# 过滤器（Filter）

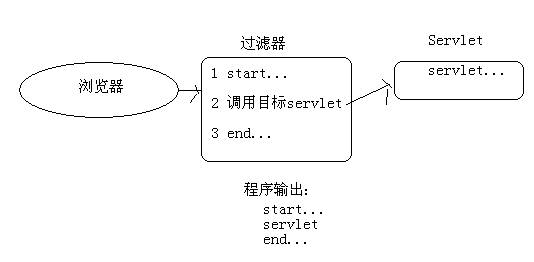
## 过滤器概述

### 什么是过滤器

过滤器是JavaWeb三大组件之一，它与Servlet很相似！不过过滤器是用来拦截请求的，而不是处理请求的。

当用户请求某个Servlet时，会先执行部署在这个请求上的Filter，如果Filter“放行”，那么会继承执行用户请求的Servlet；如果Filter不“放行”，那么就不会执行用户请求的Servlet。

其实可以这样理解，当用户请求某个Servlet时，Tomcat会去执行注册在这个请求上的Filter，然后是否“放行”由Filter来决定。可以理解为，Filter来决定是否调用Servlet！当执行完成Servlet的代码后，还会执行Filter后面的代码，决定是否响应给客户端。



### 过滤器之hello world

　　其实过滤器与Servlet很相似，我们回忆一下如果写的第一个Servlet应用！写一个类，实现Servlet接口！没错，写过滤器就是写一个类，实现Filter接口。

|  |
| --- |
| **public** **class** HelloFilter **implements** Filter {  //不去理会它--服务器创建Filter调用  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {}  //当访问被拦截资源时，doFilter()方法会被调用！我们先不去管它的参数是什么作用！  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("Hello Filter");  }  //不去理会它----服务器销毁Filter调用  **public** **void** destroy() {}  } |

第二步也与Servlet一样，在web.xml文件中部署Filter：

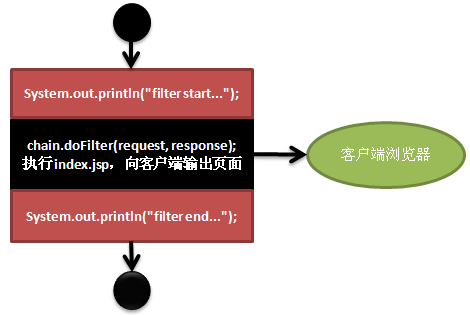
|  |
| --- |
| <filter>  <filter-name>helloFilter</filter-name>  <filter-class>cn.itcast.filter.HelloFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>helloFilter</filter-name>  //指定要拦截的路径！当用户访问index.jsp页面时，HelloFilter就会被执行  <url-pattern>/index.jsp</url-pattern>  </filter-mapping> |

应该没有问题吧，都可以看懂吧！

OK了，现在可以尝试去访问index.jsp页面了，看看是什么效果！

当用户访问index.jsp页面时，会执行HelloFilter的doFilter()方法！在我们的示例中，index.jsp页面是不会被执行的，如果想执行index.jsp页面，那么我们需要放行！

|  |
| --- |
| **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  //执行index.jsp之前执行  System.*out*.println("filter start...");  //放行！表示执行index.jsp  chain.doFilter(request, response);  //在执行index.jsp后执行这一句  System.*out*.println("filter end...");  } |



　　有很多同学总是错误的认为，一个请求在给客户端输出之后就算是结束了，这是不对的！其实很多事情都需要在给客户端响应之后才能完成！

## 过滤器详细

### 过滤器的生命周期

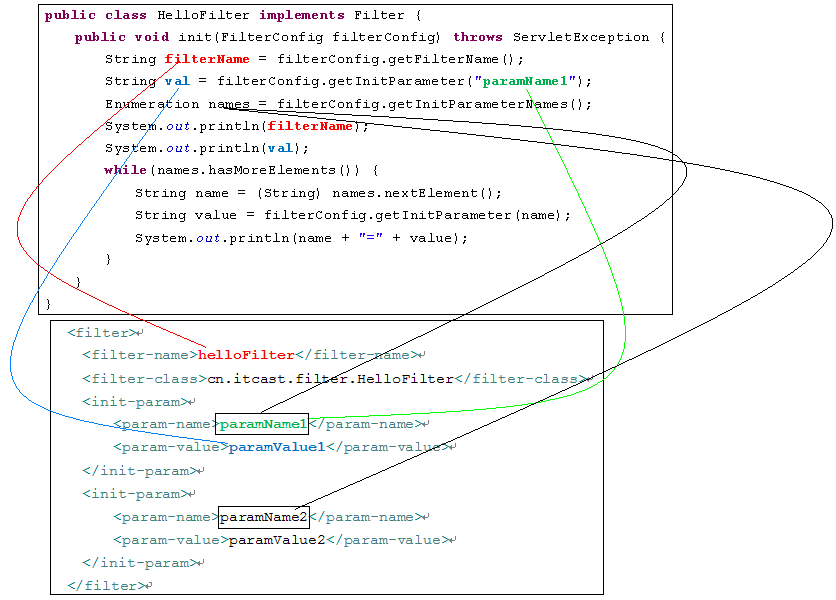
我们已经学习过Servlet的生命周期，那么Filter的生命周期也就没有什么难度了！

* init(FilterConfig)：在服务器启动时会创建Filter实例，并且每个类型的Filter只创建一个实例，从此不再创建！在创建完Filter实例后，会马上调用init()方法完成初始化工作，这个方法只会被执行一次；
* doFilter(ServletRequest req,ServletResponse res,FilterChain chain)：这个方法会在用户每次访问“目标资源（<url->pattern>index.jsp</url-pattern>）”时执行，如果需要“放行”，那么需要调用FilterChain的doFilter(ServletRequest,ServletResponse)方法，如果不调用FilterChain的doFilter()方法，那么目标资源将无法执行；
* destroy()：服务器会在创建Filter对象之后，把Filter放到缓存中一直使用，通常不会销毁它。一般会在服务器关闭时销毁Filter对象，在销毁Filter对象之前，服务器会调用Filter对象的destory()方法。

### FilterConfig

你已经看到了吧，Filter接口中的init()方法的参数类型为FilterConfig类型。它的功能与ServletConfig相似，与web.xml文件中的配置信息对应。下面是FilterConfig的功能介绍：

* ServletContext getServletContext()：获取ServletContext的方法；
* String getFilterName()：获取Filter的配置名称；与<filter-name>元素对应；
* String getInitParameter(String name)：获取Filter的初始化配置，与<init-param>元素对应；
* Enumeration getInitParameterNames()：获取所有初始化参数的名称。



### FilterChain

doFilter()方法的参数中有一个类型为FilterChain的参数，它只有一个方法：doFilter(ServletRequest,ServletResponse)。

前面我们说doFilter()方法的放行，让请求流访问目标资源！但这么说不严密，其实调用该方法的意思是，“我（当前Filter）”放行了，但不代表其他人（其他过滤器）也放行。

也就是说，一个目标资源上，可能部署了多个过滤器，就好比在你去北京的路上有多个打劫的匪人（过滤器），而其中第一伙匪人放行了，但不代表第二伙匪人也放行了，所以调用FilterChain类的doFilter()方法表示的是执行下一个过滤器的doFilter()方法，或者是执行目标资源！

如果当前过滤器是最后一个过滤器，那么调用chain.doFilter()方法表示执行目标资源，而不是最后一个过滤器，那么chain.doFilter()表示执行下一个过滤器的doFilter()方法。

### 多个过滤器执行顺序

一个目标资源可以指定多个过滤器，过滤器的执行顺序是在web.xml文件中的部署顺序：

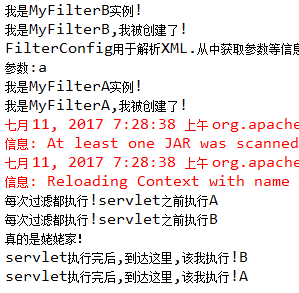
|  |
| --- |
| <filter>  <!--因为MyFilter1配置在前面，所以先执行MyFilter1的doFilter()方法。-->  <filter-name>myFilter1</filter-name>  <filter-class>cn.itcast.filter.MyFilter1</filter-class>  </filter>  <filter-mapping>  <filter-name>myFilter1</filter-name>  <url-pattern>/index.jsp</url-pattern>  </filter-mapping>  <filter>  <filter-name>myFilter2</filter-name>  <filter-class>cn.itcast.filter.MyFilter2</filter-class>  </filter>  <filter-mapping>  <filter-name>myFilter2</filter-name>  <url-pattern>/index.jsp</url-pattern>  </filter-mapping> |
| **public** **class** MyFilter1 **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request, HttpServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("filter1 start...");  chain.doFilter(request, response);//放行，执行MyFilter2的doFilter()方法  System.*out*.println("filter1 end...");  }  } |
| **public** **class** MyFilter2 **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request, HttpServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("filter2 start...");  chain.doFilter(request, response);//放行，执行目标资源  System.*out*.println("filter2 end...");  }  } |
| <body>  This is my JSP page. <br>  <h1>index.jsp</h1>  <%System.out.println("index.jsp"); %>  </body> |

当有用户访问index.jsp页面时，输出结果如下：

|  |
| --- |
| filter1 start...  filter2 start...  index.jsp  filter2 end...  filter1 end... |

但Filter的init不在此执行书序中,web,xml中的配置顺序只决定了doFilter()的执行顺序

|  |
| --- |
| <filter>  <filter-name>MyFilterA</filter-name>  <filter-class>com.filter.demo.MyFilterA</filter-class>  </filter>  <filter-mapping>  <filter-name>MyFilterA</filter-name>  <!--指定要拦截的路径！当用户访问index.jsp页面时，MyFilterA就会被执行 -->  <url-pattern>/index.jsp</url-pattern>  </filter-mapping>  <filter-mapping>  <filter-name>MyFilterB</filter-name>  <!--使用通配符拦截所有Servlet -->  <url-pattern>/\*</url-pattern>  </filter-mapping>  <filter>  <filter-name>MyFilterB</filter-name>  <filter-class>com.filter.demo.MyFilterB</filter-class>  <!--参数配置 -->  <init-param>  <param-name>A</param-name>  <param-value>a</param-value>  </init-param>  <init-param>  <param-name>B</param-name>  <param-value>b</param-value>  </init-param>  </filter> |
| package com.filter.demo;  import java.io.IOException;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.annotation.WebFilter;  /\*\*  \* Servlet Filter implementation class MyFilterA  \*/  public class MyFilterA implements Filter {  /\*\*  \* Default constructor.  \*/  public MyFilterA() {  System.out.println("我是MyFilterA实例!");  }  /\*\*  \* Filter会在tomcat关闭时被销毁  \* Filter被销毁之前执行，只执行一次  \*/  public void destroy() {  System.out.println("哎呀不好,Tomcat要关闭了,我要死了!A");  }  /\*\*  \* @see Filter#doFilter(ServletRequest, ServletResponse, FilterChain)  \* \* 每次过滤都会执行！  \*/  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  System.out.println("每次过滤都执行!servlet之前执行A");  //此方法用于将servlet放行至下一个Filter,或者让servlet执行,若没有此操作,servlet不会被放行  chain.doFilter(request, response);  System.out.println("servlet执行完后,到达这里,该我执行!A");  }  /\*\*  \* @see Filter#init(FilterConfig)  \* Filter会在tomcat启动时被创建！  \* Filter实例创建之后，马上执行！只执行一次！  \*/  public void init(FilterConfig fConfig) throws ServletException {  System.out.println("我是MyFilterA,我被创建了!");  }  } |
| **package** com.filter.demo;  **import** java.io.IOException;  **import** javax.servlet.Filter;  **import** javax.servlet.FilterChain;  **import** javax.servlet.FilterConfig;  **import** javax.servlet.ServletException;  **import** javax.servlet.ServletRequest;  **import** javax.servlet.ServletResponse;  **public** **class** MyFilterB **implements** Filter{  **public** MyFilterB() {  System.*out*.println("我是MyFilterB实例!");  }  /\*\*  \* Filter会在tomcat关闭时被销毁  \* Filter被销毁之前执行，只执行一次  \*/  **public** **void** destroy() {  System.*out*.println("哎呀不好,Tomcat要关闭了,我要死了!B");  }  /\*\*  \* **@see** Filter#doFilter(ServletRequest, ServletResponse, FilterChain)  \* \* 每次过滤都会执行！  \*/  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("每次过滤都执行!servlet之前执行B");  //此方法用于将servlet放行至下一个Filter,或者让servlet执行,若没有此操作,servlet不会被放行,但仍会将filter执行完  chain.doFilter(request, response);  System.*out*.println("servlet执行完后,到达这里,该我执行!B");  }  /\*\*  \* **@see** Filter#init(FilterConfig)  \* Filter会在tomcat启动时被创建！  \* Filter实例创建之后，马上执行！只执行一次！  \*/  **public** **void** init(FilterConfig fConfig) **throws** ServletException {  System.*out*.println("我是MyFilterB,我被创建了!");  System.*out*.println("FilterConfig用于解析XML.从中获取参数等信息:FilterName:"+fConfig.getFilterName());  System.*out*.println("参数:"+fConfig.getInitParameter("A"));  }  } |





### 四种拦截方式

我们来做个测试，写一个过滤器，指定过滤的资源为b.jsp，然后我们在浏览器中直接访问b.jsp，你会发现过滤器执行了！

但是，当我们在a.jsp中request.getRequestDispathcer(“/b.jsp”).forward(request,response)时，就不会再执行过滤器了！也就是说，默认情况下，只能直接访问目标资源才会执行过滤器，而forward执行目标资源，不会执行过滤器！

|  |
| --- |
| **public** **class** MyFilter **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request,  HttpServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  System.*out*.println("myfilter...");  chain.doFilter(request, response);  }  } |
| <filter>  <filter-name>myfilter</filter-name>  <filter-class>cn.itcast.filter.MyFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  </filter-mapping> |
| <body>  <h1>b.jsp</h1>  </body> |
| <h1>a.jsp</h1>  <%  request.getRequestDispatcher("/b.jsp").forward(request, response);  %>  </body> |

http://localhost:8080/filtertest/b.jsp -->直接访问b.jsp时，会执行过滤器内容；

http://localhost:8080/filtertest/a.jsp --> 访问a.jsp，但a.jsp会forward到b.jsp，这时就不会执行过滤器！

其实过滤器有四种拦截方式！分别是：REQUEST、FORWARD、INCLUDE、ERROR。

* REQUEST：直接访问目标资源时执行过滤器。包括：在地址栏中直接访问、表单提交、超链接、重定向，只要在地址栏中可以看到目标资源的路径，就是REQUEST；
* FORWARD：转发访问执行过滤器。包括RequestDispatcher#forward()方法、<jsp:forward>标签都是转发访问；
* INCLUDE：包含访问执行过滤器。包括RequestDispatcher#include()方法、<jsp:include>标签都是包含访问；
* ERROR：当目标资源在web.xml中配置为<error-page>中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。

可以在<filter-mapping>中添加0~n个<dispatcher>子元素，来说明当前访问的拦截方式。

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  //b.jsp为目标资源，当直接请求b.jsp时，会执行过滤器  <dispatcher>REQUEST</dispatcher>  //当转发到b.jsp页面时，会执行过滤器  <dispatcher>FORWARD</dispatcher>  </filter-mapping> |
| //当没有给出拦截方式时，那么默认为REQUEST  <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  </filter-mapping> |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  //转发到b.jsp页面时，会执行过滤器！因为已经给出了<dispatcher>FORWARD</dispatcher>了，那么就没有默认的REQUEST了！所以只有在转发到b.jsp时才会执行过滤，而转发到b.jsp时，不会执行  <dispatcher>FORWARD</dispatcher>  </filter-mapping> |

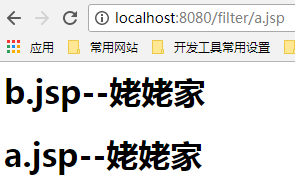
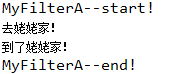
其实最为常用的就是REQUEST和FORWARD两种拦截方式，而INCLUDE和ERROR都比较少用！其中INCLUDE比较好理解，我们这里不再给出代码，学员可以通过FORWARD方式修改，来自己测试。而ERROR方式不易理解，下面给出ERROR拦截方式的例子：

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  //拦截方式为ERROR  <dispatcher>ERROR</dispatcher>  </filter-mapping>  <error-page>  <error-code>500</error-code>  //把b.jsp执行为500的错误页面  <location>/b.jsp</location>  </error-page> |
| <body>  <h1>a.jsp</h1>  <%  **if**(**true**)  //当用户访问a.jsp页面时会抛出异常，即500了！  这时服务器会转发到b.jsp，在这之前会执行过滤器！  **throw** **new** RuntimeException("嘻嘻~");  %>  </body> |

###### 实践说明:

下面分别为a.jsp b.jsp MyFilterA

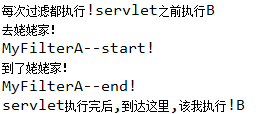
|  |
| --- |
| <h1>a.jsp--姥姥家</h1>  <%  System.out.println("去姥姥家！");  %>  <%request.getRequestDispatcher("/b.jsp").include(request, response); %>  </body> |
| <h1>b.jsp--姥姥家</h1>  <%  System.out.println("到了姥姥家！");  %> |
| **package** com.filter.filterStyle;  **public** **class** MyFilterA **implements** Filter{  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("MyFilterA--start!");  chain.doFilter(request, response);  System.*out*.println("MyFilterA--end!");    }  } |
| <filter>  <filter-name>AMyFilter</filter-name>  <filter-class>com.filter.filterStyle.MyFilterA</filter-class>  </filter>  <filter-mapping>  <filter-name>AMyFilter</filter-name>  <!--指定要拦截的路径！当用户访问index.jsp页面时，MyFilterA就会被执行 -->  <url-pattern>/\*</url-pattern>  </filter-mapping> |



查看上面的代码我们发现涉及到了两个jsp页面,那么这时我们就要想了是不是会执行两次拦截呢,可实际操作却发现并非如此,只执行了一次,那么这是为什么呢?

通过阅读上面我们知道Filter有四种拦截方式,在这四种拦截方式中Filter默认对REQUEST拦截,其他三种拦截则是需要在web.xml中配置才能拦截,不配置则不会拦截,而在上面很明显是一个INCLUDE拦截,但是我们并未在web.xml中配置,如果想让Filter对请求包含也拦截,那么只需要在web.xml中配置即可.

|  |
| --- |
| <filter>  <filter-name>AMyFilter</filter-name>  <filter-class>com.filter.filterStyle.MyFilterA</filter-class>  </filter>  <filter-mapping>  <filter-name>AMyFilter</filter-name>  <!--指定要拦截的路径！当用户访问index.jsp页面时，MyFilterA就会被执行 -->  <url-pattern>/\*</url-pattern>  <dispatcher>INCLUDE</dispatcher>  </filter-mapping> |



此时MyFilterA就只会对请求包含进行拦截,a.jsp倒没有拦截.

### 过滤器的应用场景

过滤器的应用场景：

* 执行目标资源之前做**预处理**工作，例如设置编码，这种试**通常都会放行**，只是在目标资源执行之前做一些准备工作；
* 通过条件判断是否放行，例如校验当前用户是否已经登录，或者用户IP是否已经被禁用；
* 在目标资源执行后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理；

### 设置目标资源

在web.xml文件中部署Filter时，可以通过“\*”来执行目标资源：表示过滤所有资源

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

这一特性与Servlet完全相同！通过这一特性，我们可以在用户访问敏感资源时，执行过滤器，例如：<url-pattern>/admin/\*<url-pattern>，可以把所有管理员才能访问的资源放到/admin路径下，这时可以通过过滤器来校验用户身份。

还可以为<filter-mapping>指定目标资源为某个Servlet，例如：

|  |
| --- |
| <servlet>  <servlet-name>**myservlet**</servlet-name>  <servlet-class>cn.itcast.servlet.MyServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>**myservlet**</servlet-name>  <url-pattern>/abc</url-pattern>  </servlet-mapping>  <filter>  <filter-name>myfilter</filter-name>  <filter-class>cn.itcast.filter.MyFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>myfilter</filter-name>  //这里没有指定<url-pattern>，而是指定<servlet-name>！注意，它与某个Servlet的配置名称相同！  <servlet-name>**myservlet**</servlet-name>  </filter-mapping> |

当用户访问http://localhost:8080/filtertest/abc时，会执行名字为myservlet的Servlet，这时会执行过滤器。

### Filter小结

Filter的三个方法：

* void init(FilterConfig)：在Tomcat启动时被调用；
* void destroy()：在Tomcat关闭时被调用；
* void doFilter(ServletRequest,ServletResponse,FilterChain)：每次有请求时都调用该方法；

FilterConfig类：与ServletConfig相似，用来获取Filter的初始化参数

* ServletContext getServletContext()：获取ServletContext的方法；
* String getFilterName()：获取Filter的配置名称；
* String getInitParameter(String name)：获取Filter的初始化配置，与<init-param>元素对应；
* Enumeration getInitParameterNames()：获取所有初始化参数的名称。

FilterChain类：

* void doFilter(ServletRequest,ServletResponse)：放行！表示执行下一个过滤器，或者执行目标资源。可以在调用FilterChain的doFilter()方法的前后添加语句，在FilterChain的doFilter()方法之前的语句会在目标资源执行之前执行，在FilterChain的doFilter()方法之后的语句会在目标资源执行之后执行。

四各拦截方式：REQUEST、FORWARD、INCLUDE、ERROR，默认是REQUEST方式。

* REQUEST：拦截直接请求方式；
* FORWARD：拦截请求转发方式；
* INCLUDE：拦截请求包含方式；
* ERROR：拦截错误转发方式。

# 过滤器应用案例

## 分ip统计网站的访问次数

### 1　说明

　　网站统计每个IP地址访问本网站的次数。

### 2　分析

因为一个网站可能有多个页面，无论哪个页面被访问，都要统计访问次数，所以使用过滤器最为方便。

因为需要分IP统计，所以可以在过滤器中创建一个Map，使用IP为key，访问次数为value。当有用户访问时，获取请求的IP，如果IP在Map中存在，说明以前访问过，那么在访问次数上加1，即可；IP在Map中不存在，那么设置次数为1。

把这个Map存放到ServletContext中！

### 3　代码

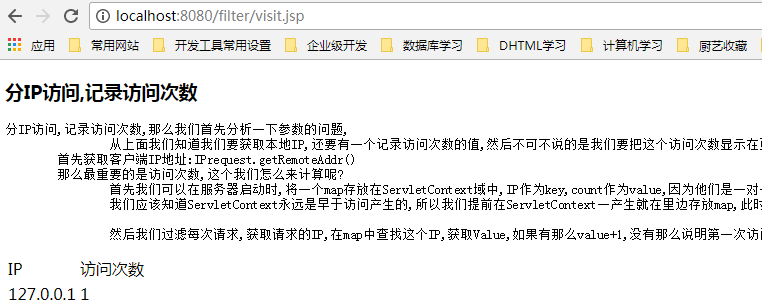
index.jsp

|  |
| --- |
| <body>  <h1>分IP统计访问次数</h1>  <table align=*"center"* width=*"50%"* border=*"1"*>  <tr>  <th>IP地址</th>  <th>次数</th>  </tr>  //循环遍历在ServletContext中的map，其中key是ip地址，value是访问次数  <c:forEach items="${applicationScope.ipCountMap }" var=*"entry"*>  <tr>  <td>${entry.key }</td>  <td>${entry.value }</td>  </tr>  </c:forEach>  </table>  </body> |

IPFilter

|  |
| --- |
| **public** **class** IPFilter **implements** Filter {  **private** ServletContext context;  **public** **void** init(FilterConfig fConfig) **throws** ServletException {  //保存ServletContext  context = fConfig.getServletContext();  //创建一个Map，保存到ServletContext中  Map<String, Integer> ipCountMap = Collections  .*synchronizedMap*(**new** LinkedHashMap<String, Integer>());  context.setAttribute("ipCountMap", ipCountMap);  }  @SuppressWarnings("unchecked")  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  String ip = req.getRemoteAddr();//获取请求方的ip  //在context中获取Map  Map<String, Integer> ipCountMap = (Map<String, Integer>) context  .getAttribute("ipCountMap");  //在Map中获取当前ip的访问次数  果这个ip在map中不存在，那么设置访问次数为1  Integer count = ipCountMap.get(ip);  **if** (count == **null**) {  count = 1;  } **else** {  //否则在原有次数上加1  count += 1;  }  ipCountMap.put(ip, count);  //把map存放到context中  context.setAttribute("ipCountMap", ipCountMap);  chain.doFilter(request, response);放行！  }  **public** **void** destroy() {}  } |
| <filter>  <display-name>IPFilter</display-name>  <filter-name>IPFilter</filter-name>  <filter-class>cn.itcast.filter.ip.IPFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>IPFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

|  |
| --- |
| <h3>分IP访问,记录访问次数</h3>  <pre>分IP访问,记录访问次数,那么我们首先分析一下参数的问题,  从上面我们知道我们要获取本地IP,还要有一个记录访问次数的值,然后不可不说的是我们要把这个访问次数显示在页面上,那么是不是还需要一个域呢  首先获取客户端IP地址:IPrequest.getRemoteAddr()  那么最重要的是访问次数,这个我们怎么来计算呢?  首先我们可以在服务器启动时,将一个map存放在ServletContext域中,IP作为key,count作为value,因为他们是一对一的关系  我们应该知道ServletContext永远是早于访问产生的,所以我们提前在ServletContext一产生就在里边存放map,此时我们就用到了监听器,监听ServletContext产生    然后我们过滤每次请求,获取请求的IP,在map中查找这个IP,获取Value,如果有那么value+1,没有那么说明第一次访问,将IP放入map  </pre>  <table>  <tr>  <td>IP</td>  <td>访问次数</td>  </tr>  <c:forEach items=*"*${applicationScope.map }*"* var=*"entry"*>  <tr>  <td>${entry.key }</td>  <td>${entry.value }</td>  </tr>  </c:forEach>  </table> |
| <listener>  <listener-class>com.filter.practice.visitCount.CountListener</listener-class>  </listener>  <filter>  <filter-name>CountFilter</filter-name>  <filter-class>com.filter.practice.visitCount.CountFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>CountFilter</filter-name>  <url-pattern>/visit.jsp</url-pattern>  </filter-mapping> |
| package com.filter.practice.visitCount;  public class CountListener implements ServletContextListener{  @Override  public void contextDestroyed(ServletContextEvent arg0) {  // TODO Auto-generated method stub    }  @Override  public void contextInitialized(ServletContextEvent event) {  System.out.println("把空map保存到servletcontext对象中...");  /\*  \* 1. 创建一个空Map，保存到ServletContext中  \*/  Map<String, Integer> map = new LinkedHashMap<>();  ServletContext sc = event.getServletContext();  sc.setAttribute("map", map);  }  }  **package** com.filter.practice.visitCount;  **public** **class** CountFilter **implements** Filter{  **private** FilterConfig fConfig;  @Override  **public** **void** destroy() {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  System.*out*.println("start!");  //获取IP  String ip = req.getRemoteAddr();  System.*out*.println(ip);  //获取map  Map<String,Integer> map = (Map<String, Integer>) fConfig.getServletContext().getAttribute("map");  //如果map中存在IP这个key,那么count+1,否则把IP放入map  **if**(map.containsKey(ip)){  //假如这个session是新创建的,说明是再次访问,不是已经访问,然后session还没过期  **if**(req.getSession().isNew()) {  **int** cnt = map.get(ip);  map.put(ip, cnt + 1);  }  }**else** {  //不存在,说明是第一次访问,此时将IP放入map,value是1  map.put(ip, 1);  }  chain.doFilter(req, response);  System.*out*.println("end");  }  @Override  **public** **void** init(FilterConfig fConfig) **throws** ServletException {  **this**.fConfig = fConfig;  }  } |



## 粗粒度权限控制（拦截是否登录、拦截用户名admin权限）

RBAC 🡪 基于角色的权限控制

* + tb\_user
  + tb\_role
  + tb\_userrole
  + tb\_menu(增、删、改、查)
  + tb\_rolemenu

### 1　说明

我们给出三个页面：welcome.jsp、user.jsp、admin.jsp。

* + welcome.jsp：谁都可以访问，没有限制；
  + user.jsp：只有登录用户才能访问；
  + admin.jsp：只有管理员才能访问。

### 2　分析

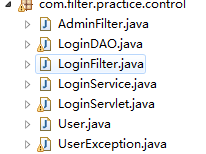
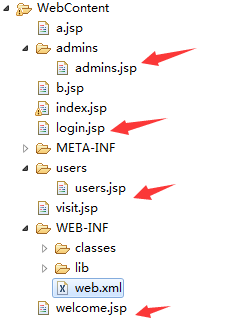
设计User类：username、password、grade，其中grade表示用户等级，1表示普通用户，2表示管理员用户。

当用户登录成功后，把user保存到session中。

创建LoginFilter，它有两种过滤方式：

* + 如果访问的是user.jsp，查看session中是否存在user；
  + 如果访问的是admin.jsp，查看session中是否存在user，并且user的grade等于2。

### 3　代码



User.java

|  |
| --- |
| **package** com.filter.practice.control;  **public** **class** User {  **private** String username;  **private** String password;  **private** String grade;  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String getGrade() {  **return** grade;  }  **public** **void** setGrade(String grade) {  **this**.grade = grade;  }  } |
| **package** com.filter.practice.control;  **public** **class** UserException **extends** Exception{  **public** UserException() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** UserException(String message) {  **super**(message);  // **TODO** Auto-generated constructor stub  }  } |
| package com.filter.practice.control;  public class LoginDAO {  QueryRunner qr = new TxQueryRunner();  public User findByUsername(User user){  try {  String sql = "select \* from user1 where username=?";  String username = user.getUsername();  User list = qr.query(sql,new BeanHandler<User>(User.class),username);  return list;  } catch (Exception e) {  throw new RuntimeException(e);  }  }  } |
| **package** com.filter.practice.control;  **public** **class** LoginService {  **private** LoginDAO dao = **new** LoginDAO();  **public** User findByUsername(User user) **throws** UserException{  User user1 = dao.findByUsername(user);  **if**(user1==**null**) **throw** **new** UserException("用户名不存在");  **return** user1;  }  } |
| package com.filter.practice.control;  /\*\*  \* Servlet implementation class LoginServlet  \*/  @WebServlet("/LoginServlet")  public class LoginServlet extends BaseServlet {  private LoginService service = new LoginService();  public String login(HttpServletRequest request,HttpServletResponse response){  User user = CommonUtils.toBean(request.getParameterMap(), User.class);  try {  //判断是否有这个用户  User user1 = service.findByUsername(user);  //执行到这里说明,有这个用户,那么把用户信息保存到session  request.getSession().setAttribute("user", user1);  //重定向  // request.getRequestDispatcher("/users/users.jsp").forward(request, response);  return "f:/users/users.jsp";  } catch (UserException e) {  //进行到这里,说明没有这个用户,把错误信息保存,在页面显示  request.setAttribute("msg", e.getMessage());  return "f:/login.jsp";  }    }  } |
| package com.filter.practice.control;  /\*\*  \* welcome.jsp也点击链接判断session中是否有用户名！如果用户没有登录，那么session中就没有用户名,此时提醒用户登录  \* @author Yorick  \*  \*/  public class LoginFilter implements Filter{  @Override  public void destroy() {  // TODO Auto-generated method stub    }  @Override  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  //处理servlet前,验证是否登录  HttpServletRequest req = (HttpServletRequest) request;  User user = (User) req.getSession().getAttribute("user");  if(user==null){  req.setAttribute("msg", "对不起,您尚未登录,请登录!");  req.getRequestDispatcher("/login.jsp").forward(req, response);//转发到login.jsp  }else {//如果username存在，说明用户已经登录了！  chain.doFilter(req, response);//放行！  }  }  @Override  public void init(FilterConfig arg0) throws ServletException {  // TODO Auto-generated method stub    }  } |
| package com.filter.practice.control;  public class AdminFilter implements Filter{  @Override  public void destroy() {  // TODO Auto-generated method stub    }  @Override  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  //需要需要这个过滤器,说明登录了,此时需要的是从session中获取用户grade,如果不是3,那么就认为没有权限  HttpServletRequest req = (HttpServletRequest) request;  User user = (User) req.getSession().getAttribute("user");  try {  if(user==null){  throw new UserException("您没有登录却到达这个页面,请致电我们,我们对系统加以升级改进!");  }  String grade = user.getGrade();  int g = Integer.parseInt(grade);  if(g!=3){  throw new UserException("对不起,您没有这个权限!");  }else {  chain.doFilter(req, response);  }  } catch (UserException e) {  req.getServletContext().setAttribute("msg", e.getMessage());  // req.getRequestDispatcher("/users/users.jsp").forward(req, response);  HttpServletResponse resp = (HttpServletResponse) response;  resp.sendRedirect("/filter/users/users.jsp");  }  }  @Override  public void init(FilterConfig arg0) throws ServletException {  // TODO Auto-generated method stub    }  } |

login.jsp

|  |
| --- |
| <h1>登录页面</h1>  <p>${msg }</p>  <form action=*"*<c:url value=*'/LoginServlet'*/>*"* method=*"post"*>  <input type=*"hidden"* name=*"method"* value=*"login"*>  用户名：<input type=*"text"* name=*"username"*/>  <input type=*"submit"* value=*"登录"*/>  </form> |

welcome.jsp

|  |
| --- |
| <h1>游客可以访问！</h1>  <h3>你只是个游客！</h3>  ${msg }  <a href=*"*<c:url value=*'/users/users.jsp'*/>*"*>会员专区</a>  <a href=*"*<c:url value=*'/admins/admins.jsp'*/>*"*>管理员专区</a>  <a href=*"*<c:url value=*'/login.jsp'*/>*"*>请登录</a>  </body> |

/user/user.jsp

|  |
| --- |
| <h1>会员可以访问</h1>${msg }  <h3>您好，${sessionScope.user.getUsername() }会员先生</h3>  <a href=*"*<c:url value=*'/welcome.jsp'*/>*"*>游客区</a>  <a href=*"*<c:url value=*'/admins/admins.jsp'*/>*"*>管理员专区</a> |

/admin/admin.jsp

|  |
| --- |
| <h1>管理员才能访问</h1>  <h3>尊敬的管理员先生，欢迎${sessionScope.user.getUsername() }大架</h3>  <a href=*"*<c:url value=*'/welcome.jsp'*/>*"*>游客区</a>  <a href=*"*<c:url value=*'/users/users.jsp'*/>*"*>会员专区</a> |
| <filter>  <filter-name>LoginFilter</filter-name>  <filter-class>com.filter.practice.control.LoginFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>LoginFilter</filter-name>  <url-pattern>/users/\*</url-pattern>  <dispatcher>REQUEST</dispatcher>  <dispatcher>FORWARD</dispatcher>  <dispatcher>INCLUDE</dispatcher>  </filter-mapping>  <filter>  <filter-name>AdminFilter</filter-name>  <filter-class>com.filter.practice.control.AdminFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>AdminFilter</filter-name>  <url-pattern>/admins/\*</url-pattern>  <dispatcher>REQUEST</dispatcher>  <dispatcher>FORWARD</dispatcher>  <dispatcher>INCLUDE</dispatcher>  </filter-mapping> |

## 禁用资源缓存

浏览器只是要缓存页面，这对我们在开发时测试很不方便，所以我们可以过滤所有资源，然后添加去除所有缓存！

|  |
| --- |
| **public** **class** NoCacheFilter **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request,  HttpServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  response.setHeader("cache-control", "no-cache");  response.setHeader("pragma", "no-cache");  response.setHeader("expires", "0");  chain.doFilter(request, response);  }  } |

但是要注意，有的浏览器可能不会理会你的设置，还是会缓存的！这时就要在页面中使用时间戳来处理了。

## 解决全站字符乱码（POST和GET中文编码问题）

### 1　说明

乱码问题：

* + 获取请求参数中的乱码问题；
  + POST请求：request.setCharacterEncoding(“utf-8”)；
  + GET请求：new String(request.getParameter(“xxx”).getBytes(“iso-8859-1”), “utf-8”)；
  + 响应的乱码问题：response.setContextType(“text/html;charset=utf-8”)。

　　基本上在每个Servlet中都要处理乱码问题，所以应该把这个工作放到过滤器中来完成。

### 2　分析

其实全站乱码问题的难点就是处理GET请求参数的问题。

如果只是处理POST请求的编码问题，以及响应编码问题，那么这个过滤器就太！太！太简单的。

|  |
| --- |
| **public** **class** EncodingFilter **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request,  HttpServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  //获取配置文件中的初始化参数：charset  String charset = **this**.getInitParameter("charset");  //如果没有给Filter配置charset参数，那么设置编码为UTF-8  **if**(charset == **null** || charset.isEmpty()) {  charset = "UTF-8";  }  //处理POST请求编码  request.setCharacterEncoding(charset);  //处理响应编码  response.setContentType("text/html;charset=" + charset);  chain.doFilter(request, response);  }  } |

如果是POST请求，当执行目标Servlet时，Servlet中调用request.getParameter()方法时，就会根据request.setCharacterEncoding()设置的编码来转码！这说明在过滤器中调用request.setCharacterEncoding()方法会影响在目标Servlet中的request.getParameter()方法的行为！

但是如果是GET请求，我们又如何能影响request.getParameter()方法的行为呢？这是不好做到的！我们不可能先调用request.getParameter()方法获取参数，然后手动转码后，再施加在到request中！因为request只有getParameter()，而没有setParameter()方法。

处理GET请求参数编码问题，需要在Filter中放行时，把request对象给“调包”了，也就是让目标Servlet使用我们“调包”之后的request对象。这说明我们需要保证“调包”之后的request对象中所有方法都要与“调包”之前一样可以使用，并且getParameter()方法还要有能力返回转码之后的参数。

这可能让你想起了“继承”，但是这里不能用继承，而是“装饰者模式（Decorator Pattern）”！

下面是三种对a对象进行增强的手段：

* 继承：AA类继承a对象的类型：A类，然后重写fun1()方法，其中重写的fun1()方法就是被增强的方法。但是，继承必须要知道a对象的真实类型，然后才能去继承。如果我们不知道a对象的确切类型，而只知道a对象是IA接口的实现类对象，那么就无法使用继承来增强a对象了；
* 装饰者模式：AA类去实现a对象相同的接口：IA接口，还需要给AA类传递a对象，然后在AA类中所有的方法实现都是通过代理a对象的相同方法完成的，只有fun1()方法在代理a对象相同方法的前后添加了一些内容，这就是对fun1()方法进行了增强；
* 动态代理：动态代理与装饰者模式比较相似，而且是通过反射来完成的。动态代理会在后面的基础加强中讲解，这里就不再废话了。

对request对象进行增强的条件，刚好符合装饰者模式的特点！因为我们不知道request对象的具体类型，但我们知道request是HttpServletRequest接口的实现类。这说明我们写一个类EncodingRequest，去实现HttpServletRequest接口，然后再把原来的request传递给EncodingRequest类！在EncodingRequest中对HttpServletRequest接口中的所有方法的实现都是通过代理原来的request对象来完成的，只有对getParameter()方法添加了增强代码！

JavaEE已经给我们提供了一个HttpServletRequestWrapper类，它就是HttpServletRequest的包装类，但它做任何的增强！你可能会说，写一个装饰类，但不做增强，其目的是什么呢？使用这个装饰类的对象，和使用原有的request有什么分别呢？

HttpServletRequestWrapper类虽然是HttpServletRequest的装饰类，但它不是用来直接使用的，而是用来让我们去继承的！当我们想写一个装饰类时，还要对所有不需要增强的方法做一次实现是很心烦的事情，但如果你去继承HttpServletRequestWrapper类，那么就只需要重写需要增强的方法即可了。

### 3　代码

EncodingRequest

|  |
| --- |
| **public** **class** EncodingRequest **extends** HttpServletRequestWrapper {  **private** String charset;  **public** EncodingRequest(HttpServletRequest request, String charset) {  **super**(request);  **this**.charset = charset;  }  **public** String getParameter(String name) {  HttpServletRequest request = (HttpServletRequest) getRequest();    String method = request.getMethod();  **if**(method.equalsIgnoreCase("post")) {  **try** {  request.setCharacterEncoding(charset);  } **catch** (UnsupportedEncodingException e) {}  } **else** **if**(method.equalsIgnoreCase("get")) {  String value = request.getParameter(name);  **try** {  value = **new** String(name.getBytes("ISO-8859-1"), charset);  } **catch** (UnsupportedEncodingException e) {  }  **return** value;  }  **return** request.getParameter(name);  }  } |

EncodingFilter

|  |
| --- |
| **public** **class** EncodingFilter **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request,  HttpServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  String charset = **this**.getInitParameter("charset");  **if**(charset == **null** || charset.isEmpty()) {  charset = "UTF-8";  }  response.setCharacterEncoding(charset);  response.setContentType("text/html;charset=" + charset);  EncodingRequest res = **new** EncodingRequest(request, charset);  chain.doFilter(res, response);  }  } |

web.xml

|  |
| --- |
| <filter>  <filter-name>EncodingFilter</filter-name>  <filter-class>cn.itcast.filter.EncodingFilter</filter-class>  <init-param>  <param-name>charset</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>EncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

## 页面静态化

### 1　说明

你到“当当”搜索最多的是什么分类，没错，就是Java分类！你猜猜，你去搜索Java分类时，“当当”会不会去查询数据库呢？当然会了，不查询数据库怎么获取Java分类下的图书呢！其实每天都有很多人去搜索“Java分类”的图书，每次都去访问数据库，这会有性能上的缺失！如果是在访问静态页面（html）那么就会快的多了！静态页面本身就比动态页面快很多倍，而且动态页面总是要去数据库查询，这会更加降低速度！

页面静态化是把动态页面生成的html保存到服务器的文件上，然后再有相同请求时，不再去执行动态页面，而是直接给用户响应上次已经生成的静态页面。而且静态页面还有助与搜索引擎找到你！

### 2　查看图书分类

我们先来写一个小例子，用来查看不同分类的图书。然后我们再去思考如何让动态页面静态化的问题。

index.jsp

|  |
| --- |
| <body>  <a href=*"*<c:url value=*'/BookServlet'*/>*"*>全部图书</a><br/>  <a href=*"*<c:url value=*'/BookServlet?category=1'*/>*"*>JavaSE分类</a><br/>  <a href=*"*<c:url value=*'/BookServlet?category=2'*/>*"*>JavaEE分类</a><br/>  <a href=*"*<c:url value=*'/BookServlet?category=3'*/>*"*>Java框架分类</a><br/>  </body> |
|  |

BookServlet.java

|  |
| --- |
| **public** **class** BookServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //创建BookService  BookService bookService = **new** BookService();  List<Book> bookList = **null**;  //获取链接参数  String param = request.getParameter("category");  **if**(param == **null** || param.isEmpty()) {  bookList = bookService.findAll();  } **else** {  **int** category = Integer.*parseInt*(param);  bookList = bookService.findByCategory(category);  }    request.setAttribute("bookList", bookList);  request.getRequestDispatcher("/show.jsp").forward(request, response);  }  } |

show.jsp

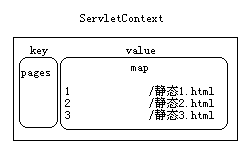
|  |
| --- |
| <table border=*"1"* align=*"center"* width=*"50%"*>  <tr>  <th>图书名称</th>  <th>图书单价</th>  <th>图书分类</th>  </tr>    <c:forEach items="${bookList }" var=*"book"*>  <tr>  <td>${book.bname }</td>  <td>${book.price }</td>  <td>  <c:choose>  <c:when test="${book.category **eq** 1}"><p style="color:*red*;">JavaSE分类</p></c:when>  <c:when test="${book.category **eq** 2}"><p style="color:*blue*;">JavaEE分类</p></c:when>  <c:when test="${book.category **eq** 3}"><p style="color:*green*;">Java框架分类</p></c:when>  </c:choose>  </td>  </tr>  </c:forEach>  </table> |
|  |

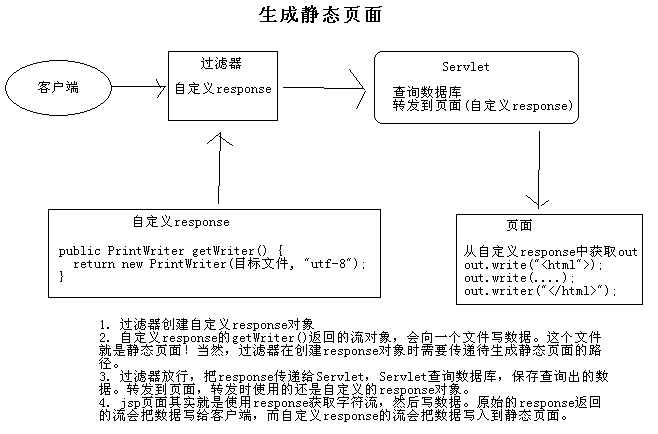
### 3　分析

　　我们的目标是在用户第一次访问页面时生成静态页面，然后让请求重定向到静态页面上去。当用户再次访问时，直接重定向到静态页面上去。

我们需要为不同的请求生成静态页面，例如用户访问BookServlet?category=1时，我们要生成静态页面，当用户访问BookServlet?category=2时，也要生成静态页面。即不同的参数生成不同的静态页面！

我们可以使用category为key，静态页面的路径为value，保存到一个Map中，然后再把Map保存到ServletContext中。没有对应的静态页面时，我们生成静态页面，再重定向到静态页面，如果存在静态页面，那么直接重定向即可。





StaticResponse.java

|  |
| --- |
| **public** **class** StaticResponse **extends** HttpServletResponseWrapper {  **private** PrintWriter pw;  **public** StaticResponse(HttpServletResponse response, String filepath)  **throws** FileNotFoundException, UnsupportedEncodingException {  **super**(response);  pw = **new** PrintWriter(filepath, "UTF-8");  }  **public** PrintWriter getWriter() **throws** IOException {  **return** pw;  }  **public** **void** close() **throws** IOException {  pw.close();  }  } |

StaticFilter.java

|  |
| --- |
| **public** **class** StaticFilter **implements** Filter {  **private** ServletContext sc;    **public** **void** destroy() {  }  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  HttpServletResponse res = (HttpServletResponse) response;  String key = "key\_" + request.getParameter("category");    Map<String,String> map = (Map<String, String>) sc.getAttribute("pages");  **if**(map == **null**) {  map = **new** HashMap<String,String>();  sc.setAttribute("pages", map);  }    **if**(map.containsKey(key)) {  res.sendRedirect(req.getContextPath() + "/staticPages/" + map.get(key));  **return**;  }  String html = key + ".html";  String realPath = sc.getRealPath("/staticPages/" + html);  StaticResponse sr = **new** StaticResponse(res, realPath);  chain.doFilter(request, sr);  sr.close();  res.sendRedirect(req.getContextPath() + "/staticPages/" + html);  map.put(key, html);  }  **public** **void** init(FilterConfig fConfig) **throws** ServletException {  **this**.sc = fConfig.getServletContext();  }  } |