# HTTP协议和tomcat服务器

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【理解】能够阐述什么是过滤器**
2. **【应用】能够独立编写过滤器代码**
3. **【应用】了解多个过滤器的执行顺序**
4. **【应用】了解过滤器的生命周期**
5. **【理解】了解过滤器的应用场景**

# 过滤器介绍

## 什么是过滤器

生活中的例子：

滤网，筛子，渔网

生活中的过滤器: 排除我们不需要的东西，留下，需要的。

高考：只有分数够高的同学才能进入理想的大学。有一部分同学被拦截在大学之外。（起到拦截的作用）

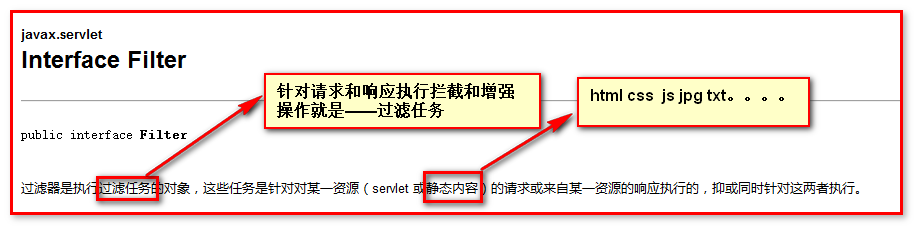
极客营： 一开始大家都是小白，进入极客营学习，经历了5个月的学习，毕业之后，具有了一定的编码能力。

（对每一个经过的学员，都增强了学员的编码能力，起到了增强的作用）

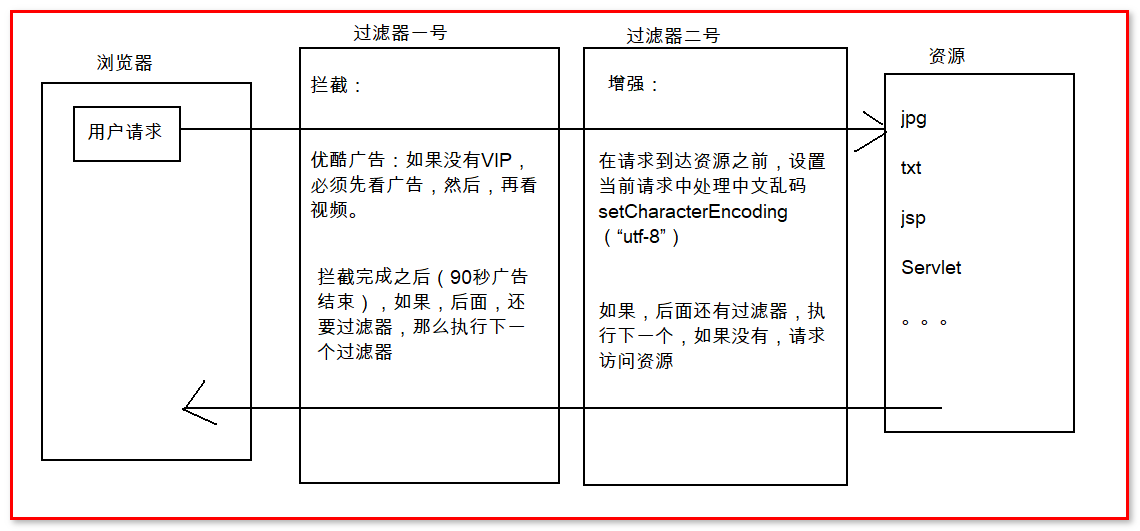
JavaWeb中的过滤器的概念： 对请求和响应进行拦截或者增强的对象，就是过滤器。（重点）

JavaWeb中的过滤器是什么呢？

Filter接口：功能——对请求和响应进行增强，或者进行拦截。



## JavaWEB中的过滤器运行图解（重点）



# Filter的快速入门（重点：必须掌握）

## Filter定义以及创建步骤介绍

定义一个过滤器：

\* 第一步：创建一个类，实现过滤器接口

\* 第二步：具体需要执行的过滤任务，写在doFilter

\* 第三步：过滤器需要在web.xml中配置

代码演示：

**package** cn.igeek.filter;

**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;

/\*\*

\*

\* Filter 是在 Web 应用程序的部署描述符中配置的

\*

\* 部署描述符:web.xml

\*

\* 定义一个过滤器：

\* 第一步：创建一个类，实现过滤器接口

\* 第二步：具体需要执行的过滤任务，写在doFilter

\* 第三步：过滤器需要在web.xml中配置

\* \*/

**public** **class** MyFilter **implements** Filter{

/\*\*

\* 初始化方法

\* \*/

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

// **TODO** Auto-generated method stub

}

/\*\*

\* 执行过滤任务的方法

\* \*/

@Override

**public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

System.*out*.println("MyFilter....doFilter.....");

}

/\*\*

\* 销毁的方法

\* \*/

@Override

**public** **void** destroy() {

// **TODO** Auto-generated method stub

}

}

Filter 是在 Web 应用程序的部署描述符中配置的——过滤器创建好之后，需要在web.xml中做配置

### 在web.xml文件中配置过滤器

<filter>

<filter-name>MyFilter</filter-name>

<filter-class>cn.igeek.filter.MyFilter</filter-class>

</filter>

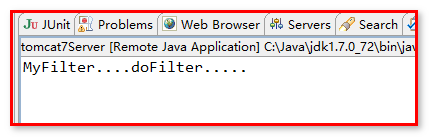
<filter-mapping>

<filter-name>MyFilter</filter-name>

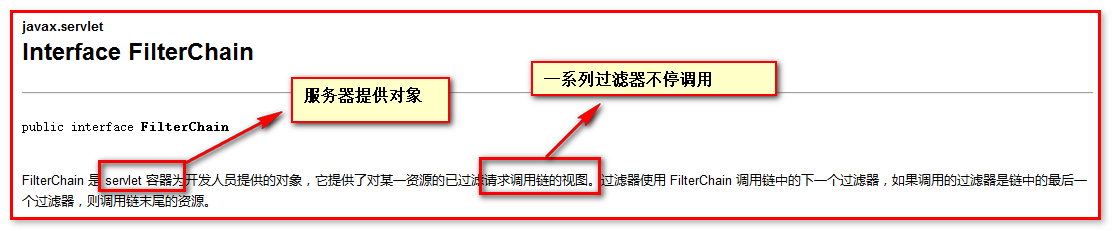
<url-pattern>/1.txt</url-pattern>

</filter-mapping>

## Filter拦截操作效果



## 过滤器放行的对象:FilterChain功能介绍

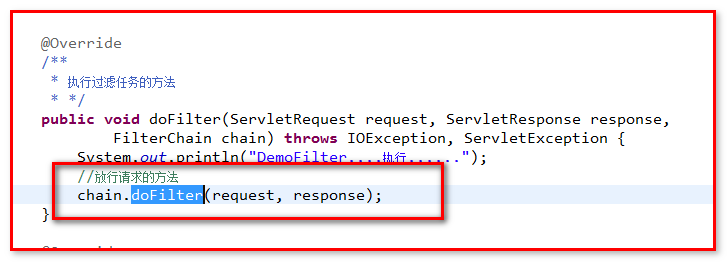


总结：过滤放行，全靠它

**FilterChain的doFilter方法**：

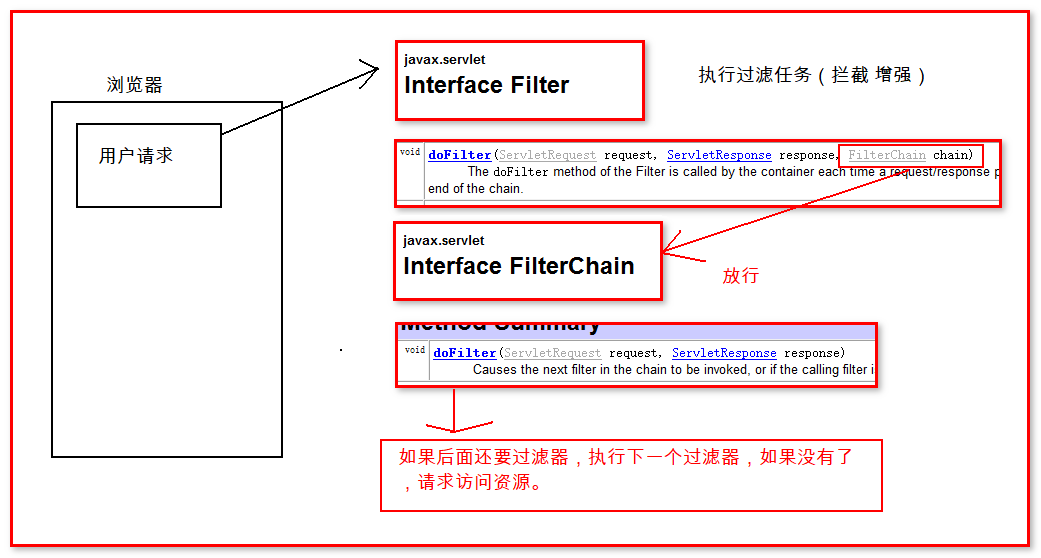


代码实现



效果：

过滤器放行执行过程：



# Filter生命周期

## Filter生命周期

回顾servlet的生命周期：

创建： 第一次被访问的时候

销毁： 服务器关闭的时候，或者当前项目从服务器中移除

回顾session的生命周期：

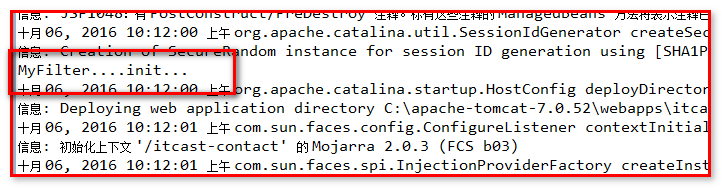
创建： 第一次调用getsession方法，第一次访问jsp

销毁： 服务器非正常关闭，超过生存时间，调用销毁（自杀）的方法

Filter：

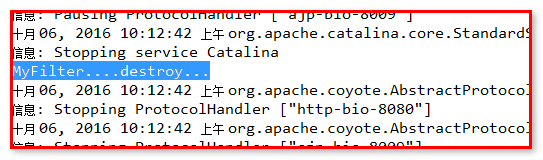
创建：在服务器启动的时候

服务器启动截图：



销毁： 在服务器关闭的时候，过滤器销毁。

服务器关闭截图：



# FilterConfig介绍

servletConfig对象：获取servlet相关的配置信息。

FilterConfig定义：获取filter相关的配置信息。



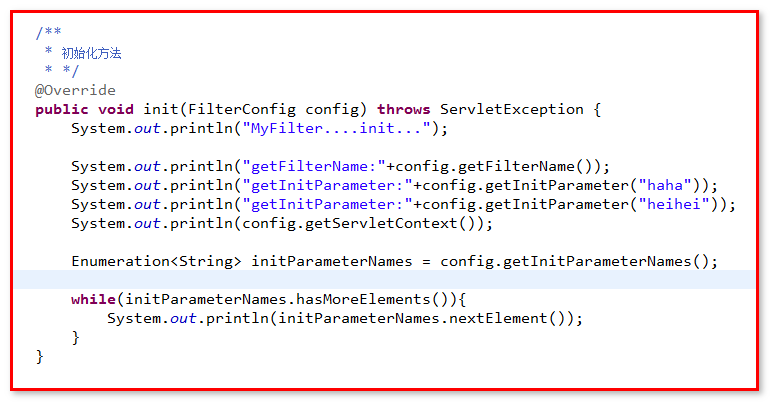
API介绍：



API代码演示：

1）设置过滤器初始化参数

2）通过filterconfig对象来获取参数



参数配置：

<filter>

<filter-name>MyFilter</filter-name>

<filter-class>cn.igeek.filter.MyFilter</filter-class>

<init-param>

<param-name>haha</param-name>

<param-value>哈哈</param-value>

</init-param>

<init-param>

<param-name>heihei</param-name>

<param-value>嘿嘿</param-value>

</init-param>

</filter>

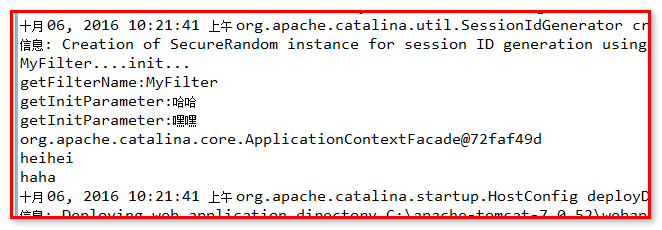
<filter-mapping>

<filter-name>MyFilter</filter-name>

<url-pattern>/1.txt</url-pattern>

</filter-mapping>

效果演示：



# Filter配置详解（web.xml中的配置）（重点）

## 关于url-pattern配置

**过滤器如何匹配请求的路径?**

回顾servlet的url-pattern：

全路径匹配——

地址栏：localhost:8080/项目根路径/资源路径 localhost:8080/igeek-filter2/1.txt

通配符的匹配——

地址栏：localhost:8080/项目根路径/abc/\*

以上两种匹配方式，配置路径的时候必须以”/”开头



后缀名匹配—— \*.do: \*.do \*.txt \*.action

地址栏：localhost:8080/项目根路径/\*.txt

后缀名匹配方式，配置路径的时候不能以”/”开头

Filter的url-pattern配置与servlet一致。

**过滤器的执行顺序？**

测试方式：

1. 两个过滤器，拦截同一个请求
2. 调整两个过滤器的配置，再来看执行的顺序

urlParam 优先>ServletName 配置路径相同的

字母在前的优先

总结：

过滤器执行的顺序是按照，web.xml中filter-mapping标签的书写顺序执行（从上往下执行）

## 关于servlet-name配置

什么是servlet-name配置？

定义：针对指定的servlet进行拦截或者增强操作的配置

Servlet：

**package** cn.igeek.servlet;

**import** java.io.IOException;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** DemoServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

System.*out*.println("DemoServlet....doGet.....");

}

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

doGet(request, response);

}

}

Filter：

**package** cn.igeek.filter;

**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** DemoFilter **implements** Filter{

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

}

@Override

**public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

System.*out*.println("DemoFilter.....doFilter....");

chain.doFilter(request, response);

}

@Override

**public** **void** destroy() {

}

}

Web.xml配置：

<servlet>

<servlet-name>DemoServlet</servlet-name>

<servlet-class>cn.igeek.servlet.DemoServlet</servlet-class>

</servlet>

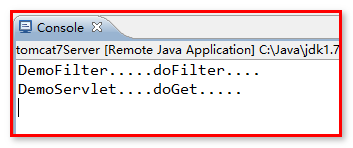
<servlet-mapping>

<servlet-name>DemoServlet</servlet-name>

<url-pattern>/demo</url-pattern>

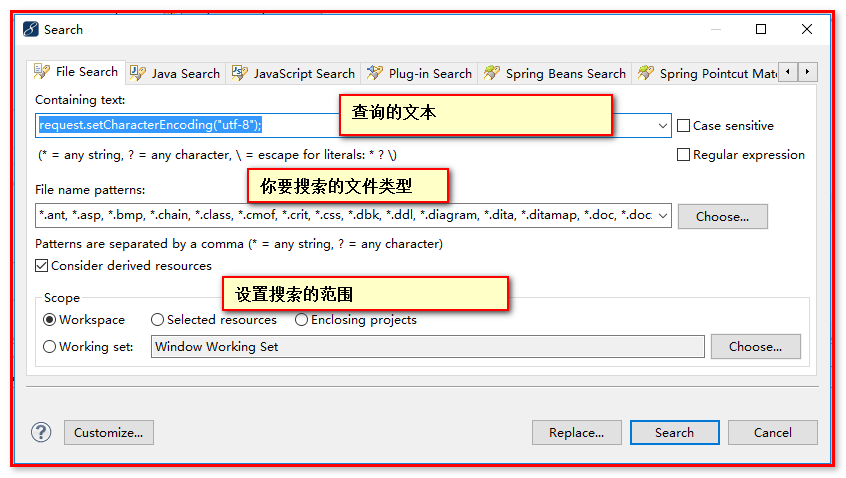
</servlet-mapping>

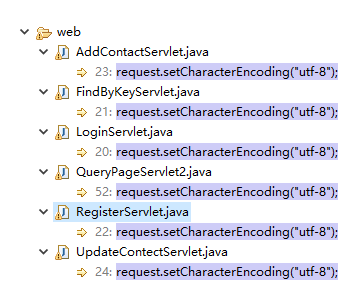
效果：



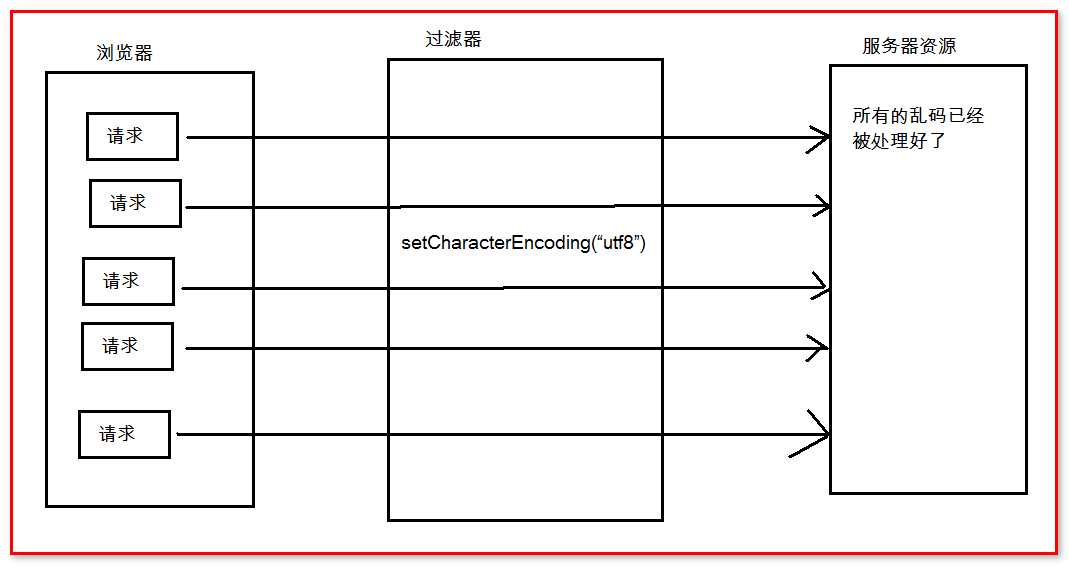
# 案例2--解决shop项目中乱码

需求：请求参数在每一个servlet中单独中文乱码处理，代码重复





优化的思路，使用一个过滤器，在请求到达servlet之前，先对request对象进行设置编码



要对所有的请求都要进行设置编码，都要拦截，进行增强，url-pattern:/\*

过滤器代码：

**package** cn.igeek.filter;

**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.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** EncodingFilter **implements** Filter{

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

}

@Override

**public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

HttpServletRequest req = (HttpServletRequest)request;

HttpServletResponse res = (HttpServletResponse)response;

//处理乱码，处理post请求乱码

req.setCharacterEncoding("utf-8");

chain.doFilter(req, res);

}

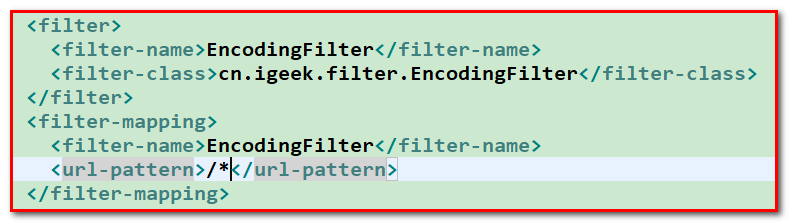
@Override

**public** **void** destroy() {

}

}

Web.xml配置：



实现完整版的乱码处理，过滤器：（了解，不做要求）

第一步：还以要有一个过滤器，对请求进行乱码处理的任务（增强任务），写在doFilter方法中

补充（装饰（包装）设计模式心法）：

1. 定义一个类，实现被装饰对象的接口
2. 定义一个成员变量，记住被装饰对象的引用
3. 定义构造方法，传入被装饰对象的实例
4. 改写要修改的方法
5. 不需要改写的方法，调用被装饰对象的原来的方法

复杂过滤器实现：

|  |
| --- |
| **package** cn.igeek.filter;  **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.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** cn.igeek.domain.MyRequest;  **public** **class** EncodingFilter **implements** Filter{  @Override  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  // 对请求和响应，处理中文乱码  HttpServletRequest req = (HttpServletRequest)request;  HttpServletResponse res = (HttpServletResponse)response;    res.setContentType("text/html;charset=utf-8");      //针对不同的请求方式，处理乱码，使用一个新的request对象，在新的request对象中，处理乱码  //设置构造函数有req对象，是为了保证，获取到请求中的数据，然后处理乱码  MyRequest myRequest = **new** MyRequest(req);  //测试获取请求参数  System.***out***.println("name1:"+myRequest.getParameter("name"));  System.***out***.println("name2:"+myRequest.getParameter("name"));    //处理完成请求和响应乱码，只需要放行  chain.doFilter(myRequest, res);    }  @Override  **public** **void** destroy() {  }  } |

自定义增强类：

|  |
| --- |
| **package** cn.igeek.domain;  **import** java.io.UnsupportedEncodingException;  **import** java.util.Map;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletRequestWrapper;  /\*\*  \* **@author** 王嘉男  \*  \* **@time** 2017年5月26日  \*  \* MyRequest如何实现：  \*  \* 1 实现httpservletrequest接口——通过继承HttpServletRequestWrapper  \* 2 设置一个变量，用来保存服务器创建的request对象  \* 3 处理乱码，就是改写获取请求参数的三个方法  \* 4 不需要改写的方法，使用原来的对象方法  \*/  **public** **class** MyRequest **extends** HttpServletRequestWrapper{  //为了后期可以使用服务器创建的request对象中的请求参数  //设置一个变量，用来保存服务器创建的request对象  **private** HttpServletRequest request ;  //定义一个标记，控制编码的次数，只执行一次  **private** **boolean** flag = **false**;  **public** MyRequest(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }    @Override  **public** Map<String, String[]> getParameterMap() {    //获取请求方式，根据不同方式处理乱码  String method = request.getMethod();  **if**(method.equalsIgnoreCase("post")){  **try** {  request.setCharacterEncoding("utf-8");  **return** request.getParameterMap();  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  **return** **super**.getParameterMap();  }  }**else** **if**(method.equalsIgnoreCase("get")){    //将所有的请求参数获取出来，然后，一个一个地处理乱码  Map<String, String[]> map = request.getParameterMap();  //如果控制编码，让他只执行一次？  //flag 默认是false，所以往下执行  //flag 第二次执行，true ，直接结束，不在执行。  **if**(flag){  **return** map;  }  //变量map集合获取数据  **if**(map != **null**){  **for**(String key :map.keySet()){  String[] values = map.get(key);  **if**(values != **null**){  **for** (**int** i = 0; i < values.length; i++) {  **try** {  String string = **new** String(values[i].getBytes("iso-8859-1"),"utf-8");  values[i] = string;  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  //如果出异常，希望后边，还没有循环到的数据，继续处理乱码  //结束当前循环，开启下一个循环  **continue**;  }  }  }  }  }  flag = **true**;  **return** map;  }**else**{  **return** **super**.getParameterMap();  }  }    @Override  **public** String[] getParameterValues(String name) {  Map<String, String[]> map = **this**.getParameterMap();  **if**(map != **null**){  String[] values = map.get(name);  **return** values;  }  **return** **super**.getParameterValues(name);  }    @Override  **public** String getParameter(String name) {  String[] values = **this**.getParameterValues(name);  **if**(values != **null**){  **return** values[0];  }  **return** **super**.getParameter(name);  }    } |

注意：最后还有去掉原来设置编码的代码。