**[WCF技术剖析之二：再谈IIS与ASP.NET管道](http://www.cnblogs.com/artech/archive/2009/06/20/1507165.html)**

http://www.cnblogs.com/artech/archive/2009/06/20/1507165.html

在2007年9月份，我曾经写了三篇详细介绍IIS架构和ASP.NET运行时管道的文章，深入介绍了IIS 5.x与IIS 6.0HTTP请求的监听与分发机制，以及ASP.NET运行时管道对HTTP请求的处理流程：

[[原创]ASP.NET Process Model之一：IIS 和 ASP.NET ISAPI](http://www.cnblogs.com/artech/archive/2007/09/09/887528.html)   
[[原创]ASP.NET Process Model之二：ASP.NET Http Runtime Pipeline - Part I](http://www.cnblogs.com/artech/archive/2007/09/13/891262.html)   
[[原创]ASP.NET Process Model之二：ASP.NET Http Runtime Pipeline - Part II](http://www.cnblogs.com/artech/archive/2007/09/13/891266.html)

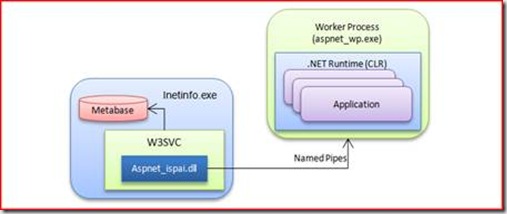
很多人留言为何没有IIS 7的介绍。在写作《WCF深入剖析》中，为了剖析基于IIS的WCF服务寄宿（Hosting），再次对相关内容进行了研究，在这里一并与大家分享。

**IIS 5.x与ASP.NET**

我们先来看看IIS 5.x是如何处理基于ASP.NET资源（比如.aspx,.asmx等）请求的，整个过程基本上可以通过**图1**体现。

IIS 5.x运行在进程InetInfo.exe中，在该进程中一个最重要的服务就是名为World Wide Web Publishing Service（简称W3SVC）的Windows Service。W3SVC的主要功能包括HTTP请求的监听、工作进程的管理以及配置管理（通过从Metabase中加载相关配置信息）等。

当检测到某个HTTP Request后，先根据扩展名判断请求的是否是静态资源（比如.html,.img,.txt,.xml等），如果是则直接将文件内容以HTTP Response的形式返回。如果是动态资源（比如.aspx,asp,php等等），则通过扩展名从IIS的脚本影射（Script Map）找到相应的ISAPI Dll。



**图1 IIS 5.x与ASP.NET**

ISAPI是Internet服务器API（Internet Server Application Programming Interface）的缩写，是一套本地的（Native）Win32 API，具有较高的执行性能,是IIS和其他动态Web应用或者平台之间的纽带。比如ASP ISAPI桥接IIS与ASP，而ASP.NET ISAPI则连接着IIS与ASP.NET。ISPAI定义在一个Dll中，ASP.NET ISAPI对应的Dll为Aspnet\_isapi.dll,你可以在目录“%windir%\Microsoft.NET\Framework\{version no}\”中找到该Dll。

ISAPI支持ISAPI扩展（ISAPI Extension）和ISAPI筛选（ISAPI Filter），前者是真正处理HTTP请求的接口，后者则可以在HTTP请求真正被处理之前查看、修改、转发或者拒绝请求，比如IIS可以利用ISAPI筛选进行请求的验证（Authentication）。

如果我们请求的是一个基于ASP.NET的资源类型，比如：.aspx Web Page、 .asmx Web Service或者.svc WCF Service等，Aspnet\_isapi.dll会被加载，ASP.NET ISAPI扩展会创建ASP.NET的工作进程（如果该进程尚未启动），对于IIS 5.x来说，该工作进程为aspnet.exe。IIS进程与工作进程之间通过命名管道（Named Pipes）进程通信，以获得最好的性能。

在工作进程初始化过程中，.NET 运行时（CLR）被加载，从而构建了一个托管的环境。对于某个Web应用的初次请求，CLR会为其创建一个AppDomain。在此AppDomain中，HTTP运行时（HTTP Runtime）被加载并用以创建相应的应用。对于寄宿于IIS 5.x的所有Web 应用都运行在同一个进程（工作进程Aspnet\_wp.exe）的不同AppDomain中。

**IIS 6与ASP.NET**

通过上面的介绍，我们可以看出IIS 5.x至少存在着如下两个方面的不足：

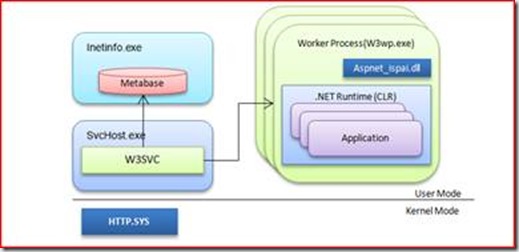
* ISAPI Dll被加载到InetInfo.exe进程中，它和工作进程之间是一种典型的跨进程通信方式，尽管采用性能最好的命名管道，但是仍然会带来性能的瓶颈；
* 所有的ASP.NET应用，运行在相同的进程（aspnet\_wp.exe）中的不同的应用程序域（AppDomain）中，基于应用程序域的隔离级别不能从根本上解决一个应用程序对另一个程序的影响，在更多的时候，我们需要不同的Web应用运行在不同的进程中。

在IIS 6.0中，为了解决第一个问题，ISAPI.dll被直接加载到工作进程中。为了解决第2个问题，引入了应用程序池（Application Pool）的机制。我们可以为一个或者多个Web应用创建应用程序池，每一个应用程序池对应一个独立的工作进程，从而为运行在不同应用程序池中的Web应用提供基于进程的隔离级别。IIS 6.0的工作进程名称为w3wp.exe。

当然，除了上面两点改进之外，IIS 6.0还有其他一些值得称道的地方，其中最重要的一点就是创建了一个新的HTTP监听器：HTTP协议栈（HTTP Protocol Stack，HTTP.SYS）。HTTP.SYS运行在Windows的内核模式（Kernel Mode）下，作为驱动程序而存在。它是Windows 2003的TCP/IP网络子系统的一部分，从结构上，它属于TCP之上的一个网络驱动程序。严格地说，HTTP.SYS已经不属于IIS的范畴了，所以HTTP.SYS的配置信息并不保存在IIS的元数据库（Metabase），而是定义在注册表中。HTTP.SYS的注册表项位于下面的路径中：HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/Services/HTTP。HTTP.SYS能够带来如下的好处：

* **持续监听：**由于HTTP.SYS是一个网络驱动程序，始终处于运行状态，对于用户的HTTP请求，能够及时作出反应；
* **更好的稳定性：**HTTP.SYS运行在操作系统内核模式下，并不执行任何用户代码，所以其本身不会受到Web应用、工作进程和IIS进程的影响；
* **内核模式下数据缓存：**如果某个资源被频繁请求，HTTP.SYS会把响应的内容进行缓存，缓存的内容可以直接响应后续的请求。由于这是基于内核模式的缓存，不存在内核模式和用户模式的切换，响应速度将得到极大的改进。

**图2**体现了IIS的结构和处理HTTP请求的流程。从中可以看出，与IIS 5.x不同，W3SVC从InetInfo.exe进程脱离出来（对于IIS6.0来说，InetInfo.exe基本上可以看作单纯的IIS管理进程），运行在另一个进程SvcHost.exe中。不过W3SVC的基本功能并没有发生变化，只是在功能的实现上作了相应的改进。与IIS 5.x一样，元数据库（Metabase）依然存在于InetInfo.exe进程中。



**图2 IIS 6与ASP.NET**

当HTTP.SYS监听到用户的HTTP请求后，将其分发给W3SVC。W3SVC解析出请求的URL，并根据从Metabase获取的URL与Web应用之间的映射关系得到目标应用，并进一步得到目标应用运行的应用程序池或者工作进程。如果工作进程不存在（尚未创建或者被回收），则为该请求创建新的工作进程，工作进程的这种创建方式被称为请求式创建。在工作进程的初始化过程中，相应的ISAPI.dll被加载，对于ASP.NET应用来说，被加载的ISAPI.dll为Aspnet\_ispai.dll。ASP.NET ISAPI再负责进行CLR的加载、AppDomain创建、Web Application的初始化等。

**IIS 7.0与ASP.NET**

IIS 7.0对请求的监听和分发机制上又进行了革新性的改进，主要体现在对于Windows进程激活服务（Windows Process Activation Service，WAS）的引入，将原来（IIS 6.0）W3SVC承载的部分功能分流给了WAS。具体来说，通过上面的介绍，我们知道对于IIS 6.0来说，W3SVC主要承载着三大功能：

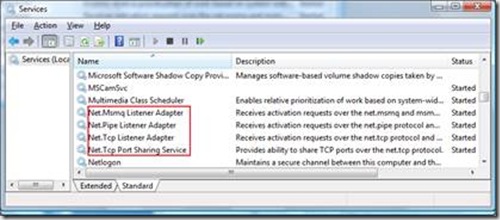
* **HTTP请求接收：**接收HTTP.SYS监听到的HTTP请求；
* **配置管理：**从元数据库（Metabase）中加载配置信息对相关组件进行配置；
* **进程管理：**创建、回收、监控工作进程。

在IIS 7.0，后两组功能被移入WAS中，接收HTTP请求的任务依然落在W3SVC头上。WAS的引入为IIS 7.0一项前所未有的特性：同时处理HTTP和非HTTP请求。在WAS中，通过一个重要的接口：监听器适配器接口（Listener Adapter Interface）抽象出不同协议监听器监听到的请求。至于IIS下的监听器，除了基于网络驱动的HTTP.SYS提供HTTP请求监听功能外，WCF提供了3种类型的监听器：TCP监听器、命名管道（Named Pipes）监听器和MSMQ监听器，分别提供了基于TCP、命名管道和MSMQ传输协议的监听功能。与此3种监听器相对的，是3种监听器适配器（Adapter）提供监听器与监听器适配器接口之间的适配。从这个意义上讲，IIS 7.0中的W3SVC更多地为HTTP.SYS起着监听适配器的功能。WCF提供的这3种监听器和监听适配器定义在程序集SMHost.exe中，你可以通过下面的目录找到该程序集：%windir%\Microsoft.NET\Framework\v3.0\Windows Communication Foundatio。

WCF提供的这3种监听器和监听适配器最终以Windows Service的形式体现，虽然它们定义在一个程序集中，我们依然通过服务工作管理器（SCM，Service Control Manager）对其进行单独的启动、终止和配置。SMHost.exe提供了4个重要的Windows Service：

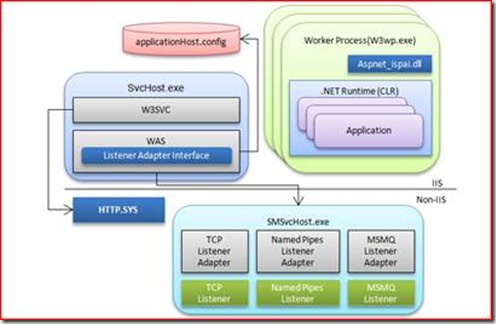
* **NetTcpPortSharing：**为WCF提供TCP端口共享，关于端口共享；
* **NetTcpActivator：**为WAS提供基于TCP的激活请求，包含TCP监听器和对应的监听适配器；
* **NetPipeActivator：**为WAS提供基于命名管道的激活请求，包含命名管道监听器和对应的监听适配器；
* **NetMsmqActivator：**为WAS提供基于MSMQ的激活请求，包含MSMQ监听器和对应的监听适配器。

**图3**为上述的4个Windows Service在服务控制管理器（SCM）中的呈现。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image006_2.jpg)

**图3 定义在SMHost.exe中的Windows Service**

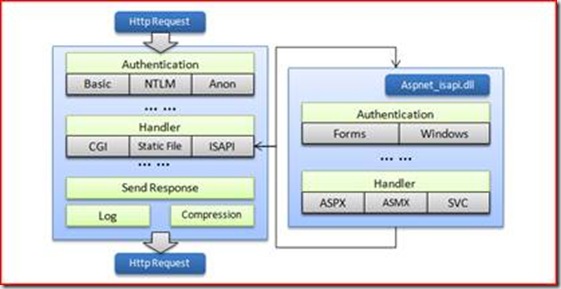
**图4**揭示了IIS 7.0的整体构架以及整个请求处理流程。无论是从W3SVC接收到的HTTP请求，还是通过WCF提供的监听适配器接收到的请求，最终都会传递到WAS。如果相应的工作进程（或者应用程序池）尚未创建，其创建之；否则将请求分发给对应的工作进程进行后续的处理。WAS在进行请求处理过程中，通过内置的配置管理模块加载相关的配置信息对相关的组建进行配置，与IIS 5.x和IIS 6.0基于Metabase的配置信息存储不同的是，IIS 7.0大都将配置信息存放于XML形式的配置文件中。基本的配置存放在applicationHost.cofig中。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image008_2.jpg)

**图4 IIS 7与ASP.NET**

**ASP.NET集成**

从上面对IIS 5.x和IIS 6.0的介绍中，我们不难发现这一点，IIS与ASP.NET是两个相互独立的管道（Pipeline），在各自管辖范围内，它们各自具有自己的一套机制对HTTP请求进行处理。两个管道通过ISAPI实现“联通”：IIS是第一道屏障，当对HTTP请求进行必要的前期处理（比如身份验证等）后，通过ISAPI将请求分发给ASP.NET管道。当ASP.NET在自身管道范围内完成对HTTP请求的处理后，处理后的结果再返回到IIS，IIS对其进行后期处理（比如日志记录、压缩等），最终生成HTTP响应（HTTP Response）。从另一个角度讲，IIS运行在非托管的环境中，而ASP.NET管道则是托管的，从这个意义上讲，ISAPI还是连接非托管环境和托管环境的纽带。**图5**反映了IIS 6.0与ASP.NET之间的桥接关系。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image012_2.jpg)

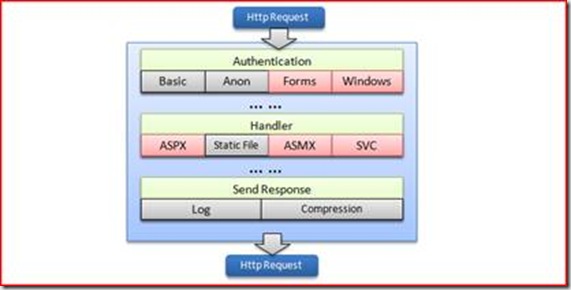
**图5 基于IIS 6.0与ASP.NET双管道设计**

IIS 5.x和IIS 6.0下把两个管道进行隔离至少带来了下面一些局限与不足：

* **相同操作的重复执行：**IIS与ASP.NET之间具有一些重复的操作，比如身份验证；
* **动态文件与静态文件处理的不一致：**因为只有基于ASP.NET的动态文件（比如.aspx、.asmx、.svc等等）的HTTP请求才能通过ASP.NET ISAPI进入ASP.NET管道，而对于一些静态文件（比如.html、.xml、.img等）的请求，则由IIS直接响应，那么ASP.NET管道中的一些功能将不能用于这些基于静态文件的请求，比如，我们希望通过Forms认证应用于基于图片文件的请求；
* **IIS难以扩展：**对于IIS的扩展基本上就体现在自定义ISAPI，但是对于大部分人来说，这不是一件容易的事情。因为ISAPI是基于Win32的非托管的API，并非一种面向应用的编程接口。通常我们希望的是诸如定义ASP.NET的HttpModule和HttpHandler一样，通过托管代码的方式来扩展IIS。

对于Windows平台下的IIS来讲，ASP.NET无疑是一等公民，它们之间不应该是“井水不犯河水”的关系，而应该是“你中有我，我中有你”的关系。为此，在IIS 7.0中，实现了两者的集成。对于集成模式下的IIS 7.0，我们获得如下的好处。

* 允许我们通过本地代码（Native Code）和托管代码（Managed Code）两种方式定义IIS Module，这些IIS Module注册到IIS中形成一个通用的请求处理管道。由这些IIS Module组成的这个管道能够处理所有的请求，不论请求基于怎样的资源类型。比如，可以将FormsAuthenticationModule提供的Forms认证应用到基于.aspx，CGI和静态文件的请求。
* 将ASP.NET提供的一些强大的功能应用到原来难以企及的地方，比如将ASP.NET的URL重写功能置于身份验证之前；
* 采用相同的方式去实现、配置、检测和支持一些服务器特性（Feature），比如Module、Handler映射、错误定制配置（Custom Error Configuration）等。



**图6 基于IIS 7.0与ASP.NET集成管道设计**

**图6**演示了在ASP.NET集成模式下，IIS整个请求处理管道的结构。我们可以看到，原来ASP.NET提供的托管组件可以直接应用在IIS管道中。

**ASP.NET管道**

以IIS 6.0为例，在工作进程w3wp.exe中，利用Aspnet\_ispai.dll加载.NET运行时（如果.NET运行时尚未加载）。IIS 6引入了应用程序池的概念，一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或者多个Web应用，每个Web应用映射到一个IIS虚拟目录。与IIS 5.x一样，每一个Web应用运行在各自的应用程序域中。

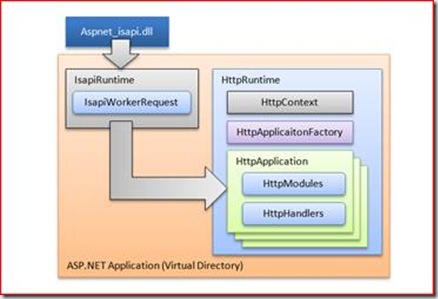
如果HTTP.SYS接收到的HTTP请求是对该Web应用的第一次访问，当成功加载了运行时后，会通过AppDomainFactory为该Web应用创建一个应用程序域（AppDomain）。随后，一个特殊的运行时IsapiRuntime被加载。IsapiRuntime定义在程序集System.Web中，对应的命名空间为System.Web.Hosting。IsapiRuntime会接管该HTTP请求。

IsapiRuntime会首先创建一个IsapiWorkerRequest对象，用于封装当前的HTTP请求，并将该IsapiWorkerRequest对象传递给ASP.NET运行时：HttpRuntime，从此时起，HTTP请求正式进入了ASP.NET管道。根据IsapiWorkerRequest对象，HttpRuntime会创建用于表示当前HTTP请求的上下文（Context）对象：HttpContext。

随着HttpContext被成功创建，HttpRuntime会利用HttpApplicationFactory创建新的或者获取现有的HttpApplication对象。实际上，ASP.NET维护着一个HttpApplication对象池，HttpApplicationFactory从池中选取可用的HttpApplication用户处理HTTP请求，处理完毕后将其释放到对象池中。HttpApplicationFactory负责处理当前的HTTP请求。

在HttpApplication初始化过程中，会根据配置文件加载并初始化相应的HttpModule对象。对于HttpApplication来说，在它处理HTTP请求的不同的阶段会触发不同的事件（Event），而HttpModule的意义在于通过注册HttpApplication的相应的事件，将所需的操作注入整个HTTP请求的处理流程。ASP.NET的很多功能，比如身份验证、授权、缓存等，都是通过相应的HttpModule实现的。

而最终完成对HTTP请求的处理实现在另一个重要的对象中：HttpHandler。对于不同的资源类型，具有不同的HttpHandler。比如.aspx页对应的HttpHandler为System.Web.UI.Page，WCF的.svc文件对应的HttpHandler为System.ServiceModel.Activation.HttpHandler。上面整个处理流程如**图7**所示。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image016_2.jpg)

**图7 ASP.NET 处理管道**

**HttpApplication**

HttpApplication是整个ASP.NET基础架构的核心，它负责处理分发给它的HTTP请求。由于一个HttpApplication对象在某个时刻只能处理一个请求，只有完成对某个请求的处理后，HttpApplication才能用于后续的请求的处理。所以，ASP.NET采用对象池的机制来创建或者获取HttpApplication对象。具体来讲，当第一个请求抵达的时候，ASP.NET会一次创建多个HttpApplication对象，并将其置于池中，选择其中一个对象来处理该请求。当处理完毕，HttpApplication不会被回收，而是释放到池中。对于后续的请求，空闲的HttpApplication对象会从池中取出，如果池中所有的HttpApplication对象都处于繁忙的状态，ASP.NET会创建新的HttpApplication对象。

HttpApplication处理请求的整个生命周期是一个相对复杂的过程，在该过程的不同阶段会触发相应的事件。我们可以注册相应的事件，将我们的处理逻辑注入到HttpApplication处理请求的某个阶段。我们接下来介绍的HttpModule就是通过HttpApplication事件注册的机制实现相应的功能的。**表1**按照实现的先后顺利列出了HttpApplication在处理每一个请求时触发的事件名称。

表1

|  |  |
| --- | --- |
| 名称 | 描述 |
| BeginRequest | HTTP管道开始处理请求时，会触发BeginRequest事件 |
| AuthenticateRequest，PostAuthenticateRequest | ASP.NET先后触发这两个事件,使安全模块对请求进行身份验证 |
| AuthorizeRequest，PostAuthorizeRequest | ASP.NET先后触发这两个事件,使安全模块对请求进程授权 |
| ResolveRequestCache，PostResolveRequestCache | ASP.NET先后触发这两个事件，以使缓存模块利用缓存的直接对请求直接进程响应（缓存模块可以将响应内容进程缓存，对于后续的请求，直接将缓存的内容返回，从而提高响应能力）。 |
| PostMapRequestHandler | 对于访问不同的资源类型，ASP.NET具有不同的HttpHandler对其进程处理。对于每个请求，ASP.NET会通过扩展名选择匹配相应的HttpHandler类型，成功匹配后，该实现被触发 |
| AcquireRequestState，PostAcquireRequestState | ASP.NET先后触发这两个事件，使状态管理模块获取基于当前请求相应的状态，比如SessionState |
| PreRequestHandlerExecute，PostRequestHandlerExecute | ASP.NET最终通过一请求资源类型相对应的HttpHandler实现对请求的处理，在实行HttpHandler前后，这两个实现被先后触发 |
| ReleaseRequestState，PostReleaseRequestState | ASP.NET先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态 |
| UpdateRequestCache，PostUpdateRequestCache | ASP.NET先后触发这两个事件，以使缓存模块将HttpHandler处理请求得到的相应保存到输出缓存中 |
| LogRequest，PostLogRequest | ASP.NET先后触发这两个事件为当前请求进程日志记录 |
| EndRequest | 整个请求处理完成后，EndRequest事件被触发 |

对于一个ASP.NET应用来说，HttpApplication派生于global.asax文件，我们可以通过创建global.asax文件对HttpApplication的请求处理行为进行定制。global.asax采用一种很直接的方式实现了这样的功能，这种方式既不是我们常用的方法重写（Method Overriding）或者事件注册，而是直接采用方法名匹配。在global.asax中，我们按照这样的方法命名规则进行事件注册：Application\_{Event Name}。比如Application\_BeginRequest方法用于处理HttpApplication的BeginRequest事件。如果通过VS创建一个global.asax文件，下面是默认的定义。

1: <%@ Application Language="C#" %>

2: <script runat="server">

3: void Application\_Start(object sender, EventArgs e) {}

4: void Application\_End(object sender, EventArgs e) {}

5: void Application\_Error(object sender, EventArgs e) {}

6: void Session\_Start(object sender, EventArgs e) {}

7: void Session\_End(object sender, EventArgs e) {}

8: </script>

**HttpModule**

ASP.NET为创建各种.NET Web应用提供了强大的平台，它拥有一个具有高度可扩展性的引擎，并且能够处理对于不同资源类型的请求。那么，是什么成就了ASP.NET的高可扩展性呢？ HttpModule功不可没。

从功能上讲，HttpModule之于ASP.NET，就好比ISAPI Filter之于IIS一样。IIS将接收到的请求分发给相应的ISAPI Extension之前，注册的ISAPI Filter会先截获该请求。ISAPI Filter可以获取甚至修改请求的内容，完成一些额外的功能。与之相似地，当请求转入ASP.NET管道后，最终负责处理该请求的是与请求资源类型相匹配的HttpHandler对象，但是在Handler正式工作之前，ASP.NET会先加载并初始化所有配置的HttpModule对象。HttpModule在初始化的过程中，会将一些功能注册到HttpApplication相应的事件中，那么在HttpApplication整个请求处理生命周期中的某个阶段，相应的事件会被触发，通过HttpModule注册的事件处理程序也得以执行。

所有的HttpModule都实现了IHttpModule接口，下面是IHttpModule的定义。其中Init方法用于实现HttpModule自身的初始化，该方法接受一个HttpApplication对象，有了这个对象，事件注册就很容易了。

1: public interface IHttpModule

2: {

3: void Dispose();

4: void Init(HttpApplication context);

5: }

ASP.NET提供的很多基础构件（Infrastructure）功能都是通过相应的HttpModule实现的，下面类列出了一些典型的HttpModule:

* **OutputCacheModule**：实现了输出缓存（Output Caching）的功能；
* **SessionStateModule**：在无状态的HTTP协议上实现了基于会话（Session）的状态；
* **WindowsAuthenticationModule + FormsAuthenticationModule + PassportAuthentication- Module**：实现了3种典型的身份认证方式：Windows认证、Forms认证和Passport认证；
* **UrlAuthorizationModule + FileAuthorizationModule**：实现了基于Uri和文件ACL（Access Control List）的授权。

而另外一个重要的HttpModule与WCF相关，那么就是System.ServiceModel. Activation.HttpModule。HttpModule定义在System.ServiceModel程序集中，在默认的情况下，HttpModule完成了基于IIS的寄宿工作。

除了这些系统定义的HttpModule之外，我们还可以自定义HttpMoudle。通过Web.config，我们可以很容易地将其注册到我们的Web应用中。

**HttpHandler**

如果说HttpModule相当于IIS的ISAPI Filter的话，我们可以说HttpHandler则相当于IIS的ISAPI Extension，HttpHandler在ASP.NET中扮演请求的最终处理者的角色。对于不同资源类型的请求，ASP.NET会加载不同的Handler来处理，也就是说.aspx page与.asmx web service对应的Handler是不同的。

所有的HttpHandler都实现了接口IHttpHandler。下面是IHttpHandler的定义，方法ProcessRequest提供了处理请求的实现。

1: public interface IHttpHandler

2: {

3: void ProcessRequest(HttpContext context);

4: bool IsReusable { get; }

5: }

对于某些HttpHandler，具有一个与之相关的HttpHandlerFactory，用于创建或者获取相应的HttpHandler。HttpHandlerFactory实现接口IHttpHandlerFactory，方法GetHandler用于创建新的HttpHandler，或者获取已经存在的HttpHandler。

1: public interface IHttpHandlerFactory

2: {

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

4: void ReleaseHandler(IHttpHandler handler);

5: }

HttpHandler和HttpHandlerFactory的类型都可以通过相同的方式配置到Web.config中。下面一段配置包含对3种典型的资源类型的HttpHandler配置：.aspx,.asmx和.svc。可以看到基于WCF Service的HttpHandler类型为：System.ServiceModel.Activation.HttpHandler。

1: <?xml version="1.0" encoding="utf-8" ?>

2: <configuration>

3: <system.web>

4: <httpHandlers>

5: <add path="\*.svc" verb="\*" type="System.ServiceModel.Activation.HttpHandler, System.ServiceModel, Version=3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" validate="false"/>

6: <add path="\*.aspx" verb="\*" type="System.Web.UI.PageHandlerFactory" validate="True"/>

7: <add path="\*.asmx" verb="\*" type="System.Web.Services.Protocols.WebServiceHandlerFactory, System.Web.Services, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" validate="False"/>

8: </httpHandlers>

9: </system.web>

10: </configuration>

作者：蒋金楠