttpApplication)source;
        HttpContext context = application.Context;
              application.CompleteRequest();
              context.Response.StatusCode=500;
              context.Response.StatusDescription=”Internal Server
Error!”;
    //context.Response.Write("HelloWorldModule1: Beginning of
Request<br>");
    }
```

这样，就会终止一个 HTTP 请求了。但是需要注意的是，即使调用了 HttpApplication.CompleteRequest()方法终止了一个 HTTP 请求，ASP.NET Framework 仍然会触发 HttpApplication 后面的这三个事件：EndRequest 事件、PreSendRequestHeaders 事件、PreSendRequestContent 事件。

如果多个自定义的 HttpModule 存在的话，比如上面我们给出的例子：IhttpModule.dll 以及 IhttpModule2.dll。如果我们在 IhttpModule.dll 中终止了一个 HTTP 请求的话，这个 HTTP 请求也不会再触发 IhttpModule2.dll 中的诸如 BeginRequest 等等事件了，但是仍然会触发 IhttpModule.dll 以及 IhttpModule2.dll 的 EndRequest 事件、PreSendRequestHeaders 事件、PreSendRequestContent 事件。

我们可以看看下面的示意图：

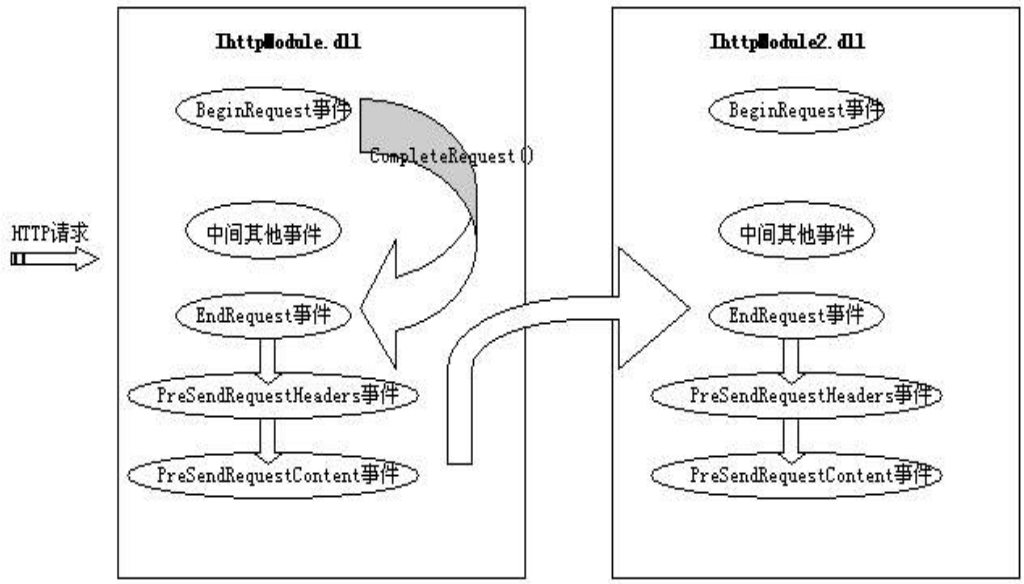

图 4.2.4

现在我们已经在 HttpModule 的神秘世界历险了一番，相信各个读者对于深奥的 HttpModule 不再陌生了，接下来我们将继续在 ASP.NET Framework 的世界里历险,下一站的目的地是神秘的 IhttpHandler,也就是 HTTP 请求处理中心了！

4.3、ASP.NET　Framework 深度历险　－　初次接触神秘的 IHttpHandler

我们在上一小节在 HttpModule 的世界历险的时候就提到过，HttpHandler 是一个 HTTP 请求的真正处理中心，也正是在这个 HttpHandler 容器当中，ASP.NET Framework 才真正的对客户端请求的服务器页面做出编译和执行，并将处理过后的信息附加在 HTTP 请求信息流中再次返回到 HttpModule 中。

在本小节，我们将会一同来初步了解一下神秘的 HttpHandler。

4.3.1　IHttpHandler 是什么？

我们一直在使用 HttpHandler，而不是 IHttpHandler，请不要混淆这两者。前者只是我们用来描述使用的一个名称而已，而后者则是 ASP.NET Framework 中一个重要的接口的名称，而我们历险的重点自然是在 IHttpHandler。

IHttpHandler 是 ASP.NET Framework 提供的一个接口，它定义了如果要实现一个 HTTP 请求的处理所需要必需实现的一些系统约定。也就是说，如果你想

要自行处理某种类型的 HTTP 请求信息流的话，你需要实现这些系统约定才能够做到，这也正是接口在.NET 中的作用。
我们先来看看 IHttpHandler 的真实面目吧：

语法：public interface IHttpHandler
需求：
名称空间: System.Web
平台: Windows 2000, Windows XP Professional,
Windows .NET Server family
装配件: System.Web (in System.Web.dll)
成员字段：IsReusable

成员方法：void ProcessRequest(HttpContext context);

HttpHandler 同 HttpModule 类似，系统都默认提供了很多的系统 HttpHandler 类，用来处理不同的 HTTP 请求。

同样的，这些默认的系统 HttpHandler 类也和系统的 HttpModule 一样在 machine.config 文件中进行配置的，下面我们就来仔细看看了：
在文件 machine.config 中可以看到下面的配置段：

```
<httpHandlers>
 <add verb="*" path="trace.axd"
type="System.Web.Handlers.TraceHandler"/>
 <add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
 <add verb="*" path="*.ashx"
type="System.Web.UI.SimpleHandlerFactory"/>
 <add verb="*" path="*.asmx"
type="System.Web.Services.Protocols.WebServiceHandlerFactory,
System.Web.Services, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
 <add verb="*" path="*.rem"
type="System.Runtime.Remoting.Channels.Http.HttpRemotingHandlerFacto
ry, System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" validate="false"/>
 <add verb="*" path="*.soap"
type="System.Runtime.Remoting.Channels.Http.HttpRemotingHandlerFacto
ry, System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" validate="false"/>
 <add verb="*" path="*.asax" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.ascx" type="System.Web.HttpForbiddenHandler"/>
```

```
 <add verb="*" path="*.config"
type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.cs" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.csproj"
type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.vb" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.vbproj"
type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.webinfo"
type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.asp" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.licx" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.resx" type="System.Web.HttpForbiddenHandler"/>
 <add verb="*" path="*.resources"
type="System.Web.HttpForbiddenHandler"/>
 <add verb="GET,HEAD" path="*" type="System.Web.StaticFileHandler"/>
 <add verb="*" path="*"
type="System.Web.HttpMethodNotAllowedHandler"/>
</httpHandlers>
```

可以看到，ASP.NET Framework 为 ASP.NET 应用程序的正常运行提供了很多的系统默认 HttpHandler 类，用来适应不同类型的 HTTP 请求。比如，我们最熟悉的*.aspx 文件，用来处理此类型的 HTTP 请求，ASP.NET Framework 将会交给一个名为 System.Web.UI.PageHandlerFactory 的 HttpHandler 类来处理。

我们在这里来讲解一下<httpHandlers>配置代码的用法：

主配置标签语法：

```
<httpHandlers>
   <add verb="verb list" path="path/wildcard" type="type,assemblyname"
validate="" />
   <remove verb="verb list" path="path/wildcard" />
   <clear />
</httpHandlers>
```

子配置标签语法：

<add>：详细指定需要的动词（GET/HEAD 等等）以及相应的请求文件路径。其中的路径文件支持通配符*。

<remove>：移除一个到 HttpHandler 上的映射。并且不支持通配符。

<clear>：移除所有的到 HttpHandler 上的映射。

而 HttpHandler 和 HttpModule 一样,系统会在最初始由 ASP.NET Framework 首先加载 machine.config 中的系统 HttpHandler，而后会加载 Web 应用程序所在目录的 web.config 中的用户自定义的 HttpHandler 类。

但是同 HttpModule 不同的是：一旦定义了自己的 HttpHandler 类，那么它对系统的 HttpHandler 的关系将是“覆盖”关系的，也就是说如果我们自己定义了一个对*.aspx 请求的自定义 HttpHandler 类的话，那么系统将会将对此 HTTP 请求的处理权限完全交给我们自己定义的这个 HttpHandler 类来处理，而我们的 HttpHandler 类则需要完全的解析这个 HTTP 请求，并做出处理，如果不这样的话，那么，这个 HTTP 请求将会被空 HTTP 信息流所代替，或者被我们自定义的 HttpHandler 类中的信息数据流所代替。这一点我们会在后面的章节中详细探讨。我们在这里仅仅给出一个系统默认 HttpHandler 和我们的自定义 HttpHandler 之间的关系图：

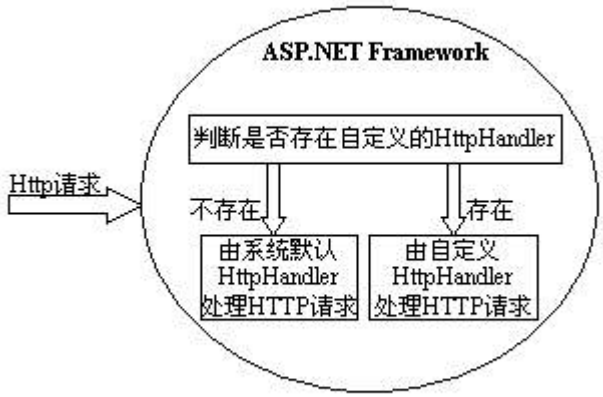

图 4.3.1

在下一小节中，我们将来探讨一下 IHttpHandler 是如何处理一个 HTTP 请求的。

4.3.2 IHttpHandler 如何处理 HTTP 请求

在上一小节中我们初步介绍了 IHttpHandler 的知识，接下来我们会探讨一下当一个 HTTP 请求被传递到一个 HttpHandler 容器之后，HttpHandler 容器是如何对这个 HTTP 请求进行解析和处理的。

我们在上面小节中可以知道，一个 HttpHandler 容器是实现了一个名为 IHttpHandler 的接口，而我们也了解了 IHttpHandler 接口需要实现的描述。我们能注意到，在 IHttpHandler 中最为重要的一个成员方法就是：

ProcessRequest。从字面上面我们可以很容易的得知，这个方法就是 HttpHandler 用来处理一个 HTTP 请求的，实事的确如此。当一个 HTTP 请求经由 HttpModule 容器传递到 HttpHandler 容器中的时候，ASP.NET Framework 会调用 HttpHandler 的 ProcessRequest 成员方法来对这个 HTTP 请求做真正的处理，我们以一个 ASPX 页面来讲，正是在这里一个 ASPX 页面才被系统处理解析，并将处理完成的结果继续（注意：这里是继续，也就是说仍在一个 HttpModule 的生命周期内）经由 HttpModule 传递下去，直至到达客户端。

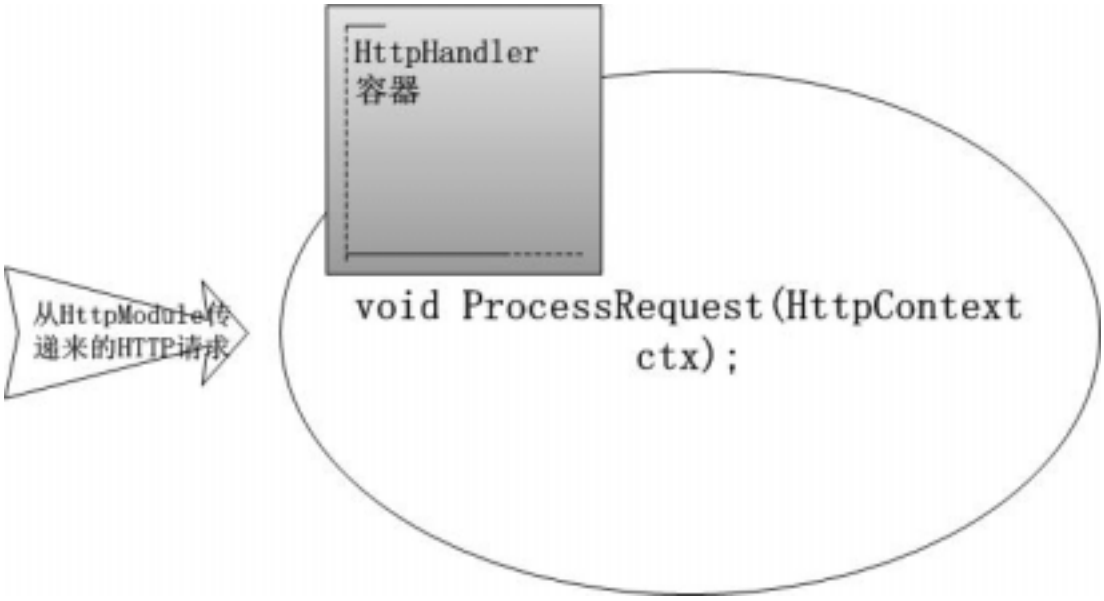

图 4.3.2

我们从 4.3.1 小节中的配置文件中可以了解到对于 ASPX 页面，ASP.NET Framework 在默认情况下是交给 System.Web.UI.PageHandlerFactory 这个 HttpHandlerFactory 来处理的，需要注意的这里并不是我们一直在讨论的 HttpHandler 容器,而是一个 HttpHandler 工厂,这一点我们会在下面详细讨论，这里我们仅仅简单的介绍一下什么是 HttpHandler 工厂。所谓一个 HttpHandlerFactory,是指当一个 HTTP 请求到达这个 HttpHandler 工厂的时候，HttpHandlerFactory 会提供出一个 HttpHandler 容器，交由这个 HttpHandler 容器来处理这个 HTTP 请求，这种处理方式我们在第三章中的 Factory 设计模式提到过，这样做的好处是大大减轻了系统的压力，提高了系统的适应性和灵活度。

但不论怎样，一个 HTTP 请求都是最终交给一个 HttpHandler 容器中的 ProcessRequest 方法来处理的。因此我们在下面的小节中将会深入的探讨一下 IHttpHandler。

4.4、ASP.NET Framework 深度历险 – IHttpHandler 深入

在 4.3 小节，我们了解到了 IHttpHandler 是如何来处理一个 HTTP 请求的，在本小节我们会更加深入讨论有关 IHttpHandler 的一些技术细节。

在本小节内，我们先来共同实现一个简单的 HttpHandler 容器，以便能对 HttpHandler 容器有一个感性的认识。

通过实现 IHttpHandler 接口我们可创建自定义 HTTP 处理程序，而该接口只包含两个方法。通过调用 IsReusable，IHttpHandlerFactory 可查询处理程序以确定是否可使用同一实例为多个请求提供服务。ProcessRequest 方法将 HttpContext 实例用作参数，这使它能够访问 Request 和 Response 内部对象。

我们来看看下面的这个例子：

1）在 VS.NET 中建立一个新的“ 类库 ”项目，命名为：MyHandler

2）在代码区域键入如下代码：

```
using System;
using System.Web;

namespace MyNamespace
{
/// <summary>
/// 目的：实现简单的自定义 HttpHandler 容器
///
/// </summary>
public class MyHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
              HttpResponse Response=context.Response;
              HttpRequest Request=context.Request;
              Response.Write("<h1><b>Hello world!</b></h1>");
    }
    public bool IsReusable
    {
              get{return true;}
    }
}
}
```

3)编译这个项目，我们得到了 MyHandler.dll 这个文件。

4）将这个 MyHandler.dll 文件拷贝到我们的测试 Web 应用程序目录中的\bin 子目录。

5）配置我们的测试 Web 应用程序配置文件 Web.Config 如下：

在配置文件 Web.Config 中增加如下信息：

```
<httpHandlers>
  <add verb="*" path="*" type="MyNamespace.MyHandler,MyHandler"/>
</httpHandlers>
```

其中的 type 属性以逗号分隔开的分别是：自定义 HttpHandler 中的类名称以及最终的程序集文件（也就是我们编译得到的那个 DLL 文件）。

而其中的 path 属性则表示我们的自定义 HttpHandler 将会接管何种类型的文件请求。

这样，经过上面的配置之后，对于所有类型的页面文件 HTTP 请求都将会被我们的自定义 HttpHandler 捕获。而我们的自定义 HttpHandler 则忽略了所有的 HTTP 请求，替代输出的是一段经典的“Hello world!”语句。

读者可以试着运行这段例子，我们给出的这个简单例子运行结果如下：

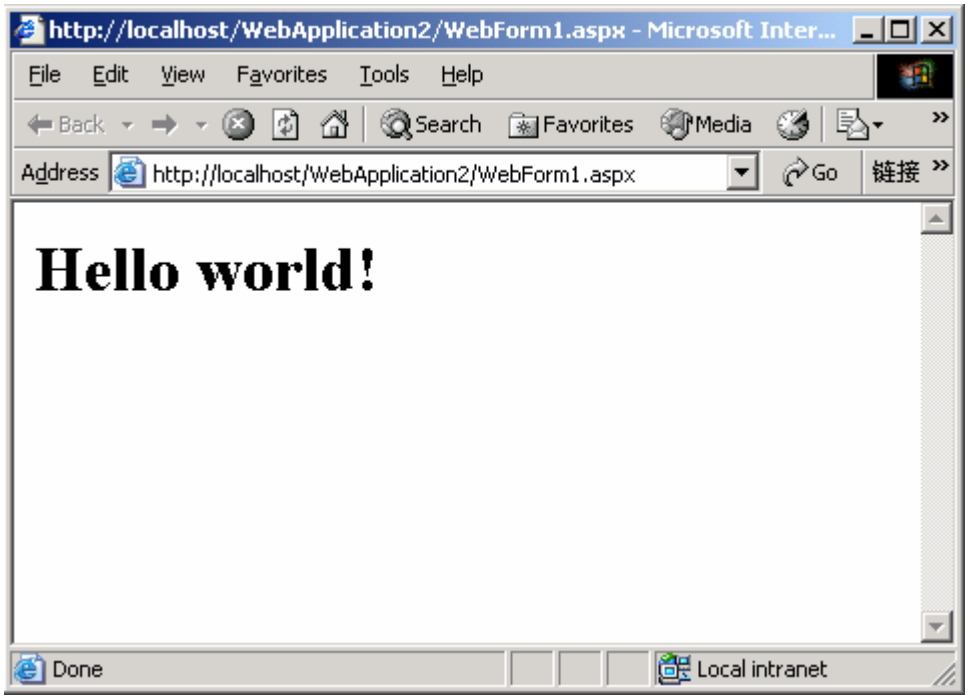

图 4.3.3

这样，一个我们自定义的 HttpHandler 容器就完全运行的和我们料想的那样一样了，也最终证实了一个 HTTP 请求的真正处理中心是在一个 HttpHandler 容器中的。

从上面的例子代码中我们可以看到，ProcessRequest 方法传递进来的是一个 HttpContext 对象，而我们通过这个 HttpContext 对象可以方便的使用我们平常在 ASP.NET 页面中经常直接使用的 Response 对象以及 Request 对象，那么我们能够使用 Session 对象吗？如果你只是简单的类似使用 Response 对象那样直接在我们的自定义 HttpHandler 容器中使用一个 Session 对象的话，编译器将会编译出错。

在一个 HttpHandler 容器中如果需要访问会话状态对象 Session，你必须实现一个 IRequiresSessionState 接口。

IRequiresSessionState 接口指定目标 HTTP 处理程序接口具有对会话状态值的读写访问权限。这只是一个标记接口，没有任何方法。

IRequiresSessionState 接口的语法如下：
命名空间: System.Web.SessionState
平台: Windows 2000, Windows XP Professional, Windows .NET Server family
程序集: System.Web (在 System.Web.dll 中)

也就是说我们只有在我们的程序中实现了这个接口，才有权利访问 Session 对象。这个接口和其他的普通接口不太一样的是，他没有任何方法，仅仅起到一个标记的作用而已。
我们修改我们的程序如下，以便能够在我们的自定义 HttpHandler 容器中存取 Session 对象：

```
using System;
using System.Web;
using System.Web.SessionState;
namespace MyNamespace
{
/// <summary>
/// 目的：实现简单的自定义 HttpHandler 容器
///
/// </summary>
public class MyHandler : IHttpHandler ,IRequiresSessionState
{
    public void ProcessRequest(HttpContext context)
    {
                        //声明我们自己的 Response 对象
                        HttpResponse Response=context.Response;

                        //声明我们自己的 Request 对象
                        HttpRequest Request=context.Request;
```

```
                        //声明我们自己的 Session 对象
                        HttpSessionState Session=context.Session;

                        //打印一段话
                        Response.Write("<h1><b>Hello
world!</b></h1><br>");

                        //给 Session 对象赋值
                        Session["test"]="this is a session test";

                        //打印测试 Session 对象的值

    Response.Write("<h1><b>Session[\"test\"]="+Session["test"].ToStr
ing()+"</b></h1>");
    }
    public bool IsReusable
    {
              get{return true;}
    }
}
}
```

这样，在我们的自定义 HttpHandler 容器中就可以正常存取 Session 对象了。

重新编译这个项目，将生成的 MyHandler.dll 文件覆盖到我们刚才的测试 Web 应用程序的\bin 子目录中，再次运行测试 Web 应用程序我们会得到如下结果：

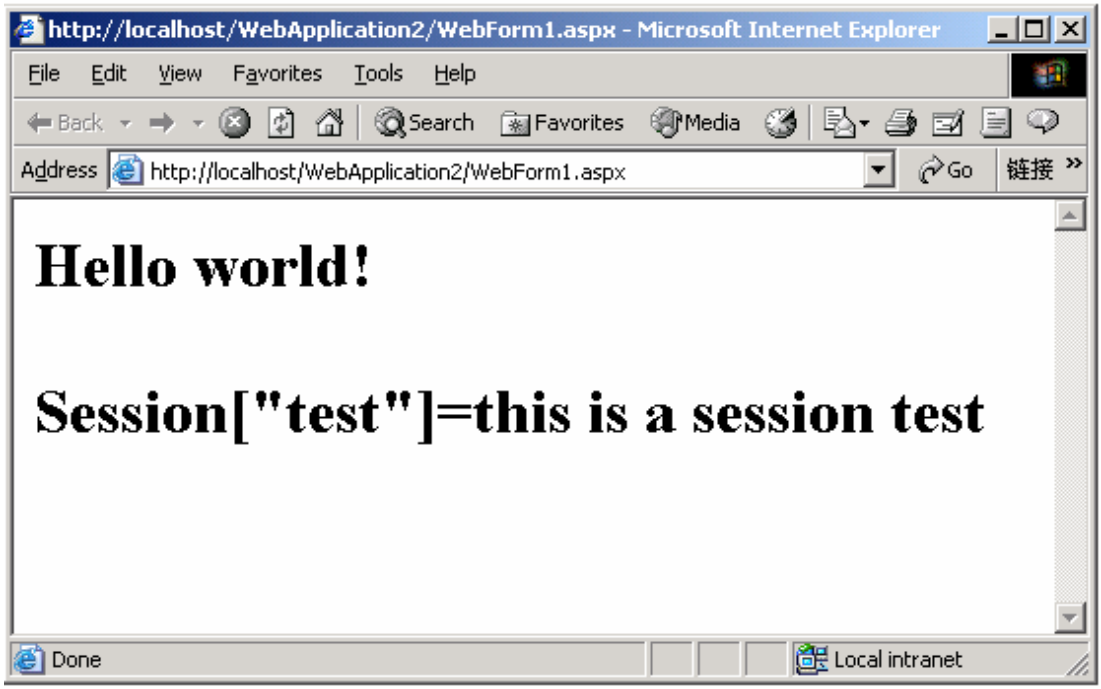

图 4.3.4

在上面我们曾经提到过，ASP.NET Framework 实际上并不是直接将相关的页面资源 HTTP 请求定位到一个其内部默认的 IHttpHandler 容器之上的，而是定位到了其内部默认的 IHttpHandler 工厂上了。

这个 IHttpHandler 工厂的作用就是对很多的系统已经实现了的 IHttpHandler 容器进行调度和管理的。这样做的优点是大大增强了系统的负荷处理能力，能够很好的提升效率。

接下来，我们就看看 IHttpHandler 工厂和 IHttpHandler 容器之间的关系吧。

首先我们来了解一下 IHttpHandler 工厂的语法定义：

命名空间: System.Web

平台: Windows 2000, Windows XP Professional, Windows .NET Server family

程序集: System.Web (在 System.Web.dll 中)

公共方法：

1) IHttpHandler GetHandler(HttpContext context,string requestType,string url,string pathTranslated);

作用：返回实现 IHttpHandler 接口的类的实例。

参数：

context ：HttpContext 类的实例，它提供对用于为 HTTP 请求提供服务的内部服务器对象（如 Request、Response、Session 和 Server） 的引用。

requestType ：客户端使用的 HTTP 数据传输方法（GET 或 POST）。
url ：所请求资源的 RawUrl。
pathTranslated ：所请求资源的 PhysicalApplicationPath。

返回值：
处理请求的新的 IHttpHandler 对象。

2）ReleaseHandler
作用：使工厂可以重用现有的处理程序实例。

4.5、ASP.NET Framework 深度历险 – IHttpModule 以及 IHttpHandler 应用实例

4.6、ASP.NET Framework 深度历险 – 深入 ASP.NET 事件模型机制

ASP.NET 相对于以前古老的 ASP 来讲，一个很重要的革命性技术就是 ASP.NET 是完全基于事件驱动的一种技术，而这点是 ASP 是不可比拟的，深入的探索 ASP.NET 的事件模型机制是本章的内容。

4.6.1 ASP.NET 事件模型初步认识

ASP.NET 之所以对于以前的 ASP 是一个革命性的巨变，在很大程度上面是由于 ASP.NET 技术是一种完全基于事件驱动的全新技术，如果你以前曾经接触过 Delphi 或者 VB，你一定就会了解事件驱动技术是怎样的一种技术了。但是需要特别提醒读者注意的事，ASP.NET 的事件模型机制以不同于一般的 Client/Server 时代的事件模型机制。在 Client/Server 时代，所有的事件捕捉和触发以及处理都是交给客户端的应用程序来实现的，而服务端则用来执行复杂的耗费时间的计算，从而也决定了 Client/Server 时代的事件模型机制的相对简单和容易实现。

ASP.NET 是完全基于 HTTP 协议的，而 HTTP 协议是一种无连接的 Web 协议，也就是说每当一次客户端的 Http 请求处理结束之后，服务器端就会自动和客户端断开连接。从而我们应当可以知道，ASP.NET 的事件驱动是和 Client/Server 时代的事件驱动有所不同的一种基于 HTTP 协议的技术，在 ASP.NET 中事件的触发和事件的处理是分别在客户端和服务端进行的。一个事件在客户端被触发之后，会通过 HTTP 协议以 POST 的方式发送到服务器端，而服务器端则通过 ASP.NET 页面架构（ASP.NET page framework）来进行相应的处理和反馈。

说到这里，我们来回忆一下曾经写过的一些 ASP.NET 程序，如果你仔细留意的话，应当会发现，每一个页面在我们使用 VS.NET 生成的时候，会自动的将所有的 Web Control 以及相应的标签放置到一个默认的<Form>当中，并且默认的传递方式是 POST，也可以是 GET 方式。你可以试着移去或者更改这个<Form>标签，但是当你重新编译并运行这个页面的时候，你应当发现系统仍然自动的将这个<Form>标签更改为自己需要的了，这也就验证了上面我们刚刚提到的 ASP.NET 需要客户端以 POST/GET 的方式通过 HTTP 协议将事件信息发送到服务器端，从而服务器端才能处理这些事件。

ASP.NET 页面架构在服务器端接收到这些来自客户端的事件信息之后，会自动的判别并决定调用相应的方法来进行事件处理。我们通过下图可以清晰的了解到这一点：

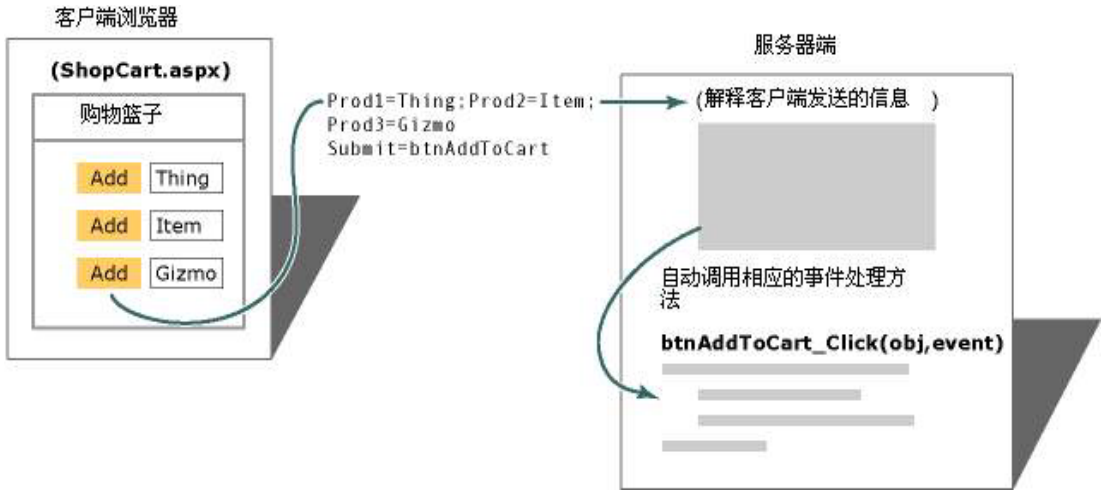

图 4.6.1

从上图我们可以看到，ASP.NET Framework 负责了从客户端事件捕获、传递、事件信息解释的全部过程，从另外一个方面来说，开发 ASP.NET 应用程序的时候，你不必亲自去管理这些事件模型的技术细节，而是可以将更大的时间和精力投入到商业逻辑的分析设计当中去，节约了大量的时间。当然由于事件的整个处理过程属于 ASP.NET 的核心技术，我们也可能去强行的干预到。由于 ASP.NET 为我们做了所有的事件管理和处理工作，我们就可以像往常的普通应用程序那样来自如的编写各种事件的响应代码了。

4.6.2 ASP.NET 的事件模型深入了解

在上一节我们初步的了解了一下 ASP.NET 事件模型机制的知识，在这一节我们来详细的深入进去看看 ASP.NET 事件驱动模型的内部到底是怎样的一回事。

从本质上面来讲，ASP.NET 的事件模型就是 Web Forms 的事件模型，我们先来看下面的一个例子代码片断：

（如果没有特殊指明我们的例子代码均采用代码绑定机制）

页面代码 WebForm1.aspx：

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false" Inherits="WebApplication1.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
              <title>WebForm1</title>
              <meta name="GENERATOR" Content="Microsoft Visual Studio
7.0">
              <meta name="CODE_LANGUAGE" Content="C#">
              <meta name="vs_defaultClientScript"
content="JavaScript">
              <meta name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
    </HEAD>
    <body MS_POSITIONING="GridLayout">
              <form id="Form1" method="post" runat="server">
                        <asp:Button id="Button1" style="Z-INDEX: 101;
LEFT: 243px; POSITION: absolute; TOP: 114px" runat="server" Text="事
件模型测试"></asp:Button>
              </form>
    </body>
</HTML>
```

CodeBehind 代码 WebForm1.aspx.cs：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
namespace WebApplication1
```

```
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
              protected System.Web.UI.WebControls.Button Button1;
              private void Page_Load(object sender, System.EventArgs e)
              {
                        // Put user code to initialize the page here
              }

              #region Web Form Designer generated code
              override protected void OnInit(EventArgs e)
              {
                        //
                        // CODEGEN: This call is required by the ASP.NET
Web Form Designer.
                        //
                        InitializeComponent();
                        base.OnInit(e);
              }

              /// <summary>
              /// Required method for Designer support - do not modify
              /// the contents of this method with the code editor.
              /// </summary>
              private void InitializeComponent()
              {
                        this.Button1.Click += new
System.EventHandler(this.Button1_Click);
                        this.Load += new
System.EventHandler(this.Page_Load);
              }
              #endregion
              private void Button1_Click(object sender,
System.EventArgs e)
              {
                        Response.Write("Hello,world!");
```

```
        }
    }
}
```

上面的一段最简单的代码就是在页面打印一句古老的“Hello,world!”。

运行的结果如下：

图 4.6.2

上面的最简单的代码运行结果是在页面上面打印一句最简单的“Hello,world!”。仔细的查阅上面的 CodeBehind 代码，我们可以简单的了解到如下几点：

1、ASP.NET Framework 是通过如下方式来调用事件处理方法的：

```
this.Button1.Click += new System.EventHandler(this.Button1_Click);
```

2、事件 Button1_Click 的参数共有两个：

```
Button1_Click(object sender, System.EventArgs e)
```

虽然我们给出的例子相当的简单，但是事件驱动的模型机制却是完全一致的，下面就然我们来由浅入深的逐步的深入到 ASP.NET 的事件模型机制的内部看一看。

ASP.NET 的所有的事件处理方法都只有相同的两个参数：object 类型参数以及 System.EventArgs 类型的参数。注意，我们提到的所有的事件处理方法都只有这两个相同的参数，唯一会有区别的是，其他的某些事件处理方法的参数 2 会是 System.EventArgs 的一个子类，以便适应特殊的处理要求。

在此我们再一次强调一下，ASP.NET 技术是完全构建在.NET 框架技术之上的，它的所有的技术细节均来源于.NET 框架。正如刚才我们看到的事件处理方法的两个参数，他们都是一个类（Class），因而你应当不难理解刚才所讲的“ 其他的某些事件处理方法的参数 2 会是 System.EventArgs 的一个子类 ”，也就是在某些情况下，参数会是继承于 System.EventArgs 类的一个子类。

或许有些读者看到这里会有些疑问，为什么所有的事件处理方法都必须要有相同的参数呢？原因还在我们一再强调的那句话“ ASP.NET 技术是完全构建在.NET 框架技术之上的 ”。

在我们给出的代码中，是通过代表（Delegate）的机制来将事件和事件处理方法绑定在一起的，而代表（Delegate）则是.NET 框架的一个机制。在 ASP.NET 中需要通过.NET 提供的 EventHandler 来做到这个绑定的：

```
[Serializable]
public delegate void EventHandler(object sender, EventArgs e);
```

由此我们可以清楚的了解到为什么所有的 ASP.NET 事件处理方法都需要有相同的两个参数了，因为它们都是通过 EventHandler 来实现事件和处理方法的绑定的。

在我们继续前进之前，我们先来透彻的了解一下事件和代表之间的关系。

一个事件（Event）是一个对象发送的一个消息，用来表示一个动作发生了。而一个动作可以被用户操作或者其他程序所触发。触发事件的对象被事件发送者（Event Sender）调用；捕获处理事件的对象被事件接收者（Event Receiver）调用。

在事件通讯当中，事件的发送者并不知道哪个对象或者方法将要去接收/处理发送过去的事件。因而在事件源和事件接收者之间就需要一个中间人存在，这一点类似于“ 指针 ”。而.NET 架构专门定义了一个特殊的类型用来提供一个指向函数的指针，就是“ 代表 ”，也叫做“ 委托 ”。

一个代表就是一个类（class），他提供了一个面向某个方法的引用参考。和其他一般普通的类不一样，一个代表类拥有一个签名（Signature），从而使得代表能够提供面向符合自己签名的方法引用参考。所谓“ 签名 ”指的是：一个函数方法的参数个数、参数类型的集合。因此，一个代表也可以被认为是一个类型安全的函数指针或者回调函数。当然，代表并不仅仅是为事件本身而设计的，他还有其他更为强大的功用，而在这里我们仅仅探讨事件捕获类型的代表。

下面的例子简单的告诉我们一个事件代表是如何定义的：

```
// AlarmEventHandler 是一个警报事件的代表
// AlarmEventArgs 是用来捕获来自于警报事件的数据的代表类，他继承于父类
//EventArgs.
public delegate void AlarmEventHandler(object sender, AlarmEventArgs
e);
```

从上面的代码我们可以看到，一个代表的定义语法和我们一般的函数定义很类似，但是关键字 delegate 会通知编译器这是一个代表类型。

在.NET 框架中，约定事件代表（Event Delegate）拥有两个参数，一个表示事件源，另外一个表示事件的数据。

一个代表的定义提供代表的签名，CLR(common language runtime)会使用这个签名。

一个代表的实例可以被绑定到任何一个符合代表签名的方法上，比如：

```
public class WakeMeUp
{
// AlarmRang has the same signature as AlarmEventHandler.
public void AlarmRang(object sender, AlarmEventArgs e){...};
...
}
```

下面是将事件绑定到相应事件处理方法的代码：

```
 WakeMeUp w = new WakeMeUp();
AlarmEventHandler alhandler = new AlarmEventHandler(w.AlarmRang);
```

这样，在 alhandler 被触发的时候，它将会依次调用方法 w.AlarmRang。至于更为详细代表机制和技术细节则不属于本书的探讨范围之内了，读者可以自行查阅相关的技术资料来了解。

现在我们应当可以理解这句代码的含义了：

```
this.Button1.Click += new System.EventHandler(this.Button1_Click);
```

这句代码表示将事件 this.Button1.Click 绑定在方法 this.Button1_Click 上面了，也就是说一旦触发按钮的 Click 事件，服务器端将会自动的调用方法 Button1_Click 来处理。

ASP.NET Framework 提供给我们的服务器端事件并不是很多，比如我们在 ASP 时代时常用到的 OnMouseOver 等等事件均没有提供，这是什么原因呢？其实从上面我们的讲解当中就应当了解到，在 ASP.NET Framework 下，事件驱动模型机制的实现是在客户端和服务端分别实现的，之间需要通过 HTTP 协议方式来传递事件信息，因而如果频繁的触发各类事件会对整个 Web 站点产生很大的流量压力，比如刚刚提到的 OnMouseOver 事件，每次微小的鼠标移动都将会触发它，这对于服务器端的压力是非常大的，因而系统并没有提供。这些特殊的需要的事件我们需要在客户端自动的实现处理，从而大大减轻服务器端的压力，也带来的更多的灵活性。

接下来我们来谈谈事件的触发顺序，也许读者会有这样的疑问：事件的触发顺序还有必要来讨论吗？我们平常的事件触发不都是自然的有自己的顺序的吗？比如吃饭，我当然要先触发“ 张开嘴 ”这个事件，再触发“ 送入食物 ”这个事件了。

我们平常生活中的事件触发的确如此，但是在 ASP.NET 的世界当中却不是这样的，正如我们上面提到的，如果频繁的和服务器端进行事件信息的传递会

大大降低服务器的处理效率和性能，因而有些事件比如 OnMouseOver 在 ASP.NET 中并没有提供，但是有些事件虽然也会频繁的触发但是必须提供，比如很多情况下要用到的 Change 事件。对于这种情况，ASP.NET Framework 提供了一个折衷的办法，就是对于这类事件在触发的时候，并不是立即将事件信息发送到服务器，而是缓存在客户端，等到再一次的事件信息被发送到服务器端的时候一同发送回去。因此，这些缓存着的事件以及刚刚被触发的事件在服务器端被接收到的时候，ASP.NET Framework 并不会按照特定的顺序去解释执行处理这些事件。

下面我们来通过一个实际的例子来验证这一点：

页面文件：WebForm1.aspx

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false" Inherits="WebApplication1.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
              <title>WebForm1</title>
              <meta name="GENERATOR" Content="Microsoft Visual Studio
7.0">
              <meta name="CODE_LANGUAGE" Content="C#">
              <meta name="vs_defaultClientScript"
content="JavaScript">
              <meta name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
    </HEAD>
    <body MS_POSITIONING="GridLayout">
              <form id="Form1" method="post" runat="server">
                        <asp:Button id="Button1" style="Z-INDEX: 101;
LEFT: 233px; POSITION: absolute; TOP: 302px" runat="server" Text="事
件模型测试"></asp:Button>
                        <asp:DropDownList id="DropDownList1"
style="Z-INDEX: 102; LEFT: 257px; POSITION: absolute; TOP: 171px"
runat="server">
                                  <asp:ListItem>Item 1</asp:ListItem>
                                  <asp:ListItem>Item 2</asp:ListItem>
                                  <asp:ListItem>Item 3</asp:ListItem>
                                  <asp:ListItem>Item 4</asp:ListItem>
                                  <asp:ListItem>Item 5</asp:ListItem>
                                  <asp:ListItem>Item 6</asp:ListItem>
                        </asp:DropDownList>
```

```
                        <asp:TextBox id="TextBox1" style="Z-INDEX: 103;
LEFT: 210px; POSITION: absolute; TOP: 219px" runat="server">我首先改
变这个</asp:TextBox>
            </form>
    </body>
</HTML>

CodeBehind 文件：WebForm1.aspx.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace WebApplication1
{
    /// <summary>
    /// 测试事件触发处理顺序
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
            protected System.Web.UI.WebControls.DropDownList
DropDownList1;
            protected System.Web.UI.WebControls.TextBox TextBox1;
            protected System.Web.UI.WebControls.Button Button1;

            private void Page_Load(object sender, System.EventArgs e)
            {
                    // Put user code to initialize the page here
            }

            #region Web Form Designer generated code
            override protected void OnInit(EventArgs e)
            {
```

```
                        //
                        // CODEGEN: This call is required by the ASP.NET
Web Form Designer.
                        //
                        InitializeComponent();
                        base.OnInit(e);
                }

                /// <summary>
                /// Required method for Designer support - do not modify
                /// the contents of this method with the code editor.
                /// </summary>
                private void InitializeComponent()
                {
                        this.Button1.Click += new
System.EventHandler(this.Button1_Click);
                        this.DropDownList1.SelectedIndexChanged += new
System.EventHandler(this.DropDownList1_SelectedIndexChanged);
                        this.TextBox1.TextChanged += new
System.EventHandler(this.TextBox1_TextChanged);
                        this.Load += new
System.EventHandler(this.Page_Load);

                }
                #endregion

                private void Button1_Click(object sender,
System.EventArgs e)
                {
                        Response.Write("Hello,world!<br>");
                }

                private void DropDownList1_SelectedIndexChanged(object
sender, System.EventArgs e)
                {
                        Response.Write("DropDownList Change 事件被触发
了！<br>");
                }
```

```
            private void TextBox1_TextChanged(object sender,
System.EventArgs e)
            {
                    Response.Write("TextBox1_TextChanged 事件被触
发了！<br>");
            }
    }
}
```

在上面的例子中，我们在页面上面放置了一个下拉框以及一个 TextBox 文本框，我们首先改变 TextBox 中的文字，在改变下拉框的选项，我们可以看到这两个 Change 事件并没有立即触发 POST 动作，而是被客户端暂时缓存了，当我们点击测试按钮的时候，才刷新整个页面，也就是提交给了服务器端进行消息处理，但是处理的顺序并不是我们刚才进行的那样，可以看到下面的结果：

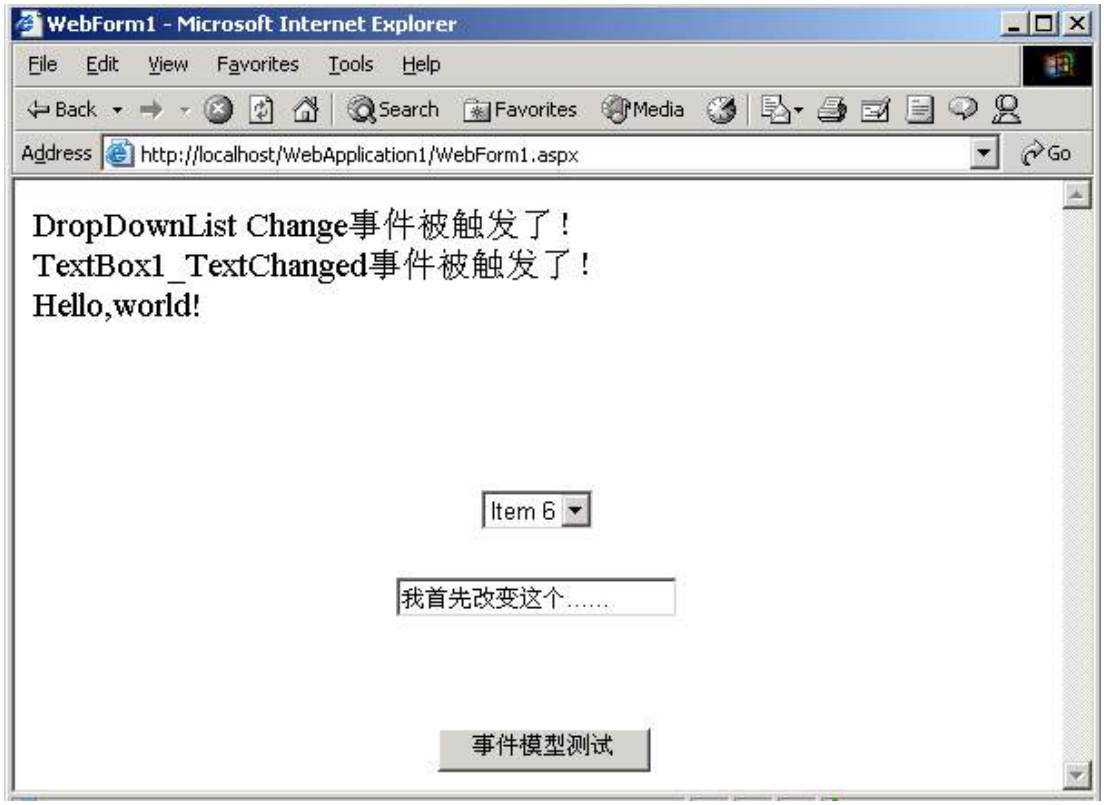

图 4.6.3

读者可以试着自己将上面的代码拷贝到 VS.NET 中运行一下看看结果怎样。

4.7、ASP.NET　Framework 深度历险　–　深入 ASP.NET 状态管理模型

4.8、ASP.NET　Framework 深度历险　–　深入 ASP.NET 安全管理模型

4.9、ASP.NET　Framework 深度历险　–　利用 MSMQ、SOAP 实现分布式应用系统

4.10、ASP.NET　Framework 深度历险　–ASP.NET 中的设计模式（Design Pattern）应用