了解 ASP.NET 底层架构

进入底层
这篇文章以非常底层的视角讲述了 Web 请求（request）在 ASP.NET 框架中是如何流转的,从 Web 服务器,通过 ISAPI 直到请求处理器(handler)和你的代码.看看在幕后都发生了些什么,不要再把 ASP.NET 看成一个黑盒了.

ASP.NET 是一个非常强大的构建 Web 应用的平台,它提供了极大的灵活性和能力以致于可以用它来构建所有类型的 Web 应用.绝大多数的人只熟悉高层的框架如 WebForms 和 WebServices-这些都在 ASP.NET 层次结构在最高层.在这篇文章中我将会讨论 ASP.NET 的底层机制并解释请求(request)是怎么从 Web 服务器传送到 ASP.NET 运行时然后如何通过 ASP.NET 管道来处理请求.

对我而言了解平台的内幕通常会带来满足感和舒适感,深入了解也能帮助我写出更好的应用.知道可以使用哪些工具以及他们是怎样作为整个复杂框架的一部分来互相配合的可以更容易地找出最好的解决方案,更重要的是可以在出现问题时更好的解决它们.这篇文章的目标是从系统级别了解 ASP.NET 并帮助理解请求(request)是如何在 ASP.NET 的处理管道中流转的.同样,我们会了解核心引擎和 Web 请求如何在那里结束.这些信息大部分并不是你在日常工作时必须了解的,但是它对于理解 ASP.NET 架构如何把请求路由到你的代码(通常是非常高层的)中是非常有益的.

不管怎么样,ASP.NET 从更低的层次上提供了更多的灵活性.HTTP 运行时和请求管道在构建 WebForms 和 WebServices 上提供了同样的能力-它们事实上都是建立在.NET 托管代码上的.而且所有这些同样的功能对你也是可用的,你可用决定你是否需要建立一个比 WebForms 稍低一点层次的定制的平台.

WebForms 显然是最简单的构建绝大多数 Web 接口的方法,不过如果你是在建立自定义的内容处理器(handler),或者有在处理输入输出内容上有特殊的要求,或者你需要为另外的应用建立一个定制的应用程序服务接口,使用这些更低级的处理器(handler)或者模块(module)能提供更好的性能并能对实际请求处理提供更多的控制.在 WebForms 和 WebServices 这些高层实现提供它们那些能力的同时,它们也对请求增加了一些额外负担,这些都是在更底层可以避免的.

ASP.NET 是什么

让我们以一个简单的定义开始:什么是 ASP.NET?我喜欢这样定义 ASP.NET:

> **ASP.NET 是一个复杂的使用托管代码来从头到尾处理 Web 请求的引擎.**
> **它并不只是 WebForms 和 WebServies...**

ASP.NET 是一个请求处理引擎.它接收一个发送过来的请求,把它传给内部的管道直到终点,作为一个开发人员的你可以在这里附加一些代码来处理请求.这个引擎是和 HTTP/Web 服务器完全分隔的.事实上,HTTP 运行时是一个组件,使你可以摆脱 IIS 或者任何其他的服务器程序,将你自己的程序寄宿在内.例如,你可以将 ASP.NET 运行时寄宿在一个 Windows form 程序中(查看

http://www.west-wind.com/presentations/aspnetruntime/aspnetruntime.asp 可以得到更加详细的信息)

运行时提供了一个复杂但同时非常优雅的在管道中路由请求的机制.其中有很多相关的对象,大多数都是可扩展的(通过继承或者事件接口),在几乎所有的处理流程上都是如此.所以这个框架具有高度可扩展性.通过这个机制,挂接到非常底层的接口(比如缓存,认证和授权)都变得可能了.你甚至可以在预处理或者处理后过滤内容,也可以简单的将符合特殊标记的请求直接路由你的代码或者另一个 URL 上.存在着许多不同的方法来完成同一件事,但是所有这些方法都是可以简单直接地实现的,同时还提供了灵活性,可以得到最好的性能和开发的简单性.

整个 ASP.NET 引擎是完全建立在托管代码上的,所有的扩展功能也是通过托管代码扩展来提供的

整个 ASP.NET 引擎是完全建立在托管代码上的,所有的扩展功能也是通过托管代码扩展来提供的.这是对.NET 框架具有构建复杂而且高效的框架的能力的最好的证明.ASP.NET 最令人印象深刻的地方是深思熟虑的设计,使得框架非常的容易使用,又能提供挂接到请求处理的几乎所有部分的能力.

通过 ASP.NET 你可以从事从前属于 ISAPI 扩展和 IIS 过滤器领域的任务-有一些限制,但是比起 ASP 来说是好多了.ISAPI 是一个底层的 Win32 风格的 API,有着非常粗劣的接口而且难以用来开发复杂的程序.因为 ISAPI 非常底层,所以它非常的快,但是对于应用级的开发者来说是十分难以管理的.所以,ISAPI 通常用来提供桥接的接口,来对其他应用或者平台进行转交.但是这并不意味者 ISAPI 将消亡.事实上,ASP.NET 在微软的平台上就是通过 ISAPI 扩展来和 IIS 进行交互的,这个扩展寄宿着.NET 运行时和 ASP.NET 运行时.ISAPI 提供了核心的接口,ASP.NET 使用非托管的 ISAPI 代码通过这个接口来从 Web 服务器获取请求,并发送响应回客户端.ISAPI 提供的内容可以通过通用对象(例如 HttpRequest 和 HttpResponse)来获取,这些对象通过一个定义良好并有很好访问性的接口来暴露非托管数据.

从浏览器到 ASP.NET

让我们从一个典型的 ASP.NET Web 请求的生命周期的起点开始.当用户输入一个 URL,点击了一个超链接或者提交了一个 HTML 表单(form)(一个 POST 请求,相对于前两者在一般意义上都是 GET 请求).或者一个客户端程序可能调用了一个基于 ASP.NET 的 WebService(同样由 ASP.NET 来处理).在 Web 服务器端,IIS5 或 6,获得这个请求.在最底层,ASP.NET 和 IIS 通过 ISAPI 扩展进行交互.在 ASP.NET 环境中这个请求通常被路由到一个扩展名为.aspx 的页面上,但是这个流程是怎么工作的完全依赖于处理特定扩展名的 HTTP Handler 是怎么实现的.在 IIS 中.aspx 通过'应用程序扩展'(又称为脚本映射)被映射到 ASP.NET 的 ISAPI 扩展 DLL-aspnet_isapi.dll.每一个请求都需要通过一个被注册到 aspnet_isapi.dll 的扩展名来触发 ASP.NET(来处理这个请求).

依赖于扩展名 ASP.NET 将请求路由到一个合适的处理器(handler)上,这个处理器负责获取这个请求.例如,WebService 的.asmx 扩展名不会将请求路由到磁盘上的一个页面,而是一个由特殊属性(Attribute)标记为 WebService 的类上.许多其他处理器和 ASP.NET 一起被安装,当然你也可以自定义处理器.所有这些 HttpHandler 在 IIS 中被配置为指向 ASP.NET ISAPI 扩展,并在 web.config(译著:ASP.NET 中自带的 handler 是在 machine.config 中配置的,当然可以在 web.config 中覆盖配置)被配置来将请求路由到指定的 HTTP Handler 上.每个 handler 都

是一个处理特殊扩展的.NET 类,可以从一个简单的只包含几行代码的 Hello World 类,到非常复杂的 handler 如 ASP.NET 的页面或者 WebService 的 handler.当前,只要了解 ASP.NET 的映射机制是使用扩展名来从 ISAPI 接收请求并将其路由到处理这个请求的 handler 上就可以了.

对在 IIS 中自定义 Web 请求处理来说,ISAPI 是第一个也是最高效的入口

ISAPI 连接

ISAPI 是底层的非托管 Win32 API.ISAPI 定义的接口非常简单并且是为性能做了优化的.它们是非常底层的-处理指针和函数指针表来进行回调-但是它们提供了最底层和面向效率的接口,使开发者和工具提供商可以用它来挂接到 IIS 上.因为 ISAPI 非常底层所以它并不适合来开发应用级的代码,而且 ISAPI 倾向于主要被用于桥接接口,向上层工具提供应用服务器类型的功能.例如,ASP 和 ASP.NET 都是建立在 ISAPI 上的,Cold Fusion,运行在 IIS 上的多数 Perl,PHP 以及 JSP 实现,很多第三方解决方案(如我的 Wisual FoxPro 的 Web 连接框架)都是如此.ISAPI 是一个杰出的工具,可以为上层应用提供高效的管道接口,这样上层应用可以抽象出 ISAPI 提供的信息.在 ASP 和 ASP.NET 中,将 ISAPI 接口提供的信息抽象成了类型 Request 和 Response 这样的对象,通过它们来读取 ISAPI 请求中对应的信息.将 ISAPI 想像成管道.对 ASP.NET 来说,ISAPI dll 是非常的”瘦”的,只是作为一个路由机制来将原始的请求转发到 ASP.NET 运行时.所有那些沉重的负担和处理,甚至请求线程的管理都发生在 ASP.NET 引擎内部和你的代码中.

作为协议,ISAPI 同时支持 ISAPI 扩展和 ISAPI 过滤器(Filter).扩展是一个请求处理接口,提供了处理 Web 服务器的输入输出的逻辑-它本质上是一个处理(事物?)接口.ASP 和 ASP.NET 都被实现为 ISAPI 扩展.ISAPI 过滤器是挂接接口,提供了查看进入 IIS 的每一个请求的能力,并能修改请求的内容或者改变功能型的行为,例如认证等.顺便提一下,ASP.NET 通过了两种概念映射了类似 ISAPI 的功能:Http Handler 类似扩展,Http Module 类似过滤器.我们将在后面详细讨论它们.

ISAPI 是开始一个 ASP.NET 请求的最初的入口.ASP.NET 映射了好几个扩展名到它的 ISAPI 扩展,此扩展位于.NET 框架的目录下:

<.NET FrameworkDir>\aspnet_isapi.dll

你可以在 IIS 服务管理界面上看到这些映射,如图 1.查看网站根目录的属性中的主目录配置页,然后查看配置|映射.

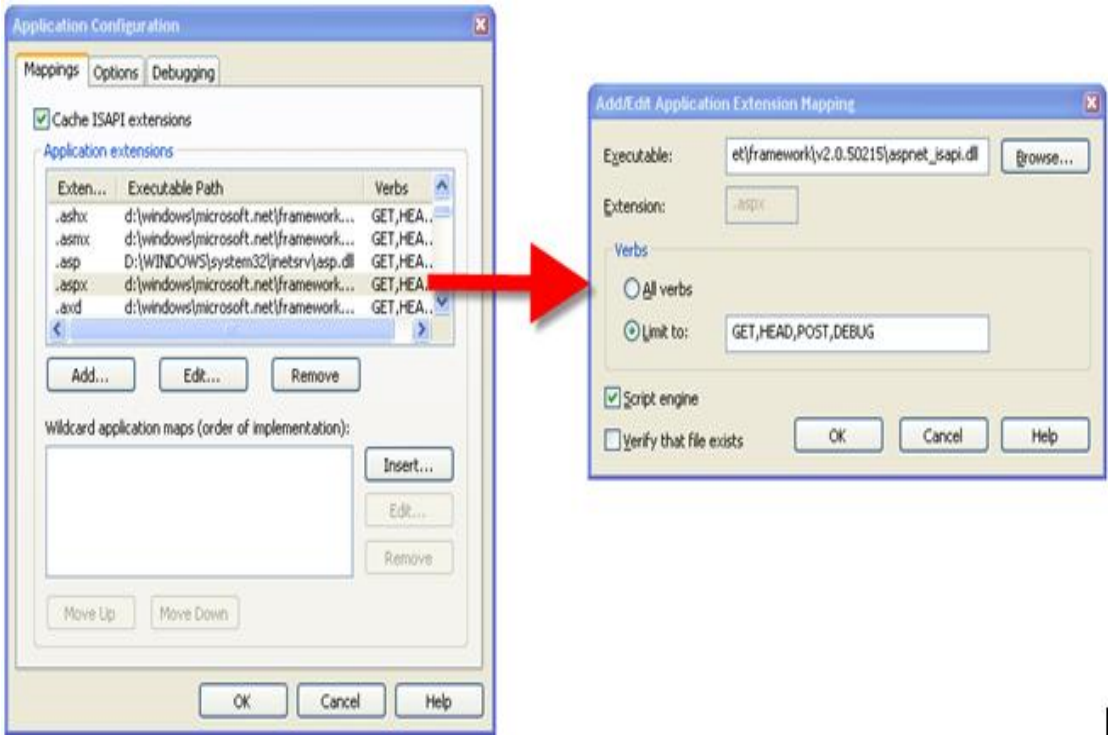


图 1:IIS 映射了多种扩展名如.ASPX 到 ASP.NET 的 ISAPI 扩展.通过这个机制请求会在 Web 服务器这一层被路由到 ASP.NET 的处理管道.

由于.NET 需要它们中的一部分,你不应该设置手动这些扩展名.使用 aspnet_regiis.exe 这个工具来确保所有的映射都被正确的设置了:

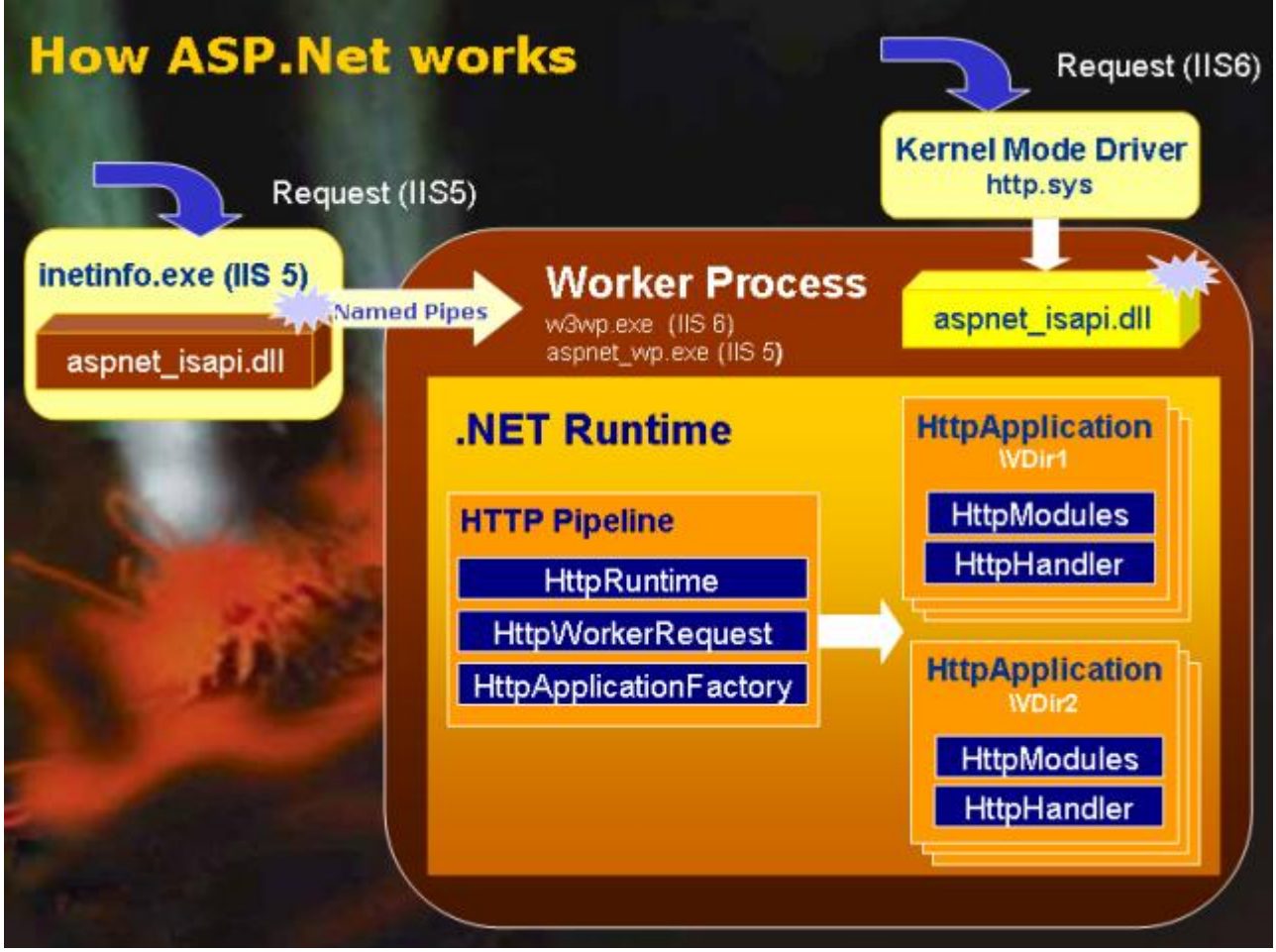
```
cd <.NetFrameworkDirectory>
aspnet_regiis – i
```

这个命令将为整个 Web 站点注册特定版本的 ASP.NET 运行时,包括脚本 (扩展名) 映射和客户端脚本库(包括进行控件验证的代码等).注意它注册的是<.NetFrameworkDirectory>中安装的特定版本的 CLR(如 1.1,2.0).aspnet_regiis 的选项令您可以对不同的虚拟目录进行配置.每个版本的.NET 框架都有自己不同版本的 aspnet_regiis 工具,你需要运行对应版本的 aspnet_regiis 来为 web 站点或者虚拟目录来配置指定版本的.NET 框架.从 ASP.NET2.0 开始提供了 ASP.NET 配置页面,可以通过这个页面在 IIS 管理控制台来交互的配置.NET 版本.

> **IIS6 通配符应用程序映射**
>
> **如果你有一个 ASP.NET 应用程序需要处理虚拟目录的(或者是整个 Web 站点,如果配置为根目录的话)每一个请求,IIS6 引入了新的称为通配符应用程序映射的概念.一个映射到通配符的 ISAPI 扩展在每个请求到来时都会被触发,而不管扩增名是什么.这意味着每个页面都会通过这个扩展来处理.这是一个强大的功能,你可以用这个机制来创建虚拟 Url 和不使用文件名的 unix 风格的 URL.然而,使用这个设置的时候要注意,因为它会把所有的东西都传给你的应用,包括静态 htm 文件,图片,样式表等等.**

IIS 5 和 6 以不同的方式工作

当一个请求来到时,IIS 检查脚本映射(扩展名映射)然后把请求路由到 aspnet_isapi.dll.这个 DLL 的操作和请求如何进入 ASP.NET 运行时在 IIS5 和 6 中是不同的.图 2 显示了这个流程的一个粗略概览.



在 IIS5 中,aspnet_isapi.dll 直接寄宿在 inetinfo.exe 进程中,如果你设置了 Web 站点或虚拟目录的隔离度为中或高,则会寄宿在 IIS 单独的(被隔离的)工作进程中.当第一个 ASP.NET 请求来到,DLL(aspnet_isapi.dll)会开始另一个新进程 aspnet_wp.exe 并将请求路由到这个进程中来进行处理.这个进程依次加载并寄宿.NET 运行时.每个转发到 ISAPI DLL 的请求都会通过命名管道调用被路由到这个进程来.

图 2-从较高层次来看请求从 IIS 到 ASP.NET 运行时,并通过请求处理管道的流程.IIS5 和 IIS6 通过不同的方式与 ASP.NET 交互,但是一旦请求来到 ASP.NET 管道,整个处理流程就是一样的了.

不同于以前版本的服务器,IIS6 为 ASP.NET 做了全面的优化

IIS6-应用程序池万岁

IIS6 对处理模型做了意义重大的改变,IIS 不再直接寄宿象 ISAPI 扩展这样的外部可执行代码.IIS 总是创建一个独立的工作线程-一个应用程序池-所有的处理都发生在这个进程中,包括 ISAPI dll 的执行.应用程序池是 IIS6 的一个很大的改进,因为它允许对指定线程中将会执行什么代码进行非常细粒度的控制.应用程序池可以在每个虚拟路径上或者整个 Web 站点上进行配置,这样你可以将每个 Web 应用隔离到它们自己的进程中,这样每个应用都将和其他运行在同一台机器上的 Web 应用完全隔离.如果一个进程崩溃了,不会影响到其他进程(至少在 Web 处理的观点上来看是如此).

不止如此,应用程序池还是高度可配置的.你可以通过设置池的执行扮演级别(execution impersonation level)来配置它们的运行安全环境,这使你可以定制赋予一个 Web 应用的权限(同样,粒度非常的细).对于 ASP.NET 的一个大的改进是,应用程序池覆盖了在 machine.config 文件中大部分的 ProcessModel 节的设置.这一节的设置在 IIS5 中非常的难以管理,因为这些设置是全局的而且不能在应用程序的 web.config 文件中被覆盖.当运行 IIS6 是,ProcessModel 相关的设置大部分都被忽略了,取而代之的是从应用程序池中读取.注意这里说的是大部分-有些设置,如线程池的大小还有 IO 线程的设置还是从 machine.config 中读取,因为它们在线程池的设置中没有对应项.

因为应用程序池是外部的可执行程序,这些可执行程序可以很容易的被监控和管理.IIS6 提供了一系列的进行系统状况检查,重启和超时的选项,可以很方便的用来检查甚至在许多情况下可以修正程序的问题.最后 IIS6 的应用程序池并不像 IIS5 的隔离模式那样依赖于 COM+,这样做一来可以提高性能,二来提高了稳定性(特别对某些内部需要调用 COM 组件的应用来说)

尽管 IIS6 的应用程序池是单独的 EXE,但是它们对 HTTP 操作进行了高度的优化,它们直接和内核模式下的 HTTP.SYS 驱动程序进行通讯.收到的请求被直接路由给适当的应用程序池.InetInfo 基本上只是一个管理程序和一个配置服务程序-大部分的交互实际上是直接在 HTTP.SYS 和应用程序池之间发生,所有这些使 IIS6 成为了比 IIS5 更加的稳定和高效的环境.特别对静态内容和 ASP.NET 程序来说这是千真万确的.

一个 IIS6 应用程序池对于 ASP.NET 有着天生的认识,ASP.NET 可以在底层的 API 上和它进行交互,这允许直接访问 HTTP 缓存 API,这样做可以将 ASP.NET 级别的缓存直接下发到 Web 服务器.

在 IIS6 中,ISAPI 扩展在应用程序池的工作进程中运行. .NET 运行时也在同一个进程中运行,所以 ISAPI 扩展和.NET 运行时的通讯是发生在进程内的,这样做相比 IIS5 使用的命名管道有着天生的性能优势.虽然 IIS 的寄宿模型有着非常大的区别,进入托管代码的接口却异常的相似-只有路由消息的过程有一点区别.

ISAPIRuntime.ProcessRequest()
函数是进入 ASP.NET 的第一站

进入.NET 运行时

进入.NET 运行时的真正的入口发生在一些没有被文档记载的类和接口中(译著:当然,你可以用 Reflector 来查看 J).除了微软,很少人知道这些接口,微软的家伙们也并不热衷于谈论这些细节,他们认为这些实现细节对于使用 ASP.NET 开发应用的开发人员并没有什么用处.
工作进程(IIS5 中是 ASPNET_WP.EXE,IIS6 中是 W3WP.EXE)寄宿.NET 运行时和 ISAPI DLL,它(工作进程)通过调用 COM 对象的一个小的非托管接口最终将调用发送到 ISAPIRuntime 类的一个实例上(译注:原文为 an instance subclass of the ISAPIRuntime class,但是 ISAPIRuntime 类是一个 sealed 类,疑为作者笔误,或者这里的 subclass 并不是子类的意思).进入运行时的第一个入口就是这个没有被文档记载的类,这个类实现了 IISAPIRuntime 接口(对于调用者说明来说,这个接口是一个 COM 接口)这个基于 Iunknown 的底层 COM 接口是从 ISAPI 扩展到 ASP.NET 的一个预定的接口.图 3 展示了 IISAPIRuntime 接口和它的调用签名.(使用了 Lutz Roeder 出色的.NET Reflector 工具 http://www.aisto.com/roeder/dotnet/).这是一个探索这个步步为营过程的很好的方法.

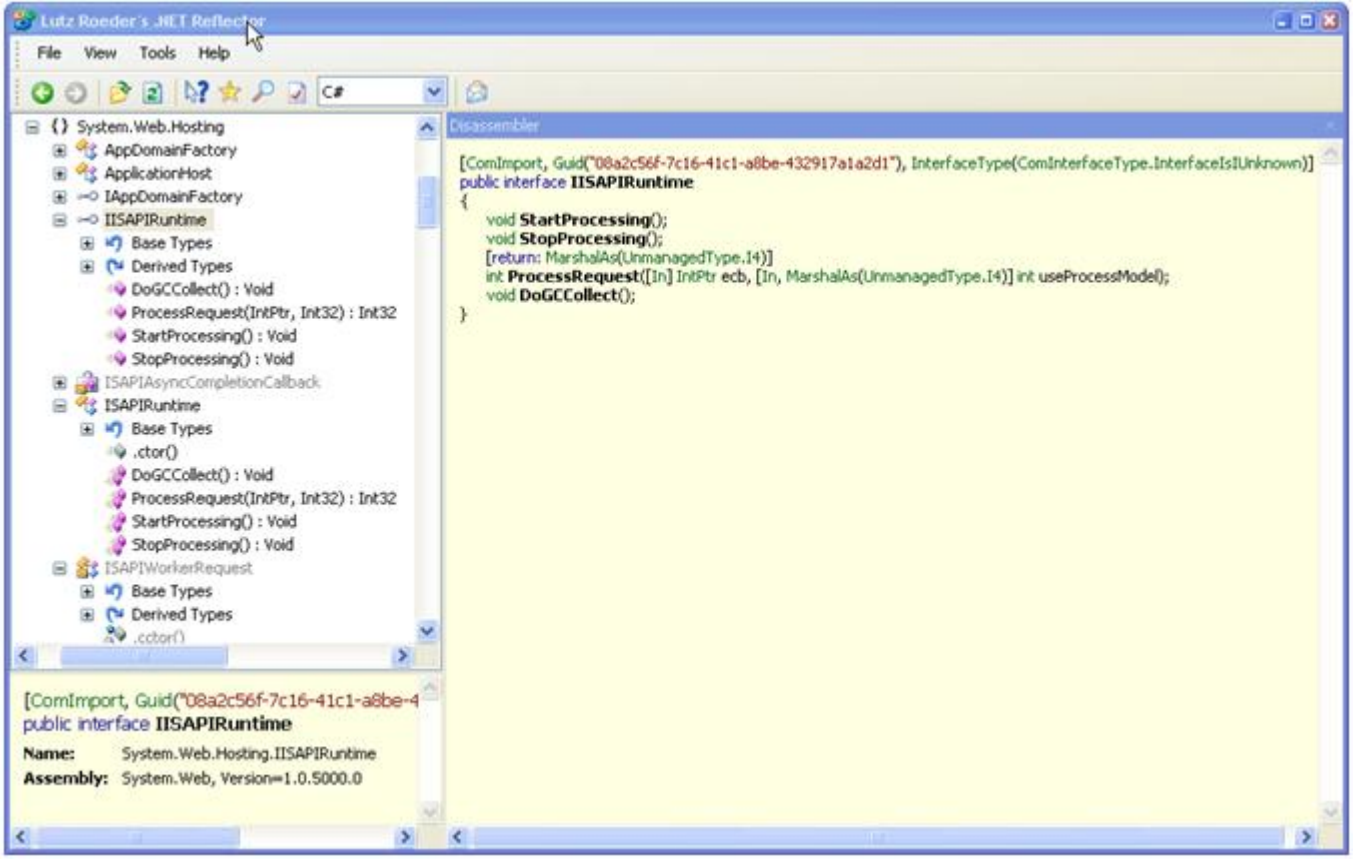


图 3- 如果你想深入这个接口,打开 Reflector,指向 System.Web.Hosting 命名空间. ISAPI DLL 通过调用一个托管的 COM 接口来打开进入 ASP.NET 的入口,ASP.NET 接收一个指向 ISAPI ECB 的非托管指针.这个 ECB 包含访问完整的 ISAPI 接口的能力,用来接收请求和发送响应回到 IIS.

IISAPIRuntime 接口作为从 ISAPI 扩展来的非托管代码和 ASP.NET 之间的接口点(IIS6 中直接相接,IIS5 中通过命名管道).如果你看一下这个类的内部,你会找到含有以下签名的 ProcessRequest 函数:

```
[return: MarshalAs(UnmanagedType.I4)]
int ProcessRequest([In] IntPtr ecb,
                   [In, MarshalAs(UnmanagedType.I4)] int useProcessModel);
```

其中的 ecb 参数就是 ISAPI 扩展控制块(Extention Control Block),被当作一个非托管资源传递给 ProcessRequest 函数.这个函数接过 ECB 后就把它做为基本的输入输出接口,和 Request 和 Response 对象一起使用.ISAPI ECB 包含有所有底层的请求信息,如服务器变量,用于表单(form)变量的输入流和用于回写数据到客户端的输出流.这一个 ecb 引用基本上提供了用来访问 ISAPI 请求所能访问的资源的全部功能,ProcessRequest 是这个资源(ecb)最初接触到托管代码的入口和出口.

ISAPI 扩展异步地处理请求.在这个模式下 ISAPI 扩展马上将调用返回到工作进程或者 IIS 线程上,但是在当前请求的生命周期上 ECB 会保持可用.ECB 含有使 ISAPI 知道请求已经被处理完的机制(通过 ecb.ServerSupportFunction 方法)(译注:更多信息,可以参考开发 ISAPI 扩展的文章),这使得 ECB 被释放.这个异步的处理方法可以马上释放 ISAPI 工作线程,并将处理传递到由 ASP.NET 管理的一个单独的线程上.

ASP.NET 接收到 ecb 引用并在内部使用它来接收当前请求的信息,如服务器变量,POST 的数据,同样它也返回信息给服务器.ecb 在请求完成前或超时时间到之前都保持可访问(stay alive),这样 ASP.NET 就可以继续和它通讯直到请求处理完成.输出被写入 ISAPI 输出流(使用 ecb.WriteClient())然后请求就完成了,ISAPI 扩展得到请求处理完成的通知并释放 ECB.这个实现是非常高效的,因为.NET 类本质上只是对高效的、非托管的 ISAPI ECB 的一个非常”瘦”(thin)的包装器.

装载.NET-有点神秘

让我们从这儿往回退一步:我跳过了.NET 运行时是怎么被载入的.这是事情变得有一点模糊的地方.我没有在这个过程中找到任何的文档,而且因为我们在讨论本机代码,没有很好的办法来反编译 ISAPI DLL 并找出它(装载.NET 运行时的代码)来.

我能作出的最好的猜测是当 ISAPI 扩展接受到第一个映射到 ASP.NET 的扩展名的请求时,工作进程装载了.NET 运行时.一旦运行时存在,非托管代码就可以为指定的虚拟目录请求一个 ISAPIRuntime 的实例(如果这个实例还不存在的话).每个虚拟目录拥有它自己的应用程序域(AppDomain),当一个独立的应用(指一个 ASP.NET 程序)开始的时候 ISAPIRuntime 从启动过程就一直在应用程序域中存在.实例化(译注:应该是指 ISAPIRuntime 的实例化)似乎是通过 COM 来进行的,因为接口方法都被暴露为 COM 可调用的方法.

当第一个针对某虚拟目录的请求到来时,System.Web.Hosting.AppDomainFactory.Create()函数被调用来创建一个 ISAPIRuntime 的实例.这就开始了这个应用的启动进程.这个调用接收这个应用的类型,模块名称和虚拟目录信息,这些信息被 ASP.NET 用来创建应用程序域并启动此虚拟目录的 ASP.NET 程序.这个 HttpRuntime 实例(译注:原文为 This HttpRuntime derived object,但 HttpRuntime 是一个 sealed 类,疑为原文错误)在一个新的应用程序域中被创建.每个虚拟目录(即一个 ASP.NET 应用程序寄)宿在一个独立的应用程序域中,而且他们也只有在特定的 ASP.NET 程序被请求到的时候才会被载入.ISAPI 扩展管理这些 HttpRuntime 对象的实例,并根据请求的虚拟目录将内部的请求路由到正确的那个 HttpRuntime 对象上.

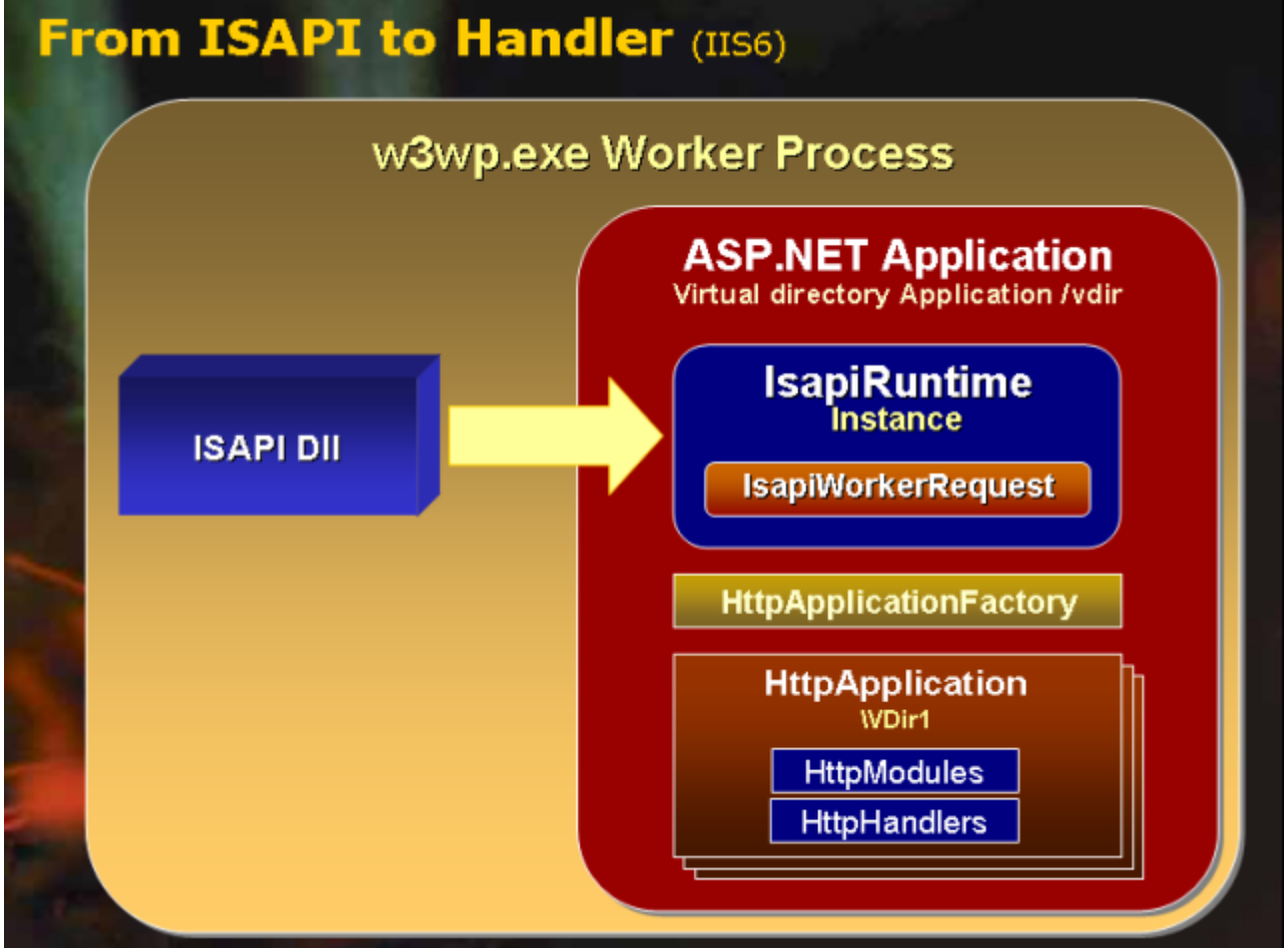


图 4-ISAPI 请求使用一些没有文档记载的类,接口并调用许多工厂方法传送到 ASP.NET 的 HTTP 管道的过程.每个 Web 程序/虚拟目录在它自己的应用程序域中运行,调用者(译注:指 ISAPI DLL)保持一个 IISAPIRuntime 接口的引用来触发 ASP.NET 的请求处理.

回到运行时

在这里我们有一个在 ISAPI 扩展中活动的,可调用的 ISAPIRuntime 对象的实例.每次运行时是启动的并运行着的时候(译注:相对的,如果运行时并没有启动,就需要象上一章所说的那样载入运行时),ISAPI 的代码调用 ISAPIRuntime.ProcessRequest()方法,这个方法是真正的进入 ASP.NET 管道的入口.这个流程在图 4 中显示.

记住 ISAPI 是多线程的,所以请求也会通过 AppDomainFactory.Create()(译注:原文为 ApplicationDomainFactory,疑有误)函数中返回的引用在多线程环境中被处理.列表 1 显示了 ISAPIRuntime.ProcessRequest()方法中反编译后的代码,这个方法接收一个 ISAPI ecb 对象和服务类型(WorkerRequestType)作为参数.这个方法是线程安全的,所以多个 ISAPI 线程可以同时在这一个被返回的对象实例上安全的调用这个方法.

列表 1:ProcessRequest 方法接收一个 ISAPI Ecb 并将其传给工作线程

```
public int ProcessRequest(IntPtr ecb, int iWRType)
{
    HttpWorkerRequest request1 =
ISAPIWorkerRequest.CreateWorkerRequest(ecb, iWRType);

    string text1 = request1.GetAppPathTranslated();
```

```
    string text2 = HttpRuntime.AppDomainAppPathInternal;
    if (((text2 == null) || text1.Equals(".")) ||
         (string.Compare(text1, text2, true,
CultureInfo.InvariantCulture) == 0))
    {
       HttpRuntime.ProcessRequest(request1);
       return 0;
    }

    HttpRuntime.ShutdownAppDomain("Physical application path changed
from " +
       text2 + " to " + text1);
    return 1;
}
```

这里实际的代码并不重要,记住这是从内部框架代码中反编译出来的,你不能直接处理它,它也有可能在将来发生改变.它只是用来揭示在幕后发生了什么.ProcessRequest 方法接收非托管的 ECB 引用并将它传送给 ISAPIWorkerRequest 对象,此对象负责为当前请求创建创建请求上下文.在列表 2 中显示了这个过程.

System.Web.Hosting.ISAPIWorkerRequest 类是 HttpWorkerRequest 类的一个抽象子类(译注:HttpWorkerRequest 和 ISAPIWorkerRequest 都是抽象类,并且 ISAPIWorkerRequest 继承自 HttpWorkerRequest),它的工作是构建一个作为 Web 应用输入的输入输出的抽象视角.注意这里有另一个工厂方法:CreateWorkerRequest,通过判断接受到的第二个参数来创建对应的 WorkerRequest 对象.有三个不同的版本:ISAPIWorkerRequestInProc,ISAPIWorkerRequestInProcForIIS6,ISAPIWorkerRequestOutOfProc.每次有请求进入,这个对象被创建并作为请求和响应对象的基础,它会接收它们的数据和由 WorkerRequest 提供的数据流.

抽象的 HttpWorkerRequest 类在低层接口上提供一个高层的抽象,这样就封装了数据是从哪里来的,可以是一个 CGI Web 服务器,Web 浏览器控件或者是一些你用来给 HTTP 运行时”喂”数据的自定义的机制.关键是 ASP.NET 能用统一的方法来接收信息.

在使用 IIS 的情况下,这个抽象是建立在 ISAPI ECB 块周围.在我们的请求处理过程中,ISAPIWorkerRequest 挂起 ISAPI ECB 并根据需要从它那里取出信息.列表 2 显示了请求字符串值(query string value)是如何被取出来的.

列表 2:使用非托管数据的 ISAPIWorkerRequest 方法

```
// *** Implemented in ISAPIWorkerRequest
public override byte[] GetQueryStringRawBytes()
{
    byte[] buffer1 = new byte[this._queryStringLength];
    if (this._queryStringLength > 0)
    {
       int num1 = this.GetQueryStringRawBytesCore(buffer1,
this._queryStringLength);
       if (num1 != 1)
       {
```

```
            throw new HttpException( "Cannot_get_query_string_bytes");
        }
    }
    return buffer1;
}

// *** Implemented in a specific implementation class
ISAPIWorkerRequestInProcIIS6
internal override int GetQueryStringCore(int encode, StringBuilder
buffer, int size)
{
    if (this._ecb == IntPtr.Zero)
    {
        return 0;
    }
    return UnsafeNativeMethods.EcbGetQueryString(this._ecb, encode,
buffer, size);
}
```

ISAPIWorkerRequest 实现了一个高层次的包装方法,它调用了低层的核心方法,负责真正的访问非托管 APIs-或称为”服务级别的实现”(service level implementation).这些核心方法在特殊的 ISAPIWorkerRequest 子类中为它寄宿的环境提供特殊的实现.这实现了简单的扩展的(pluggable)环境,这样一来当以后新的 Web 服务器接口或其他平台成为了 ASP.NET 的目标时附加的实现类可以在被简单的提供出来.这里还有一个协助类(helper class)System.Web.UnsafeNativeMethods.里面许多对 ISAPI ECB 结构的操作实现了对 ISAPI 扩展的非托管操作.

HttpRuntime,HttpContext 和 HttpApplication

当一个请求到来时,它被路由到 ISAPIRuntime.ProcessRequest()方法.这个方法调用 HttpRuntime.ProcessRequest 方法,它作一些重要的事情(用 Reflector 查看 System.Web.HttpRuntime.ProcessRequestInternal 方法):

- 为请求创建一个新的 HttpContext 实例
- 获取一个 HttpApplication 实例
- 调用 HttpApplication.Init()方法来设置管道的事件
- Init()方法触发开始 ASP.NET 管道处理的 HttpApplication.ResumeProcessing() 方法

首先一个新的 *HttpContext* 对象被创建并用来传递 ISAPIWorkerRequest(ISAPI ECB 的包装器).这个上下文在整个请求的生命周期总都是可用的并总可以通过静态属性 HttpContext.Currect 来访问.正像名字所暗示的那样,*HttpContext* 对象代表了当前活动请求的上下文因为他包含了在请求生命周期中所有典型的你需要访问的重要对象:Request,Response,Application,Server,Cache.在请求处理的任何时候 HttpContext.Current 给你访问所有这些的能力.

HttpContext 对象也包含一个非常有用的 Items 集合,你可以用它来保存针对特定请求的数据.上下文对象在请求周期的开始时被创建,在请求结束时被释放,所有在 Items 集合中保存的数据

只在这个特定的请求中可用.一个很好的使用的例子是请求日志机制,当你通过想通过在 Global.asax 中挂接 Application_BeginRequest 和 Application_EndRequest 方法记录请求的开始和结束时间(象在列表 3 中显示的那样).HttpContext 对你就非常有用了-如果你在请求或页面处理的不同部分需要数据,你自由的使用它.

列表 3-使用 HttpContext.Items 集合使你在不同的管道事件中保存数据

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
    //*** Request Logging
    if (App.Configuration.LogWebRequests)
        Context.Items.Add("WebLog_StartTime",DateTime.Now);
}

protected void Application_EndRequest(Object sender, EventArgs e)
{
    // *** Request Logging
    if (App.Configuration.LogWebRequests)
    {
        try
        {
            TimeSpan Span = DateTime.Now.Subtract(
                                (DateTime)
Context.Items["WebLog_StartTime"] );
            int MiliSecs = Span.TotalMilliseconds;

            // do your logging
                 WebRequestLog.Log(App.Configuration.ConnectionString,
                                true,MilliSecs);
    }
}
```

一旦上下文被设置好,ASP.NET 需要通过 HttpApplication 对象将收到的请求路由到适合的应用程序/虚拟目录.每个 ASP.NET 应用程序必须被设置到一个虚拟目录(或者 Web 根目录)而且每个"应用程序"是被单独的处理的.

HttpApplication 类似仪式的主人-它是处理动作开始的地方

ASP.NET2.0 中的变化

ASP.NET2.0 并没有对底层架构做很多改变.主要的新特性是 HttpApplication 对象有了一系列新的事件-大部分是预处理和后处理事件钩子-这使得应用程序事件管道变得更加的颗粒状了.ASP.NET2.0 也支持新的 ISAPI 功能- HSE_REQ_EXEC_URL-这允许在 ASP.NET 处理的内部重定向到另外的 URL 上.这使得 ASP.NET 可以在 IIS 中设置一个通配符扩展,并处理所有的请求,其中一部分被 HTTP 处理器(handler)处理,另一部分被新的 DefaultHttpHandler 对象处理. DefaultHttpHandler 会在内部调用 ISAPI 来定位到原始的 URL 上.这允许 ASP.NET 可以在其他的页面,如 ASP,被调用前处理认证和登录等事情.

域的主人:HttpApplication

每个请求都被路由到一个 HttpApplication 对象上.HttpApplicationFactory 类根据应用程序的负载为你的 ASP.NET 应用创建一个 HttpApplication 对象池并为每个请求分发 HttpApplication 对象的引用.对象池的大小受 machine.config 文件中 ProcessModel 键中的 MaxWorkerThreads 设置限制,默认是 20 个(译注:此处可能有误,根据 Reflector 反编译的代码,池的大小应该是 100 个,如果池大小小于 100,HttpApplicationFactory 会创建满 100 个,但是考虑到会有多个线程同时创建 HttpApplication 的情况,实际情况下有可能会超过 100 个).

对象池以一个更小的数字开始;通常是一个然后增长到和同时发生的需要被处理的请求数量一样.对象池被监视,这样在大负载下它可能会增加到最大的实例数量,当负载降低时会变回一个更小的数字.

HttpApplication 是你的 Web 程序的外部包装器,而且它被映射到在 Global.asax 里面定义的类上.它是进入 HttpRuntime 的第一个入口点.如果你查看 Global.asax(或者对应的代码类)你会发现这个类直接继承自 HttpApplication:

```
public class Global : System.Web.HttpApplication
```

HttpApplication 的主要职责是作为 Http 管道的事件控制器,所以它的接口主要包含的是事件.事件挂接是非常广泛的,包括以下这些:

- BeginRequest
- AuthenticateRequest
- AuthorizeRequest
- ResolveRequestCache
- AquireRequestState
- PreRequestHandlerExecute
- **...Handler Execution...**
- PostRequestHandlerExecute
- ReleaseRequestState
- UpdateRequestCache
- EndRequest

每个事件在 Global.assx 文件中以 Application_前缀开头的空事件作为实现.例如,Application_BeginRequest(), Application_AuthorizeRequest()..这些处理器为了便于使用而提供因为它们是在程序中经常被使用的,这样你就不用显式的创建这些事件处理委托了.

理解每个 ASP.NET 虚拟目录在它自己的应用程序域中运行,而且在应用程序域中有多个从 ASP.NET 管理的池中返回的 HttpApplication 实例同时运行,是非常重要的.这是多个请求可以被同时处理而不互相妨碍的原因.

查看列表 4 来获得应用程序域,线程和 HttpApplication 之间的关系.

列表 4-显示应用程序域,线程和 HttpApplication 实例之间的关系

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Put user code to initialize the page here
    this.ApplicationId = ((HowAspNetWorks.Global)
           HttpContext.Current.ApplicationInstance).ApplicationId ;
    this.ThreadId = AppDomain.GetCurrentThreadId();
```

```
    this.DomainId = AppDomain.CurrentDomain.FriendlyName;

    this.ThreadInfo = "ThreadPool Thread: " +

System.Threading.Thread.CurrentThread.IsThreadPoolThread.ToString() +
                      "<br>Thread Apartment: " +

System.Threading.Thread.CurrentThread.ApartmentState.ToString();

    // *** Simulate a slow request so we can see multiple
    //     requests side by side.
    System.Threading.Thread.Sleep(3000);
}
```

这是随 sample 提供的 demo 的一部分,运行的结果在图 5 中显示.运行两个浏览器,打开这个演示页面可以看到不同的 ID.

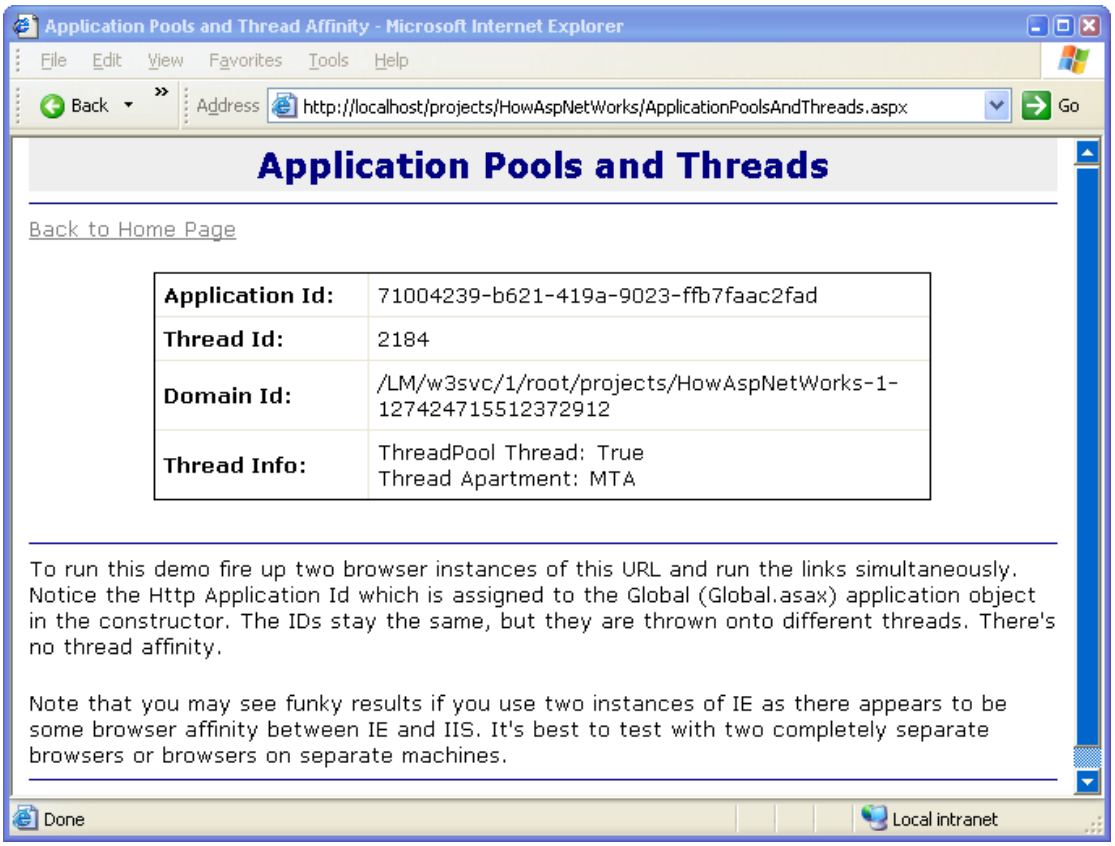


图 5-你可以通过同时运行多个浏览器来简单的查看应用程序域,应用程序池实例和请求线程是如何交互的.当多个请求同时发起,你可以看到线程 ID 和应用程序 ID 变化了,但是应用程序域还是同一个.

你可能注意到在大多数请求上,当线程和 HttpApplication ID 变化时应用程序域 ID 却保持不变,虽然它们也可能重复(指线程和 HttpApplication ID).HttpApplication 是从一个集合中取出,在随后到来的请求中可以被复用的,所以它的 ID 有时是会重复的.注意 Application 实例并不和特定的线程绑定-确切的说它们是被指定给当前请求的活动线程.

线程是由.NET 的线程池管理的,默认是多线程套间(MTA)线程.你可以在 ASP.NET 的页面上通过指定@Page 指令的属性 ASPCOMPAT=”true”来覆盖套间属性.ASPCOMPAT 意味着为 COM 组件提供一个安全的执行环境,指定了这个属性,就会为这些请求使用特殊的单线程套间(STA).STA 线程被存放在单独的线程池中,因为它们需要特殊的处理.

这些 HttpApplication 对象全部在同一个应用程序域中运行的事实是非常重要的.这是为什么 ASP.NET 可以保证对 web.config 文件或者单独的 ASP.NET 页面的修改可以在整个应用程序域中生效.改变 web.config 中的一个值导致应用程序域被关闭并重启.这可以保证所有的 HttpApplication 可以”看到”这个修改,因为当应用程序域重载入的时候,所做的修改(译注:即被修改的文件)会在启动的时候被重新读入.所有的静态引用也会被重载,所以如果程序通过 App Configuration settings 读取值,这些值也会被刷新.

为了在 sample 中看到这点,点击 ApplicationPoolsAndThreads.aspx 页面并记下应用程序域 ID.然后打开并修改 web.config(加入一个空格并保存).然后重新载入页面.你会发现一个新的应用程序域已经被创建了.

本质上当上面的情况发生时,Web 应用/虚拟目录是完整的”重启”了.所有已经在管道中被处理得请求会继续在现存的管道中被处理,当任何一个新的请求来到时,它会被路由到新的应用程序域中.为了处理”被挂起的请求”,ASP.NET 在请求已超时而它(指请求)还在等待时强制关闭应用程序域.所有事实上是可能出现一个应用程序对应两个应用程序域,此时旧的那个正在关闭而新的正在启动.两个应用程序域都继续为它们的客户服务,直到老的那个处理玩正在等待处理的请求并关闭,此时只有一个应用程序域在运行.

“流过”ASP.NET 管道

HttpApplication 触发事件来通知你的程序有事发生,以此来负责请求流转.这作为 HttpApplication.Init()函数的一部分发生(用 Reflector 查看 System.Web.HttpApplication.InitInternal()方法和 HttpApplication.ResumeSteps()方法来了解更多详情),连续设置并启动一系列事件,包括执行所有的处理器(handler).这些事件处理器映射到 global.asax 中自动生成的哪些事件中,同时它们也映射到所有附加的 HttpModule(它们本质上是 HttpApplication 对外发布的额外的事件接收器(sink)).

HttpModule 和 HttpHandler 两者都是根据 Web.config 中对应的配置被动态载入并附加到事件处理链中.HttpModule 实际上是事件处理器,附加到特殊的 HttpApplication 事件上,然而 HttpHandler 是用来处理”应用级请求处理”的终点.

HttpModule 和 HttpHandler 两者都是在 HttpApplication.Init()函数调用的一部分中被载入并附加到调用链上.图 6 显示了不同的事件,它们是何时发生的以及它们影响管道的哪一部分.

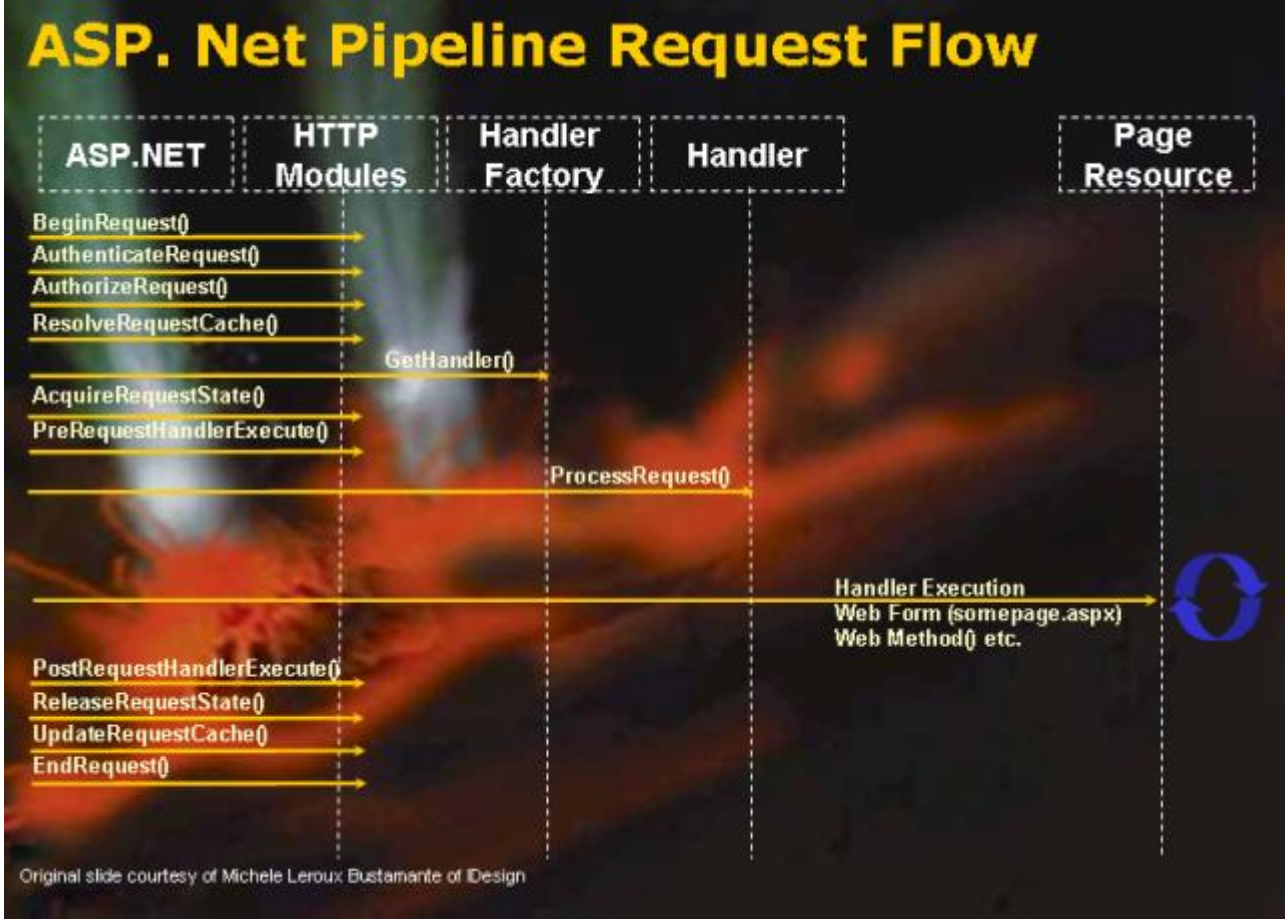


图6-事件在ASP.NET http 管道中流转的过程.HttpApplication 对象的事件驱动请求在管道中流转.Http Module 可以拦截这些事件并覆盖或者扩展现有的功能.

HttpContext, HttpModules 和 HttpHandlers

httpApplication 它本身对发送给应用程序的数据一无所知-它只是一个通过事件来通讯的消息对象.它触发事件并通过 HttpContext 对象来向被调用函数传递消息.实际的当前请求的状态数据由前面提到的 HttpContext 对象维护.它提供了所有请求专有的数据并从进入管道开始到结束一直跟随请求.图 7 显示了 ASP.NET 管道中的流程.注意上下文对象(即 HttpContext),这个从请求开始到结束一直都是你"朋友"的对象,可以在一个事件处理函数中保存信息并在以后的事件处理函数中取出.

一旦管道被启动,HttpApplication 开始象图六那样一个个的触发事件.每个事件处理器被触发,如果事件被挂接,这些处理器将执行它们自己的任务.这个处理的主要任务是最终调用挂接到此特定请求的 HttpHandler.处理器(handler)是 ASP.NET 请求的核心处理机制,通常也是所有应用程序级别的代码被执行的地方.记住 ASP.NET 页面和 Web 服务框架都是作为 HttpHandler 实现,这里也是处理请求的的核心之处.模块(module)趋向于成为一个传递给处理器(handler)的上下文的预处理或后处理器.ASP.NET 中典型的默认处理器包括预处理的认证,缓存以及后处理中各种不同的编码机制.

有很多关于 HttpHandler 和 HttpModule 的可用信息,所以为了保持这篇文章在一个合理的长度,我将提供一个关于处理器的概要介绍.

HttpModule

当请求在管道中传递时,HttpApplicaion 对象中一系列的事件被触发.我们已经看到这些事件在 Global.asax 中作为事件被发布.这种方法是特定于应用程序的,可能并不总是你想要的.如果你要建立一个通用的可用被插入任何 Web 应用程序的 HttpApplication 事件钩子,你可用使用 HttpModule,这是可复用的,不需要特定语应用程序代码的,只需要 web.config 中的一个条目.

模块本质上是过滤器(fliter)-功能上类似于 ISAPI 过滤器,但是它工作在 ASP.NET 请求级别上.模块允许为每个通过 HttpApplication 对象的请求挂接事件.这些模块作为外部程序集中的类存贮.,在 web.config 文件中被配置,在应用程序启动时被载入.通过实现特定的接口和方法,模块被挂接到 HttpApplication 事件链上.多个 HttpModule 可以被挂接在相同的事件上,事件处理的顺序取决于它们在 Web.config 中声明的顺序.下面是在 Web.config 中处理器定义.

```
<configuration>
  <system.web>
    <httpModules>
   <add name= "BasicAuthModule"
      type="HttpHandlers.BasicAuth,WebStore" />
    </httpModules>
  </system.web>
</configuration>
```

注意你需要指定完整的类型名和不带 dll 扩展名的程序集名.

模块允许你查看每个收到的 Web 请求并基于被触发的事件执行一个动作.模块在修改请求和响应数据方面做的非常优秀,可用为特定的程序提供自定义认证或者为发生在 ASP.NET 中的每个请求增加其他预处理/后处理功能.许多 ASP.NET 的功能,像认证和会话(Session)引擎都是作为 HttpModule 来实现的.

虽然 HttpModule 看上去很像 ISAPI 过滤器,它们都检查每个通过 ASP.NET 应用的请求,但是它们只检查映射到单个特定的 ASP.NET 应用或虚拟目录的请求,也就是只能检查映射到 ASP.NET 的请求.这样你可以检查所有 ASPX 页面或者其他任何映射到 ASP.NET 的扩展名.你不能检查标准的.HTM 或者图片文件,除非你显式的映射这些扩展名到 ASP.NET ISAPI dll 上,就像图 1 中展示的那样.一个常见的此类应用可能是使用模块来过滤特定目录中的 JPG 图像内容并在最上层通过 GDI+来绘制'样品'字样.

实现一个 HTTP 模块是非常简单的:你必须实现之包含两个函数(Init()和 Dispose())的 IHttpModule 接口.传进来的事件参数中包含指向 HTTPApplication 对象的引用,这给了你访问 HttpContext 对象的能力.在这些方法上你可以挂接到 HttpApplication 事件上.例如,如果你想挂接 AuthenticateRequest 事件到一个模块上,你只需像列表 5 中展示的那样做

列表 5:基础的 HTTP 模块是非常容易实现的

```
public class BasicAuthCustomModule : IHttpModule
{

    public void Init(HttpApplication application)
    {
        // *** Hook up any HttpApplication events
```

```
        application.AuthenticateRequest +=
              new EventHandler(this.OnAuthenticateRequest);
    }
    public void Dispose() { }

    public void OnAuthenticateRequest(object source, EventArgs
eventArgs)
    {
        HttpApplication app = (HttpApplication) source;
        HttpContext Context = HttpContext.Current;
        … do what you have to do…                          }
}
```

记住你的模块访问了 HttpContext 对象,从这里可以访问到其他 ASP.NET 管道中固有的对象,如请求(Request)和响应(Response),这样你还可以接收用户输入的信息等等.但是记住有些东西可能是不能访问的,它们只有在处理链的后段才能被访问.

你可以在 Init()方法中挂接多个事件,这样你可以在一个模块中实现多个不同的功能.然而,将不同的逻辑分到单独的类中可能会更清楚的将模块进行模块化(译注:这里的模块化和前面的模块没有什么关系)在很多情况下你实现的功能可能需要你挂接多个事件-例如一个日志过滤器可能在 BeginRequest 事件中记录请求开始时间,然后在 EndRequest 事件中将请求结束写入到日志中.
注意一个 HttoModule 和 HttpApplication 事件中的重点:Response.End()或 HttpApplication.CompleteRequest()会在 HttpApplication 和 Module 的事件链中"抄近道".看"注意 Response.End()"来获得更多信息.

> **注意 Response.End()**
> 当创建 HttpModule 或者在 Global.asax 中实现事件钩子的时候,当你调用 Response.End 或 Appplication.CompleteRequest 的时候要特别注意.这两个函数都结束当前请求并停止触发在 HTTP 管道中后续的事件,然后发生将控制返回到 Web 服务器中.当你在处理链的后面有诸如记录日志或对内容进行操作的行为时,因为他们没有被触发,有可能使你上当.例如,sample 中 logging 的例子就会失败,因为如果调用 Response.End()的话,EndRequest 事件并不会被触发.

HttpHandlers

模块是相当底层的,而且对每个来到 ASP.NET 应用程序的请求都会被触发.Http 处理器更加的专注并处理映射到这个处理器上的请求.

Http 处理器需要实现的东西非常简单,但是通过访问 HttpContext 对象它可以变得非常强大.Http 处理器通过实现一个非常简单的 IHttpHandler 接口(或是它的异步版本,IHttpAsyncHandler),这个接口甚至只含有一个方法-ProcessRequest()-和一个属性 IsReusable.关键部分是 ProcessRequest(),这个函数获取一个 HttpContext 对象的实例作为参数.这个函数负责从头到尾处理 Web 请求.

单独的,简单的函数?太简单了,对吧?好的,简单的接口,但并不弱小!记住 WebForm 和 WebService 都是作为 Http 处理器实现的,所以在这个看上去简单的接口中包装了很强大的能力.关键是这样一个事实,当一个请求来到 Http 处理器时,所有的 ASP.NET 的内部对象都被准备和设置好来处理请求了.主要的是 HttpContext 对象,提供所有相关的请求功能来接收输入并输

出回 Web 服务器.

对一个 HTTP 处理器来说所有的动作都在这个单独的 ProcessRequest()函数的调用中发生.这像下面所展示的这样简单:

```
public void ProcessRequest(HttpContext context)
{
 context.Response.Write("Hello World");
}
```

也可以像一个可以从 HTML 模板渲染出复杂表单的 WebForm 页面引擎那么完整,复杂.通过这个简单,但是强大的接口要做什么,完全取决于你的决定.

因为 Context 对象对你是可用的,你可用访问 Request,Response,Session 和 Cache 对象,所以你拥有所有 ASP.NET 请求的关键特性,可以获得用户提交的内容并返回你产生的内容给客户端.记住 HttpContext 对象-它是你在整个 ASP.NET 请求的生命周期中的"朋友".

处理器的关键操作应该是将输出写入 Response 对象或者更具体一点,是 Response 对象的 OutputStream.这个输出是实际上被送回到客户端的.在幕后,ISAPIWorkerRequest 管理着将输出流返回到 ISAPI ecb 的过程.WriteClient 方法是实际产生 IIS 输出的方法.

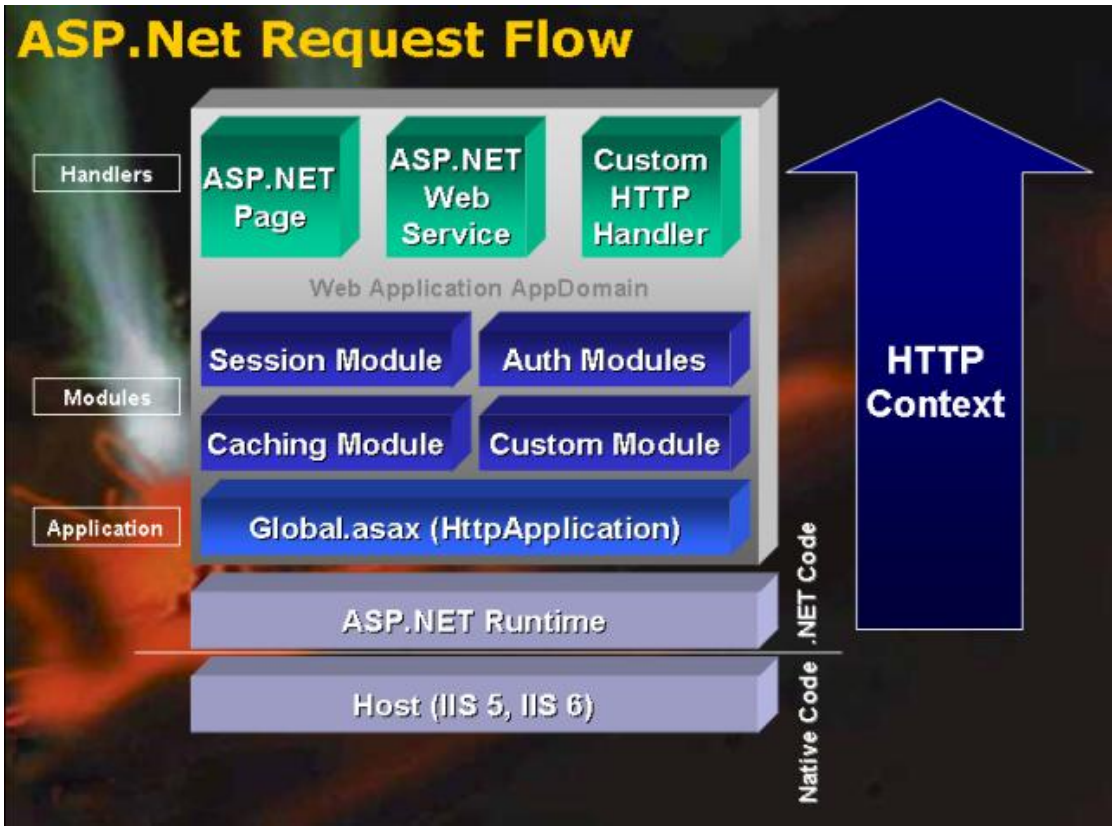


图 7-ASP.NET 请求管道通过一系列事件接口来转发请求,提供了更大的灵活性.Application 当请求到来并通过管道时作为一个载入 Web 应用并触发事件的宿主容器.每个请求都沿着配置的 Http 过滤器和模块的路径走(译注:原文为 Http Filters And Modules,应该是指 Http Module 和 Http Handler).过滤器可以检查每个通过管道的请求,Handler 允许实现应用程序逻辑或者像 Web Form 和 WebService 这样的应用层接口.为了向应用提供输入输出,Context 对象在这个处理过程中提供了特定于请求的的信息.

WebForm 使用一系列在框架中非常高层的接口来实现一个 Http 处理器,但是实际上 WebForm 的 Render()方法简单的以使用一个 HtmlTextWriter 对象将它的最终结果输出到 context.Response.OutputStream 告终.所以非常梦幻的,终究即使是向 WebForm 这样高级的工具也只是在 Request 和 Response 对象之上进行了抽象而已.

到了这里你可能会疑惑在 Http handler 中你到底需要处理什么.既然 WebForm 提供了简单可用的 Http Handler 实现,那么为什么需要考虑更底层的东西而放弃这扩展性呢?

WebForm 对于产生复杂的 HTML 页面来说是非常强大的,业务层逻辑需要图形布局工具和基于模块的页面.但是 WebForm 引擎做了一系列 overhead intensive 的任务.如果你想要做的是从系统中读入一个文件并通过代码将其返回的话,不通过 WebForm 框架直接返回文件会更有效率.如果你要做的是类似从数据库中读出图片的工作,并不需要使用页面框架-你不需要模板而且确定不需要 Web 页面并从中捕捉用户事件.

没有理由需要建立一个页面对象和 Session 并捕捉页面级别的事件-所有这些需要执行对你的任务没有帮助的额外的代码.

所以自定义处理器更加有效率.处理器也可用来做 WebForm 做不到的事情,例如不需要在硬盘上有物理文件就可用处理请求的能力,也被称为虚拟 Url.要做到这个,确认你在图 1 中展示的应用扩展对话框中关掉了"检查文件存在"选项.

这对于内容提供商来说非常常见,象动态图片处理,XML 服务,URL 重定向服务提供了 vanity Urls,下载管理以及其他,这些都不需要 WebForm 引擎.

> 异步 HTTP Handler
> 在这篇文章中我大部分都在讨论同步处理,但是 ASP.NET 运行时也可以通过异步 HTTP handler 来支持异步操作.这些处理器自动的将处理"卸载"到独立的线程池的线程中并释放主 **ASP.NET** 线程,使 **ASP.NET** 线程可以处理其他的请求.不幸的是在 **1.x** 版的.**NET** 中,"卸载"后的处理还是在同一个线程池中,所以这个特性之增加了一点点的性能.为了创建真正的异步行为,你必须创建你自己的线程并在回调处理中自己管理他们.
> 当前版本的 ASP.NET 2.0 Beta 2 在 IhttpHandlerAsync(译注:此处应该是指 IHttpAsyncHandler,疑为作者笔误)接口和 Page 类两方面做了一些对异步处理的改进,提供了更好的性能,但是在最终发布版本中这些是否会保留.

我说的这些对你来说够底层了吗?

唷-我们已经走完了整个请求处理过程了.这过程中有很多底层的信息,我对 HTTP 模块和 HTTP 处理器是怎么工作的并没有描述的非常详细.挖掘这些信息相当的费时间,我希望在了解了 ASP.NET 底层机制后,你能获得和我一样的满足感.

在结束之前让我们快速的回顾一下我在本文中讨论的从 IIS 到处理器(handler)的过程中,事件发生的顺序

- IIS 获得请求
- 检查脚本映射中,此请求是否映射到 aspnet_isapi.dll
- 启动工作进程 (IIS5 中为 aspnet_wp.exe,IIS6 中为 w3wp.exe)
- .NET 运行时被载入
- IsapiRuntime.ProcessRequest()被非托管代码调用

- 为每个请求创建一个 IsapiWorkerRequest
- HttpRuntime.ProcessRequest()被工作进程调用
- 以 IsapiWorkerRequest 对象为参数创建 HttpContext 对象
- 调用 HttpApplication.GetApplicationInstance()来从池中取得一个对象实例
- 调用 HttpApplication.Init()来开始管道事件并挂接模块和处理器
- HttpApplicaton.ProcessRequest 被调用以开始处理.
- 管道中的事件被依次触发
- 处理器被调,ProcessRequest 函数被触发
- 控制返回到管道中,后处理事件被依次触发

有了这个简单的列表,记住这些东西并把他们拼在一起就变得容易多了.我时常看看它来加深记忆.所以现在,回去工作,做一些不那么抽象的事情...

虽然我都是基于 ASP.NET1.1 来进行讨论的,不过在 ASP.NET2.0 中这些处理过程看上去并没有发生太大的变化.