**HttpContext**

**HttpContext介绍**

**-> HttpContext是什么**

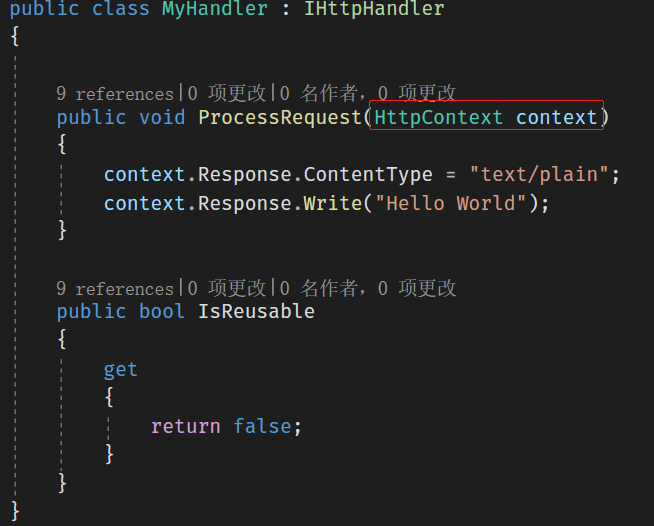
1. HttpContext是ASP.NET提供给我们的一个类，里面封装了当前这个会话中HTTP请求的上下文信息，它贯穿了整个HTTP RunTime的请求生命周期，并在不同的流程中由不同的处理单元（HttpModule、HttpApplication、HttpHandler）进行处理

**-> HttpContext是什么时候创建出来的**

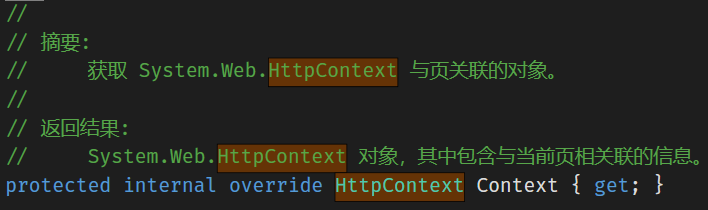
1. 一个HTTP请求的上下文信息从Unmanaged Environment传递到Managed Environment中是以一个封装了HTTP请求上下文信息的指针（EPC）来作为沟通桥梁的，当这个EPC指针通过【System.Web.Hosting.ISAPIRuntime.ProcessRequest】进入ISAPI Runtime的时候，ASP.NET就会为这个EPC创建一个【HttpWorkerRequest】的类，并携带着这个类进入到HTTP RunTime中，最后会通过【System.Web.HttpRuntime.ProcessRequestInternal】函数去创建一个【HttpContext】

**-> 如何在一个HTTP请求会话中获取到HttpContext**

1. \*.ashx：因为一般处理程序是一个实现了IHttpHandler接口的类，所以当我们创建一个一般处理程序的时候会默认带有一个【ProcessRequest】函数让我们去编写内部的实现，而这个【ProcessRequest】函数中会有一个【HttpContext】的Siganature让我们去获取【HttpContex】的实例信息



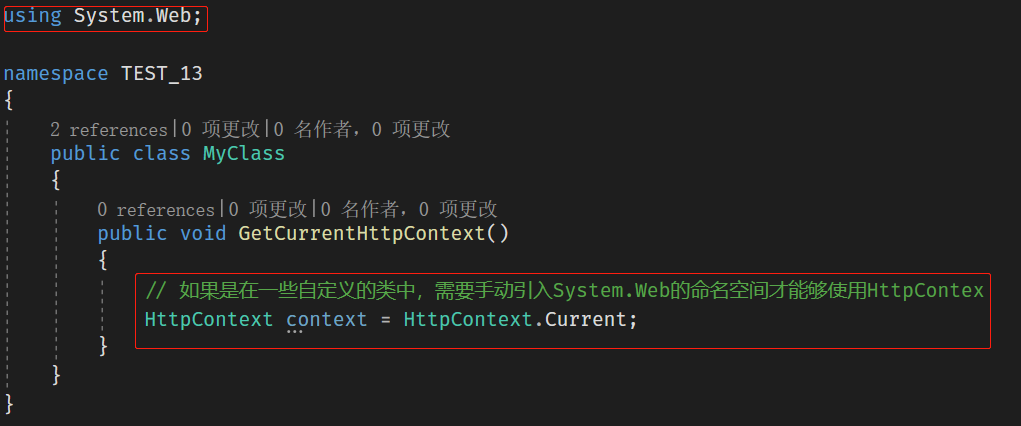
1. \*.aspx：WebForm的父类Page实现了IHttpHandler接口，他也有属于自己实现的【ProcessRequest】函数，并且在其内部也封装了名为【Context】的属性，我们可以通过这一属性去获取【HttpContext】的实例信息



1. 通过【HttpContext.Current】获取当前应用程序域HTTP会话的【HttpContex】

-> 在其它类中使用需要手动的导入【System.Web】的命名空间

-> 【HttpContext.Current】只能够获取当前请求线程的HTTP上下文的实例，因为每一次请求.NET Framework都会为当前请求分配一个线程进行运行，我们之所以处处都能够获取到HttpContext的原因是因为代码都是运行在一个线程内的，但是如果我们在一个我们所创建的线程中获取该实例的时候，会返回一个【null】



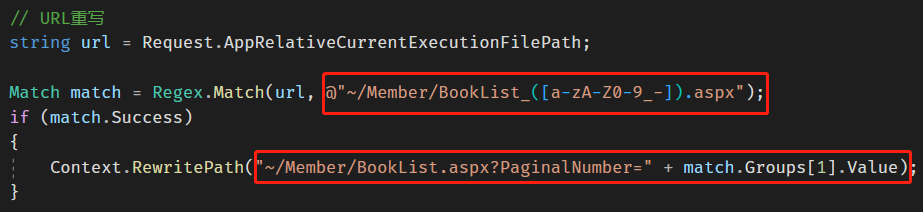
**HttpContext的使用**

**-> 关于该篇文档中的注意事项**

1. 在本篇中所出现的【键值对集合】均不是以往的【Dictionarty<T,T>】，而是一种类似【Dictionarty<T,T>】的集合，其添加、获取元素的方式和原来的【Dictionarty<T,T>】是一模一样的，并且我们可以使用一些【Dictionarty<T,T>】的成员诸如：Add()、Remove()、Count…。需要我们注意的是，**当我们操作这里的【键值对集合】访问一个不存在的键的时候，并不会抛出异常，而是会返回一个null，并且当我们为一个不存在的键直接赋值的时候，就自动为【键值对集合】添加当前键的元素，区别于【Dictionarty<T,T>】，当我们想要操作或访问一个不存在的键的时候就直接抛异常了**

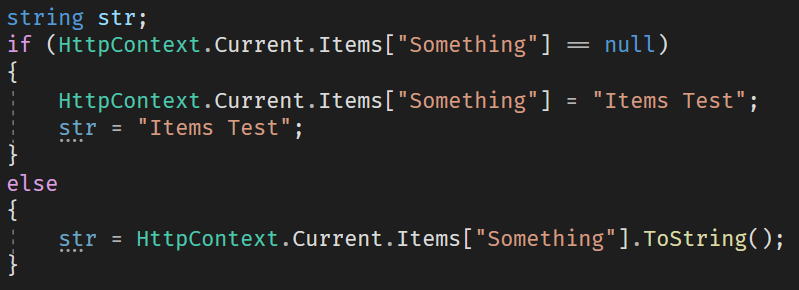
**-> 函数**

1. Httpcontext.RewritePath(string rewriteUrl)：重写当前所请求的URL路径，使HttpHandler在创建的时候创建我们所重写的新的Url的资源实例（例如：重写前的Url为MyHandler.ashx，重写后的Url为Page.aspx，那么在请求管道第8个事件中创建HttpHandler进行后续处理的时候就只会创建我们所重写的Page.aspx的实例了），并且更新HttpContext中相应的成员为我们所重写新的Url的相应参数的体现（例如：重写前的Url没有带Get请求的参数格式，则重写前的HttpContext.Request.QueryString是什么都没有的，而重写后的Url添加了Get请求参数的格式，则重写后的HttpContext.Request.QueryString也会有相应的体现）



**-> HttpContex中的成员**

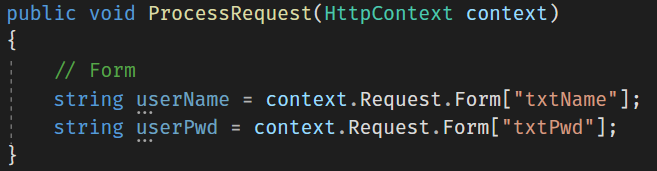
1. Httpcontext.Items：用于存储只是 **应用于当前上下文** 所需要存储的一些数据，可以把这个数据看作是一个只是应用于当前线程的一个静态存储集



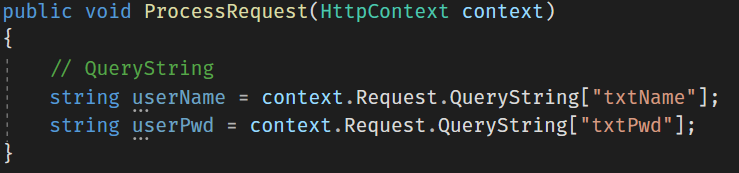
1. Httpcontext.Request：用于操作当前HTTP的请求信息

**-> 属性**

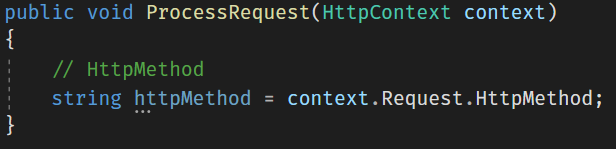
1. HttpContext.Request.Form：获取以【Post】方式提交数据的【键值对集合】



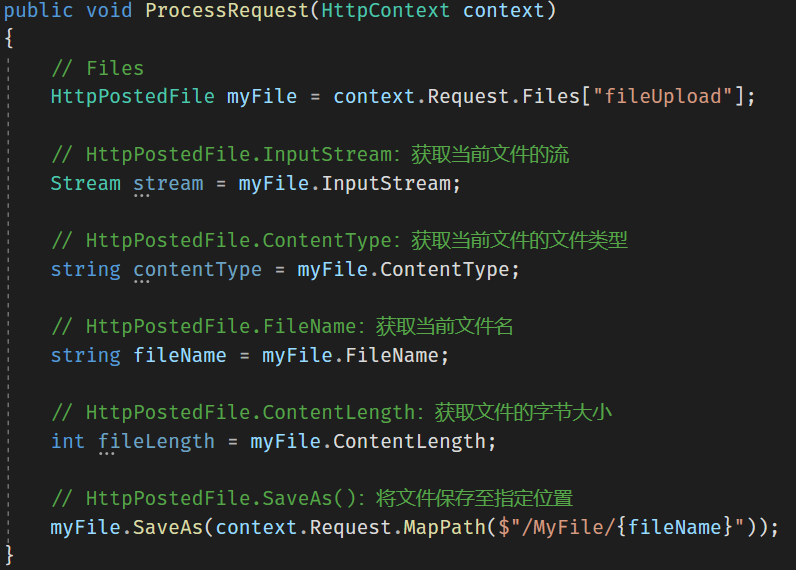
1. HttpContext.Request.QueryString：获取以【Get】方式提交数据的【键值对集合】



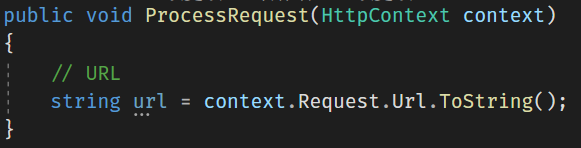
1. HttpContext.Request.HttpMethod：获取当前HTTP请求方法名



1. HttpContext.Request.Files：获取由客户端通过文件上传域表单元素所提交至服务器的文件【键值对集合】【HttpFile Collection】，所上传的文件最终经过ASP.NET会被封装在【HttpPostedFile】的类中（【HttpFileCollection】值的类型）

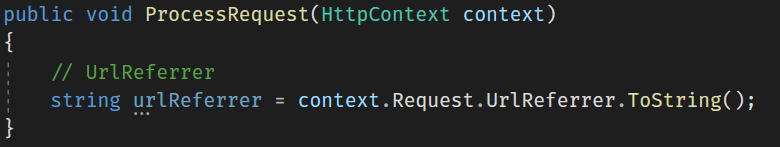


1. HttpContext.Request.Url：获取当前所请求的地址

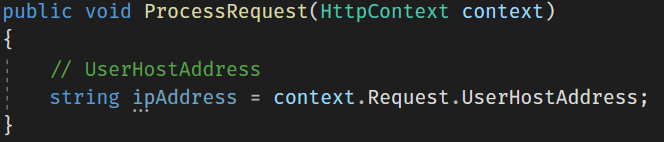


1. HttpContext.Request.UrlReferrer：获取请求的来源地址，也就是上一次所请求的地址

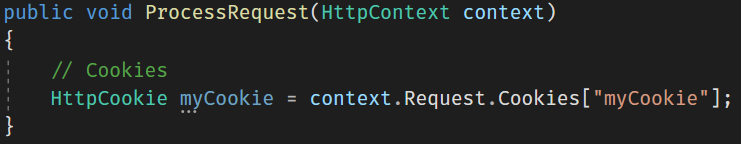
-> 此属性可用于防盗链，但是只能用于最基本的防盗链的手段，因为其实现原理是根据HTTP请求头部字段【Referer】来获取的，但是请求首部字段是可以通过编程人员进行手动窜写伪造的，所以要百分之百解决防盗链的问题是一件十分困难的事情



1. HttpContext.Request.UserHostAddress：获取发送请求主机的IPv6格式的地址



1. HttpContext.Request.Cookies：获取从请求报文中所接收到的Cookie的【键值对集合】【HttpCookieCollection】 ，该集合的键类型为【HttpCookie】，关于此类型的更多详解请查看下一节点【关于其他成员】中【HttpCookie】的描述

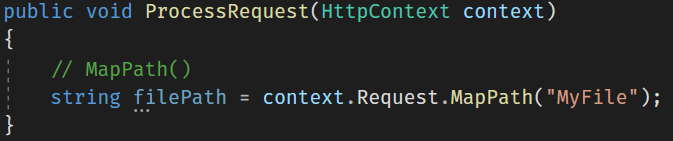


1. HttpContext.Request.AppRelativeCurrentExecutionFilePath：获取当前所请求资源的虚拟路径，并且开头以"~"号作 为开头，如：~/Member/Booklist\_1.aspx

微信截图_20190717235950

**-> 函数**

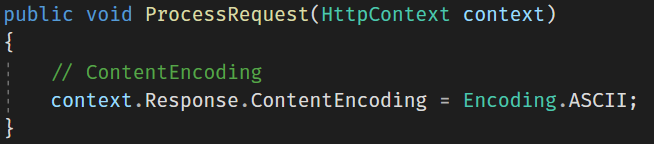
1. HttpContext.Request.MapPath(string virtualPath)：根据虚拟路径【virtualPath】和当前所请求资源文件所在的绝 对路径拼接成一个新的绝对路径



1. Httpcontext.Response：用于操作当前HTTP的响应信息

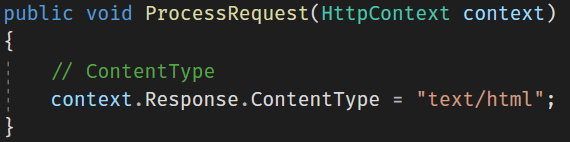
**-> 属性**

1. HttpContext.Response.ContentEncoding：设置HTTP响应报文内容的编码格式



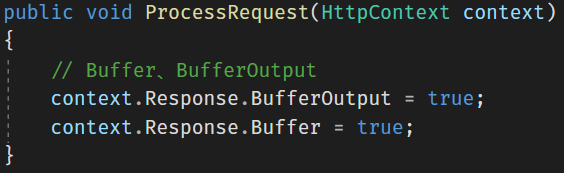
1. HttpContext.Response.ContentType：指定HTTP响应报文的实体数据类型

-> 关于更多的实体数据类型请[查看这里](../HTTP协议/Content-Type.doc)

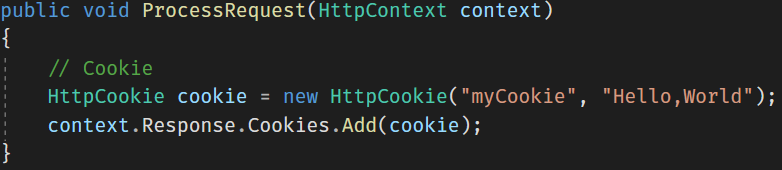


1. HttpContext.Response.Buffer/BufferOutput：设置是否启用响应缓存模式进行输出

-> 为提高服务器的性能，默认情况下ASP.Net向浏览器输出数据的时候并不会立即输出到浏览器，而是会将数据缓存至 缓冲区中，到合适的时机或者响应结束才会将缓冲区中的数据一起发送到浏览器

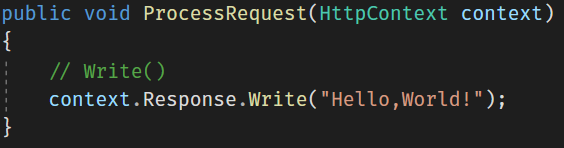


1. HttpContext.Response.Cookies：操作需要发送响应给客户端的Cookie的【键值对集合】【HttpCookieColl ectio n】，该集合的键类型为【HttpCookie】，关于此类型更多详解请查看下一节点【关于其他成员】中【HttpCookie】的描述

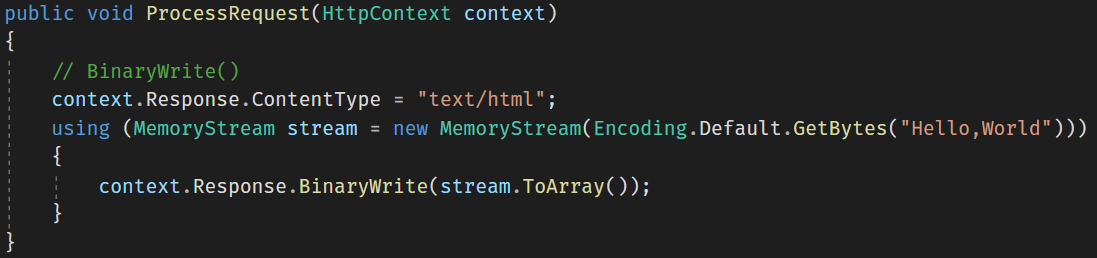


**-> 函数**

1. HttpContext.Response.Write(string content)：把字符串content写入至HTTP响应体



1. HttpContext.Response.BinaryWrite(byte[] buffer)：把二进制数据写入至HTTP响应体中

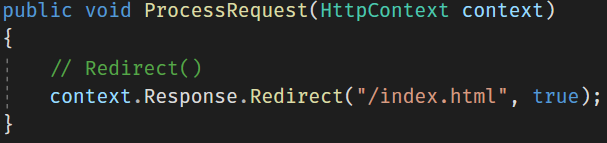


1. HttpContext.Response.Redirect(string url,bool endResponse)：指定重定向的URL，并根据endResponse来确 定是否立刻结束之后的代码开始一次重定向的流程

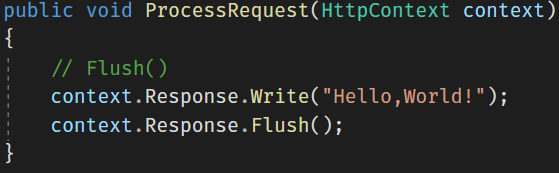
-> 重定向：当浏览器接收到的一段响应状态码为[302]且响应头中带有[Location]字段的响应报文的时候，浏览器则会根 据【Location】字段所指定的位置再次发送一次新的GET请求，这就是重定向的流程

-> Redirect函数的底层实现原理是通过抛出一个[ThreadAbortException]的异常来结束当前函数下文的代码的

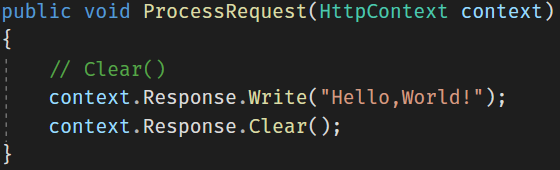
-> 在MVC开发中，就算是设置了endResponse为true，后续的代码还是会执行，这涉 及到MVC的工作原理的问题



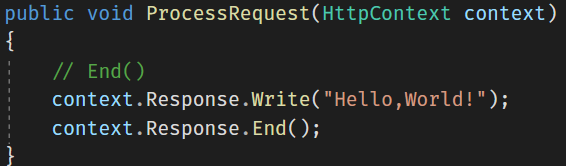
1. HttpContext.Response.Flush()：立刻将缓冲区的数据发送至浏览器



1. HttpContext.Response.Clear()：清空缓冲区中等待发送至浏览器的数据



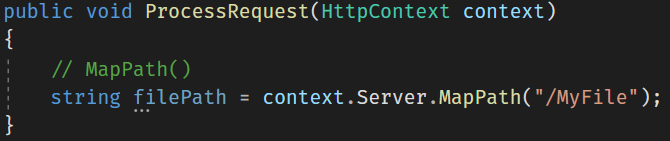
1. HttpContext.Response.End()：抛出一个异常，终止当前HttpRuntime的线程，并且立刻返回响应信息给Client



1. HttpContext.Server：用于一些服务器上的功能性的操作

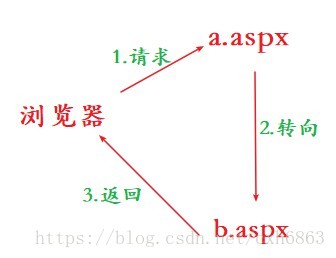
**-> 函数**

1. HttpContext.Server.MapPath(string virtualPath)：根据虚拟路径【virtualPath】和当前所请求资源文件所在的绝 对路径拼接成一个新的绝对路径

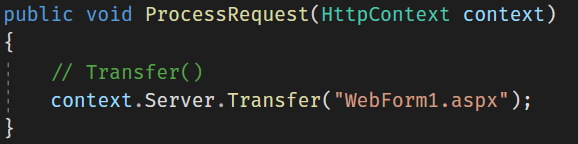


1. HttpContext.Server.Transfer(string virtualPath)：停止在调用该函数后下文代码内容输出，并保留在调用该函数前所 执行的结果，跳转至虚拟路径virtualPath所指定的网页文件并处理里面的内容，最后，所跳转的网页文件结合上一个网页 文件所保留的结果合并发送回给浏览器进行内容的渲染解析

-> 实现图

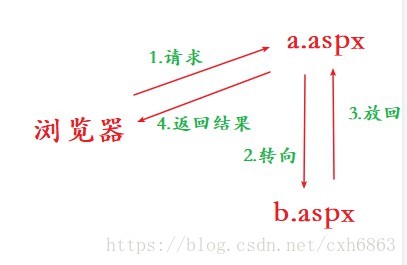


-> 代码

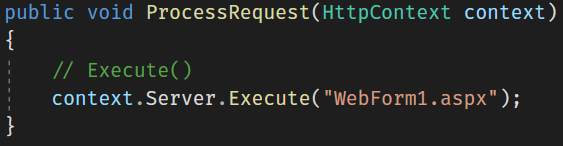


1. HttpContext.Server.Execute(string virtualPath)：跳转并执行虚拟目录virtualPath所指定的网页文件并读取里面的 内容，并把结果返回给调用该函数的网页文件，最后调用该函数的网页文件再把合并两个网页的执行结果一起发送回给浏 览器进行渲染解析

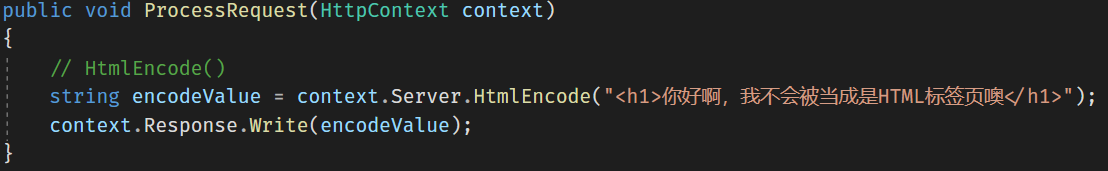
-> 实现图



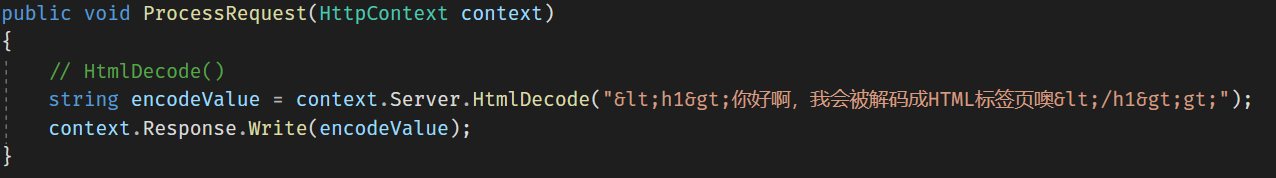
-> 代码



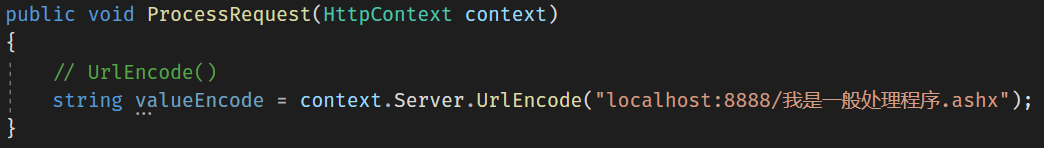
1. HttpContext.Server.HtmlEncode(string encodeValue)：将标准的HTML字符串encodeValue进行转义符的编码处 理，返回一种不会被浏览器当成是HTML标签处理的一种普通格式的字符串，该函数可作为跨站脚本攻击XSS的一种防御 手段



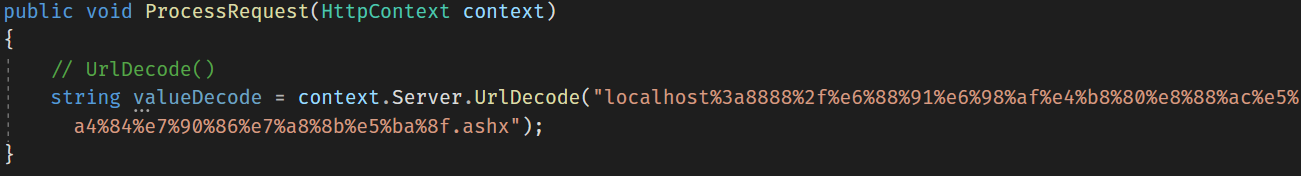
1. HttpContext.Server.HtmlDecode(string decodeValue)：将已经被转义符编码处理过的字符串decodeValue解码回 一种标准格式的HTML字符串并返回



1. HttpContext.Server.UrlEncode(string urlEncode)：将字符串urlEncode中所存在的中文字符进行编码处理并返回， 在低版本的浏览器当中，URL是不允许出现中文的需要我们手动的进行转换才能够访问存在中文的域名，而高版本的浏览 器会自动地为我们吧中文字符进行编码处理



1. HttpContext.Server.UrlDecode(string urlDecode)：将中文字符已经被编码过的字符串urlDecode进行解码处理并 返回



1. HttpContext.Server.GetLastError()：获取最后一次所抛出的异常信息

-> 该方法能够用于获取Web Application中最后一次所抛出的异常信息，通常配合在【Global.asax】中的Application Error Event Handler所使用

-> 如果一个Web Application中的一段代码抛出了一个异常，我们可以不在当前上下文中使用 try/catch 进行异常的捕获，而是交给[Application Error Event Handler](HttpModule和HttpHandler.docx)，通过它的内部再去调用【GetLastError】函数，去获取最后后一次所引发的异常信息，这样我们就可以不用在我们的代码中到处去写 try/catch 进行针对性的异常信息捕获了



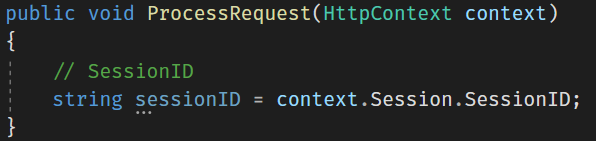
1. HttpContext.Session：用于操作Session

-> ASP.NET中操作Session必须要显式/隐式的继承自 【System.Web.SessionState.IRequiresSessionState】这个接口，否则该属性的值为null

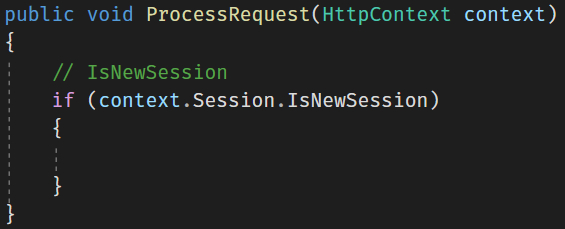
-> 关于Session的更多信息，请[查看这里](../../WEB前端/Session/Session.docx)

**-> 属性**

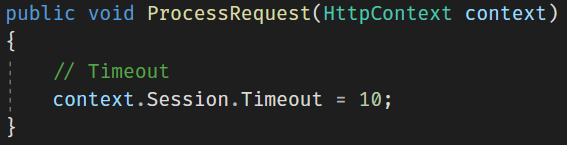
1. HttpContext.Session.SessionID：获取当前会话的SessionID



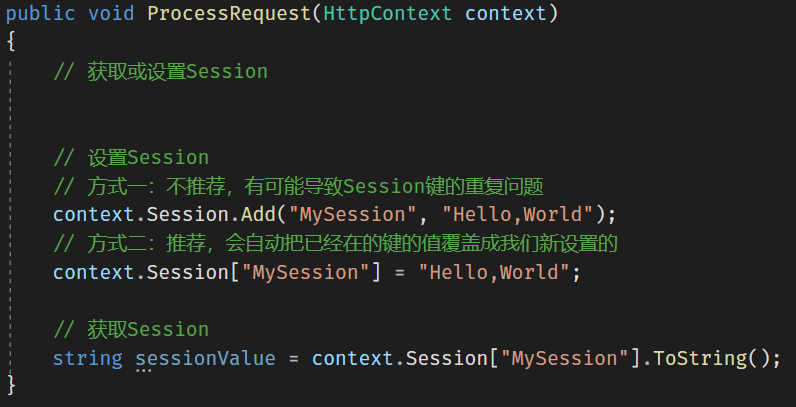
1. HttpContext.Session.IsNewSession：获取当前SessionID是否是一个新的SessionID，也就是检验此次HTTP请求 是否是一次新的会话



1. HttpContext.Session.TimeOut：获取或设置当前SessionID存储在Session池中的过期时间



**-> 添加或获取指定键的Session的值**



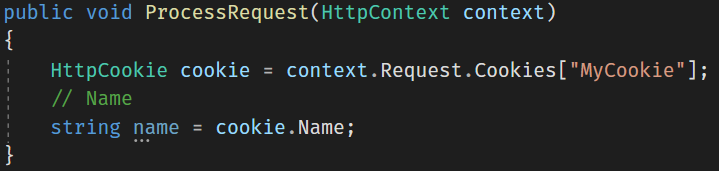
**-> 关于其他成员**

1. HttpCookie：【HttpCookie】为ASP.NET封装了用于操作、获取Cookie的信息

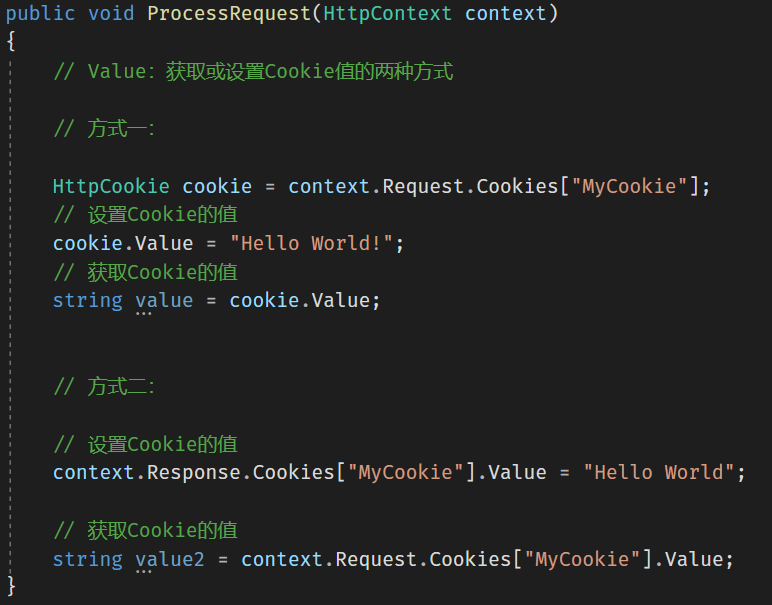
-> 关于Cookie的更多信息，请[查看这里](../../WEB前端/Cookie/Cookie.docx)

**-> 属性**

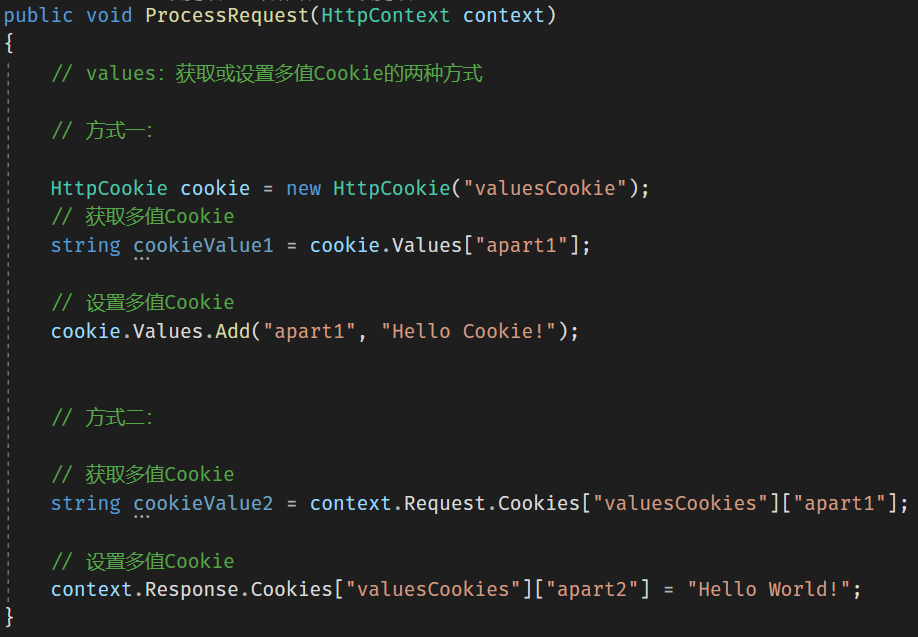
1. HttpCookie.Name：获取或设置当前Cookie的键名



1. HttpCookie.Value：获取或设置当前Cookie的值

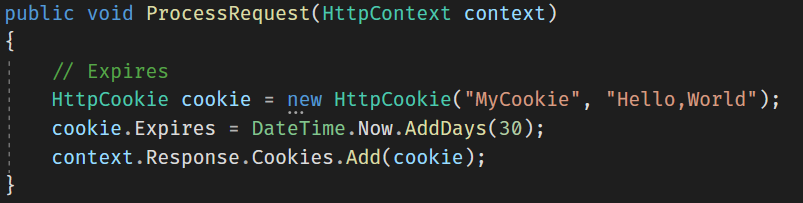


1. HttpCookie.Values：获取或设置多值Cookie



1. HttpCookie.Expires：获取或设置当前Cookie的过期时间

-> 当设置了Cookie的过期时间，浏览器在进行cookie存储的时候会把cookie存储的用户磁盘的响应临时目录下，反 之浏览器在进行cooki存储的时候会保存至浏览器所占用的内存当中



1. HttpCookie.Path：获取或设置当前Cookie的Path属性

-> 该属性可以用于控制浏览器在访问当前站点资源的时候，当前cookie只能在我们所指定虚拟路径下进行发送，默认“/”（意思是当前站点目录下的所有资源请求，浏览器都会把当前cookie放在HTTP请求报文中发送给服务器）



1. HttpCookie.Domain：获取或设置当前Cookie相关联的域，可实现Cookie的跨域机制

-> 关于Cookie的跨域机制请[查看这里](../../WEB前端/Cookie/Cookie.docx)

