# 17.Filter

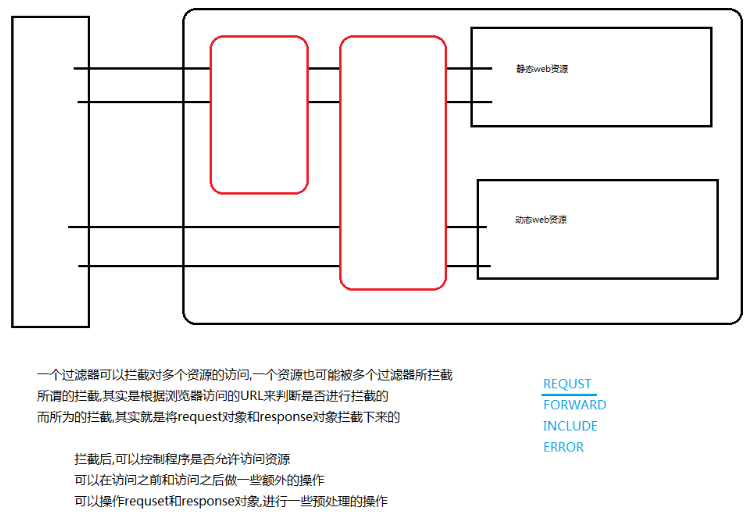
## 17.1.过滤器概念

**过滤器的基本概念**

Filter也称之为过滤器，它是Servlet技术中最实用的技术，WEB开发人员通过Filter技术，对web服务器管理的所有web资源：例如Jsp, Servlet, 静态图片文件或静态 html 文件等进行拦截，从而实现一些特殊的功能。例如实现URL级别的权限访问控制、过滤敏感词汇、压缩响应信息等一些高级功能。

是javaEE三大组件之一：Servlet、Filter、Listener

**过滤器的功能**



(1)过滤器可以拦截对资源的访问

(2)一个过滤器可以拦截多个资源，一个资源也可能被多个过滤器拦截

(3)所谓的拦截是根据访问的URL地址来确定访问的是哪个资源，确定是否拦截

(4)所谓的拦截其实就是拦截下来代表请求的request和响应的response

(5)拦截后：控制是否允许访问、访问之前和之外做一些额外操作

这种多个过滤器拦截一个资源的模式 称之为 责任链模式

## 17.2.过滤器开发入门

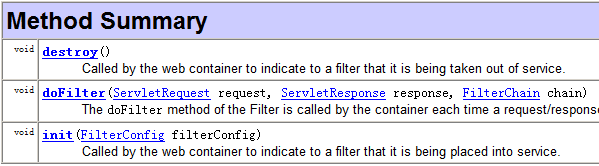
**开发过滤器的步骤**

想要开发一个过滤器，需要如下两个步骤：

(1)写一个类实现Filter接口

(2)在web.xml中配置过滤器

**写一个类实现javax.servlet.Filter接口**



init为初始化方法，在Filter对象被创建出来时，Servlet容器会调用该方法对filter进行初始化。

destory为销毁的方法，在过滤器对象被销毁之前，服务器会调用这个方法执行善后工作。

doFilter为过滤器中最核心的方法，当过滤器拦截到对资源的访问时，服务器会自动调用该方法执行过滤代码。 我们只需要在这个方法中设计过滤器的逻辑代码即可。

**在web.xml中配置一下过滤器**

<filter> -- 配置一个过滤器

<filter-name>FirstFilter</filter-name> -- 过滤器的名字

<filter-class>cn.tedu.filter.FirstFilter</filter-class> --过滤器类的全路径名

</filter>

<filter-mapping> -- 过滤器的拦截路径配置,可以配置多个

<filter-name>FirstFilter</filter-name> -- 为哪个名字的过滤器配置

<url-pattern>/\*</url-pattern> -- 拦截哪个路径资源可以配置多个

<servlet-name>XxxServlet</servlet-name> -- 拦截哪个名字的Servlet

<dispatcher></dispatcher> -- 指定过滤器拦截哪种方式对资源的访问，可 以取值为REQUEST FORWARD INCLUDE ERROR，如果不配置，默认只 拦截 REQUEST方式的访问。可以配置多个。

</filter-mapping>

**代码示例：**

|  |
| --- |
| 创建项目day16，可以先准备一个servlet，例如创建包cn.tedu.web，创建servlet：Servlet Demo1，并添加如下代码：  System.*out*.println("ServletDemo1......");  response.getWriter().write("hello....filter...."); |
| src下创建包cn.tedu.filter，包下创建class：FilterDemo01，实现Filter接口（注意，导包需要导javax.servlet.Filter包），并实现方法（也可以配置多个Filter）。代码如下：  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  System.*out*.println("init........");  }  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("doFilter...before....");  chain.doFilter(request, response);  System.*out*.println("doFilter...after...");  }  **public** **void** destroy() {  System.*out*.println("destroy.......");  } |
| 在web.xml中配置如下信息：  <filter>  <filter-name>FilterDemo01</filter-name>  <filter-class>cn.tedu.filter.FilterDemo01</filter-class>  </filter>  <filter-mapping>  <filter-name>FilterDemo01</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

## 17.3.Filter开发相关

**Filter生命周期**

在web应用启动时，会创建出web应用中配置的过滤器对象，创建出过滤器对象后立即调用init方法进行初始化的操作，之后一直存活，直到web应用被销毁时，Filter跟着被销毁，在销毁之前调用destory方法执行善后工作。在存活期间，每当拦截到资源，就执行dofilter方法 来执行过滤器的逻辑，如果不做操作 则默认拦截，可以通过调用filterchain的dofilter方法来放行对资源的访问。并且可以在dofilter方法之前或之后做一些额外的操作。

**拦截顺序**

如果一个资源被多个过滤器所拦截，多个过滤器的拦截顺序，取决与web.xml中filter-mapping 配置的顺序。

多个过滤器的执行，类似于方法一层一层调用的过程，一层一层往里钻，再一层一层往外出。

**和filter开发相关的对象**

**FilterConfig：**

init方法的参数。

代表filter在web.xml中的配置对象。

可以用来获取filter的初始化参数

可以用来获取ServletContext对象

**FilterChain：**

dofilter方法的参数

代表过滤器链

提供了doFilter方法，放行当前过滤器，执行后续过滤器，如果后续没有过滤器，则调用到最终访问的资源

示例：通过FilterConfig获取web.xml中的参数

|  |
| --- |
| 在web.xml中，在FilterDemo01标签中配置如下属性：  <filter>  <filter-name>FilterDemo01</filter-name>  <filter-class>cn.tedu.filter.FilterDemo01</filter-class>  <init-param>  <param-name>param1</param-name>  <param-value>value1</param-value>  </init-param>  <init-param>  <param-name>param2</param-name>  <param-value>value2</param-value>  </init-param>  </filter> |
| 在FilterDemo01中的init方法中添加如下代码：  String v1 = filterConfig.getInitParameter("param1");  String v2 = filterConfig.getInitParameter("param2");  System.*out*.println("param1:"+v1);  System.*out*.println("param2:"+v2); |

## 17.4.Easymall全站乱码解决过滤器

在web开发的过程中，存在请求参数乱码，和响应输出乱码。

之前的开发中，在所有的Servlet和jsp页面中，需要手动的解决这两种乱码。

可以通过开发过滤器，拦截所有的资源访问，在过滤器中解决全站乱码问题，这样只需要一个过滤器，就可以解决全站乱码，而不需要在每个资源中都单独去解决乱码了。

**响应乱码：**

response.setContentType("text/html;charset="+encode);

**请求乱码：**

post请求：

request.setCharacterEncoding(encode);

get请求：

方案1：获取请求中的参数，解决完乱码，再设置回去。

不可能为每个servlet都获取参数解决乱码 -- getParameterMap

就算都获取出来解决了乱码也没有办法将解决好乱码的请求参 数设置回去 -- 无解

方案2：request中的请求参数本身无法改变，那么，换一个思路， 想办法改造和获取请求参数相关的方法，在方法内加上解决乱码 的代码，这样通过这些方法获取请求参数时，解决好乱码再返回， 用起来就感觉，乱码被解决了一样。

改造一个已有对象身上的方法？？？？

**继承**

只能先继承再创建对象，如果用原来的类创建对象，对 这些对象没有影响。

**装饰**

可以改造已有对象身上的方法，但是当原来对象身上的 方法比较多的时候，开发起来比较麻烦。

**动态代理**

可以改造已有对象身上的方法，并且只需要改造要改 造的方法，其他方法不需要额外编写，非常方便。

代码示例：

|  |
| --- |
| 继承方式演示：  **class** Person{  **public** **void** eat(){  System.*out*.println("吃......");  }  **public** **void** study(){  System.*out*.println("学......");  }  }  //继承  **class** Person2 **extends** Person{  @Override  **public** **void** eat() {  System.*out*.println("洗手......");  **super**.eat();  System.*out*.println("上厕所......");  }  }  /\*\*  \* 测试  \*/  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  //只能重新new，不能用改造之前的person对象  Person p = **new** Person2();  p.eat();  p.study();  }  } |
| 装饰方式演示：  **interface** PersonIntf{  **public** **void** eat();  **public** **void** study();  }  **class** Person **implements** PersonIntf{  **public** **void** eat(){  System.*out*.println("吃......");  }  **public** **void** study(){  System.*out*.println("学......");  }  }  //装饰  **class** Person3 **implements** PersonIntf{  **private** Person p = **null**;  **public** Person3(Person p){  **this**.p = p;  }  **public** **void** eat() {  System.*out*.println("洗手......");  p.eat();  System.*out*.println("擦嘴......");  }  **public** **void** study() {  p.study();  }  }  /\*\*  \* 测试  \*/  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  //可以将已经创建好的对象传入  Person3 p3 = **new** Person3(p);  p3.eat();  p3.study();  }  } |
| 动态代理演示：  **interface** PersonIntf{  **public** **void** eat();  **public** **void** study();  }  **class** Person **implements** PersonIntf{  **public** **void** eat(){  System.*out*.println("吃......");  }  **public** **void** study(){  System.*out*.println("学......");  }  }  /\*\*  \* 测试  \*/  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  **final** Person p = **new** Person();  //动态代理  /\*\*  \* Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)  \* 此方法可以创建一个代理对象，返回值为object类型，需要强转（注意，强转为接口），有三个参数：  \* ClassLoader：类加载器，可以将被代理对象的类加载器传入  \* Class<?>[] interfaces：被代理对象上的所有接口，可以通过：对象.class.getClassLoader()获取  \* InvocationHandler：处理程序的接口,需要实现其中的invoke方法  \*/  PersonIntf p4 = (PersonIntf)Proxy.*newProxyInstance*(Person.**class**.getClassLoader(), Person.**class**.getInterfaces(), **new** InvocationHandler() {  /\*\*  \* invoke:代理对象执行的方法，参数有：  \* proxy：代理对象  \* method：执行的方法  \* args：参数，没有则为null  \*/  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  **if**("eat".equals(method.getName())){  System.*out*.println("洗手......");  /\*\*  \* 此方法为真正执行的目标方法，第一个参数为被代理对象（注意，在匿名内部类中使用外部对象，需要是final的），第二个参数为所有的参数。  \*/  method.invoke(p, args);  System.*out*.println("洗手......");  }**else**{  method.invoke(p, args);  }  **return** **null**;  }  });  p4.eat();  p4.study();  }  } |

用代理方式实现easymall中的乱码问题：

|  |
| --- |
| 在easymall中创建包：cn.tedu.filter，创建类：EncodingFilter，并实现Filter接口，同时，在web.xml中添加如下配置：  <!-- 通用参数配置 -->  <context-param>  <param-name>encode</param-name>  <param-value>utf-8</param-value>  </context-param>  <!-- filter配置 -->  <filter>  <filter-name>EncodingFilter</filter-name>  <filter-class>cn.tedu.filter.EncodingFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>EncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |
| 在EncodingFilter中添加如下代码：  /\*\*  \* 当前web应用编码集  \*/  **private** String encode = **null**;  /\*\*  \* 初始化方法  \*/  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  //获取ServletContext对象  ServletContext sc = filterConfig.getServletContext();  //读取初始化参数中的编码集配置  **this**.encode = sc.getInitParameter("encode");  }  /\*\*  \* 过滤方法  \*/  **public** **void** doFilter(**final** ServletRequest request, ServletResponse response,FilterChain chain) **throws** IOException, ServletException {  //解决乱码  //响应乱码  response.setContentType("text/html;charset="+encode);  //请求参数：创建动态代理  ServletRequest proxyRequest = (ServletRequest) Proxy.*newProxyInstance*(request.getClass().getClassLoader(), request.getClass().getInterfaces(), **new** InvocationHandler() {    **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  //如果是调用getParameter方法  **if**("getParameter".equals(method.getName())){  //获取请求参数  String value = (String) method.invoke(request, args);  //重新编码并返回  **return** **new** String(value.getBytes("iso8859-1"),encode);  //如果是调用getParameterValues方法  }**else** **if**("getParameterValues".equals(method.getName())){  //获取请求参数组成的数组  String[] values = (String[]) method.invoke(request, args);  //遍历数组  **for** (**int** i = 0; i < values.length; i++) {  //对数组中每一个元素重新编码，并赋值给当前元素  values[i] = **new** String(values[i].getBytes("iso8859-1"),encode);  }  //返回编码后的数组  **return** values;  //如果是调用getParameterMap方法  }**else** **if**("getParameterMap".equals(method.getName())){  //获取请求参数组成的map  Map<String,String[]> map = (Map<String, String[]>) method.invoke(request, args);  //遍历map  **for** (Map.Entry<String, String[]>entry : map.entrySet()) {  String[] value = entry.getValue();  **for** (**int** i = 0; i < value.length; i++) {  value[i] = **new** String(value[i].getBytes("iso8859-1"),encode);  }  //将编码后的value赋值给map  map.put(entry.getKey(), value);  }  **return** map;  }**else**{  //如果是其他方法，直接执行目标方法并返回  **return** method.invoke(request, args);  }  }  });  //放行时，使用代理对象作为参数  chain.doFilter(proxyRequest, response);  }  /\*\*  \* 销毁方法  \*/  **public** **void** destroy() {  } |
| 可以创建一个测试servlet和测试jsp进行测试 |

**Easymall使用装饰的方式实现乱码解决**

|  |
| --- |
| 在EncodingFilter中添加或修改如下代码：  /\*\*  \* 内部类 ServletRequest的装饰类 改造了获取请求参数相关的方法 增加了乱码解决的代码  \*/  //继承了HttpServletRequestWrapper ，这个父类本身就是 HttpServletRequest的装饰器 在其中提供方法的默认的实现 不想改造的方法 不用管 想改造的方法 覆盖父类方法即可  **class** MyServletRequest **extends** HttpServletRequestWrapper{  **private** ServletRequest request = **null**;  **private** **boolean** hasNotEncode = **true**;  //构造器 接受传入的request保存在类的内部  **public** MyServletRequest(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }  //覆盖和获取请求参数相关的方法  @Override  **public** Map<String,String[]> getParameterMap() {  **try** {  //获取请求参数组成的map  Map<String,String[]> map = request.getParameterMap();  **if**(hasNotEncode){//由于request对此map会缓存 所以解决乱码的操作 只需要做一次 此处通过hasNotEncode来控制  //遍历map  **for** (Map.Entry<String, String[]>entry : map.entrySet()) {  //获取当前遍历到的值的数组  String[] values = entry.getValue();  //遍历值的数组  **for** (**int** i = 0; i < values.length; i++) {  //解决乱码 存回数组  values[i] = **new** String(values[i].getBytes("iso8859-1"),encode);  }  }  hasNotEncode = **false**;  }  //返回解决完乱码的map  **return** map;  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  @Override  **public** String[] getParameterValues(String name) {  **return** getParameterMap().get(name);  }  @Override  **public** String getParameter(String name) {  String[] values = getParameterValues(name);  **return** values == **null** ? **null** : values[0];  }  }  /\*\*  \* 过滤方法  \*/  **public** **void** doFilter(**final** ServletRequest request, ServletResponse response,FilterChain chain) **throws** IOException, ServletException {  //解决乱码  //响应乱码  response.setContentType("text/html;charset="+encode);  //请求参数  ServletRequest myRequest = **new** MyServletRequest((HttpServletRequest) request);  //资源放行  chain.doFilter(myRequest, response);  } |
| 分别删除LoginServlet、RegistServlet、AjaxCheckUsernameServlet中的处理乱码部分代码。 |

## 17.5.Easymall30天内自动登录

在处理用户登录时，判断用户是否勾选过30天内自动登录选项，如果用户名密码正确，且勾选过该选项，则发送包含用户名密码的cookie，让浏览器保存该cookie30天。

之后用过户再来访问时，如果用户未登录，且带了自动登录的cookie，且其中的用户名密码都正确，则自动登录该用户。

**开发步骤：**

login.jsp 提供30天内自动登录的选择框

LoginServlet 在用户登录成功后，增加处理30天内自动登录的逻辑

AutoLoginFilter 拦截所有请求，判断是否未登录，是否有自动登录cookie，自动登录cookie中的用户名密码是否正确，如果都满足，则自动登录，无论是否自动登录，都放行资源。

LogoutServlet 当用户手动注销时，删除自动登录cookie。

代码示例：

|  |
| --- |
| 在LoginServlet中，进行登录后添加如下代码：  //判断是否需要30天内自动登陆  **if**("true".equals(request.getParameter("autologin"))){  Cookie c = **new** Cookie("autologin",URLEncoder.*encode*(username,"utf-8")+"#"+password);  c.setMaxAge(3600\*24\*30);  c.setPath(request.getContextPath()+"/");  response.addCookie(c);  } |
| 在filter包下创建类AutoLoginFilter，实现Filter接口，并在web.xml中进行配置，在EncodingFilter后面添加如下代码：  <filter>  <filter-name>AutoLoginFilter</filter-name>  <filter-class>cn.tedu.filter.AutoLoginFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>AutoLoginFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |
| 在AutoLoginFilter中的doFilter方法中添加如下代码：  //1.只有未登录的用户才做自动登录  HttpServletRequest req = (HttpServletRequest) request;  **if**(req.getSession(**false**) == **null** || req.getSession().getAttribute("user") == **null**){  //2.只有携带了自动登录cookie的用户才做自动登录  Cookie[] cookies = req.getCookies();  Cookie autoCookie = **null**;  **if**(cookies != **null**){  **for** (Cookie cookie : cookies) {  **if**("autologin".equals(cookie.getName())){  autoCookie = cookie;  }  }  }  **if**(autoCookie != **null**){  //3.只有自动登录cookie中保存的用户名密码都正确才做自动登录  UserService service = BasicFactory.*getFactory*().getInstance(UserService.**class**);  String username = URLDecoder.*decode*(autoCookie.getValue(),"utf-8").split("#")[0];  String password = autoCookie.getValue().split("#")[1];  User user = service.loginUser(username, password);  System.*out*.println(username);  **if**(user != **null**){  //三个条件都满足,实现自动登录  req.getSession().setAttribute("user", user);  }  }  }  //4.无论自动登录，都放行访问  chain.doFilter(request, response); |
| 在LogoutServlet中杀死session后添加如下代码：  //删除自动登录cookie  Cookie cookie = **new** Cookie("autologin","");  cookie.setMaxAge(0);  cookie.setPath(request.getContextPath()+"/");  response.addCookie(cookie); |

## 17.6.MD5加密算法

### 17.6.1.MD5概述

用户名密码保存在客户端是一种十分危险的行为。所以需要进行加密后保存。

其中MD5就是一种比较常用的加密算法。

与其说MD5算法是一种加密算法，不如说是一种数据指纹（数据摘要）算法。

**其特点如下：**

任意大小的二进制数经过MD5计算后都能得到一个独一无二的128位二进制数。

不同的数据算出的MD5绝对不相同。

相同的数据算出的MD5一定相同。

只能有明文算出密文，密文是永远也无法算成明文的。

MD5大量应用于计算机中。如数据库中保存的密码通常都是经过MD5加密后的数据。如用户下载文件时可以进行MD5校验防止数据被篡改。

在记住用户名案例中，我们可以使用MD5进行加密后再保存在客户端，从而保证数据安全。

在数据库中保存的密码也不宜直接存储为明文。也要经过MD5加密后存储。

### 17.6.2.JAVA实现MD5加密

使用md5的算法进行加密: public static String md5(String plainText) { byte[] secretBytes = null; try { secretBytes = MessageDigest.getInstance("md5").digest( plainText.getBytes()); } catch (NoSuchAlgorithmException e) { throw new RuntimeException("没有md5这个算法！"); } String md5code = new BigInteger(1, secretBytes).toString(16); for (int i = 0; i < 32 - md5code.length(); i++) { md5code = "0" + md5code; } return md5code; }

### 17.6.3.使用md5对easymall项目中的密码加密

代码如下：

|  |
| --- |
| 在WebUtils类中加入md5加密代码：  /\*\*  \* 使用md5的算法进行加密  \*/  **public** **static** String md5(String plainText) {  **byte**[] secretBytes = **null**;  **try** {  secretBytes = MessageDigest.*getInstance*("md5").digest(  plainText.getBytes());  } **catch** (NoSuchAlgorithmException e) {  **throw** **new** RuntimeException("没有md5这个算法！");  }  String md5code = **new** BigInteger(1, secretBytes).toString(16);  **for** (**int** i = 0; i < 32 - md5code.length(); i++) {  md5code = "0" + md5code;  }  **return** md5code;  } |
| 在RegistServlet中，实现注册之前，添加如下代码：  //对password进行md5加密，然后再存入数据库  user.setPassword(WebUtils.*md5*(user.getPassword()));  //4.实现注册(将用户信息保存进数据库)  UserService service = BasicFactory.*getFactory*().getInstance(UserService.**class**);  service.registUser(user); |
| 在LoginServlet中，获取请求参数password修改如下：  String password = WebUtils.*md5*(request.getParameter("password")); |

## 17.7.作业练习

1. Filter 的作用是什么？

答：它使用户可以改变一个request和修改一个response。它能够在一个request到达servlet之前预处理request，也可以在离开servlet时处理response。可以实现全站乱码解决、权限访问控制、过滤敏感词汇、压缩响应信息等功能。

1. Filter的生命周期？

答：在web应用启动时，会创建出web应用中配置的过滤器对象，创建出过滤器对象后立即调用init方法进行初始化的操作，之后一直存活，直到web应用被销毁时，Filter跟着被销毁，在销毁之前调用destory方法执行善后工作。

1. Filter开发流程？

答：Filter开发分为2步：

（1）编写java类实现Filter接口，并实现其doFilter方法。

（2）在web.xml 文件中使用<filter>和<filter-mapping>元素对编写的filter类进行配 置，并设置它所能拦截的资源。