#### [Web.xml配置详解](http://www.cnblogs.com/chinafine/archive/2010/09/02/1815980.html)

<http://www.blogjava.net/xzclog/archive/2011/09/29/359789.html>

1 定义头和根元素

　　部署描述符文件就像所有XML文件一样，必须以一个XML头开始。这个头声明可以使用的XML版本并给出文件的字符编码。  
DOCYTPE声明必须立即出现在此头之后。这个声明告诉服务器适用的servlet规范的版本（如2.2或2.3）并指定管理此文件其余部分内容的语法的DTD(Document Type Definition，文档类型定义)。  
所有部署描述符文件的顶层（根）元素为web-app。请注意，XML元素不像HTML，他们是大小写敏感的。因此，web-App和WEB-APP都是不合法的，web-app必须用小写。

2 部署描述符文件内的元素次序

　　XML 元素不仅是大小写敏感的，而且它们还对出现在其他元素中的次序敏感。例如，XML头必须是文件中的第一项，DOCTYPE声明必须是第二项，而web- app元素必须是第三项。在web-app元素内，元素的次序也很重要。服务器不一定强制要求这种次序，但它们允许（实际上有些服务器就是这样做的）完全拒绝执行含有次序不正确的元素的Web应用。这表示使用非标准元素次序的web.xml文件是不可移植的。  
下面的列表给出了所有可直接出现在web-app元素内的合法元素所必需的次序。例如，此列表说明servlet元素必须出现在所有servlet-mapping元素之前。请注意，所有这些元素都是可选的。因此，可以省略掉某一元素，但不能把它放于不正确的位置。  
l icon icon元素指出IDE和GUI工具用来表示Web应用的一个和两个图像文件的位置。  
l display-name display-name元素提供GUI工具可能会用来标记这个特定的Web应用的一个名称。  
l description description元素给出与此有关的说明性文本。  
l context-param context-param元素声明应用范围内的初始化参数。  
l filter 过滤器元素将一个名字与一个实现javax.servlet.Filter接口的类相关联。  
l filter-mapping 一旦命名了一个过滤器，就要利用filter-mapping元素把它与一个或多个servlet或JSP页面相关联。  
l listener servlet API的版本2.3增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或servlet环境时得到通知。Listener元素指出事件监听程序类。  
l servlet 在向servlet或JSP页面制定初始化参数或定制URL时，必须首先命名servlet或JSP页面。Servlet元素就是用来完成此项任务的。  
l servlet-mapping 服务器一般为servlet提供一个缺省的URL：<http://host/webAppPrefix/servlet/ServletName>。但是，常常会更改这个URL，以便servlet可以访问初始化参数或更容易地处理相对URL。在更改缺省URL时，使用servlet-mapping元素。  
l session-config 如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用HttpSession的setMaxInactiveInterval方法明确设置单个会话对象的超时值，或者可利用session-config元素制定缺省超时值。  
l mime-mapping 如果Web应用具有想到特殊的文件，希望能保证给他们分配特定的MIME类型，则mime-mapping元素提供这种保证。  
l welcom-file-list welcome-file-list元素指示服务器在收到引用一个目录名而不是文件名的URL时，使用哪个文件。  
l error-page error-page元素使得在返回特定HTTP状态代码时，或者特定类型的异常被抛出时，能够制定将要显示的页面。  
l taglib taglib元素对标记库描述符文件（Tag Libraryu Descriptor file）指定别名。此功能使你能够更改TLD文件的位置，而不用编辑使用这些文件的JSP页面。  
l resource-env-ref resource-env-ref元素声明与资源相关的一个管理对象。  
l resource-ref resource-ref元素声明一个资源工厂使用的外部资源。  
l security-constraint security-constraint元素制定应该保护的URL。它与login-config元素联合使用  
l login-config 用login-config元素来指定服务器应该怎样给试图访问受保护页面的用户授权。它与sercurity-constraint元素联合使用。  
l security-role security-role元素给出安全角色的一个列表，这些角色将出现在servlet元素内的security-role-ref元素的role-name子元素中。分别地声明角色可使高级IDE处理安全信息更为容易。  
l env-entry env-entry元素声明Web应用的环境项。  
l ejb-ref ejb-ref元素声明一个EJB的主目录的引用。  
l ejb-local-ref ejb-local-ref元素声明一个EJB的本地主目录的应用。

3 分配名称和定制的UL

在web.xml中完成的一个最常见的任务是对servlet或JSP页面给出名称和定制的URL。用servlet元素分配名称，使用servlet-mapping元素将定制的URL与刚分配的名称相关联。  
3.1 分配名称  
为了提供初始化参数，对servlet或JSP页面定义一个定制URL或分配一个安全角色，必须首先给servlet或JSP页面一个名称。可通过 servlet元素分配一个名称。最常见的格式包括servlet-name和servlet-class子元素（在web-app元素内），如下所示：

Xml代码   
<servlet>     
    <servlet-name>Test</servlet-name>     
    <servlet-class>moreservlets.TestServlet</servlet-class>     
</servlet>

<servlet>  
<servlet-name>Test</servlet-name>  
<servlet-class>moreservlets.TestServlet</servlet-class>  
</servlet>

　　这表示位于WEB-INF/classes/moreservlets/TestServlet的servlet已经得到了注册名Test。给 servlet一个名称具有两个主要的含义。首先，初始化参数、定制的URL模式以及其他定制通过此注册名而不是类名引用此servlet。其次,可在 URL而不是类名中使用此名称。因此，利用刚才给出的定义，URL <http://host/webAppPrefix/servlet/Test> 可用于 <http://host/webAppPrefix/servlet/moreservlets.TestServlet> 的场所。  
请记住：XML元素不仅是大小写敏感的，而且定义它们的次序也很重要。例如，web-app元素内所有servlet元素必须位于所有servlet- mapping元素（下一小节介绍）之前，而且还要位于5.6节和5.11节讨论的与过滤器或文档相关的元素（如果有的话）之前。类似地，servlet 的servlet-name子元素也必须出现在servlet-class之前。5.2节"部署描述符文件内的元素次序"将详细介绍这种必需的次序。  
例如，程序清单5-1给出了一个名为TestServlet的简单servlet，它驻留在moreservlets程序包中。因为此servlet是扎根在一个名为deployDemo的目录中的Web应用的组成部分，所以TestServlet.class放在deployDemo/WEB- INF/classes/moreservlets中。程序清单5-2给出将放置在deployDemo/WEB-INF/内的web.xml文件的一部分。此web.xml文件使用servlet-name和servlet-class元素将名称Test与TestServlet.class相关联。图 5-1和图5-2分别显示利用缺省URL和注册名调用TestServlet时的结果。

程序清单5-1 TestServlet.java

Java代码   
package moreservlets;      
     
import java.io.\*;      
import javax.servlet.\*;      
import javax.servlet.http.\*;      
     
public class TestServlet extends HttpServlet {      
    public void doGet(HttpServletRequest request,      
HttpServletResponse response)      
throws ServletException, IOException {      
        response.setContentType("text/html");      
        PrintWriter out = response.getWriter();      
        String uri = request.getRequestURI();      
        out.println(ServletUtilities.headWithTitle("Test Servlet") +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>URI: " + uri "</H2>\n" +"</BODY></HTML>");      
    }      
}

package moreservlets;  
import java.io.\*;  
import javax.servlet.\*;  
import javax.servlet.http.\*;  
public class TestServlet extends HttpServlet {  
public void doGet(HttpServletRequest request,  
HttpServletResponse response)  
throws ServletException, IOException {  
response.setContentType("text/html");  
PrintWriter out = response.getWriter();  
String uri = request.getRequestURI();  
out.println(ServletUtilities.headWithTitle("Test Servlet") +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>URI: " + uri "</H2>\n" +"</BODY></HTML>");  
}  
}

程序清单5-2 web.xml（说明servlet名称的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
<!-- … -->     
<servlet>     
    <servlet-name>Test</servlet-name>     
    <servlet-class>moreservlets.TestServlet</servlet-class>     
</servlet>     
<!-- … -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- … -->  
<servlet>  
<servlet-name>Test</servlet-name>  
<servlet-class>moreservlets.TestServlet</servlet-class>  
</servlet>  
<!-- … -->  
</web-app>

3.2 定义定制的URL  
大多数服务器具有一个缺省的serlvet URL：  
<http://host/webAppPrefix/servlet/packageName.ServletName>。虽然在开发中使用这个URL很方便，但是我们常常会希望另一个URL用于部署。例如，可能会需要一个出现在Web应用顶层的URL（如，http: //host/webAppPrefix/Anyname），并且在此URL中没有servlet项。位于顶层的URL简化了相对URL的使用。此外，对许多开发人员来说，顶层URL看上去比更长更麻烦的缺省URL更简短。  
事实上，有时需要使用定制的URL。比如，你可能想关闭缺省URL映射，以便更好地强制实施安全限制或防止用户意外地访问无初始化参数的servlet。如果你禁止了缺省的URL，那么你怎样访问servlet呢？这时只有使用定制的URL了。  
为了分配一个定制的URL，可使用servlet-mapping元素及其servlet-name和url-pattern子元素。Servlet- name元素提供了一个任意名称，可利用此名称引用相应的servlet；url-pattern描述了相对于Web应用的根目录的URL。url- pattern元素的值必须以斜杠（/）起始。  
下面给出一个简单的web.xml摘录，它允许使用URL <http://host/webAppPrefix/UrlTest>而不是<http://host/webAppPrefix/servlet/Test>或  
http: //host/webAppPrefix/servlet/moreservlets.TestServlet。请注意，仍然需要XML头、 DOCTYPE声明以及web-app封闭元素。此外，可回忆一下，XML元素出现地次序不是随意的。特别是，需要把所有servlet元素放在所有 servlet-mapping元素之前。

Xml代码   
<servlet>     
    <servlet-name>Test</servlet-name>     
    <servlet-class>moreservlets.TestServlet</servlet-class>     
</servlet>     
<!-- ... -->     
<servlet-mapping>     
    <servlet-name>Test</servlet-name>     
    <url-pattern>/UrlTest</url-pattern>     
</servlet-mapping>

<servlet>  
<servlet-name>Test</servlet-name>  
<servlet-class>moreservlets.TestServlet</servlet-class>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name>Test</servlet-name>  
<url-pattern>/UrlTest</url-pattern>  
</servlet-mapping>

URL模式还可以包含通配符。例如，下面的小程序指示服务器发送所有以Web应用的URL前缀开始，以..asp结束的请求到名为BashMS的servlet。

Xml代码   
<servlet>     
    <servlet-name>BashMS</servlet-name>     
    <servlet-class>msUtils.ASPTranslator</servlet-class>     
</servlet>     
<!-- ... -->     
<servlet-mapping>     
    <servlet-name>BashMS</servlet-name>     
    <url-pattern>/\*.asp</url-pattern>     
</servlet-mapping>

<servlet>  
<servlet-name>BashMS</servlet-name>  
<servlet-class>msUtils.ASPTranslator</servlet-class>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name>BashMS</servlet-name>  
<url-pattern>/\*.asp</url-pattern>  
</servlet-mapping>

3.3 命名JSP页面  
　　因为JSP页面要转换成sevlet，自然希望就像命名servlet一样命名JSP页面。毕竟，JSP页面可能会从初始化参数、安全设置或定制的URL中受益，正如普通的serlvet那样。虽然JSP页面的后台实际上是servlet这句话是正确的，但存在一个关键的猜疑：即，你不知道JSP页面的实际类名（因为系统自己挑选这个名字）。因此，为了命名JSP页面，可将jsp-file元素替换为servlet-calss元素，如下所示：

Xml代码   
<servlet>     
    <servlet-name>Test</servlet-name>     
    <jsp-file>/TestPage.jsp</jsp-file>     
</servlet>

<servlet>  
<servlet-name>Test</servlet-name>  
<jsp-file>/TestPage.jsp</jsp-file>  
</servlet>

　　命名JSP页面的原因与命名servlet的原因完全相同：即为了提供一个与定制设置（如，初始化参数和安全设置）一起使用的名称，并且，以便能更改激活 JSP页面的URL（比方说，以便多个URL通过相同页面得以处理，或者从URL中去掉.jsp扩展名）。但是，在设置初始化参数时，应该注意，JSP页面是利用jspInit方法，而不是init方法读取初始化参数的。  
例如，程序清单5-3给出一个名为TestPage.jsp的简单JSP页面，它的工作只是打印出用来激活它的URL的本地部分。TestPage.jsp放置在deployDemo应用的顶层。程序清单5-4给出了用来分配一个注册名PageName，然后将此注册名与<http://host/webAppPrefix/UrlTest2/anything> 形式的URL相关联的web.xml文件（即，deployDemo/WEB-INF/web.xml）的一部分。

程序清单5-3 TestPage.jsp

Html代码   
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">     
<HTML>     
<HEAD>     
<TITLE>JSP Test Page</TITLE>     
</HEAD>     
<BODY BGCOLOR="#FDF5E6">     
<H2>URI: <%= request.getRequestURI() %></H2>     
</BODY>     
</HTML>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<HTML>  
<HEAD>  
<TITLE>JSP Test Page</TITLE>  
</HEAD>  
<BODY BGCOLOR="#FDF5E6">  
<H2>URI: <%= request.getRequestURI() %></H2>  
</BODY>  
</HTML>

程序清单5-4 web.xml（说明JSP页命名的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
<!-- ... -->     
<servlet>     
    <servlet-name>PageName</servlet-name>     
    <jsp-file>/TestPage.jsp</jsp-file>     
</servlet>     
<!-- ... -->     
<servlet-mapping>     
    <servlet-name> PageName </servlet-name>     
    <url-pattern>/UrlTest2/\*</url-pattern>     
</servlet-mapping>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<servlet>  
<servlet-name>PageName</servlet-name>  
<jsp-file>/TestPage.jsp</jsp-file>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name> PageName </servlet-name>  
<url-pattern>/UrlTest2/\*</url-pattern>  
</servlet-mapping>  
<!-- ... -->  
</web-app>

4 禁止激活器servlet

　　对servlet或JSP页面建立定制URL的一个原因是，这样做可以注册从 init（servlet）或jspInit（JSP页面）方法中读取得初始化参数。但是，初始化参数只在是利用定制URL模式或注册名访问 servlet或JSP页面时可以使用，用缺省URL <http://host/webAppPrefix/servlet/ServletName> 访问时不能使用。因此，你可能会希望关闭缺省URL，这样就不会有人意外地调用初始化servlet了。这个过程有时称为禁止激活器servlet，因为多数服务器具有一个用缺省的servlet URL注册的标准servlet，并激活缺省的URL应用的实际servlet。  
　　有两种禁止此缺省URL的主要方法：  
l 在每个Web应用中重新映射/servlet/模式。  
l 全局关闭激活器servlet。  
重要的是应该注意到，虽然重新映射每个Web应用中的/servlet/模式比彻底禁止激活servlet所做的工作更多，但重新映射可以用一种完全可移植的方式来完成。相反，全局禁止激活器servlet完全是针对具体机器的，事实上有的服务器（如ServletExec）没有这样的选择。下面的讨论对每个Web应用重新映射/servlet/ URL模式的策略。后面提供在Tomcat中全局禁止激活器servlet的详细内容。

4.1 重新映射/servlet/URL模式  
　　在一个特定的Web应用中禁止以<http://host/webAppPrefix/servlet/> 开始的URL的处理非常简单。所需做的事情就是建立一个错误消息servlet，并使用前一节讨论的url-pattern元素将所有匹配请求转向该 servlet。只要简单地使用：

Xml代码   
<url-pattern>/servlet/\*</url-pattern>

<url-pattern>/servlet/\*</url-pattern>

作为servlet-mapping元素中的模式即可。

例如，程序清单5-5给出了将SorryServlet servlet（程序清单5-6）与所有以<http://host/webAppPrefix/servlet/> 开头的URL相关联的部署描述符文件的一部分。

程序清单5-5 web.xml（说明JSP页命名的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
<!-- ... -->     
<servlet>     
    <servlet-name>Sorry</servlet-name>     
    <servlet-class>moreservlets.SorryServlet</servlet-class>     
</servlet>     
<!-- ... -->     
<servlet-mapping>     
    <servlet-name> Sorry </servlet-name>     
    <url-pattern>/servlet/\*</url-pattern>     
</servlet-mapping>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<servlet>  
<servlet-name>Sorry</servlet-name>  
<servlet-class>moreservlets.SorryServlet</servlet-class>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name> Sorry </servlet-name>  
<url-pattern>/servlet/\*</url-pattern>  
</servlet-mapping>  
<!-- ... -->  
</web-app>

程序清单5-6 SorryServlet.java

Java代码   
package moreservlets;      
     
import java.io.\*;      
import javax.servlet.\*;      
import javax.servlet.http.\*;      
     
public class SorryServlet extends HttpServlet {      
    public void doGet(HttpServletRequest request,      
HttpServletResponse response)      
throws ServletException, IOException {      
        response.setContentType("text/html");      
        PrintWriter out = response.getWriter();      
        String title = "Invoker Servlet Disabled.";      
        out.println(ServletUtilities.headWithTitle(title) +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>" + title + "</H2>\n" +      
"Sorry, access to servlets by means of\n" +"URLs that begin with\n" +"[http://host/webAppPrefix/servlet//n](http://host/webAppPrefix/servlet/n)" +"has been disabled.\n" + "</BODY></HTML>");      
        }      
     
    public void doPost(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException {      
        doGet(request, response);      
    }      
}

package moreservlets;  
import java.io.\*;  
import javax.servlet.\*;  
import javax.servlet.http.\*;  
public class SorryServlet extends HttpServlet {  
public void doGet(HttpServletRequest request,  
HttpServletResponse response)  
throws ServletException, IOException {  
response.setContentType("text/html");  
PrintWriter out = response.getWriter();  
String title = "Invoker Servlet Disabled.";  
out.println(ServletUtilities.headWithTitle(title) +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>" + title + "</H2>\n" +  
"Sorry, access to servlets by means of\n" +"URLs that begin with\n" +"[http://host/webAppPrefix/servlet//n](http://host/webAppPrefix/servlet/n)" +"has been disabled.\n" + "</BODY></HTML>");  
}  
public void doPost(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException {  
doGet(request, response);  
}  
}

4.2 全局禁止激活器：Tomcat  
Tomcat 4中用来关闭缺省URL的方法与Tomcat 3中所用的很不相同。下面介绍这两种方法：

1．禁止激活器： Tomcat 4  
Tomcat 4用与前面相同的方法关闭激活器servlet，即利用web.xml中的url-mapping元素进行关闭。不同之处在于Tomcat使用了放在 install\_dir/conf中的一个服务器专用的全局web.xml文件，而前面使用的是存放在每个Web应用的WEB-INF目录中的标准 web.xml文件。  
因此，为了在Tomcat 4中关闭激活器servlet，只需在install\_dir/conf/web.xml中简单地注释出/servlet/\* URL映射项即可，如下所示：

Xml代码   
<servlet-mapping>     
    <servlet-name>invoker</servlet-name>     
    <url-pattern>/servlet/\*</url-pattern>     
</servlet-mapping>

<servlet-mapping>  
<servlet-name>invoker</servlet-name>  
<url-pattern>/servlet/\*</url-pattern>  
</servlet-mapping>

再次提醒，应该注意这个项是位于存放在install\_dir/conf的Tomcat专用的web.xml文件中的，此文件不是存放在每个Web应用的WEB-INF目录中的标准web.xml。

2．禁止激活器：Tomcat3  
在Apache Tomcat的版本3中，通过在install\_dir/conf/server.xml中注释出InvokerInterceptor项全局禁止缺省 servlet URL。例如，下面是禁止使用缺省servlet URL的server.xml文件的一部分。

Xml代码   
<!--      
<RequsetInterceptor className="org.apache.tomcat.request.InvokerInterceptor" debug="0" prefix="/servlet/" />     
-->

<!--  
<RequsetInterceptor className="org.apache.tomcat.request.InvokerInterceptor" debug="0" prefix="/servlet/" />  
-->

5 初始化和预装载servlet与JSP页面

这里讨论控制servlet和JSP页面的启动行为的方法。特别是，说明了怎样分配初始化参数以及怎样更改服务器生存期中装载servlet和JSP页面的时刻。

5.1 分配servlet初始化参数  
利用init-param元素向servlet提供初始化参数，init-param元素具有param-name和param-value子元素。例如，在下面的例子中，如果initServlet servlet是利用它的注册名（InitTest）访问的，它将能够从其方法中调用getServletConfig(). getInitParameter("param1")获得"Value 1"，调用getServletConfig().getInitParameter("param2")获得"2"。

Xml代码   
<servlet>     
    <servlet-name>InitTest</servlet-name>     
    <servlet-class>moreservlets.InitServlet</servlet-class>     
        <init-param>     
            <param-name>param1</param-name>     
            <param-value>value1</param-value>     
        </init-param>     
        <init-param>     
            <param-name>param2</param-name>     
            <param-value>2</param-value>     
        </init-param>     
</servlet>

<servlet>  
<servlet-name>InitTest</servlet-name>  
<servlet-class>moreservlets.InitServlet</servlet-class>  
<init-param>  
<param-name>param1</param-name>  
<param-value>value1</param-value>  
</init-param>  
<init-param>  
<param-name>param2</param-name>  
<param-value>2</param-value>  
</init-param>  
</servlet>

在涉及初始化参数时，有几点需要注意：  
l 返回值。GetInitParameter的返回值总是一个String。因此，在前一个例子中，可对param2使用Integer.parseInt获得一个int。  
l JSP中的初始化。JSP页面使用jspInit而不是init。JSP页面还需要使用jsp-file元素代替servlet-class。  
l 缺省URL。初始化参数只在通过它们的注册名或与它们注册名相关的定制URL模式访问Servlet时可以使用。因此，在这个例子中，param1和 param2初始化参数将能够在使用URL <http://host/webAppPrefix/servlet/InitTest> 时可用，但在使用URL <http://host/webAppPrefix/servlet/myPackage.InitServlet> 时不能使用。  
例如，程序清单5-7给出一个名为InitServlet的简单servlet，它使用init方法设置firstName和emailAddress字段。程序清单5-8给出分配名称InitTest给servlet的web.xml文件。

程序清单5-7 InitServlet.java

Java代码   
package moreservlets;      
     
import java.io.\*;      
import javax.servlet.\*;      
import javax.servlet.http.\*;      
     
public class InitServlet extends HttpServlet {      
    private String firstName, emailAddress;      
     
    public void init() {      
        ServletConfig config = getServletConfig();      
        firstName = config.getInitParameter("firstName");      
        emailAddress = config.getInitParameter("emailAddress");      
    }      
     
    public void doGet(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException {      
        response.setContentType("text/html");      
        PrintWriter out = response.getWriter();      
        String uri = request.getRequestURI();      
        out.println(ServletUtilities.headWithTitle("Init Servlet") +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>Init Parameters:</H2>\n" +"<UL>\n" +"<LI>First name: " + firstName + "\n" +"<LI>Email address: " + emailAddress + "\n" +"</UL>\n" + "</BODY></HTML>");      
    }      
}

package moreservlets;  
import java.io.\*;  
import javax.servlet.\*;  
import javax.servlet.http.\*;  
public class InitServlet extends HttpServlet {  
private String firstName, emailAddress;  
public void init() {  
ServletConfig config = getServletConfig();  
firstName = config.getInitParameter("firstName");  
emailAddress = config.getInitParameter("emailAddress");  
}  
public void doGet(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException {  
response.setContentType("text/html");  
PrintWriter out = response.getWriter();  
String uri = request.getRequestURI();  
out.println(ServletUtilities.headWithTitle("Init Servlet") +"<BODY BGCOLOR=\"#FDF5E6\">\n" +"<H2>Init Parameters:</H2>\n" +"<UL>\n" +"<LI>First name: " + firstName + "\n" +"<LI>Email address: " + emailAddress + "\n" +"</UL>\n" + "</BODY></HTML>");  
}  
}

程序清单5-8 web.xml（说明初始化参数的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
     
<web-app>     
<!-- ... -->     
<servlet>     
    <servlet-name>InitTest</servlet-name>     
        <servlet-class>moreservlets.InitServlet</servlet-class>     
        <init-param>     
            <param-name>firstName</param-name>     
            <param-value>Larry</param-value>     
        </init-param>     
        <init-param>     
            <param-name>emailAddress</param-name>     
            <param-value>Ellison@Microsoft.com</param-value>     
        </init-param>     
    </servlet>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<servlet>  
<servlet-name>InitTest</servlet-name>  
<servlet-class>moreservlets.InitServlet</servlet-class>  
<init-param>  
<param-name>firstName</param-name>  
<param-value>Larry</param-value>  
</init-param>  
<init-param>  
<param-name>emailAddress</param-name>  
<param-value>Ellison@Microsoft.com</param-value>  
</init-param>  
</servlet>  
<!-- ... -->  
</web-app>

5.2 分配JSP初始化参数  
给JSP页面提供初始化参数在三个方面不同于给servlet提供初始化参数。  
1）使用jsp-file而不是servlet-class。因此，WEB-INF/web.xml文件的servlet元素如下所示：

Xml代码   
<servlet>     
    <servlet-name>PageName</servlet-name>     
    <jsp-file>/RealPage.jsp</jsp-file>     
        <init-param>     
            <param-name>...</param-name>     
            <param-value>...</param-value>     
        </init-param>     
...      
</servlet>

<servlet>  
<servlet-name>PageName</servlet-name>  
<jsp-file>/RealPage.jsp</jsp-file>  
<init-param>  
<param-name>...</param-name>  
<param-value>...</param-value>  
</init-param>  
...  
</servlet>

2) 几乎总是分配一个明确的URL模式。对servlet，一般相应地使用以<http://host/webAppPrefix/servlet/> 开始的缺省URL。只需记住，使用注册名而不是原名称即可。这对于JSP页面在技术上也是合法的。例如，在上面给出的例子中，可用URL <http://host/webAppPrefix/servlet/PageName> 访问RealPage.jsp的对初始化参数具有访问权的版本。但在用于JSP页面时，许多用户似乎不喜欢应用常规的servlet的URL。此外，如果 JSP页面位于服务器为其提供了目录清单的目录中（如，一个既没有index.html也没有index.jsp文件的目录），则用户可能会连接到此 JSP页面，单击它，从而意外地激活未初始化的页面。因此，好的办法是使用url-pattern（5.3节）将JSP页面的原URL与注册的 servlet名相关联。这样，客户机可使用JSP页面的普通名称，但仍然激活定制的版本。例如，给定来自项目1的servlet定义，可使用下面的 servlet-mapping定义：

Xml代码   
<servlet-mapping>     
    <servlet-name>PageName</servlet-name>     
    <url-pattern>/RealPage.jsp</url-pattern>     
</servlet-mapping>

<servlet-mapping>  
<servlet-name>PageName</servlet-name>  
<url-pattern>/RealPage.jsp</url-pattern>  
</servlet-mapping>

3）JSP页使用jspInit而不是init。自动从JSP页面建立的servlet或许已经使用了inti方法。因此，使用JSP声明提供一个init方法是不合法的，必须制定jspInit方法。  
为了说明初始化JSP页面的过程，程序清单5-9给出了一个名为InitPage.jsp的JSP页面，它包含一个jspInit方法且放置于 deployDemo Web应用层次结构的顶层。一般，<http://host/deployDemo/InitPage.jsp> 形式的URL将激活此页面的不具有初始化参数访问权的版本，从而将对firstName和emailAddress变量显示null。但是， web.xml文件（程序清单5-10）分配了一个注册名，然后将该注册名与URL模式/InitPage.jsp相关联。

程序清单5-9 InitPage.jsp

Html代码   
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">     
<HTML>     
<HEAD><TITLE>JSP Init Test</TITLE></HEAD>     
<BODY BGCOLOR="#FDF5E6">     
<H2>Init Parameters:</H2>     
<UL>     
<LI>First name: <%= firstName %>     
<LI>Email address: <%= emailAddress %>     
</UL>     
</BODY></HTML>     
<%!      
private String firstName, emailAddress;      
     
public void jspInit() {      
ServletConfig config = getServletConfig();      
firstName = config.getInitParameter("firstName");      
emailAddress = config.getInitParameter("emailAddress");      
}      
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<HTML>  
<HEAD><TITLE>JSP Init Test</TITLE></HEAD>  
<BODY BGCOLOR="#FDF5E6">  
<H2>Init Parameters:</H2>  
<UL>  
<LI>First name: <%= firstName %>  
<LI>Email address: <%= emailAddress %>  
</UL>  
</BODY></HTML>  
<%!  
private String firstName, emailAddress;  
public void jspInit() {  
ServletConfig config = getServletConfig();  
firstName = config.getInitParameter("firstName");  
emailAddress = config.getInitParameter("emailAddress");  
}  
%>

程序清单5-10 web.xml（说明JSP页面的init参数的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
<!-- ... -->     
    <servlet>     
        <servlet-name>InitPage</servlet-name>     
            <jsp-file>/InitPage.jsp</jsp-file>     
                <init-param>     
                    <param-name>firstName</param-name>     
                    <param-value>Bill</param-value>     
                </init-param>     
                <init-param>     
                    <param-name>emailAddress</param-name>     
                    <param-value>gates@oracle.com</param-value>     
                </init-param>     
        </servlet>     
<!-- ... -->       
    <servlet-mapping>     
        <servlet-name> InitPage</servlet-name>     
        <url-pattern>/InitPage.jsp</url-pattern>     
    </servlet-mapping>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<servlet>  
<servlet-name>InitPage</servlet-name>  
<jsp-file>/InitPage.jsp</jsp-file>  
<init-param>  
<param-name>firstName</param-name>  
<param-value>Bill</param-value>  
</init-param>  
<init-param>  
<param-name>emailAddress</param-name>  
<param-value>gates@oracle.com</param-value>  
</init-param>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name> InitPage</servlet-name>  
<url-pattern>/InitPage.jsp</url-pattern>  
</servlet-mapping>  
<!-- ... -->  
</web-app>

5.3 提供应用范围内的初始化参数  
一般，对单个地servlet或JSP页面分配初始化参数。指定的servlet或JSP页面利用ServletConfig的getInitParameter方法读取这些参数。但是，在某些情形下，希望提供可由任意servlet或JSP页面借助ServletContext的getInitParameter方法读取的系统范围内的初始化参数。  
可利用context-param元素声明这些系统范围内的初始化值。context-param元素应该包含param-name、param-value以及可选的description子元素，如下所示：

Xml代码   
<context-param>     
    <param-name>support-email</param-name>     
    <param-value>blackhole@mycompany.com</param-value>     
</context-param>

<context-param>  
<param-name>support-email</param-name>  
<param-value>blackhole@mycompany.com</param-value>  
</context-param>

可回忆一下，为了保证可移植性，web.xml内的元素必须以正确的次序声明。但这里应该注意，context-param元素必须出现任意与文档有关的元素（icon、display-name或description）之后及filter、filter-mapping、listener或 servlet元素之前。

5.4 在服务器启动时装载servlet  
假如servlet或JSP页面有一个要花很长时间执行的init （servlet）或jspInit（JSP）方法。例如，假如init或jspInit方法从某个数据库或ResourceBundle查找产量。这种情况下，在第一个客户机请求时装载servlet的缺省行为将对第一个客户机产生较长时间的延迟。因此，可利用servlet的load-on- startup元素规定服务器在第一次启动时装载servlet。下面是一个例子。

Xml代码   
<servlet>     
    <servlet-name> … </servlet-name>     
    <servlet-class> … </servlet-class> <!-- Or jsp-file -->     
<load-on-startup/>     
</servlet>

<servlet>  
<servlet-name> … </servlet-name>  
<servlet-class> … </servlet-class> <!-- Or jsp-file -->  
<load-on-startup/>  
</servlet>

可以为此元素体提供一个整数而不是使用一个空的load-on-startup。想法是服务器应该在装载较大数目的servlet或JSP页面之前装载较少数目的servlet或JSP页面。例如，下面的servlet项（放置在Web应用的WEB-INF目录下的web.xml文件中的web-app元素内）将指示服务器首先装载和初始化SearchServlet，然后装载和初始化由位于Web应用的result目录中的index.jsp文件产生的 servlet。

Xml代码   
<servlet>     
    <servlet-name>Search</servlet-name>     
    <servlet-class>myPackage.SearchServlet</servlet-class> <!-- Or jsp-file -->     
        <load-on-startup>1</load-on-startup>     
/servlet>     
<servlet>     
        <servlet-name>Results</servlet-name>     
        <servlet-class>/results/index.jsp</servlet-class> <!-- Or jsp-file -->     
        <load-on-startup>2</load-on-startup>     
</servlet>

<servlet>  
<servlet-name>Search</servlet-name>  
<servlet-class>myPackage.SearchServlet</servlet-class> <!-- Or jsp-file -->  
<load-on-startup>1</load-on-startup>  
/servlet>  
<servlet>  
<servlet-name>Results</servlet-name>  
<servlet-class>/results/index.jsp</servlet-class> <!-- Or jsp-file -->  
<load-on-startup>2</load-on-startup>  
</servlet>

6 声明过滤器

servlet版本2.3引入了过滤器的概念。虽然所有支持servlet API版本2.3的服务器都支持过滤器，但为了使用与过滤器有关的元素，必须在web.xml中使用版本2.3的DTD。  
过滤器可截取和修改进入一个servlet或JSP页面的请求或从一个servlet或JSP页面发出的相应。在执行一个servlet或JSP页面之前，必须执行第一个相关的过滤器的doFilter方法。在该过滤器对其FilterChain对象调用doFilter时，执行链中的下一个过滤器。如果没有其他过滤器，servlet或JSP页面被执行。过滤器具有对到来的ServletRequest对象的全部访问权，因此，它们可以查看客户机名、查找到来的cookie等。为了访问servlet或JSP页面的输出，过滤器可将响应对象包裹在一个替身对象（stand-in object）中，比方说把输出累加到一个缓冲区。在调用FilterChain对象的doFilter方法之后，过滤器可检查缓冲区，如有必要，就对它进行修改，然后传送到客户机。  
例如，程序清单5-11帝国难以了一个简单的过滤器，只要访问相关的servlet或JSP页面，它就截取请求并在标准输出上打印一个报告（开发过程中在桌面系统上运行时，大多数服务器都可以使用这个过滤器）。

程序清单5-11 ReportFilter.java

Java代码   
package moreservlets;      
     
import java.io.\*;      
import javax.servlet.\*;      
import javax.servlet.http.\*;      
import java.util.\*;      
     
public class ReportFilter implements Filter {      
    public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)throws ServletException, IOException {      
        HttpServletRequest req = (HttpServletRequest)request;      
        System.out.println(req.getRemoteHost() +" tried to access " +req.getRequestURL() +" on " + new Date() + ".");      
        chain.doFilter(request,response);      
    }      
     
    public void init(FilterConfig config)throws ServletException {}      
     
    public void destroy() {}      
}

package moreservlets;  
import java.io.\*;  
import javax.servlet.\*;  
import javax.servlet.http.\*;  
import java.util.\*;  
public class ReportFilter implements Filter {  
public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)throws ServletException, IOException {  
HttpServletRequest req = (HttpServletRequest)request;  
System.out.println(req.getRemoteHost() +" tried to access " +req.getRequestURL() +" on " + new Date() + ".");  
chain.doFilter(request,response);  
}  
public void init(FilterConfig config)throws ServletException {}  
public void destroy() {}  
}

一旦建立了一个过滤器，可以在web.xml中利用filter元素以及filter-name（任意名称）、file-class（完全限定的类名）和（可选的）init-params子元素声明它。请注意，元素在web.xml的web-app元素中出现的次序不是任意的；允许服务器（但不是必需的）强制所需的次序，并且实际中有些服务器也是这样做的。但这里要注意，所有filter元素必须出现在任意filter-mapping元素之前， filter-mapping元素又必须出现在所有servlet或servlet-mapping元素之前。  
例如，给定上述的ReportFilter类，可在web.xml中作出下面的filter声明。它把名称Reporter与实际的类ReportFilter（位于moreservlets程序包中）相关联。

Xml代码   
<filter>     
    <filter-name>Reporter</filter-name>     
    <filter-class>moresevlets.ReportFilter</filter-class>     
</filter>

<filter>  
<filter-name>Reporter</filter-name>  
<filter-class>moresevlets.ReportFilter</filter-class>  
</filter>

一旦命名了一个过滤器，可利用filter-mapping元素把它与一个或多个servlet或JSP页面相关联。关于此项工作有两种选择。  
首先，可使用filter-name和servlet-name子元素把此过滤器与一个特定的servlet名（此servlet名必须稍后在相同的 web.xml文件中使用servlet元素声明）关联。例如，下面的程序片断指示系统只要利用一个定制的URL访问名为SomeServletName 的servlet或JSP页面，就运行名为Reporter的过滤器。

Xml代码   
<filter-mapping>     
    <filter-name>Reporter</filter-name>     
    <servlet-name>SomeServletName</servlet-name>     
</filter-mapping>

<filter-mapping>  
<filter-name>Reporter</filter-name>  
<servlet-name>SomeServletName</servlet-name>  
</filter-mapping>

其次，可利用filter-name和url-pattern子元素将过滤器与一组servlet、JSP页面或静态内容相关联。例如，相面的程序片段指示系统只要访问Web应用中的任意URL，就运行名为Reporter的过滤器。

Xml代码   
<filter-mapping>     
    <filter-name>Reporter</filter-name>     
    <url-pattern>/\*</url-pattern>     
</filter-mapping>

<filter-mapping>  
<filter-name>Reporter</filter-name>  
<url-pattern>/\*</url-pattern>  
</filter-mapping>

例如，程序清单5-12给出了将ReportFilter过滤器与名为PageName的servlet相关联的web.xml文件的一部分。名字 PageName依次又与一个名为TestPage.jsp的JSP页面以及以模式http: //host/webAppPrefix/UrlTest2/ 开头的URL相关联。TestPage.jsp的源代码已经JSP页面命名的谈论在前面的3节"分配名称和定制的URL"中给出。事实上，程序清单5- 12中的servlet和servlet-name项从该节原封不动地拿过来的。给定这些web.xml项，可看到下面的标准输出形式的调试报告（换行是为了容易阅读）。

程序清单5-12 Web.xml（说明filter用法的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
    <filter>     
        <filter-name>Reporter</filter-name>     
        <filter-class>moresevlets.ReportFilter</filter-class>     
    </filter>     
    <!-- ... -->     
    <filter-mapping>     
        <filter-name>Reporter</filter-name>     
        <servlet-name>PageName</servlet-name>     
    </filter-mapping>     
<!-- ... -->     
    <servlet>     
        <servlet-name>PageName</servlet-name>     
        <jsp-file>/RealPage.jsp</jsp-file>     
    </servlet>     
<!-- ... -->     
    <servlet-mapping>     
        <servlet-name> PageName </servlet-name>     
        <url-pattern>/UrlTest2/\*</url-pattern>     
        </servlet-mapping>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<filter>  
<filter-name>Reporter</filter-name>  
<filter-class>moresevlets.ReportFilter</filter-class>  
</filter>  
<!-- ... -->  
<filter-mapping>  
<filter-name>Reporter</filter-name>  
<servlet-name>PageName</servlet-name>  
</filter-mapping>  
<!-- ... -->  
<servlet>  
<servlet-name>PageName</servlet-name>  
<jsp-file>/RealPage.jsp</jsp-file>  
</servlet>  
<!-- ... -->  
<servlet-mapping>  
<servlet-name> PageName </servlet-name>  
<url-pattern>/UrlTest2/\*</url-pattern>  
</servlet-mapping>  
<!-- ... -->  
</web-app>

7 指定欢迎页

假如用户提供了一个像http: //host/webAppPrefix/directoryName/ 这样的包含一个目录名但没有包含文件名的URL，会发生什么事情呢？用户能得到一个目录表？一个错误？还是标准文件的内容？如果得到标准文件内容，是 index.html、index.jsp、default.html、default.htm或别的什么东西呢？  
Welcome-file-list 元素及其辅助的welcome-file元素解决了这个模糊的问题。例如，下面的web.xml项指出，如果一个URL给出一个目录名但未给出文件名，服务器应该首先试用index.jsp，然后再试用index.html。如果两者都没有找到，则结果有赖于所用的服务器（如一个目录列表）。

Xml代码   
<welcome-file-list>     
    <welcome-file>index.jsp</welcome-file>     
    <welcome-file>index.html</welcome-file>     
</welcome-file-list>

<welcome-file-list>  
<welcome-file>index.jsp</welcome-file>  
<welcome-file>index.html</welcome-file>  
</welcome-file-list>

虽然许多服务器缺省遵循这种行为，但不一定必须这样。因此，明确地使用welcom-file-list保证可移植性是一种良好的习惯。

8 指定处理错误的页面

现在我了解到，你在开发servlet和JSP页面时从不会犯错误，而且你的所有页面是那样的清晰，一般的程序员都不会被它们的搞糊涂。但是，是人总会犯错误的，用户可能会提供不合规定的参数，使用不正确的URL或者不能提供必需的表单字段值。除此之外，其它开发人员可能不那么细心，他们应该有些工具来克服自己的不足。  
error-page元素就是用来克服这些问题的。它有两个可能的子元素，分别是：error-code和exception- type。第一个子元素error-code指出在给定的HTTP错误代码出现时使用的URL。第二个子元素excpetion-type指出在出现某个给定的Java异常但未捕捉到时使用的URL。error-code和exception-type都利用location元素指出相应的URL。此 URL必须以/开始。location所指出的位置处的页面可通过查找HttpServletRequest对象的两个专门的属性来访问关于错误的信息，这两个属性分别是：javax.servlet.error.status\_code和javax.servlet.error.message。  
可回忆一下，在web.xml内以正确的次序声明web-app的子元素很重要。这里只要记住，error-page出现在web.xml文件的末尾附近，servlet、servlet-name和welcome-file-list之后即可。

8.1 error-code元素  
为了更好地了解error-code元素的值，可考虑一下如果不正确地输入文件名，大多数站点会作出什么反映。这样做一般会出现一个404错误信息，它表示不能找到该文件，但几乎没提供更多有用的信息。另一方面，可以试一下在<http://www.microsoft.com/>、<http://www.ibm.com/> 处或者特别是在<http://www.bea.com/> 处输出未知的文件名。这是会得出有用的消息，这些消息提供可选择的位置，以便查找感兴趣的页面。提供这样有用的错误页面对于Web应用来说是很有价值得。事实上rm-error-page子元素）。由form-login-page给出的HTML表单必须具有一个j\_security\_check的 ACTION属性、一个名为j\_username的用户名文本字段以及一个名为j\_password的口令字段。  
例如，程序清单5-19指示服务器使用基于表单的验证。Web应用的顶层目录中的一个名为login.jsp的页面将收集用户名和口令，并且失败的登陆将由相同目录中名为login-error.jsp的页面报告。

程序清单5-19 web.xml（说明login-config的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
     
<web-app>     
<!-- ... -->     
    <security-constraint> ... </security-constraint>     
    <login-config>     
        <auth-method> FORM </auth-method>     
        <form-login-config>     
            <form-login-page>/login.jsp</form-login-page>     
            <form-error-page>/login-error.jsp</form-error-page>     
        </form-login-config>     
    </login-config>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<security-constraint> ... </security-constraint>  
<login-config>  
<auth-method> FORM </auth-method>  
<form-login-config>  
<form-login-page>/login.jsp</form-login-page>  
<form-error-page>/login-error.jsp</form-error-page>  
</form-login-config>  
</login-config>  
<!-- ... -->  
</web-app>

9.2 限制对Web资源的访问  
现在，可以指示服务器使用何种验证方法了。"了不起，"你说道，"除非我能指定一个来收到保护的 URL，否则没有多大用处。"没错。指出这些URL并说明他们应该得到何种保护正是security-constriaint元素的用途。此元素在 web.xml中应该出现在login-config的紧前面。它包含是个可能的子元素，分别是：web-resource-collection、 auth-constraint、user-data-constraint和display-name。下面各小节对它们进行介绍。  
l web-resource-collection  
此元素确定应该保护的资源。所有security-constraint元素都必须包含至少一个web-resource-collection项。此元素由一个给出任意标识名称的web-resource-name元素、一个确定应该保护的URL的url-pattern元素、一个指出此保护所适用的 HTTP命令（GET、POST等，缺省为所有方法）的http-method元素和一个提供资料的可选description元素组成。例如，下面的 Web-resource-collection项（在security-constratint元素内）指出Web应用的proprietary目录中所有文档应该受到保护。

Xml代码   
<security-constraint>     
    <web-resource-coolection>     
    <web-resource-name>Proprietary</web-resource-name>     
        <url-pattern>/propritary/\*</url-pattern>     
    </web-resource-coolection>     
<!-- ... -->     
</security-constraint>

<security-constraint>  
<web-resource-coolection>  
<web-resource-name>Proprietary</web-resource-name>  
<url-pattern>/propritary/\*</url-pattern>  
</web-resource-coolection>  
<!-- ... -->  
</security-constraint>

重要的是应该注意到，url-pattern仅适用于直接访问这些资源的客户机。特别是，它不适合于通过MVC体系结构利用 RequestDispatcher来访问的页面，或者不适合于利用类似jsp:forward的手段来访问的页面。这种不匀称如果利用得当的话很有好处。例如，servlet可利用MVC体系结构查找数据，把它放到bean中，发送请求到从bean中提取数据的JSP页面并显示它。我们希望保证决不直接访问受保护的JSP页面，而只是通过建立该页面将使用的bean的servlet来访问它。url-pattern和auth-contraint元素可通过声明不允许任何用户直接访问JSP页面来提供这种保证。但是，这种不匀称的行为可能让开发人员放松警惕，使他们偶然对应受保护的资源提供不受限制的访问。   
l auth-constraint  
尽管web-resource-collention元素质出了哪些URL应该受到保护，但是auth-constraint元素却指出哪些用户应该具有受保护资源的访问权。此元素应该包含一个或多个标识具有访问权限的用户类别role- name元素，以及包含（可选）一个描述角色的description元素。例如，下面web.xml中的security-constraint元素部门规定只有指定为Administrator或Big Kahuna（或两者）的用户具有指定资源的访问权。

Xml代码   
<security-constraint>     
    <web-resource-coolection> ... </web-resource-coolection>     
    <auth-constraint>     
        <role-name>administrator</role-name>     
        <role-name>kahuna</role-name>     
    </auth-constraint>     
</security-constraint>

<security-constraint>  
<web-resource-coolection> ... </web-resource-coolection>  
<auth-constraint>  
<role-name>administrator</role-name>  
<role-name>kahuna</role-name>  
</auth-constraint>  
</security-constraint>

重要的是认识到，到此为止，这个过程的可移植部分结束了。服务器怎样确定哪些用户处于任何角色以及它怎样存放用户的口令，完全有赖于具体的系统。  
例如，Tomcat使用install\_dir/conf/tomcat-users.xml将用户名与角色名和口令相关联，正如下面例子中所示，它指出用户joe（口令bigshot）和jane（口令enaj）属于administrator和kahuna角色。

Xml代码   
<tomcat-users>     
    <user name="joe" password="bigshot" roles="administrator,kahuna" />     
    <user name="jane" password="enaj" roles="kahuna" />     
</tomcat-users>

<tomcat-users>  
<user name="joe" password="bigshot" roles="administrator,kahuna" />  
<user name="jane" password="enaj" roles="kahuna" />  
</tomcat-users>

l user-data-constraint  
这个可选的元素指出在访问相关资源时使用任何传输层保护。它必须包含一个transport-guarantee子元素（合法值为NONE、 INTEGRAL或CONFIDENTIAL），并且可选地包含一个description元素。transport-guarantee为NONE值将对所用的通讯协议不加限制。INTEGRAL值表示数据必须以一种防止截取它的人阅读它的方式传送。虽然原理上（并且在未来的HTTP版本中），在 INTEGRAL和CONFIDENTIAL之间可能会有差别，但在当前实践中，他们都只是简单地要求用SSL。例如，下面指示服务器只允许对相关资源做 HTTPS连接：

Xml代码   
<security-constraint>     
<!-- ... -->     
    <user-data-constraint>     
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>     
    </user-data-constraint>     
</security-constraint>

<security-constraint>  
<!-- ... -->  
<user-data-constraint>  
<transport-guarantee>CONFIDENTIAL</transport-guarantee>  
</user-data-constraint>  
</security-constraint>

l display-name  
security-constraint的这个很少使用的子元素给予可能由GUI工具使用的安全约束项一个名称。

9.3 分配角色名  
迄今为止，讨论已经集中到完全由容器（服务器）处理的安全问题之上了。但servlet以及JSP页面也能够处理它们自己的安全问题。  
例如，容器可能允许用户从bigwig或bigcheese角色访问一个显示主管人员额外紧贴的页面，但只允许bigwig用户修改此页面的参数。完成这种更细致的控制的一种常见方法是调用HttpServletRequset的isUserInRole方法，并据此修改访问。  
Servlet的 security-role-ref子元素提供出现在服务器专用口令文件中的安全角色名的一个别名。例如，假如编写了一个调用 request.isUserInRole（"boss"）的servlet，但后来该servlet被用在了一个其口令文件调用角色manager而不是boss的服务器中。下面的程序段使该servlet能够使用这两个名称中的任何一个。

Xml代码   
<servlet>     
<!-- ... -->     
    <security-role-ref>     
        <role-name>boss</role-name> <!-- New alias -->     
        <role-link>manager</role-link> <!-- Real name -->     
    </security-role-ref>     
</servlet>

<servlet>  
<!-- ... -->  
<security-role-ref>  
<role-name>boss</role-name> <!-- New alias -->  
<role-link>manager</role-link> <!-- Real name -->  
</security-role-ref>  
</servlet>

也可以在web-app内利用security-role元素提供将出现在role-name元素中的所有安全角色的一个全局列表。分别地生命角色使高级IDE容易处理安全信息。

10 控制会话超时

如果某个会话在一定的时间内未被访问，服务器可把它扔掉以节约内存。可利用HttpSession的setMaxInactiveInterval方法直接设置个别会话对象的超时值。如果不采用这种方法，则缺省的超时值由具体的服务器决定。但可利用session-config和session- timeout元素来给出一个适用于所有服务器的明确的超时值。超时值的单位为分钟，因此，下面的例子设置缺省会话超时值为三个小时（180分钟）。

Xml代码   
<session-config>     
    <session-timeout>180</session-timeout>     
</session-config>

<session-config>  
<session-timeout>180</session-timeout>  
</session-config>

11 Web应用的文档化

越来越多的开发环境开始提供servlet和JSP的直接支持。例子有Borland Jbuilder Enterprise Edition、Macromedia UltraDev、Allaire JRun Studio（写此文时，已被Macromedia收购）以及IBM VisuaAge for Java等。  
大量的web.xml元素不仅是为服务器设计的，而且还是为可视开发环境设计的。它们包括icon、display-name和discription等。  
可回忆一下，在web.xml内以适当地次序声明web-app子元素很重要。不过，这里只要记住icon、display-name和description是web.xml的web-app元素内的前三个合法元素即可。  
l icon  
icon元素指出GUI工具可用来代表Web应用的一个和两个图像文件。可利用small-icon元素指定一幅16 x 16的GIF或JPEG图像，用large-icon元素指定一幅32 x 32的图像。下面举一个例子：

Xml代码   
<icon>     
    <small-icon>/images/small-book.gif</small-icon>     
    <large-icon>/images/tome.jpg</large-icon>     
</icon>

<icon>  
<small-icon>/images/small-book.gif</small-icon>  
<large-icon>/images/tome.jpg</large-icon>  
</icon>

l display-name  
display-name元素提供GUI工具可能会用来标记此Web应用的一个名称。下面是个例子。  
<display-name>Rare Books</display-name>  
l description  
description元素提供解释性文本，如下所示：

Xml代码   
<description>     
This Web application represents the store developed for rare-books.com, an online bookstore specializing in rare and limited-edition books.      
</description>

<description>  
This Web application represents the store developed for rare-books.com, an online bookstore specializing in rare and limited-edition books.  
</description>

12 关联文件与MIME类型

服务器一般都具有一种让Web站点管理员将文件扩展名与媒体相关联的方法。例如，将会自动给予名为mom.jpg的文件一个image/jpeg的MIME 类型。但是，假如你的Web应用具有几个不寻常的文件，你希望保证它们在发送到客户机时分配为某种MIME类型。mime-mapping元素（具有 extension和mime-type子元素）可提供这种保证。例如，下面的代码指示服务器将application/x-fubar的MIME类型分配给所有以.foo结尾的文件。

Xml代码   
<mime-mapping>     
    <extension>foo</extension>     
    <mime-type>application/x-fubar</mime-type>     
</mime-mapping>

<mime-mapping>  
<extension>foo</extension>  
<mime-type>application/x-fubar</mime-type>  
</mime-mapping>

或许，你的Web应用希望重载（override）标准的映射。例如，下面的代码将告诉服务器在发送到客户机时指定.ps文件作为纯文本（text/plain）而不是作为PostScript（application/postscript）。

Xml代码   
<mime-mapping>     
    <extension>ps</extension>     
    <mime-type>application/postscript</mime-type>     
</mime-mapping>

<mime-mapping>  
<extension>ps</extension>  
<mime-type>application/postscript</mime-type>  
</mime-mapping>

13 定位TLD

JSP taglib元素具有一个必要的uri属性，它给出一个TLD（Tag Library Descriptor）文件相对于Web应用的根的位置。TLD文件的实际名称在发布新的标签库版本时可能会改变，但我们希望避免更改所有现有JSP页面。此外，可能还希望使用保持taglib元素的简练性的一个简短的uri。这就是部署描述符文件的taglib元素派用场的所在了。Taglib包含两个子元素：taglib-uri和taglib-location。taglib-uri元素应该与用于JSP taglib元素的uri属性的东西相匹配。Taglib-location元素给出TLD文件的实际位置。例如，假如你将文件chart-tags- 1.3beta.tld放在WebApp/WEB-INF/tlds中。现在，假如web.xml在web-app元素内包含下列内容。

Xml代码   
<taglib>     
    <taglib-uri>/charts.tld</taglib-uri>     
    <taglib-location>/WEB-INF/tlds/chart-tags-1.3beta.tld</taglib-location>     
</taglib>

<taglib>  
<taglib-uri>/charts.tld</taglib-uri>  
<taglib-location>/WEB-INF/tlds/chart-tags-1.3beta.tld</taglib-location>  
</taglib>

给出这个说明后，JSP页面可通过下面的简化形式使用标签库。  
<%@ taglib uri="/charts.tld" prefix="somePrefix" %>

14 指定应用事件监听程序

应用事件监听器程序是建立或修改servlet环境或会话对象时通知的类。它们是servlet规范的版本2.3中的新内容。这里只简单地说明用来向Web应用注册一个监听程序的web.xml的用法。  
注册一个监听程序涉及在web.xml的web-app元素内放置一个listener元素。在listener元素内，listener-class元素列出监听程序的完整的限定类名，如下所示：

Xml代码   
<listener>     
    <listener-class>package.ListenerClass</listener-class>     
</listener>

<listener>  
<listener-class>package.ListenerClass</listener-class>  
</listener>

虽然listener元素的结构很简单，但请不要忘记，必须正确地给出web-app元素内的子元素的次序。listener元素位于所有的servlet 元素之前以及所有filter-mapping元素之后。此外，因为应用生存期监听程序是serlvet规范的2.3版本中的新内容，所以必须使用 web.xml DTD的2.3版本，而不是2.2版本。  
例如，程序清单5-20给出一个名为ContextReporter的简单的监听程序，只要Web应用的Servlet-Context建立（如装载Web应用）或消除（如服务器关闭）时，它就在标准输出上显示一条消息。程序清单5-21给出此监听程序注册所需要的web.xml文件的一部分。

程序清单5-20 ContextReporterjava

Java代码   
package moreservlets;      
     
import javax.servlet.\*;      
import java.util.\*;      
     
public class ContextReporter implements ServletContextListener {      
    public void contextInitialized(ServletContextEvent event) {      
        System.out.println("Context created on " +  new Date() + ".");      
    }      
     
    public void contextDestroyed(ServletContextEvent event) {      
        System.out.println("Context destroyed on " +   new Date() + ".");      
    }      
}

package moreservlets;  
import javax.servlet.\*;  
import java.util.\*;  
public class ContextReporter implements ServletContextListener {  
public void contextInitialized(ServletContextEvent event) {  
System.out.println("Context created on " +  new Date() + ".");  
}  
public void contextDestroyed(ServletContextEvent event) {  
System.out.println("Context destroyed on " +   new Date() + ".");  
}  
}

程序清单5-21 web.xml（声明一个监听程序的摘录）

Xml代码   
<?xml version="1.0" encoding="ISO-8859-1"?>     
<!DOCTYPE web-app      
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"      
"<http://java.sun.com/dtd/web-app_2_3.dtd>">     
<web-app>     
<!-- ... -->     
    <filter-mapping> … </filter-mapping>     
    <listener>     
        <listener-class>package.ListenerClass</listener-class>     
    </listener>     
    <servlet> ... </servlet>     
<!-- ... -->     
</web-app>

<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"<http://java.sun.com/dtd/web-app_2_3.dtd>">  
<web-app>  
<!-- ... -->  
<filter-mapping> … </filter-mapping>  
<listener>  
<listener-class>package.ListenerClass</listener-class>  
</listener>  
<servlet> ... </servlet>  
<!-- ... -->  
</web-app>

15 J2EE元素

本节描述用作J2EE环境组成部分的Web应用的web.xml元素。这里将提供一个简明的介绍，详细内容可以参阅<http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf>的Java 2 Plantform Enterprise Edition版本1.3规范的第5章。  
l distributable  
distributable 元素指出，Web应用是以这样的方式编程的：即，支持集群的服务器可安全地在多个服务器上分布Web应用。例如，一个可分布的应用必须只使用 Serializable对象作为其HttpSession对象的属性，而且必须避免用实例变量（字段）来实现持续性。distributable元素直接出现在discription元素之后，并且不包含子元素或数据，它只是一个如下的标志。  
<distributable />  
l resource-env-ref  
resource -env-ref元素声明一个与某个资源有关的管理对象。此元素由一个可选的description元素、一个resource-env-ref- name元素（一个相对于java:comp/env环境的JNDI名）以及一个resource-env-type元素（指定资源类型的完全限定的类），如下所示：

Xml代码   
<resource-env-ref>     
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>     
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>     
</resource-env-ref>

<resource-env-ref>  
<resource-env-ref-name>jms/StockQueue</resource-env-ref-name>  
<resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>  
</resource-env-ref>

l env-entry  
env -entry元素声明Web应用的环境项。它由一个可选的description元素、一个env-entry-name元素（一个相对于java: comp/env环境JNDI名）、一个env-entry-value元素（项值）以及一个env-entry-type元素（java.lang程序包中一个类型的完全限定类名，java.lang.Boolean、java.lang.String等）组成。下面是一个例子：

Xml代码   
<env-entry>     
    <env-entry-name>minAmout</env-entry-name>     
    <env-entry-value>100.00</env-entry-value>     
    <env-entry-type>minAmout</env-entry-type>     
</env-entry>

<env-entry>  
<env-entry-name>minAmout</env-entry-name>  
<env-entry-value>100.00</env-entry-value>  
<env-entry-type>minAmout</env-entry-type>  
</env-entry>

l ejb-ref  
ejb -ref元素声明对一个EJB的主目录的应用。它由一个可选的description元素、一个ejb-ref-name元素（相对于java: comp/env的EJB应用）、一个ejb-ref-type元素（bean的类型，Entity或Session）、一个home元素（bean的主目录接口的完全限定名）、一个remote元素（bean的远程接口的完全限定名）以及一个可选的ejb-link元素（当前bean链接的另一个 bean的名称）组成。  
l ejb-local-ref  
ejb-local-ref元素声明一个EJB的本地主目录的引用。除了用local-home代替home外，此元素具有与ejb-ref元素相同的属性并以相同的方式使用。

给你的session加个监听器

今天一个学生问我怎么实现在网页里显示在线用户的名称——他已经使用了session，但是无法处理用户离开的情况，然后导致在线用户列表的无限增大。跟他说了自己在application中进行超时检查，更新application的时候就比较当前所有列表中的session是否超过自己指定的时间间隔。后来想了想，又给他提了使用给session加监听器的方法。但是提的时候自己也没有做过，所以只是说这种方式很复杂，建议他还是自己进行超时检查。刚才又看了看资料，发现实际上给session加监听器的方式很简单，不禁觉得自己有点误人子弟了，现在将方法写在这，借以告诫自己以后要严谨。  
首先写一个SessionBinder类,它实现了HttpSessionBindingListener接口的valueBound方法和valueUnbound方法，示例代码如下：

Java代码   
public class SessionBinder implements HttpSessionBindingListener {      
    public void valueBound(HttpSessionBindingEvent event){      
    //you can do anything you want!this method will be called when this binder is bind with any session.      
    }      
     
    public void valueUnbound(HttpSessionBindingEvent event) {      
    //you can do something while this session is invalidate      
    }      
}

public class SessionBinder implements HttpSessionBindingListener {  
public void valueBound(HttpSessionBindingEvent event){  
//you can do anything you want!this method will be called when this binder is bind with any session.  
}  
public void valueUnbound(HttpSessionBindingEvent event) {  
//you can do something while this session is invalidate  
}  
}

现在写好了SessionBinder，我们现在选择在一个servlet中向session中加入这个监听器——在jsp中的代码书写与此相同  
//省略前面的代码，此操作可能发生在servlet的doGet方法中，也可能是doPost方法中

Java代码   
HttpSession session = req.getSession(true);//首先获得需要加入监听器的session对象，req是HttpRequest对象      
SessionBinder sb = new SessionBinder();//建立一个监听器对象      
session.putValue("BinderObject",sb);//将监听器加入此session中，从此时开始执行sb的valueBound方法

    HttpSession session = req.getSession(true);//首先获得需要加入监听器的session对象，req是HttpRequest对象  
SessionBinder sb = new SessionBinder();//建立一个监听器对象  
session.putValue("BinderObject",sb);//将监听器加入此session中，从此时开始执行sb的valueBound方法

//省略后面的代码  
随后，如果整个session超时或者被用户中止之后，sb的valueUnbound自动执行