# 1. 过滤器(Filter)

## 1.1 过滤器简介

过滤器可以对资源的访问进行拦截。过滤器能拦截所有的对Web资源的请求和响应，包括Servlet、JSP、HTML、CSS和JS等（只要是属于该Web应用的资源）。

开发Web项目时，经常需要对用户的请求进行重复的操作，比如：判断用户是否已经登录（登录了才能展示相关页面）、设置请求的编码格式为UTF-8。

对每个请求这样重复写代码很麻烦，不利于管理。我们想对一些请求进行统一的过滤管理，这时就可考虑使用过滤器。

对请求使用了过滤器后，那么代码的执行流程就为：

（1）用户请求Servlet或JSP资源，该请求会先进入过滤器（即实现了对Servlet/JSP请求的拦截），过滤器中可对请求进行预处理。

（2）过滤器处理完毕后，放行该请求，这时请求才会到达Servlet或JSP。

（3）Servlet或JSP进行自己的处理，完毕后，这次的请求和响应还会返回到过滤器，最后由服务器响应给用户。

## 1.2 过滤器的使用

过滤器的开发步骤如下：

（1）新建一个类，实现Filter接口。该类就是过滤器类。Filter接口中有如下方法：

A：void init(FilterConfig filterConfig)：服务器启动时会执行init方法，可在init()里面写初始化内容。

方法提供了一个filterConfig对象参数，该对象可用于获取过滤器的初始化参数（即在web.xml中也可对过滤器配置参数，就像对Servlet配置参数，使用ServletConfig能拿到参数一样），FilterConfig接口提供了下面方法：

String getInitParameter(String name) ：获得指定名称的参数值

Enumeration getInitParameterNames()：获得所有参数名称的集合。

上述的用法还是和以前的一样。

B：void doFilter(request, response, FilterChain filterChain)：过滤器拦截的处理方法，传递了响应和请求的参数。主要就是写这里面的方法来预处理请求和响应数据。

另外方法参数还提供了一个filterChain对象，该对象用于将当前的请求放行，这样请求才能进入到Servlet/JSP或者下一个过滤器。调用以下方法对当前请求放行：

void doFilter(ServletRequest request, ServletResponse response)

所有的过滤器形成了过滤器链FilterChain，因此在过滤器中必须放行，若不放行，就无法进行下去。

C：void destroy()：服务器关闭时会自动调用destroy()方法来销毁过滤器实例。

现在简单编写一个过滤器如下：

|  |
| --- |
| **public class** MyFilter **implements** Filter {  **public** MyFilter() {  System.***out***.println(**"创建了过滤器实例"**);  }    @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  System.***out***.println(**"初始化过滤器"**);  *// 可获得过滤器中配置的参数。比如配置了名叫encoding的参数* System.***out***.println(filterConfig.getInitParameter(**"encoding"**));  }  *// 对于该过滤器拦截范围内的资源的，每次访问都会由容器来调用该方法。*  @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain)  **throws** IOException, ServletException {  System.***out***.println(**"执行过滤器处理方法。"**);  *// 现在不做什么处理，直接放行* filterChain.doFilter(req, resp);  *// Servlet处理完成后，又会回来。* System.***out***.println(**"Servlet/JSP响应完毕。"**);  }    @Override  **public void** destroy() {  System.***out***.println(**"销毁了Filter实例"**);  } } |

（2）在web.xml中配置Filter，即配置该过滤器要拦截的资源：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"**>  *<!-- 在filter节点中配置过滤器 -->* <**filter**>  *<!-- Filter的名称 -->* <**filter-name**>myFilter</**filter-name**>  *<!-- 指定过滤器类 -->* <**filter-class**>com.zhang.test.MyFilter</**filter-class**>  *<!-- 配置Filter的初始化参数。可设置多个，即有多个init-param标签 -->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>UTF-8</**param-value**>  </**init-param**>  </**filter**>  *<!-- 过滤器映射配置 -->* <**filter-mapping**>  *<!-- 过滤器名称，和上面一致即可 -->* <**filter-name**>myFilter</**filter-name**>  *<!-- 设置拦截的路径。设置为/\*表示拦截所有资源，包括Servlet/JSP/HTML等。 -->* <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**> </**web-app**> |

可以看出Filter和Servlet的配置方法基本一致。需要注意：

（1）/\*拦截了所有请求，包括静态资源，比如html页面、js文件等。

（2）若在web.xml配置Servlet，则Servlet的配置需要在Filter配置之后。

## 1.3 拦截指定的请求

有时不希望拦截全部请求，只希望拦截一部分请求。可用url-pattern和servlet-name说明拦截路径和要拦截的Servlet名称，并且支持通配符。一个filter-mapping中可配置多个url-pattern和servlet-name节点。示例如下：

|  |
| --- |
| <**filter-mapping**>  *<!-- 拦截指定的jsp -->* <**url-pattern**>/index.jsp</**url-pattern**>  *<!-- 拦截所有的jsp -->* <**url-pattern**>\*.jsp</**url-pattern**>  *<!-- 根据servlet的名称拦截 -->* <**servlet-name**>IndexServlet</**servlet-name**> </**filter-mapping**> |

可以自己定义组合来拦截。如果只想拦截Servlet，那么可以定个规范，比如让所有的Servlet都以Action开头，这样就好拦截了。

## 1.4 过滤器的串联

在web.xml中，可以为一个应用配置多个过滤器。如果某个资源在多个过滤器的拦截范围内，则一个过滤器放行后，会再进入下一个过滤器，进入过滤器的顺序是根据web.xml中过滤器的配置顺序来的。

## 1.5 过滤器的注解

同样，我们也能使用@WebFilter注解来对Filter进行配置，这样就不用自行配置web.xml了。缺点是使用注解不能确定过滤器执行的顺序。下面的应用中为了方便我们就会使用过滤器的注解。

## 1.6 过滤器的应用

### 1.6.1 设置Servlet请求和响应编码

之前，每次请求和响应几乎都要先设置编码，我们可以在过滤器中就实现编码的处理，使得GET和POST请求和响应不再考虑编码的情况。在过滤器中，我们将和请求和响应有关的编码全部设置好：

|  |
| --- |
| **package** com.webdemo;  **import** javax.servlet.\*; **import** java.io.IOException;  *// 拦截所有的包含servlet的请求路径，可以把所有servlet的路径这样命名以方便拦截* @WebFilter(urlPatterns = {**"/servlet/\*"**}) **public class** EncodingFilter **implements** Filter {  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException { }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) **throws** IOException, ServletException {  *// 先设置请求编码* req.setCharacterEncoding(**"UTF-8"**);  *// 预先就设置响应编码。因为等到响应回来再设置的话，就晚了。必须在write()之前设置。* resp.setCharacterEncoding(**"UTF-8"**);  resp.setContentType(**"text/html;charset=UTF-8"**);  *// 放行* chain.doFilter(req, resp);  }   @Override  **public void** destroy() { } } |

这样在Servlet中就不需要处理编码问题了。为了灵活性，也可在Filter的配置中读取要设置的编码：

|  |
| --- |
| **package** com.webdemo;  **import** javax.servlet.\*; **import** javax.servlet.annotation.WebFilter; **import** javax.servlet.annotation.WebInitParam; **import** java.io.IOException;  @WebFilter(urlPatterns = {**"/servlet/\*"**}, initParams = {@WebInitParam(name = **"encoding"**, value = **"utf-8"**)}) **public class** EncodingFilter **implements** Filter {   *// 维护过滤器的配置对象，以便在doFilter中使用* **private** FilterConfig **filterConfig**;   @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  **this**.**filterConfig** = filterConfig;  }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) **throws** IOException, ServletException {  *// 读取指定的参数* String encoding = **filterConfig**.getInitParameter(**"encoding"**);  **if** (encoding == **null**) {  encoding = **"UTF-8"**; *// 如果没有，就给定一个默认值UTF-8* }  *// 设置编码* req.setCharacterEncoding(encoding);  resp.setCharacterEncoding(encoding);  resp.setContentType(**"text/html;charset="** + encoding);  chain.doFilter(req, resp);  }   @Override  **public void** destroy() { } } |

上面的代码中使用注解配置了initParams，就相当于在web.xml配置init-param。由于initParams的类型是@WebInitParam数组，因此initParams的值用大括号包起来，里面再写@WebInitParam注解的设置。在实际中，我们还是推荐在web.xml中设置过滤器的参数，比较直观。

### 1.6.2 设置资源的缓存时间

浏览器能缓存请求到的网站资源，这样下次再访问同样的资源时就直接从本地读取，速度快效率高。但是在WEB开发中，一些动态资源是禁止被缓存的，例如Servlet等，否则每次访问动态资源会得到同样的数据。而网站中的一些静态资源在一定时间内是允许浏览器缓存的，例如HTML、CSS、JS和图片资源等，避免重复请求。

因此这里我们使用过滤器来设置资源的缓存时间。

下面的DynamicResourceCacheFilter过滤器代码可用于设置动态资源不缓存到浏览器：

|  |
| --- |
| **package** com.webdemo;  **import** javax.servlet.\*; **import** javax.servlet.annotation.WebFilter; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException;  *// 拦截servlet、jsp动态资源* @WebFilter(urlPatterns = {**"/servlet/\*"**, **"\*.jsp"**}) **public class** DynamicResourceCacheFilter **implements** Filter {  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException { }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest request;  HttpServletResponse response;  **try**{  *// 将上面的类强转成Http协议类型的类* request = (HttpServletRequest)req;  response = (HttpServletResponse)resp;  }**catch**(Exception e){  **throw new** ClassCastException(**"non-HTTP request or response"**);  }  response.setHeader(**"Expires"**, **"-1"**); *//控制缓存时间。只要比当前时间小即可，因此直接设置-1.* response.setHeader(**"Cache-Control"**, **"no-cache"**); *//HTTP1.1* response.setHeader(**"Pragma"**, **"no-cache"**); *//HTTP1.0* chain.doFilter(request, response);  }   @Override  **public void** destroy() { } } |

上述的代码中需要将ServletResponse和ServletRequest对象强转为HTTP类型的，才好使用其中的方法。实际上上面处理异常的代码是抄Tomcat的源码的，因为Tomcat实现的HttpServlet类的service方法中，就有这样的代码：

|  |
| --- |
| **public void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  HttpServletRequest request;  HttpServletResponse response;  **try** {  request = (HttpServletRequest)req;  response = (HttpServletResponse)res;  } **catch** (ClassCastException ex) {  **throw new** ServletException(**"non-HTTP request or response"**);  }   **this**.service(request, response); } |

学习别人好的代码也能提高自己。

现在再写一个可以设置静态资源缓存时间的过滤器StaticResourceCacheFilter类：

|  |
| --- |
| **package** com.webdemo;  **import** javax.servlet.\*; **import** javax.servlet.annotation.WebFilter; **import** javax.servlet.annotation.WebInitParam; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException;  @WebFilter(  urlPatterns = {**"\*.html"**, **"\*.css"**, **"\*.js"**},  initParams = {  @WebInitParam(name = **"html"**, value = **"120"**),  @WebInitParam(name = **"css"**, value = **"240"**),  @WebInitParam(name = **"js"**, value = **"120"**)  } ) **public class** StaticResourceCacheFilter **implements** Filter {   *// 维护filterConfig* **private** FilterConfig **filterConfig**;   @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  **this**.**filterConfig** = filterConfig;  }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest request;  HttpServletResponse response;  **try**{  request = (HttpServletRequest)req;  response = (HttpServletResponse)resp;  }**catch**(ClassCastException e){  **throw new** RuntimeException(**"non-HTTP request or response"**);  }   **long** time = 0;*//缓存的时间.单位是毫秒  // 截取uri后缀判断用户访问的资源。* String uri = request.getRequestURI();  String extName = uri.substring(uri.lastIndexOf(**"."**) + 1);  *// 根据扩展名设置缓存的时间* **if**(**"html"**.equals(extName)){  time = Long.*parseLong*(**filterConfig**.getInitParameter(**"html"**)) \* 60 \* 1000; *// 设置的是分钟，因此要这样计算* } **else if** (**"css"**.equals(extName)){  time = Long.*parseLong*(**filterConfig**.getInitParameter(**"css"**)) \* 60 \* 1000;  } **else if** (**"js"**.equals(extName)){  time = Long.*parseLong*(**filterConfig**.getInitParameter(**"js"**)) \* 60 \* 1000;  }  response.setDateHeader(**"Expires"**, System.*currentTimeMillis*() + time);  chain.doFilter(request, response);  }   @Override  **public void** destroy() { } } |

提醒一下，在开发时，所有的资源最好都设置成不缓存，因为要看开发的实时效果。特别是我用Chrome开发时，Chrome默认会缓存JS文件，当改动JS文件时，常常发现改动不生效，基本都是缓存的原因，需要清理缓存。当项目上线后，可适当使用浏览器缓存。

### 1.6.3 登录判断过滤器

对一些管理系统来说，除了主页、注册和登录等页面外，进入其他页面时都要判断用户是否登录了。若没有登录，则跳转到主页。可将用户登录信息保存在session中。

为了简化登录判断，我们写个过滤器来进行判断。过滤器中可设置不需要拦截的页面、拿到session进行判断等。

代码示例：

|  |
| --- |
| @WebFilter(urlPatterns = {**"/\*"**}) **public class** EncodingFilter **implements** Filter {  *// 先定义不需要过滤的页面* String[] **extraPages** = {**"index"**, **"login"**, **"signup"**};  *// 通过该方法判断该请求的地址是否需要拦截* **private boolean** needFilter(HttpServletRequest request) {  *// 得到请求地址* String uri = request.getRequestURI();  *// 截取得到页面名称* String pageName = uri.substring(uri.lastIndexOf(**"/"**) + 1, uri.length());  *// 遍历extraPages，看是否需要拦截* **boolean** flag = **false**;  **for** (String page : **extraPages**) {  **if** (page.equals(pageName)) {  flag = **true**;  **break**;  }  }  **return** flag;  }  @Override  **public void** init(FilterConfig filterConfig) {}   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain)  **throws** IOException, ServletException {  *// 将req和resp转换成HttpServletRequest和HttpServletResponse* HttpServletRequest request = (HttpServletRequest) req;  HttpServletResponse response = (HttpServletResponse) resp;  **if** (needFilter(request)) {  *// 需要拦截。则获得session* HttpSession session = request.getSession();  **if** (session == **null** || session.getAttribute(**"username"**) == **null**) {  *// 没有用户信息，则跳转到首页* request.getRequestDispatcher(**"/index"**).forward(request, response);  } **else** {  *// 放行* filterChain.doFilter(req, resp);  }  } **else** {  *// 不需拦截，直接放行* filterChain.doFilter(req, resp);  }  }   @Override  **public void** destroy() {} } |

### 1.6.4 自动登录过滤器

上面的登录判断过滤器和这里要写的自动登录过滤器是不一样的。上面的登录判断过滤器仅仅是判断当前session会话中是否存在已登录用户。而这个自动登录过滤器实现的功能说明如下：

一些论坛帖子资源网址、商城商品详情资源，就算用户不登录也是可以浏览的。如果用户不登录，网站的某块地方就让你登录，如果登录了就显示用户名。而且登录的时候，还提供“记住密码”的功能，如果勾选“记住密码”，则下次再访问网址就会自动登录显示用户名。

上述功能的实现需要同时使用session和cookie，session用于保存用户本次会话的信息（即显示用户名等），而cookie用于“记住密码”，即cookie中保存用户名和密码，这样下次用户访问该网址，虽然之前的session已失效，但是可以检查cookie中的登录信息来自动登录，再次把用户信息保存到session中，实现自动登录。

但是如果在每个Servlet中都判断cookie未免显得太麻烦，因此为了简化Servlet开发，我们使用过滤器完成这个功能。

# 2. 监听器(Listener)

## 2.1 监听器概述

一、含义：监听器用来监听服务器中特定对象的创建、销毁或者属性发生变化的情况。

二、监听器监听的对象以及作用：

监听器一般监听的是服务器自行创建的对象，即request、session和servletContext。因为自己创建使用的对象可自己控制，不需要监听。监听上述的服务器对象的创建、销毁和属性的改变，能够让开发者在特定的时机触发执行想要的自定义的操作。例如，如果想要在服务器启动时就执行一些操作，则可监听servletContext，当servletContext创建时就触发执行自定义代码。

三、有哪些监听器？Servlet提供了监听器接口，开发者只要自定义类实现监听器接口，即可创建好一个监听器。

（1）生命周期监听器。

生命周期监听器接口用于监听服务器对象的创建和销毁。Servlet提供了三个接口分别用于监听request、session和servletContext对象的创建和销毁：

1. ServletRequestListener接口：监听request对象的创建和销毁。

2. HttpSessionListener接口：监听session对象的创建和销毁。

3. ServletContextListener接口：监听servletContext对象的创建和销毁。

上述三个接口中都提供了对象创建和销毁事件发生时要调用的方法。开发者可自行写入各个时机需要执行的方法。（具体在下面讲）

（2）属性监听器。

属性监听器用于监听request、session和servletContext中属性的变化，即调用对象的setAttribute()方法时发生的属性（域）的变化情况。

1. ServletRequestAttributeListener接口：监听request对象属性的变化；

2. HttpSessionAttributeListener接口：监听session对象属性的变化；

3. ServletContextAttributeListener接口：监听servletContext对象属性的变化。

上述三个接口中都提供了增加属性、移除属性和改变属性状态时要调用的方法。开发者可自行写入各个时机需要执行的方法。

（2）Session相关监听器。

Servlet还提供了Session相关的监听器：

1. HttpSessionBidingListener：监听对象绑定到session上的事件；

2. HttpSessionActivationListener：监听session序列化及反序列化的事件（了解即可）。

## 2.2 监听器的使用

开发监听器的步骤：

（1）写一个普通Java类，实现相关监听器接口；

（2）在web.xml中配置监听器类，或者直接在监听器类上使用@WebListener注解。

### 2.2.1 生命周期监听器使用示例

例子：监听request对象的创建和销毁。分别在request对象创建和销毁时向控制台输出一行信息。

监听器类代码（使用注解）：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.ServletRequestEvent; **import** javax.servlet.ServletRequestListener; **import** javax.servlet.annotation.WebListener;  @WebListener **public class** MyRequestListener **implements** ServletRequestListener {  @Override  **public void** requestInitialized(ServletRequestEvent sre) {  *// 通过sre参数能拿到request对象。此时request对象刚创建，是拿不到域对象中值的* String name = (String) sre.getServletRequest().getAttribute(**"name"**);  System.***out***.println(**"获得的name属性："** + name);  System.***out***.println(**"request对象创建"**);  }   @Override  **public void** requestDestroyed(ServletRequestEvent sre) {  *// 这里就能拿到与对象中的值* String name = (String) sre.getServletRequest().getAttribute(**"name"**);  System.***out***.println(**"获得的name属性："** + name);  System.***out***.println(**"request对象销毁..."**);  } } |

上述代码中使用了@WebListener注解，并且在代码中测试了能否成功执行操作并拿到request和域对象。为了成功测试代码，我们在index.jsp中写入以下内容：

|  |
| --- |
| <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%** request.setAttribute(**"name"**, **"张三"**); **%>** 首页 </**body**> </**html**> |

这样，我们每访问一次index.jsp页面，控制台上就会产生如下信息：

|  |
| --- |
| 获得的name属性：null  request对象创建  获得的name属性：张三  request对象销毁... |

因为JSP页面也是一个Servlet，也会产生request对象。

使用HttpSessionListener和ServletContextListener也是类似的步骤，只是ServletContextListener的创建事件代码会在服务器启动时就执行（因为servletContext对象就是在服务器启动时创建的），而HttpSessionListener中创建事件代码也会稍后执行。

对于HttpSessionListener，需要注意的是：当浏览器关闭时，并不会调用HttpSessionListener中session销毁事件的代码，因为服务器端此时的该session对象并未被销毁。session的原理是服务器端拿到浏览器的SESSIONID，再查找服务器中是否有匹配的session对象。而浏览器关闭了再打开，之前的SESSIONID会消失，因此导致服务器并不能拿到之前的session对象，但这只是说明session失效了，不是session对象被销毁了。在Tomcat中，session默认30分钟销毁（可自行在web.xml中配置销毁时间），当然也可调用session.invalidate()方法主动销毁session对象，这样HttpSessionListener中的session销毁事件才会触发执行。

使用ServletContextListener进行监听使用较多，因为可在服务器启动时就执行一些操作。比如以后要学习的Spring框架，就是通过配置监听器来完成Spring容器加载的（Spring的ContextLoaderListener就实现了ServletContextListener接口）。

### 2.2.2 属性监听器示例

用HttpSessionAttributeListener监听属性状态变化。代码如下：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.http.HttpSessionAttributeListener; **import** javax.servlet.http.HttpSessionBindingEvent;  **public class** MySessionAttrListener **implements** HttpSessionAttributeListener {   @Override  **public void** attributeAdded(HttpSessionBindingEvent se) {  *// 向session中添加属性触发的事件  // 可直接拿到属性名和属性值* System.***out***.println(**"添加属性名为："** + se.getName());  System.***out***.println(**"添加属性值为："** + se.getValue());  }   @Override  **public void** attributeReplaced(HttpSessionBindingEvent se) {  System.***out***.println(**"重新设置属性名为："** + se.getName());  System.***out***.println(**"重新设置的属性值之前为："** + se.getValue());  *// 通过session拿到现在的值，其中也使用到se.getName()。* System.***out***.println(**"重新设置的属性值现在为："** + se.getSession().getAttribute(se.getName()));   }   @Override  **public void** attributeRemoved(HttpSessionBindingEvent se) {  System.***out***.println(**"移除的属性名为："** + se.getName());  System.***out***.println(**"移除的属性名值："** + se.getValue());  } } |

这里使用web.xml配置监听器：

|  |
| --- |
| <**listener**>  <**listener-class**>com.listener.demo.MySessionAttrListener</**listener-class**> </**listener**> |

在index.jsp中写：

|  |
| --- |
| <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%**  session.setAttribute(**"age"**, **"12"**);  session.setAttribute(**"age"**, **"20"**);  session.removeAttribute(**"age"**); **%>** 首页 </**body**> </**html**> |

这时访问JSP页面，控制台输出的信息为：

|  |
| --- |
| 添加属性名为：age  添加属性值为：12  重新设置属性名为：age  重新设置的属性值之前为：12  重新设置的属性值现在为：20  移除的属性名为：age  移除的属性名值：20 |

### 2.2.3 Session相关监听器示例

HttpSessionBindingListener监听对象绑定/解除绑定到sesison上的事件。通过示例来解释。

我们只要写一个自定义类（一般是实体类）实现HttpSessionBindingListener即可：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.http.HttpSessionBindingEvent; **import** javax.servlet.http.HttpSessionBindingListener;  **public class** Employee **implements** HttpSessionBindingListener {   *// 这个类基本上是一个实体类* **private** String **name**; *// 姓名* **public** Employee(String name) {  **this**.**name** = name;  }   @Override  **public void** valueBound(HttpSessionBindingEvent se) {  *// 该类对象绑定到session触发的事件* System.***out***.println(**"绑定了"** + se.getName());  }   @Override  **public void** valueUnbound(HttpSessionBindingEvent se) {  System.***out***.println(**"解绑了"** + se.getName());   }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

index.jsp代码如下：

|  |
| --- |
| <%@**page import**="**com.listener.demo.Employee**"%> <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%** session.setAttribute(**"emp"**, **new** Employee(**"张三"**)); *// 可触发绑定事件* session.removeAttribute(**"emp"**); *// 可触发解绑事件* **%>** 首页 </**body**> </**html**> |

运行时就会触发事件。即只要我们的自定义类对象绑定/解绑在了session中，就会触发相应事件。

需要注意的是，这里我们并没有配置“监听器”，即既没有配置注解，也没有配置web.xml。这是因为Employee这个对象是我们自己定义使用的，而不是服务器对象，因此无需“监听”。