**ASP.NET的处理模型**

**相关术语**

1. **HTTP.SYS**

-> 运行在 Kernel Model 中的一个组件 ,它负责侦听（Listen）来自于外部的HTTP请求,根据请求的URL将其分发给相应的应用程序池

-> HTTP监听器，是Windows TCP/IP网络子程序的一部分，用于持续监听HTTP请求

-> 为了提供更好的性能，HTTP.SYS内部建立了一个缓冲区， 静态的内容现在被缓存于内核模式下，将最近的HTTP请求处理结果保存起来，这使服务响应速度更快

-> 所有的服务请求会在HTTP.SYS里暂存入队列，而不是由服务程序本身来处理，这样，即使服务程序重启，尚未被处理的请求也不会丢失了

-> 只要是通过HTTP.SYS管理的端口（基本包括了那些著名的端口，比如80）都支持端口重用

1. **Application Pool 应用程序池**

-> 应用程序池是将一个或多个应用程序链接到一个或多个工作进程集合的配置

-> 每个应用程序池都对应着一个工作者进程

-> 应用程序池的概念出现在 IIS 6.0 当中，而IIS 5.0中所使用的还是应用程序域 Application Domain 的处理机制

-> 一个资源文件请求至服务器时，HTTP.SYS 会根据IIS中的 Metabase 查看基于该Request的Resource属于哪个Application Pool，如果不存在则创建之

1. **Application Domain 应用程序域**

-> 一个应用程序在其中执行的独立环境，为执行托管代码提供隔离、卸载和安全边界

-> 一个工作者进程中承载着一个或多个的应用程序域，这由我们在IIS上在部署网站的时候的配置选择有关

-> 当一个请求到达 ISAPI Extension 进行初始化CLR、构建.NETFramework Runtime时候，这个过程其实就是在为WEB应用程序构建一个应用程序域

1. **Worker Process 工作者进程**

-> 运行在一个非托管环境下的进程

-> 工作者进程在不同的IIS版本所对应的文件是不同的：IIS 6.0 : w3wp.exe / IIS 5.0 : aspnet\_wp.exe

-> 一个应用程序池对应一个工作者进程，当一个应用程序池在创建的时候，WAS Web Administrative service 会根据IIS在 Metabase 中所维护的 Application Pool 和 Worker Process 的Mapping，找到所对应的Worker Process并监听它，如果Woker Process不存在，则创建之

-> Worker Porcess 会分析当前请求资源文件的后缀并根据在该模块中所对应注册的ISAPI Extension Mapping 找到ISAPI Extension（以\*.aspx请求资源为例，所指定的ISAPI Extension 默认为 aspnet\_isapi.dll) ，并通过 ISAPI Extension 去初始化CLR，构建 .NETFramework Runtime

1. **WAS：Web Administrative Service**

-> 这是一个监控程序，它一方面可以存取放在InetInfo元数据库 Metabase 中的各种信息，另一方面也负责监控应用程序池中的工作者进程的工作状态况，必要时它会关闭一个老的工作者进程并创建一个新的取而代之

1. **ISAPI：Internet Server Application Programming Interface 英特网服务应用程序编程接口**

-> 是运行在非托管环境中的一组API，它是一种用来开发扩展IIS程序的强有力的方法，虽然ISAPI扩展绝不仅仅局限于IIS的开发，但是实际上它用的最广泛的还是开发一些与IIS合作的程序

1. **ISAPI Extension**

-> 是实现了ISAPI的拓展，对于ASP.NET来说，所使用的 ISAPI Extension 则为 aspnet\_isapi.dll

-> 在 Worker Process 加载的过程当中，会通过 aspnet\_isapi.dl 去初始化CLR，加载 .NETFramework Runtime

-> aspnet\_isapi.dl 是承载在非托管环境当中的，而.NET中的对象则运行在托管环境之上(CLR)，作为它们之间沟通的桥梁就是一个非托管指针EPC Execution Control Block

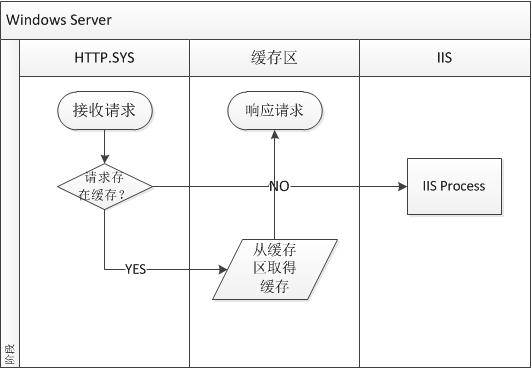
-> aspnet\_isapi.dl 在初始化CLR的过程当中是异步调用的，那么为了解决能够在调用ISAPI Extension 的过程当中也能够获取 .NETFramework中所返回的 Response，也是通过一个非托管指针EPC Execution Control Block 来实现的

**ASP.NET Process Model**

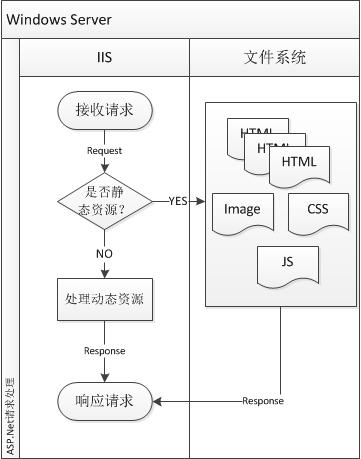
-> [缩略图](ASP.NET PROCESS MODEL.png)

-> 详解

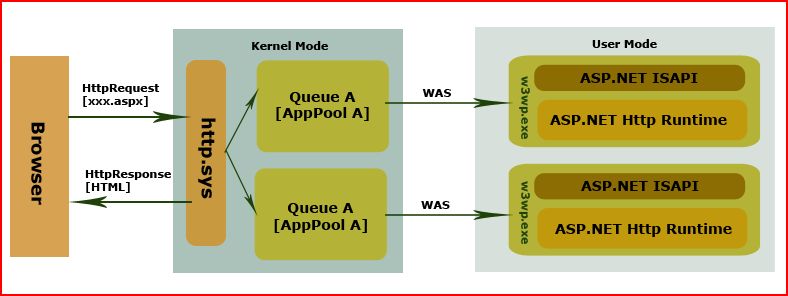
1. 以IIS6.0+而言，监听HTTP请求的工作被分发至一个运行在Kernel Module的组件HTTP.SYS上了，IIS会把自己的WebApplication的目录注册到HTTP.SYS的虚拟路径上(只是注册了静态资源的路径，不包含动态资源：\*.aspx、\*.ashx、…)，实际上是告诉HTTP.SYS当前所请求的静态资源URL的可访问性，举个例子，当我们浏览一个静态资源页面(\*.html)出现404的时候，就是通过它来决定的。那么当HTTP.SYS捕获到一个可访问的URL的HTTP请求的时候，它的首要工作是先分析当前Resource Request的IP地址和端口号，紧接着它会在自己内部所建立的一个缓冲区查找该Resource Request是否在最近被请求过，如果有的话则直接抓取所缓存的数据并返回Response给Client



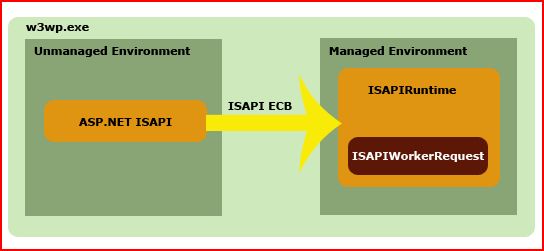
1. 如果没有的话则又会进入一次判断，判断此次Resource Request是否是一个动态资源（\*.aspx、\*.ashx…），如果不是的话（\*.html、\*.jpg…）则直接在IIS为HTTP.SYS所Host的虚拟路径下找到资源文件并返回Response给Client

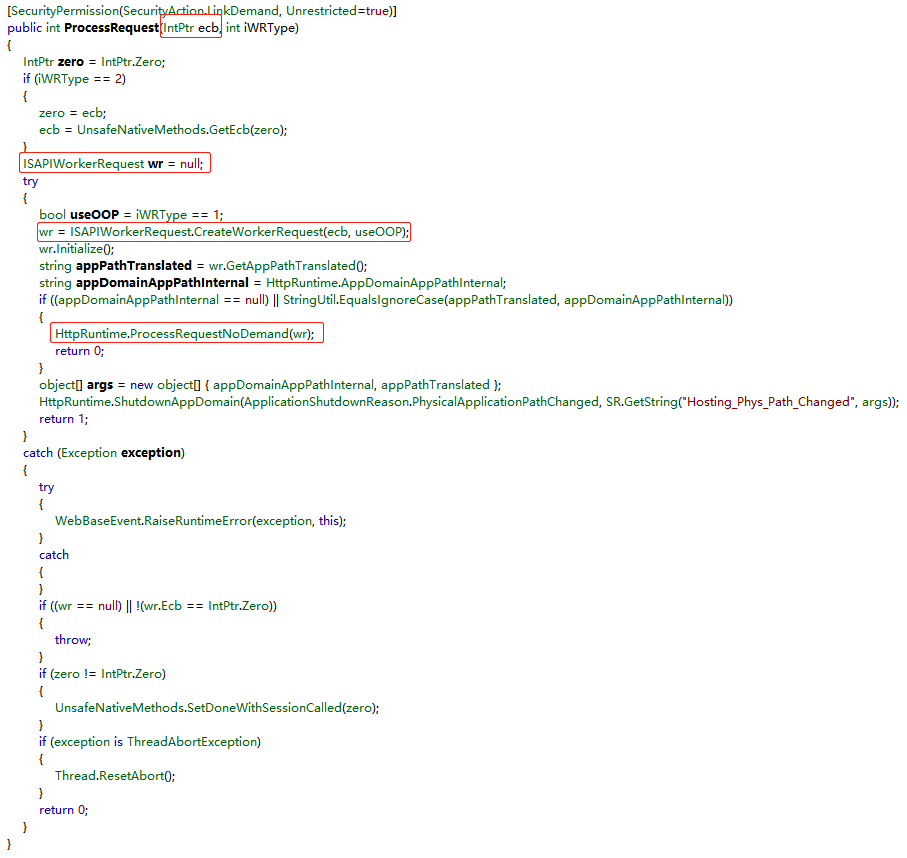


1. 那么当Resource Request为一个动态网页文件的话则根据IIS在Meta Base中所维护的Application Pool和Resource的Mapping找到对应的Application Pool，如果不存在则创建之，否则把Request加入到一个Application Pool Queue中，至此，一个Application Pool已经创建完毕。在上一节中我们介绍了WAS，WAS其主要职责就是根据IIS在Meta Base中所维护的Application Pool和Worker Porcess之间的Mapping去监听Worker Porcess的工作状态，如果它不存在，则创建之
2. 在Worker Process被创建之后，它会根据IIS在Meta Base中所维护的Resource资源后缀和ISAPI Extention的Mapping，找到并加载之，就ASP.NET而言，其对应的ISAPI Extention则为aspnet\_isapi.dll



1. aspnet\_isapi.dll是一个运行在Unmanaged Environment中的一个组件，Worker Porcess通过它经过一系列的COM级别的调用（获取Server Variable数据、通过Post Method回传Server数据、…），最终会落定会在初始化CLR身上。在CLR初始化的过程当中，会加载两个重要的dll，其一是AppManagerAppDomainFactory，通过它为当前Web Application构建出一个Application Domain，其二是System.Web.Hosting.ISAPIRuntime，这需要我们各位的关注，在ISAPIRuntime中有一个ProcessRequest函数，它是ASP.NET的一个入口，前面我们说到aspnet\_isapi.dll是一个运行在Unmanaged Environment中的，也就是说在此刻Http Resource Request其实还在Unmanaged Environment中，那么aspnet\_isapi.dll是怎么把Http Resource Request带到运行在Managed Environment中的呢？是通过一个名为EPC的C++指针，这个EPC也是上面说的ProcessRequest入口函数中的一个Siganature，通过EPC，aspnet\_isapi.dll成功地把刚刚还运行在Unmanaged Environment中的Http Resource Request带领到Managed Environment中（aspnet\_isapi.dll对ISAPIRuntime的调用是基于异步执行的，并且会把请求放入至队列中进行处理）





1. 通过上面的图片我们也可以看到ProcessRequest入口函数的实现代码，其核心已标明出来，那么我们跟着这个流程继续走（ProcessRequestNoDemand），代码又会落定在ProcessRequestInternal之中，在个函数中主要干了两件非常重要的事情
2. 第一件则是通过EPC所构建的HttpWorkerRequest去创建了HttpContext



1. 第二件事则是调用GetApplicationInstance，关于这个函数干了很多事，代码图就不贴出来了，关于[Process Model缩略图](ASP.NET Process Module.png)已表明具体哪个函数的调用，下面只做简单讲解

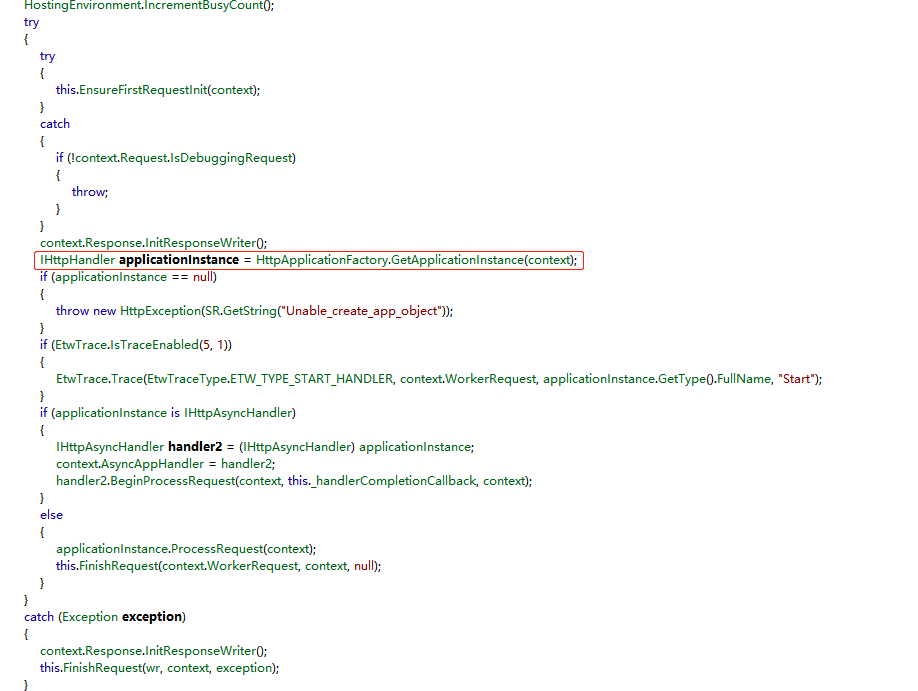
-> 判断Application Pool是否存在未被使用的HttpApplication，如果不存在则通过反射的方式创建一个基于Global.asax的HttpApplication

-> 通过EnsureAppStartCalled函数去触发HttpApplication的Application\_Start Event Handler

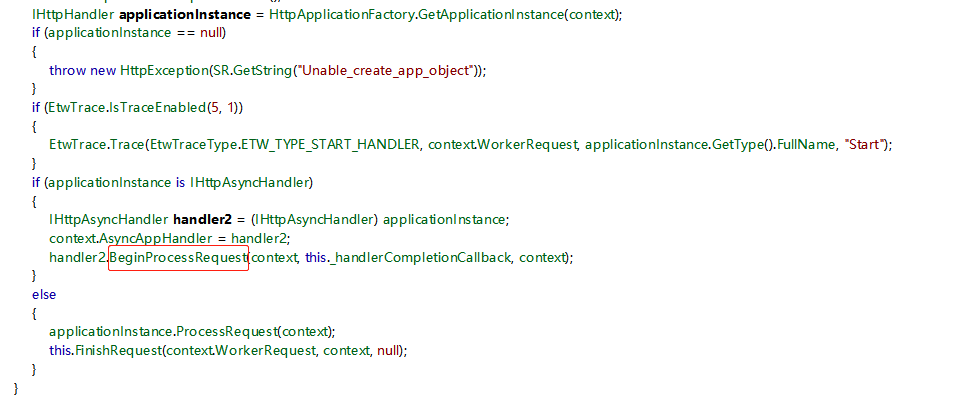
-> 通过HookupEventHandlersForApplicationAndModules完成Global.asax中的Event Handler的注册

-> 通过配置文件(默认machine.config，也可以自己定义的web.config中的<modules>的节点)中的映射关系去创建HttpModule，并循环执行每个HttpModule的init函数，完成所有HttpModule内置Event Handler的注册

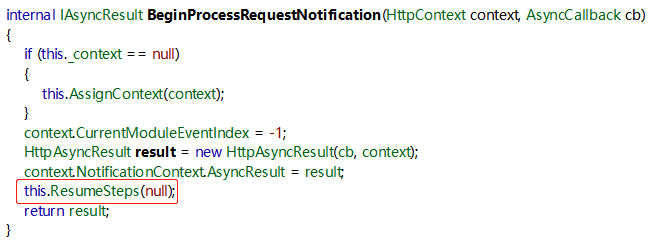
-> 通过BuildStep最终为HttpApplication完成19个管道事件的注册

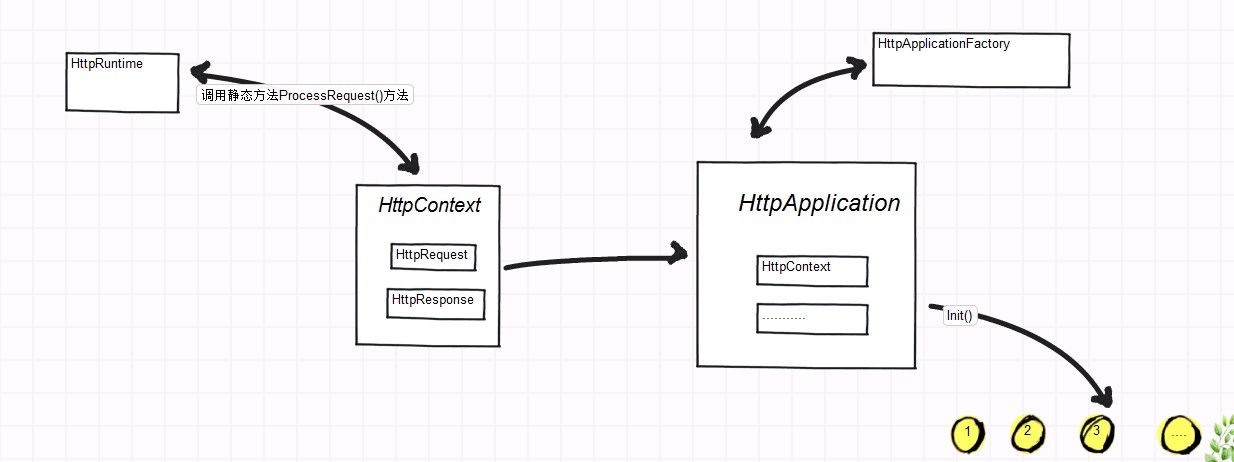


1. 在HttpApplication、HttpModules、HttpContext初始化完成之后，调用会降临到BeginProcessRequest身上



1. 那么在BeginProcessRequest的内部，会通过HttpApplication去调用ResumeSteps，至此开始进入Http PipeLine，依次执行管道中的事件，并且每一个事件处理函数都把HttpApplication作为Sender传入





1. 在Http PipeLine中，当执行到第8个管道的时候，会根据配置文件中的<handlers>节点中的映射关系，构造一个属于当前Resource Request的HttpHandler，那么在第11至12之间，会调用这个HttpHandler的ProcessRequest函数去进行后续的逻辑处理

