# Asp.net中的web.config配置

By Bendon 20101123

目录

[Asp.net中的web.config配置 1](#_Toc278291745)

[一、 配置文件保存位置 2](#_Toc278291746)

[二、 配置文件加载顺序 2](#_Toc278291747)

[三、 配置文件节点介绍 3](#_Toc278291748)

[1. <configSections> 3](#_Toc278291749)

[2. <appSettings> 5](#_Toc278291750)

[3. <connectionStrings> 5](#_Toc278291751)

[4. <system.web> 6](#_Toc278291752)

[<location> 11](#_Toc278291753)

[四、 针对配置文件的一些编程操作 11](#_Toc278291754)

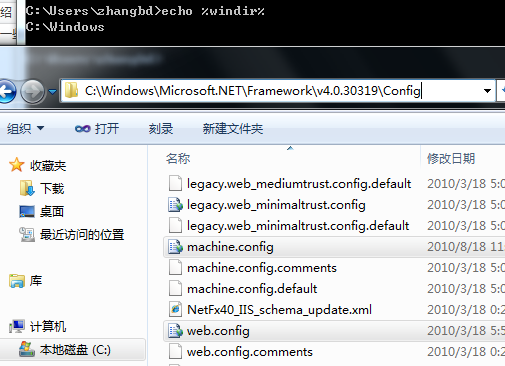
[1. 运行时进行配置文件的修改 11](#_Toc278291755)

[2. 配置节点的加密 12](#_Toc278291756)

web.config是asp.net中保存配置信息（比如数据库连接字符串等）的重要文件。它是基于xml的文本文件方式放在Web应用程序的任何目录中，并且默认不随源文件编译到Dll中，而运行环境随时监视着它是否有改变，一但有变动，系统会自动重新加载里面的最新内容。

## 配置文件保存位置

.net的默认配置文件保存在“windows目录\Microsoft.NET\Framework\对应.net版本\config”文件夹下面。不同的操作系统windows目录不一样，我们在命令行下输入“echo %windir%”查看windows目录所在的位置。



图：web.config所在的目录

Asp.net中有两个非常重要的配置文件，分别是machine.config和web.config，它们都位于config文件夹下面。这两个文件一般不需要我们手工是维护它，保持默认即可。但针对asp.net应用程序，它自身会有0个，1个或者多个web.config配置文件，多个配置文件会存在加载顺序问题。下节会介绍。

注意，传说中.net3.0和.net3.5只是在.net2.0的基础上扩充中，所以还是没用的.net2.0的配置文件。它们连config这个目录都没有。

## 配置文件加载顺序

IIS在Asp.net网站启动时，会加载配置文件中的配置信息，然后缓存这些信息，不会每次要用都去读取配置文件，只是IIS会随时监视着这些文件的变化，一量有变化，它会重新去读取并缓存配置信息。

Asp.net网站运行时会按照以下方式加载配置文件中的节点信息：

* + 1. 如果在当前运行页面所在的目录下有web.config文件，则查找是否存在所需要的节点，如果存在则返回结果，并停止下一步地查找。
    2. 如里所在目录不存在web.config配置或者配置文件里没有所需要的节点，则查找它所在的上一级目录的配置文件中的节点，直到网站根目录。（问题：IIS6中的虚拟目录算不算根目录）
    3. 如果网站根目录中都不存在web.config或者所需要的配置节点，转而到“windows目录\Microsoft.NET\Framework\对应.net版本\config\web.config”中去查找。
    4. 如果第3条中还没找到，继续到“windows目录\Microsoft.NET\Framework\对应.net版本\config\machine.config”中去查找。
    5. 如果还没找到，那就报错吧。

存在两个问题

1. IIS6中的虚拟目录算不算根目录。
2. 在系统运行时，在一个原本没有web.config的目录中手工加上一个web.config，会不会自动加载。

## 配置文件节点介绍

Web.config文件是一个xml文本文件，它的根节点为<configuration>,该节点下包含常见的子节点有：<configSections>、<appSettings>、<connectionStrings>（保存数据库连接字符串）、<location>和<system.web>。下面针对各节点配置进行介绍。

### <configSections>

configSections 元素指定了配置节和处理程序声明。由于 ASP.NET 不对如何处理配置文件内的设置作任何假设，因此这非常必要。但 ASP.NET 会将配置数据的处理委托给配置节处理程序。配置结构信息如下：

<configSections>

<!--定义配置节处理程序与配置元素之间的关联。-->

<section />

<!--定义配置节处理程序与配置节之间的关联。-->

<sectionGroup />

<!--移除对继承的节和节组的引用。-->

<remove />

<!--移除对继承的节和节组的所有引用，只允许由当前 section 和 sectionGroup 元素添加的节和节组。-->

<clear/>

</configSections>

每个 section 元素标识一个配置节或元素以及对该配置节或元素进行处理的关联 ConfigurationSection 派生类。可以在 sectionGroup 元素中对 section 元素进行逻辑分组，以对 section 元素进行组织并避免命名冲突。section 和 sectionGroup 元素包含在 configSections 元素中。

如果配置文件中包含 configSections 元素，则 configSections 元素必须是 configuration 元素的第一个子元素。

下面我们来示例写一个自定义配置信息，并完成它的SectionHandler，首先我们在<configuration>节点下添加configSections。

<configuration>

<configSections>

<sectionGroup name="mySectionGroup">

<section name="mySection" requirePermission="true"

type="ConfigTest.SectionHandler.MySectionHandler,ConfigTest.SectionHandler" />

</sectionGroup>

</configSections>

<mySectionGroup>

<mySection>

<add key="key1" value="value1" />

<add key="key2" value="value2" />

<add key="key3" value="value3" />

<add key="key4" value="value4" />

<add key="key5" value="value5" />

</mySection>

</mySectionGroup>

<system.web>

<compilation debug="true" targetFramework="4.0" />

</system.web>

</configuration>

编写自定义SectionHandler，我们为MySectionHandler返回一个Hashtable的数据。

namespace ConfigTest.SectionHandler

{

public class MySectionHandler : IConfigurationSectionHandler

{

public object Create(object parent, object configContext, System.Xml.XmlNode section)

{

Hashtable ht = new Hashtable();

foreach (XmlNode node in section.ChildNodes)

{

if (node.Name == "add")

{

ht.Add(node.Attributes["key"].Value, node.Attributes["value"].Value);

}

}

return ht;

}

}

}

在页面中使用该Section，由ConfigurationManager.GetSection得到SectionHandler返回的Hashtable。注意参数结构。

protected void Page\_Load(object sender, EventArgs e)

{

Hashtable ht = ConfigurationManager.GetSection("mySectionGroup/mySection") as Hashtable;

foreach (DictionaryEntry de in ht)

{

Response.Write(de.Key + " - " + de.Value + "<br>");

}

}

### <appSettings>

该节点下主要用来存储asp.net应用程序的一些配置信息，也可以把数据库连接字符串也放在这里，不过.net2.0提供了connectionStrings节点，所以数据库连接字符串还是不建议放在这里，下面为一个图片类型的实例。

<appSettings>

<!--图片类型扩展名-->

<add key="ImgType" value=".bmp;.jpg;.gif;.png"/>

</appSettings>

调用方法为:

string ImgType = ConfigurationManager.AppSettings["ImgType"];

### <connectionStrings>

connectionStrings和appSettings类似，不过用于保存配置数据库连接信息，下面给一个实例。

<connectionStrings>

<add name="SqlserverConnStr" connectionString="Data Source=Aron1;Initial Catalog= pubs;UserId=sa;Password=asdasd;"/>

<add name="OrcleConnStr" connectionString="Provider=msdaora;Data Source= MyOracleDB;UserId=UserName;Password=asdasd;"/>

</connectionStrings>

调用方式为：

string connStr = ConfigurationManager.ConnectionStrings["SqlserverConnStr"].ConnectionString;

### <system.web>

<system.web>为.net应用程序的行为方式配置节点，该节点包含很多子节点，很多子节点已经由.net配置好了，这里我们只来看看一些重要的配置节点。

#### <customErrors>

<customErrors defaultRedirect="GenericError.htm" mode="RemoteOnly">

<error statusCode="500" redirect="InternalError.htm"/>

</customErrors>

其中mode属性有三种值，On/Off/RemoteOnly，默认为RemoteOnly。Error节点指定给定 HTTP 状态代码的自定义错误页面。

#### <authentication>

该节点为配置 ASP.NET 身份验证方案，该方案用于识别查看 ASP.NET 应用程序的用户。Mode属性包含四种身份验证模式：

1. Windows(默认)

将 Windows 验证指定为默认的身份验证模式。将它与以下任意形式的 Microsoft Internet 信息服务 (IIS) 身份验证结合起来使用：基本、摘要、集成 Windows 身份验证 (NTLM/Kerberos) 或证书。在这种情况下，您的应用程序将身份验证责任委托给基础 IIS。

1. Forms

将 ASP.NET 基于窗体的身份验证指定为默认身份验证模式。

1. Passport

将 Microsoft Passport Network 身份验证指定为默认身份验证模式。

1. None

不指定任何身份验证。您的应用程序仅期待匿名用户，否则它将提供自己的身份验证。

下面的代码示例演示如何为基于窗体的身份验证配置站点、指定传输来自客户端的登录信息的 Cookie 的名称以及指定当初始身份验证失败时使用的登录页的名称。必须将 authorization 节包含在内才能要求对所有用户进行 Forms 身份验证，并拒绝匿名用户访问站点。

<configuration>

<system.web>

<authentication mode="Forms">

<forms name="401kApp" loginUrl="/login.aspx"/>

</authentication>

<authorization>

<deny users="?"/>

</authorization>

</system.web>

</configuration>

Login.aspx中登陆通过：

FormsAuthentication.RedirectFromLoginPage(this.TextBox1.Text, true);

#### <httpHandlers>

HttpHandlers可用于根据请求中指定的 URL 和 HTTP 谓词将传入的请求映射到相应的处理程序。可以针对某个特定的目录下指定的特殊文件进行特殊处理。

下面我们来针对网站path目录下的所有\*.abc文件夹来编写自定义HttpHandle。

先添加到配置文件：

<httpHandlers>

<add path="path/\*.abc" verb="\*" type="ConfigTest.HttpHandler.AbcHttpHandler,ConfigTest.HttpHandler"/>

</httpHandlers>

编写AbcHttpHandler:

namespace ConfigTest.HttpHandler

{

public class AbcHttpHandler : IHttpHandler, IRequiresSessionState

{

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

context.Response.Write("<h1><b>Hello HttpHandler</b></h1>");

context.Session["Test"] = "你在调用AbcHttpHandler容器中调用Session";

context.Response.Write(context.Session["Test"]);

}

}

}

系统调用结果：



图：HttpHandler测试

我们还可以使用HttpHandlerFactory来进行handler自行切换。我们先定义两个HttpHandler,分别是httpHandler1和httpHandler2。然后定义一个继承于IHttpHandlerFactory的MyHandlerFactory来动态切换httpHandler，看代码：

namespace ConfigTest.HttpHandler

{

public class MyHandlerFactory : IHttpHandlerFactory

{

public IHttpHandler GetHandler(HttpContext context, string requestType, string url, string pathTranslated)

{

if (url.IndexOf("1") > -1)

{

return new HttpHandler1();

}

else if (url.IndexOf("2") > -1)

{

return new HttpHandler2();

}

//返回默认Handler

return context.Handler;

}

public void ReleaseHandler(IHttpHandler handler)

{

// throw new NotImplementedException();

}

}

public class HttpHandler1 : IHttpHandler, IRequiresSessionState

{

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

context.Response.Write("<h1>HttpHandler1</h1>");

}

}

public class HttpHandler2 : IHttpHandler, IRequiresSessionState

{

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

context.Response.Write("<h1>HttpHandler2</h1>");

}

}

}

这里只是测试，我们设定url中存在1这个字符时用HttpHandler1,存在2这个字符里用HttpHandler2，否则返回系统默认的Handler。我们还得增加配置项：

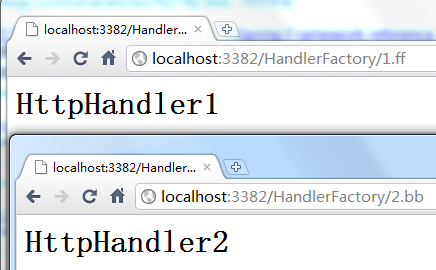
<httpHandlers>

<add path="path/\*.abc" verb="\*" type="ConfigTest.HttpHandler.AbcHttpHandler,ConfigTest.HttpHandler"/>

<add path="HandlerFactory/\*.\*" verb="\*" type="ConfigTest.HttpHandler.MyHandlerFactory,ConfigTest.HttpHandler"/>

</httpHandlers>

添加了一个httpHandler的配置项针对HandlerFactory目录下所有文件，我们运行测试一下：



图：HandlerFactory测试

#### <httpModules>

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.

<httpModules>

<add name="MyHttpModule" type="ConfigTest.HttpModule.MyHttpModule,ConfigTest.HttpModule"/>

</httpModules>

和HttpHandler一样，编写继承于IHttpModule的HttpModule：

namespace ConfigTest.HttpModule

{

public class MyHttpModule:IHttpModule

{

public void Dispose()

{

throw new NotImplementedException();

}

public void Init(HttpApplication context)

{

context.BeginRequest += new EventHandler(context\_BeginRequest);

}

void context\_BeginRequest(object sender, EventArgs e)

{

HttpApplication application = (HttpApplication)sender;

HttpContext context = application.Context;

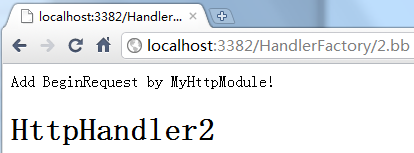
context.Response.Write("Add BeginRequest by MyHttpModule!");

}

}

}

我们只是在每个页面上添加了一句话：Add BeginRequest by MyHttpModule!看结果：



图：HttpModule测试

### <location>

Location节点是用来指定子配置的资源。如果在asp.net应用程序中想对某个目录做特殊处理，则可以用该节点来实现。举两个例子。

下面的代码示例演示如何仅将指定页的上载文件大小限制设置为 128 KB。

<configuration>

<location path="UploadPage.aspx">

<httpRuntime maxRequestLength="128"/>

</location>

</configuration>

为指定目录的图片加水印：

<configuration>

<location path="images">

<system.web>

<httpHandlers>

<add verb="\*" path="\*.jpg" type="ImageHandler"/><!--图片水印设置Handler-->

</httpHandlers>

</system.web>

</location>

</configuration>

## 针对配置文件的一些编程操作

### 运行时进行配置文件的修改

这里我们演示对appSettings节点进行修改：

public static void SetAppSetting(string key, string value)

{

Configuration config = WebConfigurationManager.OpenWebConfiguration("~");

AppSettingsSection appSetting = config.AppSettings;

//如果不存在则添加

if (appSetting.Settings[key] == null)

{

appSetting.Settings.Add(key, value);

}

else//否则修改

{

appSetting.Settings[key].Value = value;

}

config.Save(ConfigurationSaveMode.Full);

}

### 配置节点的加密

有时候我们要对关键节点进行加密，.net给我们提供了加密的方法，下面我们演示对connectionStrings节点进行加密：

public static void ProtectSection()

{

Configuration config = WebConfigurationManager.OpenWebConfiguration("~");

ConfigurationSection section = config.Sections["connectionStrings"];

section.SectionInformation.ProtectSection("RsaProtectedConfigurationProvider");

section.SectionInformation.ForceSave = true;

config.Save(ConfigurationSaveMode.Full);

}

这里我们使用的是RsaProtectedConfigurationProvider，加密后connectionStrings节点为：

<connectionStrings configProtectionProvider="RsaProtectedConfigurationProvider">

<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"

xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1\_5" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<KeyName>Rsa Key</KeyName>

</KeyInfo>

<CipherData>

<CipherValue>Sg75NxcTUSMJwZG9ypLUZh9CeSe6Qa1APhxLpZ+EMNWH4lA9AhEEVbzxAgbWvjPGeJoQfONxpkhjeNVZUTrpm9T6dJMU2vQ6EPmXMMF7Lkng62nQ1LOK+gkTbJT8Z3VsprazFteQAwiBhL8GWB4M94kO7bx6P5Ifu6xgXPYdoEQ=</CipherValue>

</CipherData>

</EncryptedKey>

</KeyInfo>

<CipherData>

<CipherValue></CipherValue>

</CipherData>

</EncryptedData>

</connectionStrings>

以上所有具体操作可以看Demo实例。