JSP/Servlet

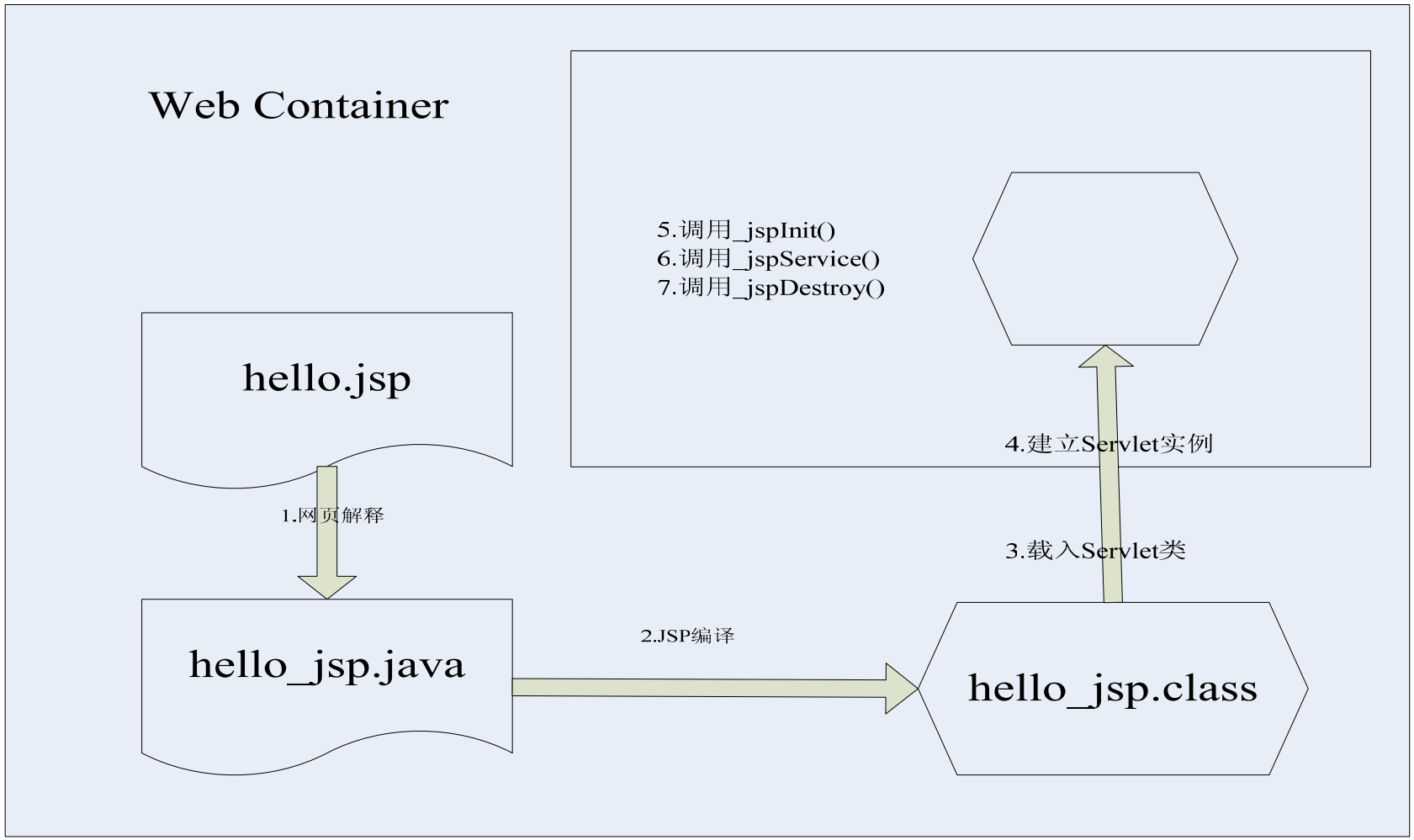
讲义3

第6章 JSP技术

JSP技术

- JSP本质上是开发Servlet的另外一种形式，它允许java代码和html代码混合使用，页面中静态的部分直接写html代码，而动态的部分用java代码来写，可以快捷的开发出web应用的视图。
- Web容器管理JSP页面生命周期主要分成两个阶段：
  - 转换阶段（translation phase）
  - 执行阶段（execution phase）
- 当客户端第一次请求一个JSP的时候，JSP页面转换成Servlet源文件，然后将这个Servlet源文件编译成Servlet的字节码文件，这个阶段称为转换阶段。
- 然后Servlet容器就执行Servlet的生命周期：加载Servlet类，实例化一个Servlet对象，执行初始化的_jspInit方法，调用_jspService方法实现请求的处理，最后执行回收的_jspDestroy方法。
- 只有第一次请求一个JSP的时候才完成转换阶段，如果是第二次就不会做转换，而是直接完成执行阶段。

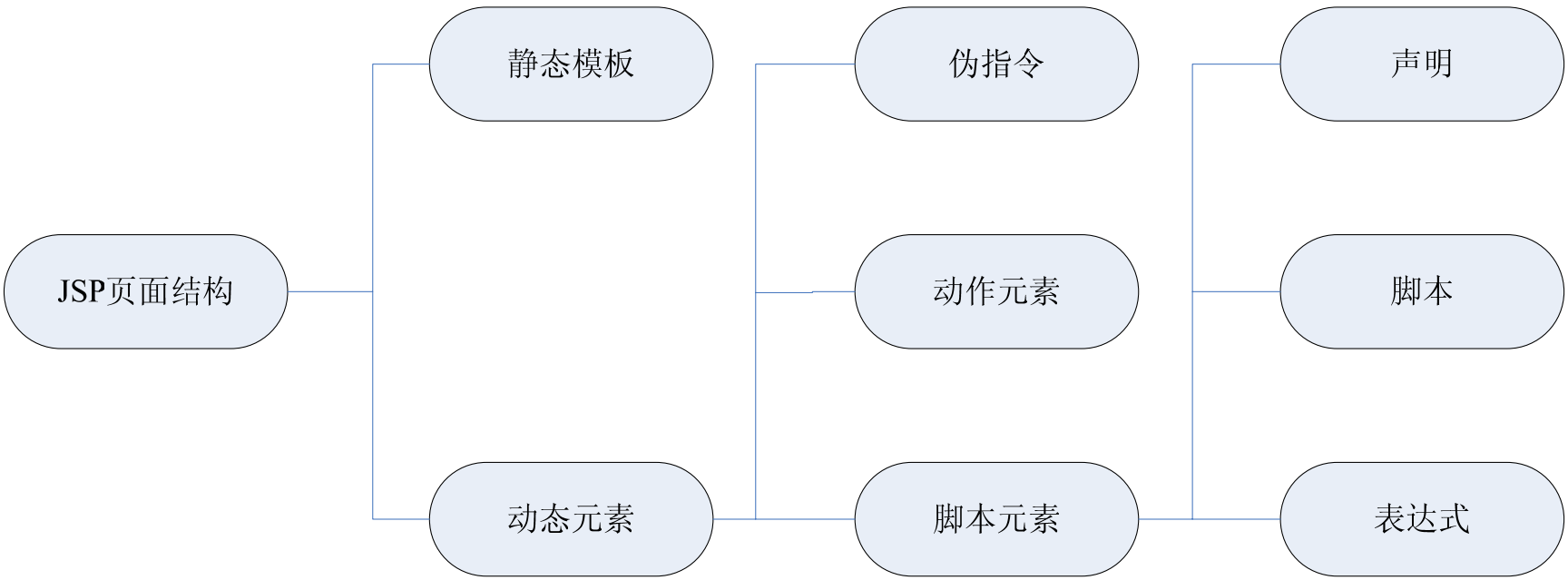
JSP执行原理


Web Container
5.调用_jspInit()
6.调用_jspService()
7.调用_jspDestroy()
hello.jsp
4.建立Servlet实例
1.网页解释
3.载入Servlet类
hello_jsp.java
2.JSP编译
hello_jsp.class

JSP语法组成

- 静态代码
  - html
  - css
  - javascript
- 动态代码
  - 指令元素
  - 动作元素
  - 脚本元素
    - 声明
    - 脚本
    - 表达式

JSP页面组成元素



指令元素

- JSP的指令元素主要用于转换阶段提供整体JSP页面的相关信息。
- 指令元素的语法格式如下
  - <%@ 指令名 属性1＝“值” 属性2＝“值”%>
- JSP中指令元素一共有三个
  - page，include，taglib

page指令

- page指令作用于整个JSP页面，定义了许多页面相关的属性，这些属性将被用于和JSP容器通信。
- page指令可以放在任何位置，作用范围都是整个jsp页面
- page指令的语法格式
  - <%@page 属性1=”值” 属性2=”值”...%>

page常用属性

- language 设置脚本语言，默认是java，也只能是java
- extends 需要继承的超类，没有用处
- import 一个很有用的属性，导入类，可以一次导入很多类。这是唯一一个可以重复设置的属性。
- session 是否可以使用session，默认true
- buffer 设置缓冲区大小，默认8kb
- autoFlush 该属性设置如果为false，会抛出异常，默认为true自动刷新缓冲区
- isThreadSafe 是否线程安全，没有用处
- info 页面信息，没有用处
- errorPage 如果该页发生异常，跳转到哪一个错误页
- isErrorPage 该页是不是另一个jsp的错误页
- contentType 当前页的mime类型和编码格式
- pageEncoding 字符编码
- isELIgnored 是否忽略EL标记

include指令

- include指令用于在JSP页面中静态包含一个文件，该文件可以是一个JSP页面、HTML网页、文本文件或一段Java代码。使用了include指令的JSP页面在转换的时候，JSP容器会将包含的文件的代码插入到当前页面中，一起进行编译
- include指令的语法格式
  - <%@ include file=”文件路径”%>

- 需要注意的是静态包含是把包含和被包含的文件最终的代码合成到一起，形成一个文件，各文件之间不能有冲突。例如多余的<html>，<body>之类或者重复声明的变量

taglib指令

- taglib指令允许页面使用用户定制的标签
- 语法如下
  - <%@taglib uri=“标签库URI” prefix=”前缀名”%>

脚本元素

- 脚本元素包括三个部分：
  - 声明
  - 脚本段
  - 表达式
- JSP2 . 0 增加了EL 表达式

脚本元素——声明

- 声明在其他脚本元素中可以使用的变量和方法，以<%!开始 %>结束

- 示例

```
<%!
private int i = 10;

public int m1(){
   return 5+2;
}
%>
```

- 需要注意的声明的变量在转换Servlet之后会成为全局变量，所以要考虑线程安全问题

脚本元素——脚本段

- 脚本段是在请求处理期间要执行的Java 代码段。脚本段可以产生输出，并将输出发送到客户端，也可以是一些流程控制语句。脚本段以＜％开始，以％＞结束

- 示例

```
<table width="100%" border="1">
    <%
        for (int i = 1; i <= 10; i++) {
            out.println("<tr><td>" + i + "</td></tr>");
        }
    %>
</table>
```

脚本元素——表达式

- 表达式脚本元素是Java 语言中完整的表达式，在请求处理时计算这些表达式，计算的结果将被转换为字符串，插入到当前的输出流中。表达式以<%=开始，以%>结束

- 示例

```
<table width="100%" border="1">
    <%for (int i = 1;i<=10;i ++){%>
    <tr>
        <td><%=i %></td>
    </tr>
    <%}%>
</table>
```

jsp文件注释

- 可以使用html注释<!--　-->
- 可以在<%%>中使用java的本身注释例如//，/*　*/
- 也可以使用专门的JSP注释<%--　--%> 用这种注释是不会输出到客户段的
  - 例如 <%-- 输出十行表格 --%>

JSP内置（隐含）对象

- 在JSP容器生成的Servlet 类的_jspService()方法中，定义了几个对象，而这些对象就是我们在编写JSP 页面时，可以使用的隐含对象。要注意的是，因为这些隐含对象是在_jspService()方法中定义的，所以我们只能在脚本段和表达式中使用这些对象

- 隐含对象一共有九个
  - request，response，session，pageContext，application，out，config，page和exception

内置对象解释

- request 相当于HttpServletRequest
- response 相当于HttpServletResponse
- session 相当于HttpSession
- application 相当于ServletContext

- config 相当于ServletConfig
- out 提供了输出流，输出数据到网页，最常见的方法就是out.print 和out.println方法
- page 相当于页面本身实例，很少用到。
- exception Throwable类型，如果本身是错误页，包含了当前异常信息
- pageContext PageContext类型，包含了访问其他对象8种对象的方法：例如获得session对象，就可以用pageContext.getSession()方法，也提供了访问任何一个访问内的属性的值的方法

内置对象的数据类型

隐含对象	类型
out	javax.servlet.jsp.JspWriter
response	javax.servlet.http.HttpServletResponse
pageContext	javax.servlet.jsp.PageContext
request	javax.servlet.http.HttpServletRequest
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
page	java.lang.Object
exception	java.lang.Throwable

第7章 动作元素

动作元素

- JSP的动作元素为请求处理阶段提供信息。动作元素遵循XML元素的语法。JSP2.0规范中一共定义了20个动作元素。这些动作元素在JSP转换为Servlet过程中将用固定的一段Java代码来替换。

<jsp:include>

- 这个动作元素用于在当前页面中包含静态和动态的资源。一旦被包含的页面执行完毕，请求处理将在调用的页面中继续执行。

- 语法格式

```
<jsp:include page=”包含的文件路径“ flush=true|false>
    [
      <jsp:param name=参数名 value=参数值/>
      ……
    ]
</jsp:include>
```

- flush参数默认为false，如果设置为true表示当缓冲区溢满时也正常输出，为false当缓冲区溢满时发生异常

两种包含文件形式的区别

- <%@include file=路径%> 静态包含，是在转换期间包含，形成一个类文件
- <jsp:include page=路径> 动态包含，在请求期间包含，是两个单独的类文件。

<jsp:forward>

- 请求转发给另外一个资源，和RequestDispatcher.forward功能相同
- 需要注意的是在转发之前，不能有任何数据已经输出到客户端，但是jsp输出数据是有缓存的，所以只要输出的数据没有超过缓存（默认8K）就没有问题
- 语法格式：

```
<jsp:forward page="转发的文件路径">
    [
      <jsp:param name=参数名 value=参数值/>
      ……
    ]
</jsp: forward >
```

JavaBean

- JavaBean 组件本质上就是一个类，只不过这个类需要遵循一些编码的约定。在JSP 页面中，既可以像使用普通类一样实例化JavaBean 类的对象，调用它的方法，也可以利用JSP 技术中提供的动作元素来访问JavaBean
- 一个标准的JavaBean 组件具有以下几个特性：
  - 它是一个公开的（public ）类。
  - 它有一个默认的构造方法，也就是不带参数的构造方法（在实例化JavaBean 对象时，需要调用默认的构造方法）。
  - 它提供setXXX()方法和getXXX()方法来让外部程序设置和获取JavaBean 的属性。换句话说，符合上述条件的类，我们都可以把它看成是JavaBean 组件。

JavaBean的属性命名规则

- 属性（Property）是JavaBean组件内部状态的抽象表示，外部程序使用属性来设置和获取JavaBean组件的状态。为了能够让外部的程序能够知道JavaBean提供了哪些属性，JavaBean的编写者必须标准的命名方式。
- 也就是为每一个属性添加对应的get和set访问器，其中属性名字的第一个字母大写，然后在每个名字前面加上相应的“get”和“set”。这样的属性是可读写属性，如果一个属性只有set方法，那么这个属性是只写属性，如果只有get方法，那么这个属性是只读属性。
- get/set命名方式的一个例外就是对于boolean类型的属性，应该使用is/set命名方式（也可以使用get/set命名方式）。

<jsp:useBean>

- 动作元素主要用于实例化JavaBean，或者定位一个已经存在的JavaBean实例，并把实例的引用赋给一个变量
- <jsp:useBean id=”name” scope=”page|request|session|application” class=“类型名称”/>
- 标签属性：
  - id属性：相当于引用变量的名称，是当前Bean实例的名字，例如上例中的user
  - scope属性：指定Bean实例的存储范围，四种范围page,request,session,applicaion，默认是page
  - class属性：指定Bean的完整类名

<jsp:useBean>动作的行为

- JSP容器在<jsp:useBean>元素指定的范围中查找指定的id的JavaBean对象。
- 如果找到相应的对象，则使用这个对象实例
- 如果没有在指定的范围中找到对象，则创建一个新的对象实例，将这个对象的引用赋值给由属性id所指定的名字的变量，并将这个对象保存到属性scope指定的范围中。

<jsp:setProperty>

- <jsp:setProperty>动作和<jsp:useBean>一起使用，用来设置JavaBean的属性值。
- <jsp:setProperty>动作使用Bean中的setXXX()方法，在Bean中设置一个或多个属性值。
- 在JSP中常常使用<jsp:setProperty>动作元素将客户端提交的数据保存到JavaBean的属性中。

<jsp:setProperty>属性

- name属性：指定Bean实例的名字
- property属性：指定Bean中需要被赋值的属性的名字，例如property=”username”，表示给username属性赋值，如果设置为*，例如property=”*”，则会自动查找传过来的参数名，按照匹配的属性名参数名进行一一赋值，但如果参数值为空字符串，则不会赋值
- param属性：指定请求参数的名字，如果请求参数名和javaBean属性名不一致，就可以指定对应的请求参数名，例如请求参数名是myusername，javaBean属性名是username，就需要指定property=”username” param=”myusername”，如果省略此参数，则认为请求参数名和属性名相同，param属性和value属性不能同时指定
- value属性：给属性显式赋值，例如想给username赋值为tom，则可以指定property=”username” value=”tom”，用此属性表示不利用请求参数赋值

<jsp:getProperty>

- 用于访问一个Bean的属性，并把属性的值转化为一个String，然后发送到输出流中。如果属性是一个对象，将调用该对象的toString()方法

- 标签属性
  - name属性：指定Bean实例的名字
  - property属性：指定属性的名字

第8章 异常处理

WEB应用的异常处理

- java web中有两种异常处理形式
  - 声明式异常
  - 程序式异常

声明式异常

- 就是在web.xml文件中用标签声明好可能会发生的异常且指定发生此异常时的异常处理程序。用这种方式可以针对HTTP错误，例如404，500错误。也可以针对java程序异常，例如空指针异常，下标越界等等。
- 使用声明式异常的标签如下：

```
<error-page>
    <error-code>|<exception-type>
    <location>
</error-page>
```

<error-code>是设置错误号，例如<error-code>404</error-code>

<exception-type>是设置异常类型的完整类名，例如<exception-type>java.lang.Exception</exception-type>

<location>是设置错误处理页的路径

程序式异常

- 就是用传统的try-catch方法来处理异常
- 可以创建一个统一的错误处理页，然后任何try-catch的处理都将错误信息转发到错误处理页中进行显示和处理

第9章 过滤器和监听器

过滤器 （Filter）

- 过滤器的主要作用是用户和请求资源之间设置一段程序，每次在到达资源之前都会所有的请求都会先被过滤器拦截执行一段程序然后在到达资源，中间的这段拦截程序就是过滤器
- Filter可以有多层，一个Filter的doFilter方法可以把请求继续传递到下一个Filter中，也可以选择让一个Filter执行后，不再调用下一个Filter

创建过滤器

- 实现Filter的步骤十分简单，只需要编写一个类扩展javax.servlet.Filter接口，然后在doFilter方法中写上对应的过滤代码，最后在web.xml文件中加上对应的注册信息就可以了
- Filter接口的三个方法
  - **public void** init(FilterConfig config)
    - 初始化方法，在过滤器被加载时执行
  - **public void** doFilter(ServletRequest request, ServletResponse response,FilterChain chain)
    - 主要方法，发出请求时会调用，传入3个参数，request,response,和FilterChain，FilterChain的作用是调用下一个过滤器，如果没有下一个过滤器，则调用原始资源。
  - **public void** destroy()
    - 过滤器销毁时调用此方法，不常用。

配置Filter

- 需要在web.xml文件中加上标签定义配置过滤器
- 语法格式：

```
<filter>
        <filter-name>过滤器名字</filter-name>
        <filter-class>过滤器的完整类名</filter-class>
</filter>
<filter-mapping>
        <filter-name>过滤器名字</filter-name>
        <url-pattern>需要过滤的资源的URL</url-pattern>
</filter-mapping>
```

配置Filter

- 其中<url-pattern>设置如果设置为/*，则表示过滤当前WEB应用的所有资源，如果只想过滤某一个目录或这某一个指定的URL也是可以的，例如/session/*表示只过滤WEB应用根目录下的session目录中的资源。
- 在一个WEB应用中可以配置多个过滤器，按照web.xml文件中配置的顺序一个一个执行
- <filter-mapping>标记可以设置多个，也就是可以过滤多个路径，如以下代码：

```
<filter-mapping>
        <filter-name>EncodingFilter</filter-name>
        <url-pattern>/filter/*</url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>EncodingFilter</filter-name>
        <url-pattern>/admin/*</url-pattern>
</filter-mapping>
```

监听器（Listener）

- Web应用程序可以使用Listener接口，来监听在web容器中的某一执行程序，并且根据应用程序的需求做出适当的响应
- 至Servlet2.4/Jsp2.0规范，一共有8个监听器接口和6个事件类

监听器接口和事件类的对应关系

Listener接口	Event类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	
HttpSessionAttributeListener	HttpSessionBindingEvent
HttpSessionBindingListener	
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

监听器分类

- 根据监听对象的不同可以分为3类

（**1**）与**ServletContext**相关的监听器

- ServletContextListener
- ServletContextAttributeListener

（**2**）与**HttpSession**相关的监听器

- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionBindingListener
- HttpSessionActivationListener

（**3**）与**ServletRequest**相关的监听器

- ServletRequestListener
- ServletRequestAttributeListener

创建监听器

- 一个类实现一个监听器接口即可成为一个对应的监听器类

- 示例

```
public class HttpSessionListenerImpl implements HttpSessionListener{
    //当有session被创建
    public void sessionCreated(HttpSessionEvent event) {
            System.out.println("有session被创建" + event.getSession().getId());
    }
    //当有session被销毁
    public void sessionDestroyed(HttpSessionEvent event) {
            System.out.println("有session被销毁" + event.getSession().getId());
    }
}
```

在web.xml文件中配置监听器标签

语法

```
<listener>
    <listener-class>监听器类完整类名</listener-class>
</listener>
```

示例

```
<listener>
    <listener-class>
        com.china.test.HttpSessionListenerImpl
    </listener-class>
</listener>
```