

# 从底层了解 ASP.NET 体系结构

作者：[Rick Strahl](#) || 翻译：[today](#) || [下载例子代码](#)

## 目录

1. [ASP.NET 是什么？](#)
2. [从浏览器到 ASP.NET](#)
3. [ISAPI 连接](#)
4. [IIS5 和 IIS6 的不同之处](#)
5. [进入 .NET 运行时](#)
6. [加载 .NET 一稍微有点神秘](#)
7. [回到运行时](#)
8. [HttpRuntime, HttpContext 以及 HttpApplication](#)
9. [Web 程序的主要部分：HttpApplication](#)
10. [穿过 ASP.NET 管道](#)
11. [HttpContext, HttpModules 和 HttpHandlers](#)
12. [HttpModules](#)
13. [HttpHandlers](#)
14. [是否已经提供了足够的底层知识？](#)

**摘要：**ASP.NET 是一个用于构建 Web 程序的强大平台，提供了巨大的弹性和能力以至于它可以构建任意的 Web 程序。许多人仅仅对处于 ASP.NET 高层次的框架如：WebForms 和 WebServices 比较熟悉，因此，在这篇文章里，我将会阐述有关 ASP.NET 比较底层的知识，并且将会解释，如何将请求从 Web Server 移交给 ASP.NET 运行时，然后通过 ASP.NET HTTP 管道处理这些请求。

对于我来说，了解一个平台的内部工作机制总是会让我感到一些满足和安慰，如同洞察，可以帮助我写出更好的程序。知道了工具有什么用途，以及它们如何组装成复杂框架的一部分，这些将会使你很容易的找到问

题的解决方案，以及在你修改和调试错误时，都显得非常重要。这篇文章的目的就是从底层了解 ASP.NET 以及帮助你理解请求如何流入 ASP.NET 处理管道里。同时，你将会了解 ASP.NET 引擎的核心，以及一个 Web 请求如何在这里结束。这里讲到的许多知识都是你日常工作中没必要知道的，但是，如果你理解了 ASP.NET 如何把请求路由到应用程序的代码里（通常比较高层次的），这将对你非常有用。

注：整个 ASP.NET 引擎完全构建在托管代码里，其所有的扩展性都是通过托管代码去构建。

使用 ASP.NET 的大多数都比较熟悉 WebForms 和 WebServices。这些高层次的实现，使得构建 Web 程序变得非常容易。ASP.NET 被设计为驱动引擎，它把底层的接口提供给 Web Server，为高层次 Web 应用程序的前端和末端提供了路由服务。WebForms 和 WebServices 是建立在 ASP.NET 框架之上，有关 HTTP 处理的两种最常用的方式。

其实，在较低的层次上，ASP.NET 也提供了足够多的灵活性。HTTP 运行时和请求管道提供了同样的能力，可以构建类似于 WebForms 和 WebServices 的实现，当然，这些已经使用 .NET 托管代码实现了。如果你需要构建一个自定义 HTTP 处理平台，而这个平台要比 WebForms 所处的层次低一点，那么你就会用到所有这些类似的功能。

构建大多的 Web 界面，使用 WebForms 无疑是最容易的方法，但是，如果你想自定义一个内容处理器，或者需要对流入和流出的内容做特殊的处理，或者需要为一个应用程序定制一个应用服务器接口，那么使用这些低层次的处理或者模块将会得到更好的性能，以及可以在真正的请求处理中获得更多的控制权。尽管那些高层次的实现，如：WebForms 和 WebServices 已提供了类似的功能，但由于它们针对请求添加了太多的控制（导致性能下降）。所以你完全可以另辟佳境，在较低层次上处理这些请求。

## ASP.NET 是什么？

让我们从最简单的定义开始，ASP.NET 是什么？我通常喜欢用如下语句来描述 ASP.NET。

ASP.NET 是完全使用托管代码处理 Web 请求的一个成熟引擎平台。它不仅只是 WebForms 和 WebServices。

ASP.NET 是一个请求处理引擎。它获取客户端请求，然后通过它内置的管道，把请求传到一个终点，在这个终点，开发者可以添加处理这个请求的逻辑代码。实际上这个引擎和 HTTP 或者 Web Server 是完全分开的。事实上，HTTP 运行时是一个组件，你可以把它宿主在 IIS 之外的应用程序上。甚至完全可以和其它的服务组合在一起。例如，你可以把 HTTP 运行时宿主在 Windows 桌面应用程序里（详细的内容请查看：

<http://www.west-wind.com/presentations/aspnetruntime/aspnetruntime.aspx>）。

通过使用内置的管道路由请求，HTTP 运行时提供了一套复杂的，但却很优雅的机制。在处理请求的每一个层面都牵涉到许多对象，但大多数对象都可以通过派生或者事件接口来扩展。所以，此框架具有非常高的可扩展性。通过这一套机制，可以进入较低层次的接口如：缓存，身份验证，授权等是有可能的。你可以在处理请求之前或之后过滤内容，或者仅仅把匹配指定签名的客户端请求直接路由到你的代码里或转向其它的 URL。针对同一件事情，可以通过不同的处理方法完成，而且实现代码都非常的直观。除此之外，在容易开发和性能之间，HTTP 运行时还提供了最佳的灵活性。

整个 ASP.NET 引擎完全构建在托管代码里，所有的扩展性功能都是通过托管代码的扩展提供。对于功能强大的 .NET 框架而言，使用自己的东西，构建一个成熟的、高性能的引擎体系结构已经成为一个遗嘱。尽管如此，但重要的是，ASP.NET 给人印象最深的是高瞻远瞩的设计，这使得在其之上的工作变得非常容易，并且提供了几乎可以钩住请求处理当中任意部分的能力。

使用 ASP.NET 可以完成一些任务，之前这些任务是使用 IIS 上的 ISAPI 扩展和过滤来完成的。尽管还有一些限制，但与 ASP 相比，已经有了很大的进步。ISAPI 是底层 Win32 样式的 API，仅它的接口就有 1 兆，这对于大型的程序开发是非常困难的。由于 ISAPI 是底层的接口，因此它的

速度也是非常的快。但对于企业级的程序开发是相当的难于管理的。所以，在一定的时间内，ISAPI 主要充当其它应用程序或平台的桥接口。但是无论如何，ISAPI 没有被废弃。事实上，微软平台上的 ASP.NET 和 IIS 的接口是通过宿主在 .NET 里的 ISAPI 扩展来通信的，然后直达 ASP.NET 运行时。ISAPI 提供了与 Web Server 通信的核心接口，然后 ASP.NET 使用非托管代码获取请求以及对客户端请求发出响应。ISAPI 提供的内容经由公共对象类似于 `HttpRequest` 和 `HttpResponse`，通过一个设计优良的、可访问的接口，以托管对象的方式暴露非托管数据。

## 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。用户通过在浏览器中键入一个 URL，点击一个超链接，提交一个 HTML 表单（一个 post 请求），或者一个客户端程序调用基于 ASP.NET 的 `WebService`（通过 ASP.NET 提供服务）。在服务器端，IIS5 或者 IIS6 将会收到这个请求。ASP.NET 的底层通过 ISAPI 扩展与 IIS 通信，然后，通过 ASP.NET，这个请求通常被路由到一个带有 `.aspx` 扩展名的页面。但是，这个处理过程如何工作，则完全依赖于 HTTP 处理器（handler）的执行。这个处理器将被安装用于处理指定的扩展。在 IIS 中，`.aspx` 经由“应用程序扩展”被映射到 ASP.NET ISAPI 的 dll 文件：`aspnet_isapi.dll`。每一个触发 ASP.NET 的请求，都必须经由一个已经注册的，并且指向 `aspnet_isapi.dll` 的扩展名来标识。

*注：ISAPI 是自定义 Web 请求处理中第一个并且具有最高性能的 IIS 入口点。*

依靠扩展名，ASP.NET 把一个请求路由到一个恰当的处理程序，该处理程序则负责处理这个请求。举个例子，`WebServices` 的扩展名 `.asmx` 不会把一个请求路由到磁盘上的某一个页面，而是会路由到在定义中附加了指定特性（`WebMethodAttribute`）的类，此特性会把它标识成一个 `Web Services` 的实现。许多其它的处理程序将随着 ASP.NET 一起被安装。当然也可以定义你自己的处理程序。在 IIS 里所有的 `HttpHandler` 被映射并指向 ASP.NET ISAPI 扩展，并且这些 `HttpHandler` 也都在 `web.config` 里配置，用于把

请求路由到指定的 HTTP 处理器里执行。每一个处理器都是一个 .NET 类，用于处理指定的扩展。而这些处理器可以处理简单到只有几行代码的 Hello World，也可以处理复杂到类似 ASP.NET 的页面以及执行 Webservice。就目前而言，仅仅需要理解扩展就是一种基本的映射机制，ASP.NET 用它可以从 ISAPI 里获取一个请求，然后把请求路由到指定处理该请求的处理器中。

## ISAPI 连接

ISAPI 是底层非托管的 Win32 API。它定义的接口非常的单一并且性能最优。用这些接口处理原始指针 (raw pointer)，而函数指针列表 (function pointer tables) 则用于回调。ISAPI 提供了最低层的、高性能的接口，开发者和工具厂商可以使用这些接口深入到 IIS 里。由于 ISAPI 是非常低层的，所以不太适合使用它构建应用级的程序。ISAPI 趋向于被当作桥接口使用，用于给高层次的工具提供应用服务类型的功能。例如，ASP 和 ASP.NET 都是被当作冷聚变 (cold fusion) 构建于 ISAPI 之上。大多 Perl、PHP 和 JSP 的执行如同许多第三方解决方案一样，可以在 IIS 运行。ISAPI 是个非常好的工具，它给高层次的应用程序提供了高性能垂直访问接口。这使得那些高层次的应用程序需要的信息可以从 ISAPI 提供的信息中提炼。在 ASP 和 ASP.NET 里，引擎可以提炼 ISAPI 接口提供的表单里的对象如：Request 和 Response，这些对象可以从 ISAPI 请求的信息中读取它们各自的内容。

作为约定，ISAPI 支持 ISAPI 扩展 (extensions) 和 ISAPI 过滤 (filters)。扩展是请求处理接口，提供了跟 Web Server 输入和输出相关的逻辑处理。从本质上来说，它是一个事务接口。ASP 和 ASP.NET 都被看作 ISAPI 扩展的实现。ISAPI 是钩子接口，它允许你查看进入 IIS 的每一个请求并且可以修改请求的内容（包括输入和输出）或者改变模块（如：身份验证等）的行为。顺便提一下，ASP.NET 通过两个方面的内容：HTTP 处理器（对应 ISAPI 扩展）和 HTTP 模块（对应 ISAPI 过滤）映射到 ISAPI。这些相关的内容我将会在后面详细描述。

ISAPI 是代码的初始点，标识 ASP.NET 一个请求的开始。ASP.NET 映射了不同的扩展到它的 ISAPI 扩展，ISAPI 扩展位于 .NET Framework 目录：

<.NET FrameworkDir>\aspnet\_isapi.dll

你可以在 IIS 服务管理器里看到这些映射，如图 1 所示。打开 Web 站点的根目录的属性，选择主目录选项卡，然后查看 配置|应用程序映射。

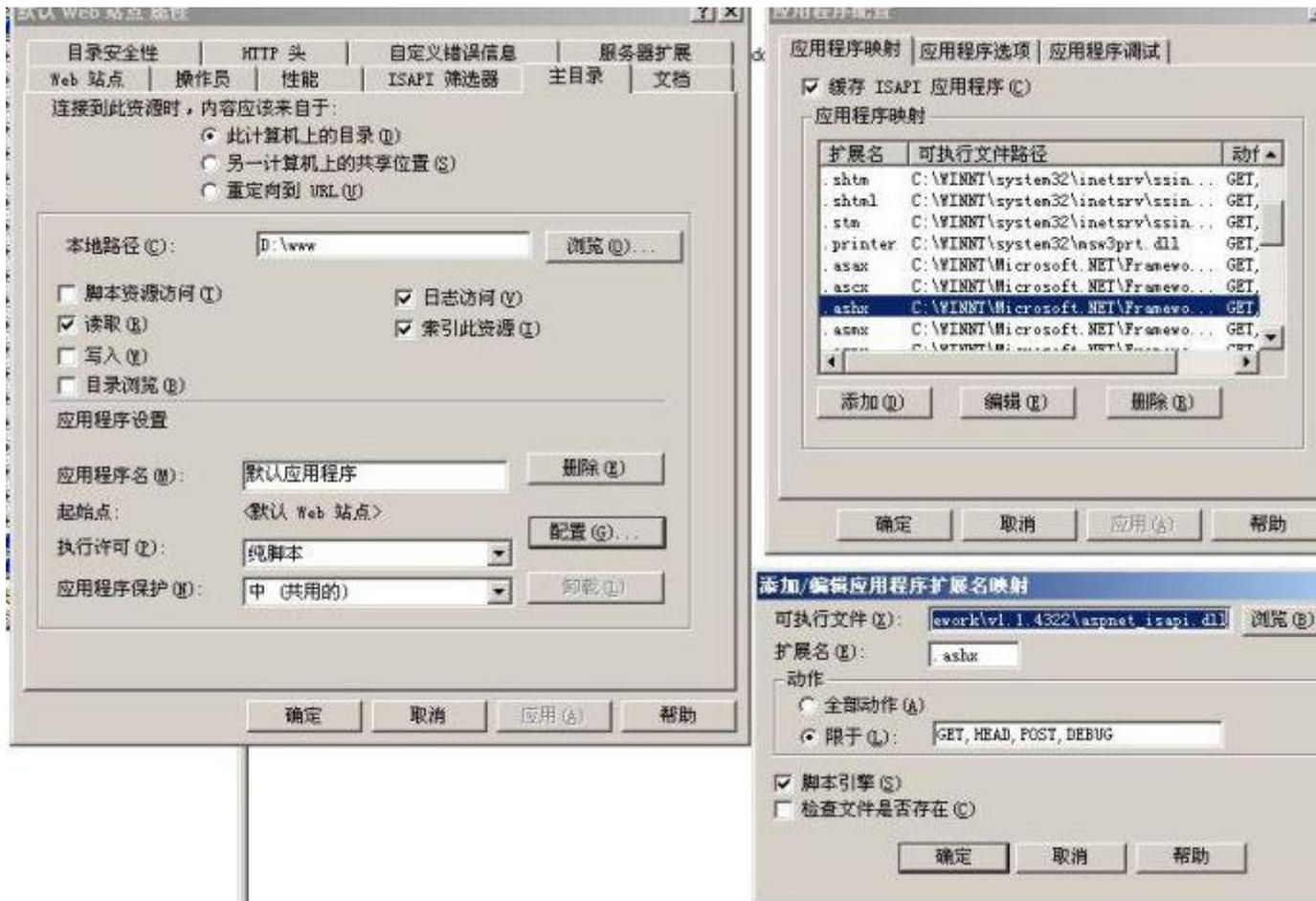


图 1: IIS 把不同的扩展名如 .aspx 映射到 ASP.NET 的 ISAPI 扩展。通过这种机制，在 Web Server 里，请求就可以被路由到 ASP.NET 的处理管道里。

尽管 .NET 需要很多扩展名，但不必手工设置它们，你可以使用 aspnet\_regiis.exe 实用工具确保所有的脚本映射都被注册。

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

这将会把 ASP.NET 运行时的个别版本，通过脚本映射注册到整个 Web 站点，并且安装客户端脚本库，这些脚本库将会被浏览器上的控件所使用。注意，它是注册了安装在上面目录里的 CLR 的那个版本。aspnet\_regiis 有一个可选项，可以使你单独配置一个虚拟目录。每一个 .NET 框架的版本，都拥有各自的 aspnet\_regiis，对于不同版本的 .NET 框架，你需要运行适当版本的 aspnet\_regiis，注册到整个站点或者虚拟目录。从 ASP.NET 2.0 开始，在 IIS 控制台里，有一个 IIS ASP.NET 配置页面，在这个页面你可以挑选 .NET 的版本。

## IIS5 和 IIS6 的不同之处

当一个请求进来的时候，IIS 会检查脚本映射，然后把请求路由到 aspnet\_isapi.dll。接下来这个 DLL 文件的操作是什么呢？在 IIS5 和 IIS6 里，这个请求又是如何到达 ASP.NET 运行时的呢？它们两者的处理方式有没有重大变化呢？图 2 展示了一个大致的流程。

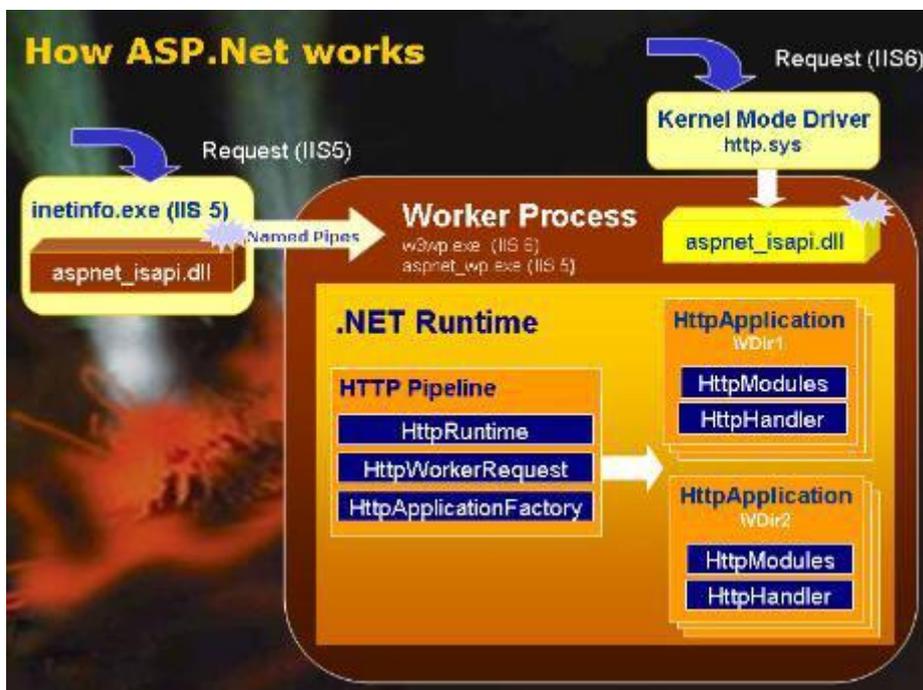


图 2: 站在比较高的角度，观看请求从 IIS 到 ASP.NET 运行时的流程，然后直达请求

处理管道。IIS5 和 IIS6 与 ASP.NET 的接口采用了不同的方式，但从请求到达 ASP.NET 管道后的整个过程是完全一样的。

IIS5 直接把 `aspnet_isapi.dll` 寄宿在 `inetinfo.exe` 进程里，或者它们中的一个将会与工作进程隔离，如果你拥有隔离权限，那么可以把 Web 站点和虚拟目录隔离的级别设置为中等或者高级。当第一个 ASP.NET 请求进来的时候，DLL 将会在另一个 `EXE-aspnet_wp.exe` 里分配一个新的进程，然后把相关信息路由到这个新分配的进程里。接着这个新的进程会依次加载和寄宿 .NET 运行时。每一个进入到 ISAPI DLL 的请求，都将通过调用命名管道路由给这个工作者进程。

*注：IIS6 与之前的 Web Server 不同，已经对 ASP.NET 进行了优化处理。IIS6 中使用了应用程序池。*

值得注意的是，IIS6 改变了这个处理模型，IIS 不再直接寄宿像 ISAPI 扩展的任何外部可执行代码。代替的是，IIS 总会保持一个单独的工作进程：应用程序池。所有的处理都发生在这个进程里，包括 ISAPI dll 的执行。对于 IIS6 而言，应用程序池是一个重大的改进，因为它们允许以更小的粒度控制一个指定进程的执行。你可以为每一个虚拟目录或者整个 Web 站点配置应用程序池，这可以使你很容易的把每一个应用程序隔离到各自的进程里，这样就可以把它与运行在同一台机器上其他程序完全隔离。从 Web 处理的角度看，如果一个进程死掉，至少它不会影响到其它的进程。

另外，应用程序池是高度可配置的。通过设置应用程序池的执行许可，可以配置它们的执行安全环境。并且可以为指定的应用程序按照相同的粒度定制这些配置。对于 ASP.NET 而言，IIS6 最大的改进是使用应用程序池代替了 `machine.config` 里的 `ProcessModel` 实体的大部分功能。在 IIS5 里，这个实体是很难管理的，因为它的设置是全局的，而且不能够在指定 Web 程序的 `web.config` 里覆盖这些设置。当 IIS6 运行的时候，`ProcessModel` 里的大部分配置将被忽略，取而代之的是读取应用程序池的配置。注意我这里说的是大部分，另外的一些配置，像线程池的大小和 IO 线程数目等仍然需要通过这个节点配置，这是因为，在服务器的应用程序池里没有提供类似功能的配置。

由于应用程序池是在外部执行的，所以这些执行可以很容易的被监控和管理。IIS6 还提供了许多性能计数器，可以对重启和超时选项进行跟踪。在许多情况下，这可以帮助应用程序纠正问题。最后，IIS6 的应用程序池的实现不依赖 COM+，正如 IIS5 隔离进程一样，这提高了程序的性能和稳定性，特别是那些需要在内部使用 COM 对象的程序。

IIS6 应用程序池也包含了 ASP.NET 固有的东西，ASP.NET 可以和新的底层 API 通信，这些 API 允许直接访问 HTTP 缓冲存储器的 API，而 HTTP 缓冲存储器的 API 可以直接进入 Web Server 的缓冲存储器，卸载 ASP.NET 级别的缓存。

在 IIS6 里，ISAPI 扩展运行在应用程序池的工作进程里。而 .NET 运行时也运行在这个进程里，所以 ISAPI 扩展和 .NET 运行时的通信是发生在进程内的。这就使得比必须使用命名管道接口的 IIS5 具有更高的性能。尽管 IIS 的宿主模型不同，但是真正进入托管代码的接口是类似的，仅仅在获取被路由的请求时有一些变动。

## 进入 .NET 运行时

进入 .NET 运行时的真正登录点发生在一些没有正式文档的类和接口之间。在微软外面的世界，这些接口鲜为人知。微软的民间也不太热衷于讨论这些细节，可能是因为他们认为这些，对于使用 ASP.NET 构建程序的开发者没有太多的影响。

工作进程 `aspnet_wp.exe` (IIS5) 和 `w3wp.exe` (IIS6) 宿主在 .NET 运行时里。ISAPI DLL 通过底层的 COM 调用一小撮非托管类型的接口，其实，最终调用的是 `ISAPIRuntime` 派生类的实例。进入运行时的第一个登录点是未归档 `ISAPIRuntime` 类，它通过 COM 把接口 `IISAPIRuntime` 暴露给调用者。这些 COM

接口是底层的 `IUnknown`，基于这些接口，就意味着从 ISAPI 扩展到 ASP.NET 之间的调用属于内部调用。图 3 是使用有名的反射工具 Reflector (<http://www.aisto.com/roeder/dotnet/>) 看到的 `IISAPIRuntime` 接口的签名。Reflector 是一个可以查看和反编译程序集的工具，使用它可以很容易的查看元数据、反编译代码，就像图 3 中看到的那样。使用它一步一步地探究处理的过程，这是个非常不错的方法。

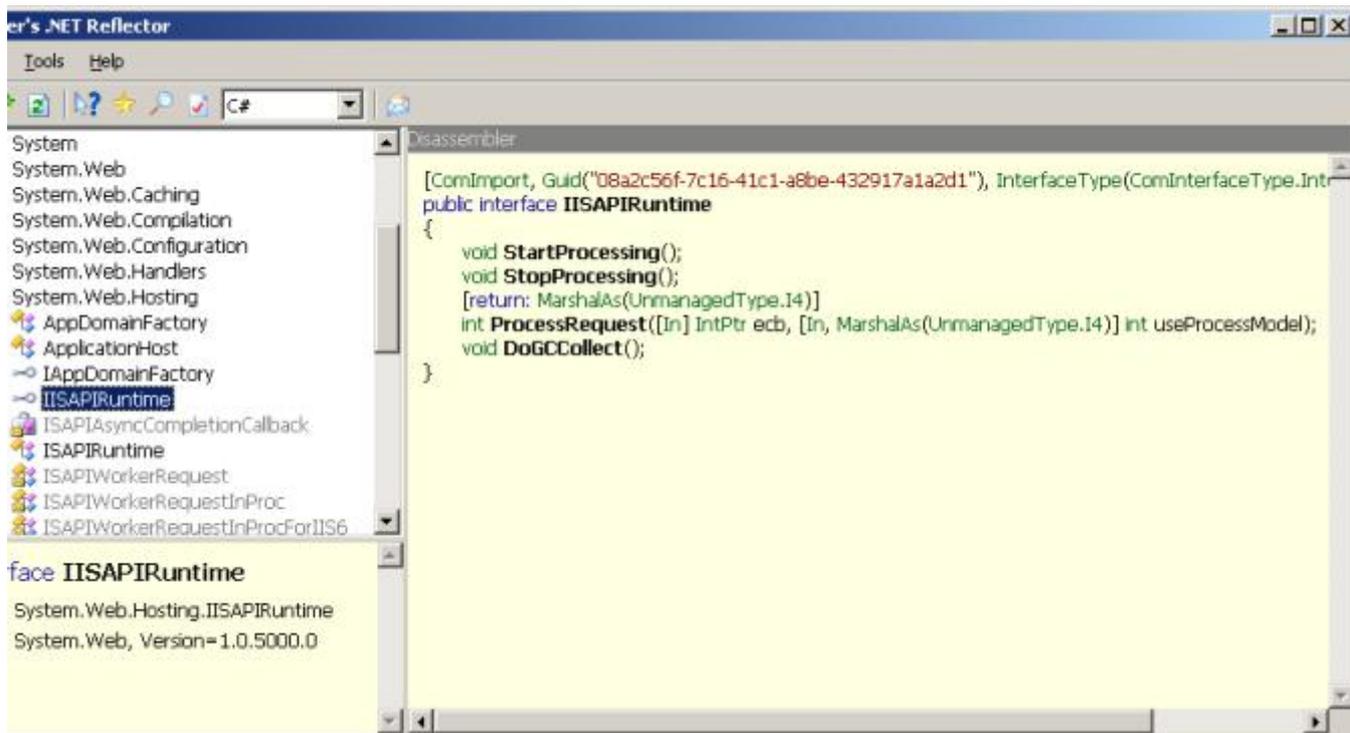


图 3: 如果你想深入了解这个底层的接口, 你可以打开 Reflector 工具, 然后指向 System.Web.Hosting 命名空间。进入 ASP.NET 的登录点以一个托管的 COM 接口出现, 该接口将在 ISAPI d11 里被调用。该登录点接收一个指向 ISAPI ECB 非托管类型的指针。ECB 拥有访问整个 ISAPI 接口的权限, 它可以获取请求的数据以及把返回的数据发回 IIS。

IISAPIRuntime 接口担当着来自于 ISAPI 扩展(在 IIS6 里是直接通信的, 在 IIS5 里间接的通过命名管道通信的)的非托管代码和托管代码之间的桥梁。如果你留意一下这个类, 你会发现 ProcessRequest 方法的签名像下面的样子:

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

ecb 参数是 ISAPI 扩展控制块(extension control block), 它被作为非托管资源传给 ProcessRequest 方法。此方法将获取 ECB, 然后把它作为基本的输入和输出接口, 用于 Request 和 Response 对象。ISAPI ECB 包含着所有底层的请求信息, 这其中包括服务器变量, 用于表单变量

(form variables) 的输入流，以及用于写数据并把数据发送到客户端的输出流中。一个单独的 ECB 引用基本上提供了一个 ISAPI 请求可以访问的所有功能。ProcessRequest 既是登录点也是登出点，在这里非托管资源最先与托管代码相联系。

ISAPI 扩展以异步的方式处理请求。所以，当 ISAPI 扩展调用了工作进程或者 IIS 的线程后，会立即返回，但会为当前有效的请求保留 ECB。因此，ECB 需要包含这样的机制，即当请求结束的时候通知 ISAPI（通过 `ecb.ServerSupportFunction` 实现），然后 ISAPI 扩展释放 ECB 资源。接着以异步的方式立即释放 ISAPI 工作线程，和卸载由 ASP.NET 托管的那个隔离的处理线程。

ASP.NET 得到 `ecb` 引用后，会在内部使用它来获取当前请求的相关信息，如服务器变量，POST 的数据以及返回输出到客户端的数据。`Ecb` 将继续存活直到这个请求结束或者 IIS 超时，在这之前，ASP.NET 将会与 `ecb` 继续保持通信。当请求结束的时候，输出的内容会写进 ISAPI 的输出流里（通过 `ecb.WriteClient()` 实现）。然后 ISAPI 扩展会被通知请求已经结束，让它知道 ECB 可以被释放了。这个执行过程是非常高效的，这是因为，.NET 类本质上只是担当着一个相当瘦小的包装器，而它包装的内容就是具有高性能的非托管 ISAPI ECB。

## 加载.NET—稍微有点神秘

让我们回到之前略过的一个话题：当请求到达时，.NET 运行时是如何被加载的。具体在哪里加载的，这是比较模糊的。关于这个处理过程，我没有找到相关的文档，由于我们现在讨论的是本地代码，所以通过反编译 ISAPI DLL 文件并把它描述出来显得不太容易。

我的最佳猜测是，在 ISAPI 扩展里，当第一个请求命中一个 ASP.NET 的映射扩展时，工作线程就会引导 .NET 运行时启动。一旦运行时存在了，非托管代码就可以为指定的虚拟目录请求一个 `ISAPIRuntime` 对象的实例，当然前提条件是，这个实例还不存在。每一个虚拟目录都会拥有一个 `AppDomain`，在 `ISAPIRuntime` 存在的 `AppDomain` 里，它将引导一个单独的程序启动。由于接口被作为 COM 可调用的方法暴露，所以实例化操作将发生在 COM 之上。

为了创建 ISAPIRuntime 的实例，当指定虚拟目录的第一个请求到达时，`System.Web.Hosting.AppDomainFactory.Create()` 方法将被调用。这将会启动程序的引导过程。这个方法接收的参数为：类型，模块名以及应用程序的虚拟路径，这些都将 ASP.NET 用于创建 AppDomain，接着会启动指定虚拟目录的 ASP.NET 程序。HttpRuntime 的根对象将会在一个新的 AppDomain 里创建。每一个虚拟目录或者 ASP.NET 程序将寄宿在属于自己的 AppDomain 里。它们仅仅在有请求到达时启动。ISAPI 扩展管理这些 HttpRuntime 对象的实例，然后基于请求的虚拟路径，把请求路由到正确的应用程序里。

## 回到运行时

这个时候，你已经拥有了一个 ISAPIRuntime 的活动实例，并且可以在 ISAPI 扩展里调用。一旦运行时启动并运行起来，ISAPI 扩展就可以调用 `ISAPIRuntime.ProcessRequest()` 方法了，而这个方法就是进入 ASP.NET 通道真正的登录点。图 4 展示了这里的流程。

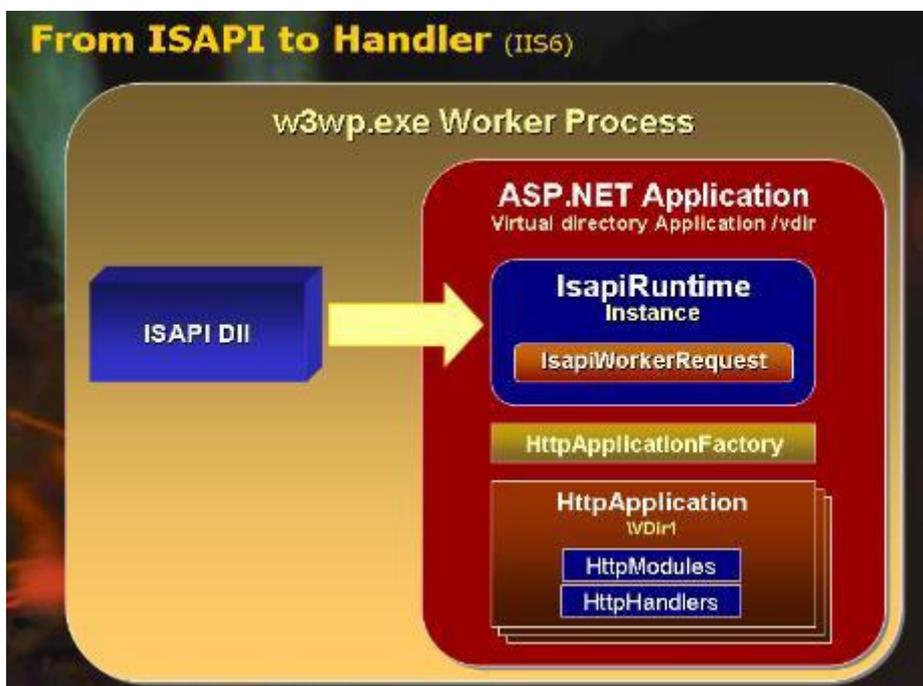


图 4：把 ISAPI 的请求转到 ASP.NET 通道需要调用很多没有正式文档的类和接口，以

及几个工厂方法。每一个 Web 程序/虚拟目录都运行在属于自己的 AppDomain 里。调用者将维护一个 IISAPIRuntime 接口的代理引用，负责触发 ASP.NET 的请求处理。

记住，ISAPI 是多线程的，因此请求可以以多线程的方式穿过 AppDomainFactory.Create() 返回的对象引用。列表 1 展现了从 IsapiRuntime.ProcessRequest 方法反编译得到的代码。这个方法接收一个 ISAPI ecb 对象和一个服务器类型参数（这个参数用于指定创建何种版本的 ISAPIWorkerRequest），这个方法是线程安全的，因此多个 ISAPI 线程可以同时安全的调用单个返回对象的实例。

列表 1: ProcessRequest 请求进入 .NET 的登录点

```
public int ProcessRequest(IntPtr ecb, int iWRTyp)
{
    // ISAPIWorkerRequest 从 HttpWorkerRequest 继承，这里创建的是
    // ISAPIWorkerRequest 派生类的一个实例
    HttpWorkerRequest request1 =
        ISAPIWorkerRequest.CreateWorkerRequest(ecb, iWRTyp
e);
    //得到请求的物理路径
    string text1 = request1.GetAppPathTranslated();
    //得到 AppDomain 的物理路径
    string text2 = HttpRuntime.AppDomainAppPathInternal;
    if (((text2 == null) || text1.Equals(".")) ||
        (string.Compare(text1, text2, true,
            CultureInfo.InvariantCulture) == 0))
    {
        HttpRuntime.ProcessRequest(request1);
        return 0;
    }
    //如果外部请求的 AppDomain 物理路径和原来 AppDomain 的路径不同，
    说明 ISAPI 维持
    //的 AppDomain 的引用已经失效了，所以，需要把原来的程序关闭，当有
    新的请求时，会
    //再次启动程序。
    HttpRuntime.ShutdownAppDomain("Physical path changed fro
```

```
m " +
                                te
xt2 + " to " + text1);
    return 1;
}
```

这里实际的代码并不重要，需要提醒的是，这里的代码是通过反编译 .NET 框架内的代码得到的，你永远也不会和这些代码打交道，而且这些代码以后可能会有所变动。这里的用意是揭示 ASP.NET 在底层发生了什么。ProcessRequest 接收了非托管参数 ecb 的引用，然后把它传给了 ISAPIWorkerRequest 对象，这个对象负责创建当前请求的内容。如列表 2 所示。

#### 列表 2: 一个 ISAPIWorkerRequest 的方法

```
// *** ISAPIWorkerRequest 里的实现代码
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;
}

// *** 再派生于 ISAPIWorkerRequest 的类
ISAPIWorkerRequestInProcIIS6 的实现// *** 代码
// *** ISAPIWorkerRequestInProcIIS6
internal override int GetQueryStringCore(int encode, StringBui
```

```
lder
buffer, int size)
{
    if (this._ecb == IntPtr.Zero)
    {
        return 0;
    }
    return UnsafeNativeMethods.EcbGetQueryString(this._ecb, enc
ode,
buffer, size);
}
```

`System.Web.Hosting.ISAPIWorkerRequest` 继承于抽象类 `HttpWorkerRequest`，它的职责是创建一个抽象的输入和输出视图，为 Web 程序的输入提供服务。注意这里的另外一个工厂方法 `CreateWorkerRequest`，它的第二个参数用于指定创建什么样的工作请求对象（即 `ISAPIWorkerRequest` 的派生类）。这里有 3 个不同的版本：`ISAPIWorkerRequestInProc`，`ISAPIWorkerRequestInProcForIIS6`，`ISAPIWorkerRequestOutOfProc`。当请求到来时，这个对象（指 `ISAPIWorkerRequest` 对象）将被创建，用于给 `Request` 和 `Response` 对象提供基础服务，而这两个对象将从数据的提供者 `WorkerRequest` 接收数据流。

抽象类 `HttpWorkerRequest` 围绕着底层的接口提供了高层的抽象（译注：抽象的目的是要把数据的处理与数据的来源解耦）。这样，就不用考虑数据的来源，无论它是一个 CGI Web Server，Web 浏览器控件还是你自定义的机制（用于把数据流入 HTTP 运行时），ASP.NET 都可以以同样的方式从中获取数据。

有关 IIS 的抽象主要集中在 ISAPI ECB 块。在我们的请求处理当中，`ISAPIWorkerRequest` 依赖于 ISAPI ECB，当有需要的时候，会从中读取数据。列表 2 展示了如何从 ECB 里获取查询字符串的值的例子。

`ISAPIWorkerRequest` 实现了一个高层次包装器方法（wrapper method），它调用了低层次的核心方法，而这些方法负责实际调用非托管 API 或者说是“服务层的实现”。核心的方法在 `ISAPIWorkerRequest` 的派生类里

得以实现。这样可以针对它宿主的环境提供特定的实现。为以后增加一个额外环境的实现类作为新的 Web Server 接口提供了便利。同样使 ASP.NET 运行在其它平台上成为可能。另外这里还有一个帮助类：`System.Web.UnsafeNativeMethods`。它的许多方法是对 ISAPI ECB 进行操作，用于执行关于 ISAPI 扩展的非托管操作。

## HttpRuntime, HttpContext 以及 HttpApplication

当一个请求到来时，它将被路由到 `ISAPIRuntime.ProcessRequest()` 方法里。这个方法会接着调用 `HttpRuntime.ProcessRequest`，在这个方法里，做了几件重要的事情（使用 Reflector 反编译 `System.Web.HttpRuntime.ProcessRequestInternal` 可以看到）。

- 为请求创建了一个新的 `HttpContext` 实例
- 获取一个 `HttpApplication` 实例
- 调用 `HttpApplication.Init()` 初始化管道事件
- `Init()` 触发 `HttpApplication.ResumeProcessing()`，启动 ASP.NET 管道处理

首先，一个新的 `HttpContext` 对象被创建，并且给它传递一个封装了 ISAPI ECB 的 `ISAPIWorkerRequest`。在请求的生命周期里，这个上下文（`context`）一直是有效的。并且可以通过静态的 `HttpContext.Current` 属性访问。正如它的名字暗示的那样，`HttpContext` 对象表示当前活动请求的上下文，因为它包含了在请求生命周期里你会用到的所有必需对象的引用，如：`Request`，`Response`，`Application`，`Server`，`Cache`。在请求处理过程的任何时候，你都可以使用 `HttpContext.Current` 访问这些对象。

`HttpContext` 对象还包含了一个非常有用的列表集合，你可以使用它存储有关特定的请求需要的数据。上下文（`context`）对象创建于一个请求生命周期的开始，在请求结束时被释放。因此，保存在列表集合里的数据仅仅对当前的请求有效。一个很好的例子，就是记录请求的日志机制，在这里，通过使用 `Global.asax` 里的 `Application_BeginRequest` 和 `Application_EndRequest` 方法，你可以从请求的开始时间至结束时间段内，对请求进行跟踪。如列表 3 所示。记住 `HttpContext` 是你的朋友，

在请求或者页面处理的不同阶段，如果你需要相关数据都可以使用它获取。

**列表 3：通过在通道事件里使用 HttpContext.Items 集合保存数据**

```
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 MilliSecs = Span.TotalMilliseconds;

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

一旦请求的上下文对象被搭建起来，ASP.NET 就需要通过一个 HttpApplication 对象，把你的请求路由到合适的程序/虚拟目录里。每一个 ASP.NET 程序都拥有各自的虚拟目录（Web 根目录），并且它们都是独立处理请求的。

## Web 程序的主要部分：HttpApplication

每一个请求都将被路由到一个 HttpApplication 对象。

HttpApplicationFactory 类会为你的 ASP.NET 程序创建一个

HttpApplication 对象池，它负责加载程序和给每一个到来的请求分发 HttpApplication 的引用。这个 HttpApplication 对象池的大小可以通过 machine.config 里的 ProcessModel 节点中的 MaxWorkerThreads 选项配置，默认值是 20。

HttpApplication 对象池尽管以比较少的数目开始启动，通常是一个。但是当同时有多个请求需要处理时，池中的对象将会随之增加。而 HttpApplication 对象池，也将被监控，目的是保持池中对象的数目不超过设置的最大值。当请求的数量减小时，池中的数目就会跌回一个较小的值。

对于你的 Web 程序而言，HttpApplication 是一个外部容器，它对应到 Global.asax 文件里定义的类。基于标准的 Web 程序，它是你实际可以看到的进入 HTTP 运行时的第一个登录点。如果你查看 Global.asax（后台代码），你就会看到这个类直接派生于 HttpApplication。

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

HttpApplication 主要用作 HTTP 管道的事件控制器，因此，它的接口主要有事件组成，这些事件包括：

- BeginRequest
- AuthenticateRequest
- AuthorizeRequest
- ResolveRequestCache
- [此处创建处理程序（即与请求 URL 对应的页）。]
- AcquireRequestState
- PreRequestHandlerExecute
- [执行处理程序。]
- PostRequestHandlerExecute
- ReleaseRequestState

- [响应筛选器（如果有的话），筛选输出。]
- UpdateRequestCache
- EndRequest

这里的每一个事件都在 Global.asax 文件中以 Application\_ 为前缀，无实现代码的方法出现。举个例子，如 Application\_BeginRequest() 和 Application\_AuthorizeRequest()。由于它们在程序中会经常用到，所以出于方便的考虑，这些事件的处理器都已经被提供了，这样你就不必再显式的创建这些事件处理器的委托了。

每一个 ASP.NET Web 程序运行在各自的 AppDomain 里，在 AppDomain 里同时运行着多个 HttpApplication 的实例，这些实例存放在 ASP.NET 管理的一个 HttpApplication 对象池里，认识到这一点，是非常重要的。这就是为什么可以同时处理多个请求，而这些请求不会互相干扰的原因。

使用列表 4 的代码，可以进一步了解 AppDomain，线程，HttpApplication 之间的关系。

列表 4: AppDomain, Threads and HttpApplication instances 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: " +
        Thread.CurrentThread.IsThreadPoolThread.ToString() +
        "<br>Thread Apartment: " +
```

```
Thread.CurrentThread.ApartmentState.ToString();  
  
// *** 为了可以同时看到多个请求一起到达，故意放慢速度  
Thread.Sleep(3000);  
}
```

这是样例程序的一部分，运行的结果如图 5 所示。为了检验结果，你应该打开两个浏览器，输入相同的地址，观察那些不同的 ID 的值。



图 5：同时运行几个浏览器，你会很容易的看到 AppDomains，application 对象以及处理请求的线程之间内在的关系。当多个请求触发时，你会看到线程和 application 的 ID 在改变，而 AppDomain 的 ID 却没有发生变化。

你将会观察到 AppDomain ID 一直保持不变，而线程和 HttpApplication 的 ID 在请求多的时候会发生改变，尽管它们会出现重复。这是因为 HttpApplications 是在一个集合里面运行，下一个请求可能会再次使用同一个 HttpApplication 实例，所以有时候 HttpApplication 的 ID 会重复。注意，一个 HttpApplication 实例对象并不依赖于一个特定的线程，它们仅仅是被分配给处理当前请求的线程而已。

线程由.NET的 ThreadPoo1 提供服务，默认情况下，线程模型为多线程单元（MTA）。你可以通过在 ASP.NET 的页面的@Page 指令里设置属性 ASPCOMPAT="true"覆盖线程单元的状态。ASPCOMPAT 意味着 COM 组件将在一个安全的环境下运行。ASPCOMPAT 使用了单线程单元（STA）的线程为请求提供服务。STA 线程在线程池里是单独设置的，这是因为它们需要特殊的处理方式。

实际上，这些 HttpApplication 对象运行在同一个 AppDomain 里是很重要的。这就是 ASP.NET 如何保证 web.config 的改变或者单独的 ASP.NET 页面得到验证可以贯穿整个 AppDomain。改变 web.config 里的一个值，将导致 AppDomain 关闭并重新启动。这确保了所有的 HttpApplication 实例可以看到这些改变，这是因为当 AppDomain 重新加载的时候，来自 ASP.NET 的那些改变将会在 AppDomain 启动的时候重新读取。当 AppDomain 重新启动的时候任何静态的引用都将重新加载。这样，如果程序是从程序的配置文件读取的值，这些值将会被刷新。

在示例程序里可以看到这些，打开一个 ApplicationPoolsAndThreads.aspx 页面，注意观察 AppDomain 的 ID。然后在 web.config 里做一些改动（增加一个空格，然后保存），重新加载这个页面（译注：由于缓存的影响可能会在原来的页面上刷新无效，需要先删除缓存再刷新即可），你就会看到一个新的 AppDomain 被创建了。

本质上，这些改变将引起 Web 程序重新启动。对于已经存在于处理管道的请求，将继续通过原来的管道处理。而对于那些新的请求，将被路由到新的 AppDomain 里。为了处理这些“挂起的请求”，在这些请求超时结束之后，ASP.NET 将强制关闭 AppDomain 甚至某些请求还没有被处理。因此，在一个特定的时间点上，同一个 HttpApplication 实例在两个 AppDomain 里存在是有可能的，这个时间点就是旧的 AppDomain 正在关闭，而新的 AppDomain 正在启动。这两个 AppDomain 将继续为客户端请求提供服务，直到旧的 AppDomain 处理完所有的未处理的请求，然后关闭，这时候才会只剩下新的 AppDomain 在运行。

## 穿过 ASP.NET 管道

HttpApplication 负责请求的传输，通过触发事件，通知应用程序正在发生的事情。这个作为 HttpApplication.Init() 方法的一部分实现的（使用 Reflector 查看 System.Web.HttpApplication.InitInternal 和 HttpApplication.ResumeSteps() 的代码可以看到这一点）。在这个方法里，创建和启动了一系列事件，包括绑定事件处理器。Global.asax 里的事件处理器会自动映射到对应的事件，它们也可以映射到额外添加的 HTTPModules，这些 HTTPModules 本质上是 HttpApplication 已发布事件的一种扩展。

通过在 web.config 里注册，HTTPModules 和 HttpHandlers 可以被动态的加载，并且可以添加到事件链条上。HTTPModules 实际上就是事件处理器，它可以钩住指定 HttpApplication 的事件。而 HttpHandlers 就是一个端点，它可以被调用处理“应用程序级的请求处理”。

HTTPModules 和 HttpHandlers 将被加载，然后添加到调用链上作为 HttpApplication.Init() 方法调用的一部分。图 6 展示了不同的事件，这些事件何时被触发以及通道上的哪些部分会受到它们的影响。

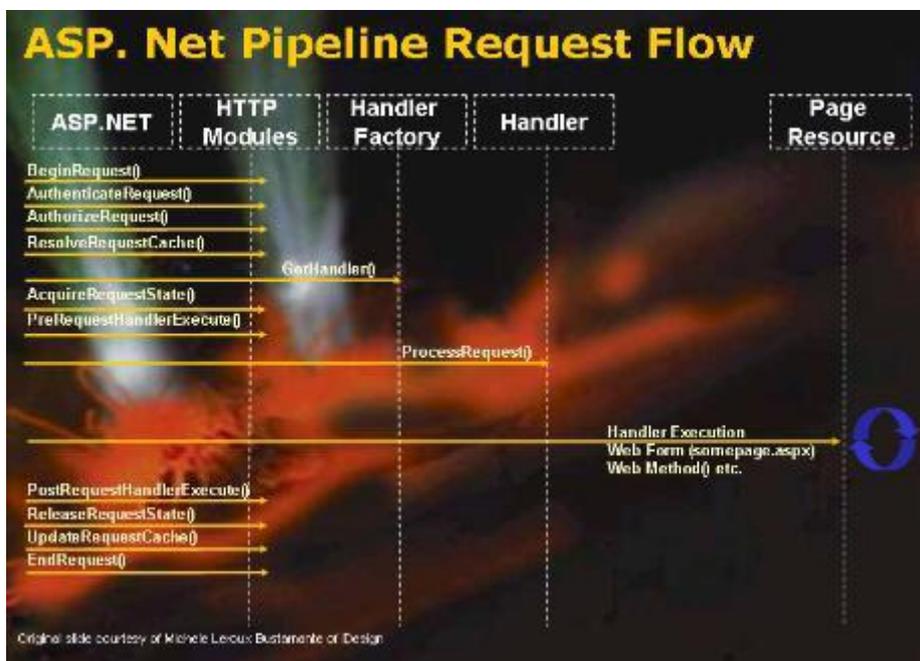


图 6：事件在 ASP.NET HTTP 管道里传输。HttpApplication 对象的事件驱动请求在管道里传输。HTTPModules 可以截获这些事件，可以覆盖或增强已存在的功能。

## HttpContext, HttpModules 和 HttpHandlers

HttpApplication 本身并不知晓发送给 Web 程序的数据。它仅仅是个消息邮递者，只负责事件之间的通信。它触发事件，然后通过传递 HttpContext 对象，把信息发送给被调用的方法。在之前我们提到，当前请求的数据是在 HttpContext 对象里保存。它提供了请求从开始到结束需要的所有数据。图 7 展示了 ASP.NET 的管道之间的传输流程。注意，从请求的开始至结束，上下文对象 (Context) 都是你的伙伴，你可以在一个事件方法里使用它保存数据，然后在之后的事件方法里获取这些数据。

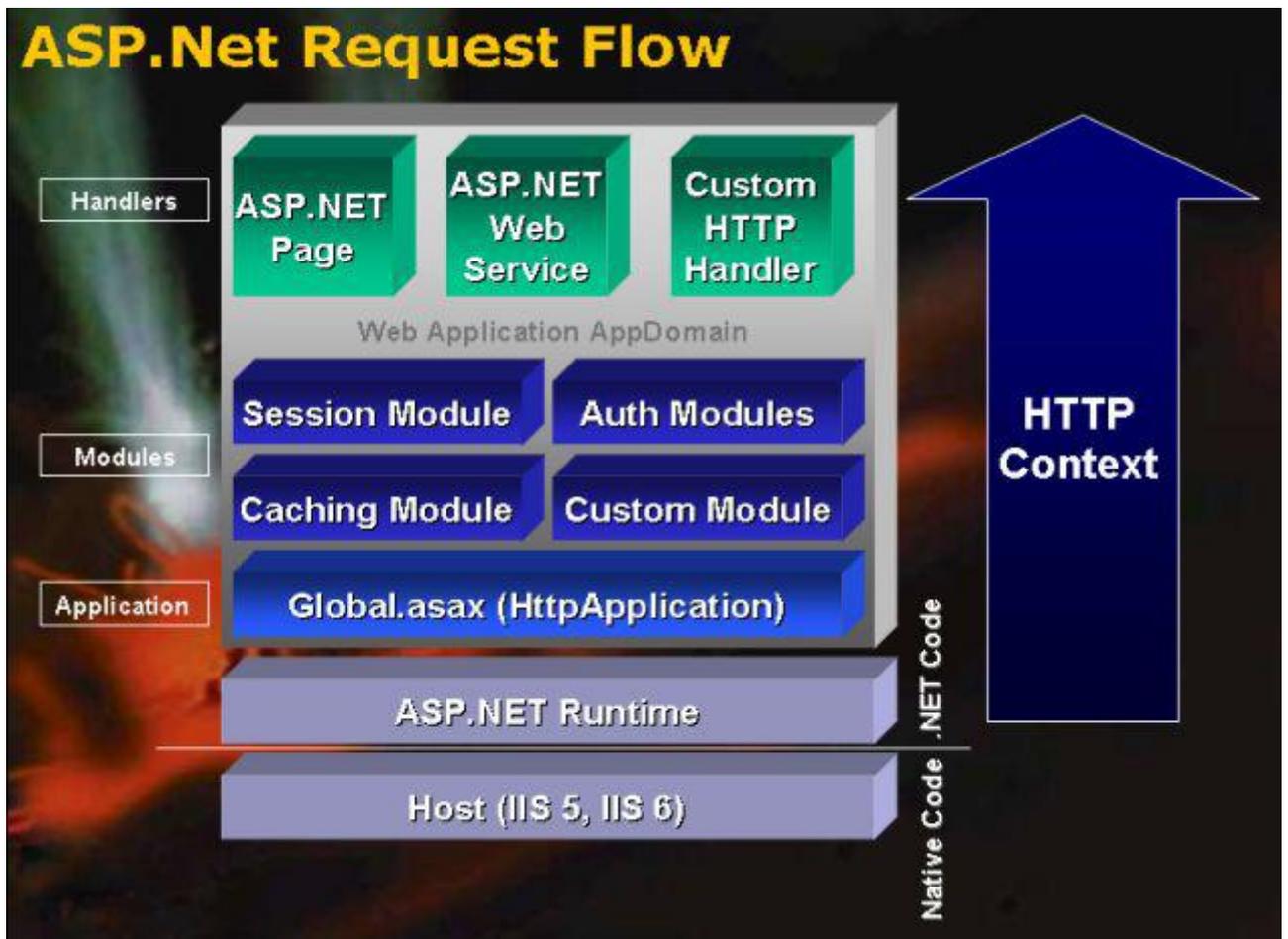


图 7: ASP.NET 管道在一系列事件接口之间传输请求，这提供了足够的灵活性。

HttpApplication 担当主容器，负责加载 Web 程序，当请求到来时触发事件以及在管道之间传输请求。经过已配置的 HTTP 过滤和模块时，每一个请求都将遵循一个公有的路径。过滤器可以检查穿梭在管道里的每一个请求，而处理器允许实现应用程序的逻辑或者应用程序级的接口像 WebForms 和 WebServices 一样。为了给程序提供输入

和输出，上下文对象（Context）给请求提供了所需的信息，它贯穿了请求生命周期的始终。

ASP.NET 管道一旦启动，HttpApplication 将逐一触发事件，如图 6 展示的那样。每一个事件都将被触发，如果事件绑定了事件处理器，那么这些事件处理器将被调用，执行它们的任务。这个过程的主要目的是通过调用 HttpHandler 处理指定的请求。对于 ASP.NET 请求而言，HttpHandler 是处理请求机制的核心，在这里任意的应用程序级的代码被执行。记住，ASP.NET 的页面和 Web Service 都是 HttpHandler 的具体实现，在这里，所有请求处理的核心功能被实现。HttpModule 则倾向于在分发给事件处理器之前或者之后对内容进行处理。在 ASP.NET 里典型的默认操作有：鉴定（Authentication），处理前的缓存操作以及各种处理后的编码操作机制。

关于 HttpModule 和 HttpHandler，这里有很多有用的信息，但为了保持这篇文章合理的尺度，我将仅仅讲述关于它们一些简短的、整体的看法。

## HttpModules

伴随着 HttpApplication 触发的一系列事件，请求将会在管道之间穿梭。你已经看到了这些发布的事件，在 Global.asax 里都有对应事件的处理方法。这个方法（步骤）是程序（ASP.NET 应用程序）携带的，尽管这个并不总是你需要的。如果你想构建一套通用的 HttpApplication 事件处理程序，并以插件的形式添加到任意的 Web 程序里。那么你可以使用 HttpModule，它是可重复使用的，不需要添加任何实现代码就可以在其它程序里使用，而你所做的仅仅在 web.config 里注册。

模块本质上是过滤器，在功能上类似于 ASP.NET 请求级别的 ISAPI 过滤。对于每一个穿过 ASP.NET 的 HttpApplication 对象的请求，模块都允许在 HttpApplication 对象触发的事件处理方法里截获这些请求。这些模块以类的形式存储在外部程序集里，可以在 web.config 里配置，当程序启动的时候加载。通过实现指定的接口和方法，模块就可以被添加到 HttpApplication 的事件链上。多个 HttpModules 可以钩住相同的事件，事件发生的顺序是它们在 web.config 里声明（配置）的顺序。如下就是在 web.config 里，一个模块的声明。

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

注意，在这里你需要指定一个完整的类型名和一个不带扩展名的程序集的名字。

模块允许你查看每一个传入的 Web 请求，基于触发的事件基础上执行操作。模块是非常有用的，它可以修改请求，输出响应的内容以及提供自定义的身份验证，另外还可以在特定的程序里，针对 ASP.NET 的每一个请求提供响应前处理和响应后处理。许多 ASP.NET 的特征像身份验证，会话引擎都是作为 HTTP 模块实现的。

HttpModules 感觉有点类似于 ISAPI 过滤器，是由于它们查看进入 ASP.NET 程序的每一个请求，但它们局限性于仅可以查看映射到某一个 ASP.NET 程序或者虚拟目录的请求，和映射到 ASP.NET 的请求。因此，你仅可以查看所有的 ASPX 页面或者任意其它自定义的已经映射到这个程序的扩展名（译注：作者可能漏掉了 ASMX，ASHX 等，这里的意思应该是所有的 ASP.NET 默认的扩展名）。但是，你不能查看标准的 .HTM 或者图像文件，除非你通过添加这些扩展名，明确的把它们映射到 ASP.NET 的 ISAPI DLL，如图一中那样。对于模块，一个常见的用处是过滤一个指定的文件夹的 JPG 图片内容，然后使用 GDI+ 在每一张返回的图片上方添加“样图”字样。

实现一个 HTTP 模块是非常简单的：你必须实现一个 IHttpModule 接口，它包含两个方法：Init() 和 Dispose()。传递的事件参数中包含着一个 HttpApplication 对象的引用，接着，它会给你访问 HttpContext 对象的权限。在这两个方法里，你可以钩住 HttpApplication 的事件。举个例子，如果你想用一个模块钩住 AuthenticateRequest 事件，那么你需要做的会像列表 5 中展示的那样。

列表 5: 一个 HTTP 模块实现起来非常的简单

```
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，你的模块已经有访问 `HttpContext` 对象的权限了。从这里到所有其它内置的 ASP.NET 管道对象如 `Response` 和 `Request`，因此你可以获取输入的数据等等。但紧记，某些对象现在可能不能使用，只有到这条链的后面的环节才会有效，

在 `Init()` 方法里，你可以钩住多个事件，因此在一个模块里，可以管理多个不同功能的操作。但是，应该尽可能的把不同的逻辑代码放到不同的类里，这样可以确保这个模块是标准的组件。在许多情况下，你实现的功能可能需要钩住多个事件。举个例子，一个日志过滤器可能需要在 `BeginRequest` 里记录请求的开始时间，在 `EndRequest` 里记录请求完成的时间。

在 `HttpModules` 里使用 `HttpApplication` 的事件时，有一点是需要注意的，`Response.End()` 和 `HttpApplication.CompleteRequest()` 方法会使 ASP.NET 跳过 `HttpApplication` 和模块的事件链。可以查看这两个方法的帮助文档获取更多的信息。

## HttpHandlers

模块是相当低层次的，它针对每一个传入 ASP.NET 程序的请求触发。HTTP 处理器则着重于处理一个指定的请求映射。通常一个页面扩展已经被映射到处理器了。

实现一个 HTTP 处理器所需要做的是非常基础的，但是通过访问 `HttpContext` 对象，就会有很多有用的功能。HTTP 处理器通过一个简单 `IHttpHandler` 接口实现（或者它的异步版本 `IHttpAsyncHandler`）。它仅仅有一个方法 `ProcessRequest()` 和一个属性 `IsReusable`。这里的关键是 `ProcessRequest()` 会得到一个 `HttpContext` 对象的实例。这个单独的方法将从开始到结束负责处理一个 Web 请求。

单一的，简单的方法？可能太简单了，是吗？可是，一个简单的接口，但它的实现可能并不简单。记住，`WebForms` 和 `WebServices` 都是作为 HTTP 处理器实现的。因此，在这个看似简单的接口里，过多的实现过程被隐藏了。关键是，事实上到现在，一个 HTTP 处理器可以访问所有为了开始处理请求而被组建和配置起来的 ASP.NET 的内置对象。关键是，`HttpContext` 对象提供了所有与请求相关的功能，可以获取流入的数据和输出数据到 Web 服务器。

对于 HTTP 处理器而言，所有的操作都通过简单地调用 `ProcessRequest()` 方法执行。可以简单到如下的样子：

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

一个完整的实现像 `WebForms` 页面引擎那样，可以根据 HTML 模版展现复杂的表单。所以，这里的关键点是你想用这个简单但功能强大的接口做什么。

因为对你而言，HttpContext 对象是可以使用的，这样你就可以访问 Request, Response, Session 和 Cache 对象，因此你已经拥有了 ASP.NET 请求的所有特征，可以自己做主如何处理用户提交的信息，然后给客户端返回处理后产生的内容。记住，在一个 ASP.NET 请求的生命期内，上下文对象始终都是你的朋友。

处理器的关键操作通常是往 Response 对象里写输出数据，或者更确切的说，是往 Response 对象的 OutputStream 里写。这个就是真正返回到客户端的输出数据。在底层，由 ISAPIWorkerRequest 负责把 OutputStream 发回给 ISAPI 的 ecb.WriteClient 方法，因为 ecb.WriteClient 方法才是真正执行 IIS 产生输出数据的。

通过使用大量的处于高层的、基础的框架接口，WebForms 实现了一个 HTTP 处理器。但最后，WebForm 的 Render() 方法却简单的使用了一个 HtmlTextWriter 对象，把最终的输出发送给 context.Response.OutputStream 对象而结束。因此，相当的奇妙，最终甚至一个高层次的工具像 WebForm，也仅仅是在 Response 和 Request 对象之上的一个高层次的抽象。

在这个时候，你可能想知道，是否需要从头实现一个完整的 HTTP 处理器？毕竟 WebForms 已经提供了一个易用的 HTTP 处理器的实现，因此为什么还要为这些大量底层的東西而烦恼，放弃这些已经提供的灵活性呢？

WebForms 是非常棒的，使用它可以生成复杂的 HTML 页面和处理业务层的逻辑而这些都是需要图形化的设计和模版化的页面。除此以外，WebForms 引擎还可以完成更多的，需要丰富的表现层的任务。如果所有你想做的是：从系统读取文件，由执行的代码把它返回。这样的任务，如果避开使用 Web Forms 页面框架，代替的是直接处理文件然后返回，相信后者才是更高效的方法。如果你做的事情仅仅是像从数据库里读取图片一样，那么完全没有必要使用页面框架来处理，因为这里的确没有 Web UI 需要你俘获离开图片的事件。

在这里找不到，组建一个页面对象和会话对象以及俘获页面层上事件的理由。对于你的任务而言，这里需要做的仅仅是执行代码，而这些与你手头上的任务毫不相干。

因此，这种情况下使用处理器会更加高效。处理器可以完成使用 WebForms 不可能完成的事情。比如：需要处理这样的请求，它们没有必要在磁盘上存在对应的物理文件，这些被请求的路径通常称为虚拟的 URL。为了使这样的请求正常的工作，你需要确保已经在应用程序的扩展对话框（如图一所示）里关闭了“确认文件是否存在”的复选框。

对于内容提供者来说，这是通用的，如：动态的图像处理，XML 服务，提供虚拟的 URL 使原来的 URL 重定向，下载管理等等，这些均不能使用 WebForms 实现。

## 是否已经提供了足够的底层知识？

哎呀！我们终于绕着请求的处理周期回到了原地。尽管我在这里没有探讨有关 HTTP 模块和 HTTP 处理器如何工作的更多细节，但仍旧提供了许多对你有帮助的底层信息。挖掘这些信息花费了我很多时间，通过了解 ASP.NET 在底层的工作模式，使我感到非常地满足，希望它也可以给你带来同样的感受。

在结束之前，还是让我们来回顾一下，我在这篇文章中讨论的从 IIS 到 HTTP 处理器的事件序列：

- IIS 得到一个请求
- 查询脚本映射扩展，然后把请求映射到 `aspnet_isapi.dll` 文件
- 代码进入工作者进程（IIS5 里是 `aspnet_wp.exe`；IIS6 里是 `w3wp.exe`）
- .NET 运行时被加载
- 非托管代码调用 `IsapiRuntime.ProcessRequest()` 方法
- 每一个请求调用一个 `IsapiWorkerRequest`
- 使用 `WorkerRequest` 调用 `HttpRuntime.ProcessRequest()` 方法
- 通过传递进来的 `WorkerRequest` 创建一个 `HttpContext` 对象
- 通过把上下文对象作为参数传递给 `HttpApplication.GetApplicationInstance()`，然后调用该方法，从应用程序池中获取一个 `HttpApplication` 实例。
- 调用 `HttpApplication.Init()`，启动管道事件序列，钩住模块和处理器
- 调用 `HttpApplication.ProcessRequest`，开始处理请求

- 触发管道事件
- 调用 HTTP 处理器和 ProcessRequest 方法
- 把返回的数据输出到管道，触发处理请求后的事件

使用手边的例子将会更容易记住这些零碎的片断是如何组合起来的。为了记住它，我会不时地看一下它。现在应该回到工作中了，去做一些不太抽象的事情吧。

尽管讨论的这些是基于 ASP.NET 1.1 的。但这里描述的底层处理在 ASP.NET 2.0 好像并没有多大改变。

最后，非常感谢来自微软的 Mike Volodarsky，是他校验了这篇文章，并且提出了一些宝贵的意见。还有 Michele Leroux Bustamante，他为 ASP.NET 管道请求流程的幻灯片提供了依据。