# 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。

下面我们在三层架构项目中完成，DAO使用DbUtils。一般为了数据安全，用户的密码不会以明文形式存储在数据库中，而是以加密形式存储在数据库中，例如使用MD5和SHA等加密算法，关于在Java中如何使用这些加密算法，请先看完附录的部分，我们稍后就使用其写好的MessageDigestUtils类。

创建用户表的SQL：

|  |
| --- |
| **create table** users (  **id int primary key**,  **username varchar**(50) **not null**,  **password varchar**(100) **not null** ); |

实体类(domain)：

|  |
| --- |
| **package** com.domain;  **import** java.io.Serializable;  **public class** User **implements** Serializable {  **private int id**;  **private** String **username**;  **private** String **password**;  // 自行编写getter和setter等方法。 } |

DAO接口：

|  |
| --- |
| **package** com.dao;  **import** com.domain.User;  **public interface** UserDao {  */\*\*  \* 根据用户名和密码返回用户信息  \** ***@param username*** *用户名  \** ***@param pwd*** *密码  \** ***@return*** *用户信息  \*/* User find(String username, String pwd);   */\*\*  \* 添加用户  \** ***@param user*** *用户  \*/* **void** add(User user); } |

DAO实现：

|  |
| --- |
| **package** com.dao.impl;  **import** com.dao.UserDao; **import** com.domain.User; **import** com.utils.DBCPUtil; **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** java.sql.SQLException;  **public class** UserDaoImpl **implements** UserDao {  QueryRunner **qr** = **new** QueryRunner(DBCPUtil.*getDataSource*());  @Override  **public** User find(String username, String pwd) {  **try** {  **return qr**.query(**"select \* from users where username=? and password=?"**, **new** BeanHandler<User>(User.**class**), username, pwd);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** add(User user) {  **try** {  **qr**.update(**"insert into users values(?,?,?)"**, user.getId(), user.getUsername(), user.getPassword());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

上述DAO实现利用了之前的DbUtils和DBCPUtil，注意引用在项目中，同时注意将有关jar包放入WEB-INF/lib下，并将db.properties文件放在src目录下。

Service接口：

|  |
| --- |
| **package** com.service;  **import** com.domain.User;  **public interface** UserService {  *// 登录* User login(String username, String pwd);  *// 添加* **void** add(User user); } |

Service实现：

|  |
| --- |
| **package** com.service.impl;  **import** com.dao.UserDao; **import** com.dao.impl.UserDaoImpl; **import** com.domain.User; **import** com.service.UserService;  **public class** UserServiceImpl **implements** UserService {  **private** UserDao **userDao** = **new** UserDaoImpl();  @Override  **public** User login(String username, String pwd) {  **return userDao**.find(username, pwd);  }   @Override  **public void** add(User user) {  **userDao**.add(user);  } } |

先往数据库中添加数据：

|  |
| --- |
| **package** com.test;  **import** com.domain.User; **import** com.service.UserService; **import** com.service.impl.UserServiceImpl; **import** com.utils.MessageDigestUtils; **import** org.junit.Test;  **public class** AppTest {  **private** UserService **userService** = **new** UserServiceImpl();  @Test  **public void** test() {  User user = **new** User();  user.setId(2);  user.setUsername(**"张三"**);  user.setPassword(MessageDigestUtils.encrypt(**"MD5"**, **"123"**.getBytes(), **false**));  userService.add(user);  } } |

由于数据库中保存的是加密的数据，因此上面使用我们的工具进行了MD5加密，因此注意也把MessageDigestUtils类引入到项目中。

下面是登录页面（login.jsp），提供记住密码选项：

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>登录</**title**> </**head**> <**body**> <**form action="${**pageContext.request.contextPath**}/servlet/loginServlet" method="post"**>  用户名：<**input name="username"**/><**br**/>  密码： <**input type="password" name="password"**/><**br**/>  <**input type="checkbox" name="rememberPwd"**/>记住密码<**br**/>  <**input type="submit" value="登录"**/> </**form**> </**body**> </**html**> |

对应的登录的Servlet：

|  |
| --- |
| **package** com.controller;  **import** com.domain.User; **import** com.service.UserService; **import** com.service.impl.UserServiceImpl; **import** com.utils.Base64Utils; **import** com.utils.MessageDigestUtils;  **import** javax.servlet.ServletException; **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.\*; **import** java.io.IOException;  @WebServlet(urlPatterns = **"/servlet/loginServlet"**) **public class** LoginServlet **extends** HttpServlet {  **private** UserService **userService** = **new** UserServiceImpl();  @Override  **protected void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  *// 获取用户名和密码* String username = request.getParameter(**"username"**);  String password = request.getParameter(**"password"**); *// 这个获得的是明文密码  // 验证登录。* User user = **userService**.login(username, MessageDigestUtils.*encrypt*(**"MD5"**, password.getBytes(), **false**));  *// 登录失败* **if**(user == **null**){  response.getWriter().write(**"错误的用户名或密码"**);  **return**;  }  *// 登录成功，先把用户信息保存在session中* HttpSession session = request.getSession();  session.setAttribute(**"user"**, user);  *// 再来判断用户是否选择记住密码，如果记住密码了，还要设置cookie。* String remember = request.getParameter(**"rememberPwd"**);  **if**(remember != **null**){  *// 设置cookie* Cookie c = **new** Cookie(**"loginInfo"**, Base64Utils.*encodeString*(username) + **"\_"** + MessageDigestUtils.*encrypt*(**"MD5"**, password.getBytes(), **false**));  c.setMaxAge(Integer.***MAX\_VALUE***);  c.setPath(request.getContextPath());  response.addCookie(c);  }  *//重定向到主页* response.sendRedirect(request.getContextPath());  } } |

在上面登录的Servlet中我们要注意两点。一是调用service的登录方法时，同样要把密码用MD5加密，因为数据库存的就是密文，否则查询不到。二是保存cookie时，用户名用Base64进行编码，防止中文无法存储的问题，密码还是以MD5的方式保存在cookie中，防止密码泄露。

其中，上面用到的Base64编码解码是我写好的工具类，专门用于将字符串编码成Base64和将Base64解码成字符串：

|  |
| --- |
| **package** com.utils;  **import** java.io.IOException; **import** java.util.Base64;  **public class** Base64Utils {  *// 编码字符串* **public static** String encodeString(String str) **throws** IOException {  **return** Base64.*getEncoder*().encodeToString(str.getBytes());  }   *// 将Base64编码解码，并以字符串方式返回。* **public static** String decodeString(String base64Str) **throws** IOException {  **byte**[] res = Base64.*getDecoder*().decode(base64Str);  **return new** String(res);  } } |

应用的首页（index.jsp）如下：

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <**html**>  <**head**>  <**title**>首页</**title**>  </**head**>  <**body**>  <**c:if test="${empty** sessionScope.user**}"**>  <**a href="${**pageContext.request.contextPath**}/login.jsp"**>登录</**a**>  </**c:if**>  <**c:if test="${**sessionScope.user != **null }"**>  欢迎您：**${**sessionScope.user.username**}** </**c:if**>  </**body**> </**html**> |

为了简化其他页面的自动登录，所以编写的过滤器类如下：

|  |
| --- |
| **package** com.filter;  **import** com.domain.User; **import** com.service.UserService; **import** com.service.impl.UserServiceImpl; **import** com.utils.Base64Utils;  **import** javax.servlet.\*; **import** javax.servlet.http.Cookie; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** javax.servlet.http.HttpSession; **import** java.io.IOException;  *// 拦截动态资源* **public class** AutoLoginFilter **implements** Filter {  *// 用户登录的service* **private** UserService **userService** = **new** UserServiceImpl();  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException { }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) **throws** IOException, ServletException {  *// 先转换类型* HttpServletRequest request = **null**;  HttpServletResponse response = **null**;  **try** {  request = (HttpServletRequest) req;  response = (HttpServletResponse) resp;  } **catch** (ClassCastException e) {  **throw new** RuntimeException(**"non-HTTP request or response"**);  }  *// 通过session判断是否登录了* HttpSession session = request.getSession();  User sessionUser = (User) session.getAttribute(**"user"**);  **if** (sessionUser == **null**) {  *// 当前会话中没有用户信息，则寻找Cookie。* Cookie userInfoCookie = **null**;  Cookie[] cookies = request.getCookies();  *// 下面必须判断cookies不为null。因为浏览器第一次访问网站时，过滤器获得的cookie就是null。  // 这也是for循环中判断遍历对象是否为null的一个技巧，可以使用。这样就不需要在if中进行循环了，代码简洁一点。* **for** (**int** i = 0; cookies!= **null** && i < cookies.**length**; i++) {  **if** (**"loginInfo"**.equals(cookies[i].getName())) {  userInfoCookie = cookies[i];  }  }  *// 如果找到了cookie，就用其中的信息进行登录* **if** (userInfoCookie != **null**) {  *// 截取用户名和密码* String[] userInfoStrs = userInfoCookie.getValue().split(**"\_"**);  *// 注意下面的用户名进行Base64解码，密码由于cookie中本来存的就是密文，所以直接传递给service即可。* User user = **userService**.login(Base64Utils.*decodeString*(userInfoStrs[0]), userInfoStrs[1]);  **if** (user != **null**) {  *// 这里就是登录成功了，那么直接把用户信息保存到session中即可，这样Servlet等动态资源就自动知道用户登录了。* session.setAttribute(**"user"**, user);  }  }  }  *// 不管是否登录，都放行* chain.doFilter(request, response);  }   @Override  **public void** destroy() { } } |

上述的过滤器类是重点代码，关注点有：cookies的遍历、service登录时的参数、保存用户信息到session中实现自动登录、放行。这样就完成了自动登录，其他Servlet中只要判断用户是否登录即可，无需做自动登录查看cookie的代码。当然对于一些管理系统来说，是禁止游客访问资源的，那么我们可以把上面的代码改成如果没有登录就自动跳转到首页登录。甚至为了灵活性，可以让开发者在文件中配置允许游客访问（或者禁止游客访问）的链接，然后我们在过滤器中读取这些链接，如果游客访问的链接属于这些链接，就放行（或者跳转到登录页）。

最后我们配置过滤器（下面用到了之前的编码过滤器，注意映射的顺序。过滤器和映射可分开写，一个过滤器可配置多个映射）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app 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"  version="3.1"**>  <**filter**>  <**filter-name**>EncodingFilter</**filter-name**>  <**filter-class**>com.filter.EncodingFilter</**filter-class**>  </**filter**>  <**filter**>  <**filter-name**>AutoLoginFilter</**filter-name**>  <**filter-class**>com.filter.AutoLoginFilter</**filter-class**>  </**filter**>  <**filter-mapping**>  <**filter-name**>EncodingFilter</**filter-name**>  <**url-pattern**>/servlet/\*</**url-pattern**>  </**filter-mapping**>  <**filter-mapping**>  <**filter-name**>AutoLoginFilter</**filter-name**>  <**url-pattern**>/servlet/\*</**url-pattern**>  </**filter-mapping**>  <**filter-mapping**>  <**filter-name**>AutoLoginFilter</**filter-name**>  <**url-pattern**>\*.jsp</**url-pattern**>  </**filter-mapping**> </**web-app**> |

该案例的完整代码已经放在project-demos/auto-login.zip包中（其中的WebDemo项目）。

## 1.7 过滤器的其他配置

过滤器默认会过滤“REQUEST”类型的请求和响应，什么是REQUEST类型呢？就是一次普通的对资源的请求和响应的过程。

除了REQUEST类型的请求和响应，还有FORWARD、INCLUDE和ERROR类型的请求和响应，默认情况下，过滤器是不会拦截这些类型的请求和响应的。

（1）FORWARD类型：就是用户访问Servlet或JSP后，Servlet或JSP又把请求和响应转发到其他页面，这个第二次转发的请求和响应就是FORWARD类型。如果想过滤器过滤该类型的请求响应，需要在filter-mapping中配置：

|  |
| --- |
| <**filter-mapping**>  <**filter-name**>OneFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  <**dispatcher**>REQUEST</**dispatcher**>  <**dispatcher**>FORWARD</**dispatcher**> </**filter-mapping**> |

上述配置中既过滤了REQUEST和FORWARD类型的请求和响应，否则如果只配置FORWARD的话，那么普通的REQUEST将不被过滤。

（2）INCLUDE类型：就是用户访问Servlet或JSP后，Servlet或JSP又动态包含了其他Servlet或JSP，这种动态包含的请求和响应就是INCLUDE类型。如果想过滤器过滤该类型的请求响应，则需要在filter-mapping中加上：

|  |
| --- |
| <**filter-mapping**>  <**filter-name**>OneFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  <**dispatcher**>REQUEST</**dispatcher**>  <**dispatcher**>INCLUDE</**dispatcher**> </**filter-mapping**> |

的配置。注意只限于动态包含，因为静态包含是源码包含，访问时相当于访问了同一个Servlet。

（3）ERROR类型：就是用户访问Servlet或JSP后，由于Servlet或JSP出错，跳转到了系统配置的错误页面，这样就是ERROR类型。如果想过滤器过滤该类型的请求响应，则需要在filter-mapping中加上：

|  |
| --- |
| <**filter-mapping**>  <**filter-name**>OneFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  <**dispatcher**>REQUEST</**dispatcher**>  <**dispatcher**>ERROR</**dispatcher**> </**filter-mapping**> |

的配置。注意如果在JSP页面中自行在@page指令中设置errorPage页面，那么这时使用的是转发技术，只有跳到到为应用配置的全局错误页面，才算是ERROR类型。

当然，REQUEST、FORWARD、INCLUDE和ERROR类型可以任意搭配使用。

## 1.8 巩固装饰设计模式

在Java SE中讲过装饰设计模式，Java中IO流的设计中就用到了装饰设计模式。例如我们想对BufferedReader实现在行首添加行号、在行尾添加分号的功能，则可分别针对这两种功能实现两个类，例如添加行号的功能类：

|  |
| --- |
| **package** com.company;  **import** java.io.BufferedReader; **import** java.io.IOException;  **public class** LineNumBufferedReader **extends** BufferedReader {  *// 维护BufferedReader对象* **private** BufferedReader **bufferedReader**;  **private int lineNum** = 1; *// 首行行号从1开始  // 传递BufferedReader对象* **public** LineNumBufferedReader(BufferedReader bufferedReader) {  **super**(bufferedReader); *// 由于BufferedReader没有无参构造，因此调用super传递BufferedReader对象* **this**.**bufferedReader** = bufferedReader;  }   *// 重写readLine方法* @Override  **public** String readLine() **throws** IOException {  String data = **bufferedReader**.readLine();  *// 需要判断是否为null，如果是null，表示文件读完了，应该还是返回null。* **return** data == **null** ? data : (**lineNum**++) + **" "** + data;  } } |

同理，添加分号的类：

|  |
| --- |
| **package** com.company;  **import** java.io.BufferedReader; **import** java.io.IOException;  **public class** AddSemicolonBufferedReader **extends** BufferedReader {  *// 维护BufferedReader对象* **private** BufferedReader **bufferedReader**;  **public** AddSemicolonBufferedReader(BufferedReader bufferedReader) {  **super**(bufferedReader);  **this**.**bufferedReader** = bufferedReader;  }   @Override  **public** String readLine() **throws** IOException {  String data = **bufferedReader**.readLine();  **return** data == **null** ? data : data + **";"**;  } } |

这样，使用BufferedReader就能以扩展的方式使用上面的功能：

|  |
| --- |
| **package** com.company;  **import** org.junit.Test; **import** java.io.\*;  **public class** AppTest {  @Test  **public void** testIO() {  BufferedReader bufferedReader = **null**;  **try** {  bufferedReader = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(**"src/com/company/domain/Account.java"**), **"UTF-8"**));  *// 进行一层层的包装。如果只想要其中一个功能，只要包装一次即可* bufferedReader = **new** LineNumBufferedReader(**new** AddSemicolonBufferedReader(bufferedReader));  String str = **null**;  **while** ( (str = bufferedReader.readLine()) != **null**) {  System.***out***.println(str);  }  } **catch** (IOException e) {  **throw new** RuntimeException(e);  } **finally** {  **if** (bufferedReader != **null**) {  **try** {  bufferedReader.close();  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  }  }  } } |

从上面的代码中可以看出，装饰设计模式中也是需要维护一个对象，然后根据需要修改方法来扩展功能，这样的做法和在代理模式中很相似。但装饰设计模式和代理模式的侧重点不同，装饰设计模式强调灵活定制各种功能（例如上面的增加行号和分号功能），而代理模式强调对方法的修改。

## 1.9 包装类在过滤器中的应用

### 1.9.1 Tomcat 7之前的GET编码过滤

Tomcat7之前，GET请求数据默认以ISO-8859-1编码，这会导致乱码，下面使用过滤器解决这个问题。为了能让Servlet中调用getParameter方法能直接拿到不乱码的数据，因此需要在过滤器中对HttpServletRequest的getParameter方法进行修改（即让调用request.getParameter方法时返回正确编码的数据）。按照我们之前所学，使用代理可以实现。但现在我们使用装饰设计模式，因为Servlet的API提供了一个装饰类HttpServletResponseWrapper，其中实现了request的默认方法，这样就不需要我们默认实现其中的每一个方法了。因此我们可利用该类实现需求，则代码如下：

|  |
| --- |
| **package** com.webdemo.filter;  **import** javax.servlet.\*; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletRequestWrapper; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException; **import** java.io.UnsupportedEncodingException;  **public class** EncodingFilter **implements** Filter {  **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 = **null**;  HttpServletResponse response = **null**;  **try** {  request = (HttpServletRequest) req;  response = (HttpServletResponse) resp;  } **catch** (ClassCastException e) {  **throw new** RuntimeException(**"non-HTTP request or response"**);  }  String encoding = **filterConfig**.getInitParameter(**"encoding"**);  **if** (encoding == **null**) {  encoding = **"UTF-8"**;  }  *// 设置编码* request.setCharacterEncoding(encoding);  response.setCharacterEncoding(encoding);  response.setContentType(**"text/html;encoding="** + encoding);  *// 放行。注意，放行时，request包装成下面的经过我们处理的对象，这样Servlet中就能正确拿到数据了* chain.doFilter(**new** EncodeHttpServletResponse(request), response);  }   @Override  **public void** destroy() { } }  *// 我们的对request的装饰类* **class** EncodeHttpServletResponse **extends** HttpServletRequestWrapper {  *// 维护的request对象* **private** HttpServletRequest **request**;  **public** EncodeHttpServletResponse(HttpServletRequest request) {  **super**(request);  **this**.**request** = request;  }  *// 需要重写getParameter方法* @Override  **public** String getParameter(String name) {  String value = **request**.getParameter(name);  **if** (value == **null**) {  **return null**;  }  **if**(**"GET"**.equalsIgnoreCase(**request**.getMethod())){  *// 如果是GET请求方式就进行处理* **try** {  **byte** b[] = value.getBytes(**"ISO-8859-1"**); *// 再以该编码进行编码* value = **new** String(b, **request**.getCharacterEncoding()); *// 再以新的编码解码。这里使用request.getCharacterEncoding()是没问题的，因为request的编码在上面的过滤器类中已经设置好了，直接使用即可。* } **catch** (UnsupportedEncodingException e) {  **throw new** RuntimeException(e);  }  }  **return** value;  } } |

至于getParameterValues()这样的方法可自行参照实现。这样就彻底解决了Tomcat7之前的编码问题。

### 1.9.2 过滤脏话和HTML代码

根据上面的例子，我们也能写出过滤脏话和HTML代码的过滤器。

为了过滤脏话，可以在获得请求参数数据的时候，就把脏话过滤成“\*\*”，从源头上消灭脏话，过滤器可如下写：

|  |
| --- |
| **package** com.webdemo.filter;  **import** javax.servlet.\*; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletRequestWrapper; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException;  **public class** DirtyWordsFilter **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 = **null**;  HttpServletResponse response = **null**;  **try** {  request = (HttpServletRequest) req;  response = (HttpServletResponse) resp;  } **catch** (ClassCastException e) {  **throw new** RuntimeException(**"non-HTTP request or response"**);  }  *// 直接放行，但是request对象使用我们下面的类进行包装* chain.doFilter(**new** RemoveDirtyWordsHttpServletRequest(request), response);  }   @Override  **public void** destroy() { } }  *// 清除脏话的request类* **class** RemoveDirtyWordsHttpServletRequest **extends** HttpServletRequestWrapper {  **private** HttpServletRequest **request**;  **public** RemoveDirtyWordsHttpServletRequest(HttpServletRequest request) {  **super**(request);  **this**.**request** = request;  }   @Override  **public** String getParameter(String name) {  String value = **request**.getParameter(name);  **if** (value == **null**) {  **return null**;  }  *// 规定脏话。实际中，是从数据库中读取的* String[] dirtyWords = {**"SB"**, **"TMD"**};  *// 过滤脏话* **for** (String dirtyWord : dirtyWords) {  value.replace(dirtyWord, **"\*\*"**);  }  **return** value;  } } |

同理，可过滤HTML代码。当用户将HTML代码输入到文本框中，一般后端要进行转义处理再保存到数据库中，否则以后在页面上显示这些数据时，页面上可能会显示HTML代码并可能执行JavaScript，导致不安全。因此我们需要对HTML代码进行过滤，即转义即可：

|  |
| --- |
| **package** com.webdemo.filter;  **import** javax.servlet.\*; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletRequestWrapper; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException;  **public class** HTMLFilter **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 = **null**;  HttpServletResponse response = **null**;  **try** {  request = (HttpServletRequest) req;  response = (HttpServletResponse) resp;  } **catch** (ClassCastException e) {  **throw new** RuntimeException(**"non-HTTP request or response"**);  }  *// 直接放行，但是request对象使用我们下面的类进行包装* chain.doFilter(**new** HandleHTMLHttpServletRequest(request), response);  }   @Override  **public void** destroy() { } }  **class** HandleHTMLHttpServletRequest **extends** HttpServletRequestWrapper {  **private** HttpServletRequest **request**;  **public** HandleHTMLHttpServletRequest(HttpServletRequest request) {  **super**(request);  **this**.**request** = request;  }   @Override  **public** String getParameter(String name) {  String value = **request**.getParameter(name);  **if** (value == **null**) {  **return** value;  }  *// 进行HTML转义* value = value.replace(**"&"**, **"&amp;"**) *// 先把&替换，如果这句话在后面，那么会把前面生成的&又替换一遍，导致不正常。* .replace(**"<"**, **"&lt;"**)  .replace(**">"**, **"&gt;"**)  .replace(**"\""**, **"&quot;"**);  **return** value;  } } |

### 1.9.3 GZIP压缩过滤器

WEB开发初期就讲过GZIP压缩，现在使用过滤器压缩Servlet和JSP资源，省去在Servlet中处理的步骤。Servlet输出数据无非就是使用字节流getOutputStream()和字符流getWriter()，只要我们能拿到这两个流的数据就能完成压缩了，因此我们使用一个类继承HttpServletResponseWrapper类，重写其中的这两个方法，让其中的流数据都先存到我们的临时仓库中。

其中重写getWriter()比较简单，但重写getOuputStream()时，为了使数据存入临时仓库中，还要装饰一下ServletOutputStream类，重写其中的write(int b)方法，因为不论调用ServletOutputStream的哪个print还是write方法，最终使用的都是write(int b)方法，这个可以查看源码验证。

最终，我们的代码写成：

|  |
| --- |
| **package** com.webdemo.filter;  **import** javax.servlet.\*; **import** javax.servlet.annotation.WebFilter; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** javax.servlet.http.HttpServletResponseWrapper; **import** java.io.\*; **import** java.util.zip.GZIPOutputStream;  @WebFilter(urlPatterns = **"/\*"**) **public class** GZIPFilter **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 = **null**;  HttpServletResponse response = **null**;  **try** {  request = (HttpServletRequest) req;  response = (HttpServletResponse) resp;  } **catch** (ClassCastException e) {  **throw new** RuntimeException(**"non-HTTP request or response"**);  }  *// 放行，但放行的是修饰过的response* GZIPHttpServletResponse gzipResponse = **new** GZIPHttpServletResponse(response);  chain.doFilter(request, gzipResponse);  *// 拿到response中存储的仓库数据* **byte**[] data = gzipResponse.getByteArray();  System.***out***.println(**"压缩前："** + data.**length**);  *// 判断客户端是否支持gzip* **if** (request.getHeader(**"Accept-Encoding"**).contains(**"gzip"**)) {  *// 压缩输出* ByteArrayOutputStream buffer = **new** ByteArrayOutputStream();  GZIPOutputStream gzipOutputStream = **new** GZIPOutputStream(buffer);  gzipOutputStream.write(data);  gzipOutputStream.close(); *// 这里一定要关闭gzip流，然后再从buffer中拿到被压缩的数据* data = buffer.toByteArray(); *// 得到压缩之后的数据* System.***out***.println(**"压缩后："** + data.**length**);  *// 告知客户端的压缩方式* response.setHeader(**"Content-Encoding"**, **"gzip"**);  response.setContentLength(data.**length**); *// 告知数据长度* }  *// 最后需要统一用ServletOutputStream字节流把结果输出，这样网页才能正常显示。  // 且此时要用response对象获得字节流，否则还是向仓库中写入数据。* ServletOutputStream sos = response.getOutputStream();  sos.write(data);  }   @Override  **public void** destroy() { } }  *// 修饰HttpServletResponse* **class** GZIPHttpServletResponse **extends** HttpServletResponseWrapper {  **private** HttpServletResponse **response**;  *// 临时存放字节数据的“仓库”* **private** ByteArrayOutputStream **baos** = **new** ByteArrayOutputStream();  *// 维护字符流pw* **private** PrintWriter **pw**;  *// 传递response* **public** GZIPHttpServletResponse(HttpServletResponse response) {  **super**(response);  **this**.**response** = response;  }   @Override  **public** PrintWriter getWriter() **throws** IOException {  *// 使Servlet中调用getWriter方法输出数据时，先存入我们的仓库中。这里，使用了response的编码，因为如果有编码过滤器的话，response的编码在这之前就设置好了，直接使用即可。* **pw** = **new** PrintWriter(**new** OutputStreamWriter(**baos**, **response**.getCharacterEncoding()));  **return pw**;  }   @Override  **public** ServletOutputStream getOutputStream() **throws** IOException {  *// 这里返回的是经过装饰的ServletOutputStream类。其中重写了write方法，使得使用该对象时，也先存入到仓库中* **return new** ModifyServletOutputStream(**baos**, **response**.getOutputStream());  }   *// 这是我们自己写的方法，用于在过滤器中拿到仓库中的字节数据。字节流和字符流通用，因为都是使用baos作为仓库* **public byte**[] getByteArray() {  **try** {  *// 关闭PrintWriter，且刷新baos，确保其中的数据都保存到了baos中  // 这里使用的是pw，而不能使用“getWriter()”，因为调用getWriter()方法的话，pw又被重新new一次。* **if** (**pw** != **null**) {  **pw**.close();  }  *// 刷新baos。* **baos**.flush();  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  *// 返回baos中的字节数据* **return baos**.toByteArray();  } }  *// 重写的ServletOutputStream类* **class** ModifyServletOutputStream **extends** ServletOutputStream {  **private** ServletOutputStream **servletOutputStream**;  **private** ByteArrayOutputStream **baos**;  *// 为了能使用baos，把baos注入了进来* **public** ModifyServletOutputStream (ByteArrayOutputStream baos, ServletOutputStream servletOutputStream) {  **this**.**baos** = baos;  **this**.**servletOutputStream** = servletOutputStream;  }   *// 这是必须要实现的方法，使用servletOutputStream的即可* @Override  **public boolean** isReady() {  **return servletOutputStream**.isReady();  }   *// 这是必须要实现的方法，使用servletOutputStream的即可* @Override  **public void** setWriteListener(WriteListener writeListener) {  **servletOutputStream**.setWriteListener(writeListener);  }   *// 这是重写的重点，输出数据最终都调用这个方法* @Override  **public void** write(**int** b) **throws** IOException {  *// 就是让数据先写入到我们的仓库baos中。* **baos**.write(b);  } } |

比较复杂一点，需要注意的点有很多，多看注释。上面的过滤器对Servlet和JSP都是有效的（JSP也是Servlet），对静态资源也是有效的（静态资源也是通过默认的Servlet处理的）。静态资源的GZIP压缩也可参照网址：http://panyongzheng.iteye.com/blog/2249815。

不建议对所有资源都进行GZIP压缩处理，只有一些纯文本等类型建议压缩，因为这种文本的压缩效率很高，不建议对图片、视频、压缩文件等进行GZIP压缩，因为这些文件本身就被压缩过，再压缩的意义不大，且浪费服务器资源。

# 2. 观察者设计模式（监听器）

在学习监听器前，我们先学习观察者设计模式，因为监听器其实就是使用了观察者设计模式。

## 2.1 回顾

我们从之前学习的例子说起。编写GUI程序时遇到的第一个问题就是无法关闭窗体，我们是通过处理windowClosing窗体关闭事件解决的：

|  |
| --- |
| **package** com.demo;  **import** java.awt.\*; **import** java.awt.event.WindowAdapter; **import** java.awt.event.WindowEvent;  **public class** Main {  **public static void** main(String[] args) {  Frame frame = **new** Frame(**"标题"**);  frame.setBounds(500, 200, 400, 300);  frame.setVisible(**true**);  *// 添加事件监听* frame.addWindowListener(**new** WindowAdapter() {  @Override  **public void** windowClosing(WindowEvent e) {  *// 拿到事件源对象，dispose窗体* Frame f = (Frame) e.getSource();  f.dispose();  }  });  } } |

代码中，通过addWindowListener添加对事件的监听，这样当用户点击关闭按钮时，就会执行windowClosing事件处理的代码。

实际上事件的处理就是使用了观察者设计模式，即被观察者（窗体）一旦有了某操作（例如执行了某方法，或这里的被观察者窗体被点击关闭按钮了），就会被观察者（监听器）发现，进而执行对应的操作。

上述处理中有这样几个概念：

（1）frame对象：这是事件源，即发生事件的对象。其是被监听的目标。

（2）WindowListener（默认适配器是WindowAdapter）：这就是监听器，监听器中会对应者事件源中的相关内容。

（3）WindowEvent：这是事件对象，其中封装了事件源等数据，便于调用者拿到使用。因此WindowEvent对象有一个getSource()方法，就是用于拿到事件源对象的。

（4）addWindowListener(WindowListener listener)方法用于注册监听器。如果没有注册的话，是无任何反应的。

## 2.2 观察者设计模式案例

上述都是使用别人设计好的监听器，是比较简单的，为了清晰原理，现在自己编写观察者设计模式的示例代码，这样能更好地理解是如何实现的。

观察者模式，即能实现监听被观察者对象：当被观察者执行了某操作时，如果该操作被监听器注册了，那么监听器中编写的处理方法也将执行。

我们要实现的案例：监听学生的学习和睡觉行为，分别在学习前和睡觉前提供一杯牛奶。我们参照前面事件处理的方法编写代码。

（1）首先定义一个Student类，提供study和sleep方法：

|  |
| --- |
| **package** com.demo;  **public class** Student {  **private** String **name**;   **public** Student(String name) {  **this**.**name** = name;  }   **public void** study() {  System.***out***.println(**name** + **"开始学习"**);  }   **public void** sleep() {  System.***out***.println(**name** + **"开始睡觉"**);  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

（2）编写StudentListener学生监听器接口，声明可监听学习前和睡觉前动作。参照事件处理代码，还要在接口方法中传递一个事件对象StudentEvent。事件对象用于传递事件源，StudentEvent类下面编写。因此这里的StudentListener接口写为：

|  |
| --- |
| **package** com.demo;  **public interface** StudentListener {  *// 监听学习前* **void** beforeStudy(StudentEvent e);  *// 监听睡觉后* **void** beforeSleep(StudentEvent e); } |

（3）编写事件对象StudentEvent，其中封装事件源对象，且这个对象通过构造函数传递进来：

|  |
| --- |
| **package** com.demo;  **public class** StudentEvent {  **private** Object **source**; *// 维护事件源对象* **public** StudentEvent(Object source) {  **this**.**source** = source;  }  *// 提供getSource方法* **public** Object getSource() {  **return this**.**source**;  } } |

（4）最后，还需要改进Student代码，使得Student能够被监听。这里给Student添加一个addStudentListener方法，方法传入StudentListener对象，由该类维护，然后，在每个要监听的行为中判断是否有监听器注册，否则有，就执行监听器中的方法。因此代码如下：

|  |
| --- |
| **package** com.demo;  **public class** Student {  **private** String **name**;  **private** StudentListener **listener**; *// 维护listener* **public** Student(String name) {  **this**.**name** = name;  }   **public void** study() {  **if** (**listener** != **null**) {  **listener**.beforeStudy(**new** StudentEvent(**this**));  }  System.***out***.println(**name** + **"开始学习"**);  }   **public void** sleep() {  **if** (**listener** != **null**) {  **listener**.beforeSleep(**new** StudentEvent(**this**));  }  System.***out***.println(**name** + **"开始睡觉"**);  }   *// 提供addStudentListener方法* **public void** addStudentListener(StudentListener listener) {  **this**.**listener** = listener;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

其实addStudentListener也是使用了策略设计模式。现在我们执行如下程序：

|  |
| --- |
| **package** com.demo;  **public class** Main {  **public static void** main(String[] args) {  Student student = **new** Student(**"张三"**);  *// 添加监听* student.addStudentListener(**new** StudentListener() {  @Override  **public void** beforeStudy(StudentEvent e) {  Student obj = (Student) e.getSource();  System.***out***.println(obj.getName() + **"，给你一杯牛奶"**);  }   @Override  **public void** beforeSleep(StudentEvent e) {  Student obj = (Student) e.getSource();  System.***out***.println(obj.getName() + **"，给你一杯牛奶"**);  }  });  *// 学生执行了行为* student.study();  student.sleep();   } } |

就会发现能够监听学生的行为。当然，如果不通过addStudentListener注册监听，那么调用方法时就像什么都没有发生一样。

# 3. 监听器(Listener)

## 3.1 监听器概述和使用

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等的情况。Servlet中提供的监听器一般用来监听服务器维护的对象（例如servletContext、session等），这样能够让开发者在特定的时机执行某些自定义操作。

Servlet规范中一共提供了8个监听器，下面分类进行介绍。

### 3.1.1 生命周期监听器

Servlet提供了三个生命周期监听器，分别是ServletContextListener、HttpSessionListener和ServletRequestListener监听器接口，它们分别用于监听servletContext、session和request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。

编写监听器时，开发者只要自定义类实现监听器接口即可。

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;  **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..."**);  } } |

上述代码测试了能否成功执行操作并拿到request和域对象。这些事件对象sre等中也提供了getSource()方法用于拿到事件源对象，但是需要强转，为了方便，使用getServletRequest等方法就能直接拿到事件源对象。

当然，为了使监听器生效，需要在web.xml中注册该监听器，只要在listener-class节点中配置该监听器即可。例如：

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

为了简单，也可在类上使用@WebListener注解来配置监听器。

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ServletContextListener和HttpSessionListener也是类似的步骤，要注意的是servletContext和session对象创建和销毁的时机。

servletContext对象仅会在服务器启动时创建一次，最后在服务器正常关闭时销毁。因此，ServletContextListener的应用场景是完成系统的初始化工作。例如我们以后要学习的Spring框架，可用于管理应用中使用的各种对象（例如DAO、Service对象等）。在Web应用中，Spring框架就是通过配置监听器来完成对象的初始化工作的，Spring的监听器ContextLoaderListener就实现了ServletContextListener接口。Spring使用容器存放应用需要使用的对象，可以把容器理解为一个Map集合，键是对象的唯一名称，值是对象本身。这个以后接触到Spring框架后才有深刻的理解。

而session对象是在服务器第一次调用getSession方法时创建，服务器为每个客户端都维护一个session对象，因此某客户端第一次访问应用时，通过getSession拿不到session对象就会创建属于该客户端的session对象（访问JSP时默认就会调用getSession，因为JSP的page指令中，session属性值默认为true，即默认就会获得session）。session对象何时被销毁呢？session对象将在session过期后销毁（Tomcat默认过期时间是30分钟，可通过配置文件配置），当然调用session.invalidate()方法将主动销毁session对象。

需要注意的是，浏览器关闭时，session对象并未被销毁，因为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，浏览器中的SESSIONID这个Cookie过期失效了，因此导致服务器无法拿到之前的session对象，但这只说明之前的session无法再被使用（即失效了），但该对象并不是在此时被销毁，而是等到该session过期后（例如默认的30分钟）被销毁，此时，服务器会为该客户端新创建一个session对象，即此时会话会使用新的session对象，之前的session对象由服务器自行负责销毁。

根据session的这个特点，可使用HttpSessionListener监听器来粗略统计网站的访问量，一种一个session对象就代表着一个客户端。

### 3.1.2 属性监听器

属性监听器用于监听servletContext、session和request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。它们对应的监听器接口分别是ServletContextAttributeListener、HttpSessionAttributeListener和ServletRequestAttributeListener。

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。

属性监听器的使用方法和上述生命周期监听器的使用方法类似。

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 |

### 3.1.3 session相关的感知型监听器

Servlet还提供了Session相关的感知型监听器：

（1）HttpSessionBidingListener：实现该监听器接口的类的实例，能够感知自己何时被加入了session域对象中，和何时被session域对象删除。即能监听本对象被绑定（或解绑）到session上的事件。

（2）HttpSessionActivationListener：实现该接口的类的实例，能够感知自己何时随着session对象被钝化和激活。要求该类还需要实现序列化接口（这是session钝化和激活要求的）。

注意的是，感知型监听器类是不需要进行注册的（即不需要通过web.xml或者@WebListener注册），自动能“感知”到自己的事件，即非服务器对象无需注册“监听”。

例1：监听Employee类被session绑定/解绑事件。

只要自定义类（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 { **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。

例2：演示Student类被钝化和激活的。

类只要实现序列化和HttpSessionActivationListener接口即可：

|  |
| --- |
| **package** br.bean;  **import** javax.servlet.http.HttpSessionActivationListener; **import** javax.servlet.http.HttpSessionEvent; **import** java.io.Serializable;  **public class** Student **implements** Serializable, HttpSessionActivationListener {  @Override  **public void** sessionWillPassivate(HttpSessionEvent hse) {  *// 钝化的事件* System.***out***.println(**"Student对象被钝化了"**);  }   @Override  **public void** sessionDidActivate(HttpSessionEvent hse) {  *// 激活的事件* System.***out***.println(**"Student对象被激活了"**);  } } |

然后使用在session中使用一下Student类，以便演示钝化和激活：

|  |
| --- |
| <%@ **page import**="**br.bean.Student**" %> <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> **<%** session.setAttribute(**"name"**, **new** Student()); **%>** </**body**> </**html**> |

此时，使用shutdown命令正常关闭Tomcat，则触发钝化事件，随后，重新启动Tomcat，则会触发激活事件。

# 附：MD5和SHA的加密

开发中常会用到加密技术，保证数据的安全性。MD5和SHA可以用来对任何数据进行签名，生成一个独一无二的字符串（很难重复）。MD5和SHA一般用来校验资源的正确性和加密敏感数据。

MD5和SHA算法能签名任何资源（说白了就是对二进制数据签名），签名则会生成固定长度的字节数据（Java中表现为字节数组）。其中MD5签名会生成16个字节的数据，SHA中分为SHA-1（就是平时说的SHA）、SHA256和SHA512，SHA-1签名生成20个字节的数据，而SHA-256和SHA-512签名分别生成32和64字节的数据。

在显示某资源的MD5或SHA值时，一般将MD5或SHA等数据以字节为单位表示成十六进制字符串，这样便于识别。下面就以“显示字符串123的MD5值”为例说明表示的过程（了解此过程便于理解后面的Java代码）：

首先，字符串“123”进行MD5签名后，产生的字节数组是（每个字节表示成一个十进制数据，一个字节8位，则无符号的数据范围是0~255，共有16个这样的数。至于是如何产生这些数据的，这是较低层了，目前不关注。）：

|  |
| --- |
| 32, 44, 185, 98, 172, 89, 7, 91, 150, 75, 7, 21, 45, 35, 75, 112 |

对应的十六进制是：

|  |
| --- |
| 20, 2c, b9, 62, ac, 59, 7, 5b, 96, 4b, 7, 15, 2d, 23, 4b, 70 |

规定每个字节要用两个十六进制数表示，则十六进制不满1位的（即十进制小于16的）应该在前面添一个人0，这样将每个字节的数据拼接到一起，就形成了长度为32的MD5字符串值，即表示成：

|  |
| --- |
| 202cb962ac59075b964b07152d234b70 |

因此，“202cb962ac59075b964b07152d234b70”就是字符串资源“123”的MD5值。平时我们见到的MD5或SHA值都是以这样的十六进制字符串表示的。对于SHA值的表示，也是这样的过程，只是SHA-1会表示成长度为40的字符串，SHA-256和SHA-512分别会表示成长度为64和128的字符串。毫无疑问，SHA-512的安全性是它们中最高的。

MD5和SHA虽可加密，但不可解密，即它们是不可逆的。通俗地进行理解很简单，以MD5为例，由于任何资源都可生成MD5值，而资源是无穷的，但是区区32位的MD5肯定是又穷的（可根据排列组合计算），因此他们必然不是一一对应的关系，因此无法解密，一个MD5值可能对应着无数的资源。虽然这么说，但由于出现相同MD5的几率还是太小了，因此一般考虑他们是唯一的。如果担心MD5不能满足要求，还可考虑使用SHA-256、SHA-512等。有人说网上提供了MD5等的解密，但其实它们不是真正地解密，而是基于“字典”的解密。比如我知道了字符串123的MD5值是“202cb962ac59075b964b07152d234b70”，那么我把它们的关系存起来，下次如果有人解密“202cb962ac59075b964b07152d234b70”这个MD5值，就返回字符串123这个结果。

MD5和SHA一般有两种应用：

（1）利用不可逆性，把用户的密码以MD5等密文形式存储在数据库中。这样即使数据库被攻破，也不会泄露用户原始密码。这就是为什么你忘记密码时，银行只让你重新设置密码，而无法告知你原来的密码，因为银行也不知道。

（2）用于校验文件。如果两个文件的MD5值一样，可认为它们是同一份文件，否则它们不是同一份文件，有可能被篡改。例如很多官方网站提供文件下载时，常说明其MD5值是什么，这个就是用于校验文件的。如果你下载好的文件的MD5值与官网提供的MD5值相同，则认为是同一个文件。同理，你也可理解为什么一些网盘有“秒传”的功能，就是你有时上传一个很大的文件，几秒钟就上传好了。因为你上传的这个文件之前肯定有人上传过了，网盘服务器中会记录其MD5值，然后你再上传时，MD5值一样，就免去上传的步骤了，直接把这个资源也标记为属于你的。

那么，如何用Java进行MD5和SHA的加密呢？利用MessageDigest类可实现该功能，例如：

|  |
| --- |
| @Test **public void** testEnc() {  MessageDigest messageDigest = **null**;  **try** {  *// 使用指定的签名名称获得MessageDigest对象。不区分大小写* messageDigest = MessageDigest.*getInstance*(**"MD5"**);  } **catch** (NoSuchAlgorithmException e) {  **throw new** RuntimeException(**"不受支持的签名类型"**);  }  *// 针对资源进行加密，调用digest方法，其中传递字节数组（二进制资源）  // 方法返回的也是字节数组* **byte**[] encByte = messageDigest.digest(**"123"**.getBytes());  System.***out***.println(Arrays.*toString*(encByte)); } |

上述加密后我们得到了MD5的字节数组，是16个字节长度。得到的输出结果如下：

|  |
| --- |
| [32, 44, -71, 98, -84, 89, 7, 91, -106, 75, 7, 21, 45, 35, 75, 112] |

注意的是，其中有负数，这是因为Java中的byte类型是有符号的（-128~127），因此表示一个字节的数据时，会有负数。但Java中没有无符号数据类型，怎么把其中的负数转成正数处理呢？这时，需要把byte数据与“0xff”相与，因为0xff二进制表示都是1且0xff是int类型，因此相与的结果就是对应的int类型的正数（不能用Java的byte类型接收，因为byte类型总是有符号的。也不能强转成int类型，强转的话还是负数）。

接下来，按照我们之前的说法，就可以获得一个资源的十六进制MD5值，代码如下：

|  |
| --- |
| @Test **public void** shaTest() {  MessageDigest messageDigest = **null**;  **try** {  messageDigest = MessageDigest.*getInstance*(**"md5"**);  } **catch** (NoSuchAlgorithmException e) {  **throw new** RuntimeException(**"不受支持的签名类型"**);  }  **byte**[] encByte = messageDigest.digest(**"123"**.getBytes());  StringBuilder sb = **new** StringBuilder(); *// 用于拼接MD5字符串* **for** (**byte** b : encByte) {  **int** i = 0xff & b; *// 确保将所有的数据转换成正数  // 按照之前讲的，如果该字节数据（即i）小于16，即不足十六进制的1位，则先补一个0* **if** (i < 16) {  sb.append(0);  }  *// 然后将该字节数据转成16进制拼接到sb上（无论i是否小于16。i小于16只是补高位的0，即使i为0，也要把地位的0写上）* sb.append(Integer.*toHexString*(i));  }  *// 输出结果* System.***out***.println(sb.toString()); } |

以上就实现了把资源加密成MD5值。如果要想以SHA加密也很简单，只要把上面的getInstance("md5")换成getInstance("sha")即可。getInstance(String name)中的name支持的字符串有（不区分大小写）：

（1）MD5：表示MD5加密；

（2）SHA、SHA-1和SHA1：表示SHA-1加密；

（3）SHA-256：表示SHA-256加密；

（4）SHA-512：表示SHA-512加密。

（5）其他的比如SHA-224、SHA384等。

如果输入其他的字符，可能导致JDK不支持，则会抛出NoSuchAlgorithmException异常。

我们完全可以把加密方法进行抽取：

|  |
| --- |
| **package** com.company.util;  **import** java.security.MessageDigest; **import** java.security.NoSuchAlgorithmException;  **public class** MessageDigestUtils {  **private** MessageDigestUtils() {}   */\*\*  \* 加密。支持MD5和SHA  \** ***@param algorithm*** *加密类型，如MD5、SHA、SHA-256、SHA-512.  \** ***@param data*** *要加密的数据。是一个字节数组，不仅仅可加密字符串资源。  \** ***@param isUpperCase*** *是否将其中的十六进制符号转换成大写的。  \** ***@return*** *加密的字符串值  \*/* **public static** String encrypt(String algorithm, **byte**[] data, **boolean** isUpperCase) {  MessageDigest messageDigest = **null**;  StringBuilder sb = **new** StringBuilder();  **try** {  messageDigest = MessageDigest.*getInstance*(algorithm);  } **catch** (NoSuchAlgorithmException e) {  **throw new** RuntimeException(e);  }  **byte**[] encByte = messageDigest.digest(data);  **for** (**byte** b : encByte) {  **int** i = 0xff & b;  **if** (i < 16) {  sb.append(**"0"**);  }  sb.append(Integer.*toHexString*(i));  }  **return** isUpperCase ? sb.toString().toUpperCase() : sb.toString(); *// 这样写，因为默认十六进制就是小写的* } } |

使用示例：

|  |
| --- |
| **package** com.company;  **import** com.company.util.MessageDigestUtils; **import** org.junit.Test;  **import** java.io.ByteArrayOutputStream; **import** java.io.FileInputStream; **import** java.io.IOException;  **public class** AppTest {  *// 测试SHA - 字符串加密* @Test  **public void** testSHA() {  *// 对hello进行SHA加密，不大写十六进制* String sha1 = MessageDigestUtils.*encrypt*(**"sha-1"**, **"hello"**.getBytes(), **false**);  System.***out***.println(sha1);  }  *// 测试md5，计算文件的MD5* @Test  **public void** calcMd5() {  *// 读取文件* **try** {  FileInputStream fis = **new** FileInputStream(**"L:/nginx-1.12.1.zip"**);  ByteArrayOutputStream baos = **new** ByteArrayOutputStream(); *// 用于存放文件数据的字节数输出流* **byte**[] buff = **new byte**[1024 \* 10];  **int** length = 0;  **while** ( (length = fis.read(buff)) != -1 ) {  baos.write(buff, 0, length);  }  fis.close(); *// 关闭输入流  // 计算MD5，以大写形式返回* String md5 = MessageDigestUtils.*encrypt*(**"SHA1"**, baos.toByteArray(), **true**);  baos.close(); *// 关闭字节数组输出流* System.***out***.println(md5); *// 输出MD5值* } **catch** (IOException e) {  e.printStackTrace();  }  } } |

上面的代码示例中，分别对字符串资源和文件资源计算SHA-1值。但是上面的代码是有点小问题的，因为在计算文件的MD5等值时，我们是把整个文件的字节数据保存在内存中，然后由MessageDigestUtils去计算MD5。这样计算小文件没问题，如果文件较大，是不允许这样做的，内存承受不了，而且JVM会抛出“java.lang.OutOfMemoryError: Java heap space”异常。

为了解决这个问题，我们需要在使用MessageDigest时，调用update方法让底层自动进行分批处理。注意不要自行进行分批处理，那样你只会得到N个MD5等值。因此，我们针对大文件处理，在MessageDigestUtils中新增了一个encrypt重载，代码如下：

|  |
| --- |
| */\*\*  \* 加密。支持MD5和SHA  \** ***@param algorithm*** *加密类型，如MD5、SHA、SHA-256、SHA-512.  \** ***@param data*** *要加密的数据。是一个输入流，可用于大批量数据的计算MD5值。  \** ***@param isUpperCase*** *是否将其中的十六进制符号转换成大写的。  \** ***@return*** *加密的字符串值  \*/* **public static** String encrypt(String algorithm, InputStream data, **boolean** isUpperCase) {  MessageDigest messageDigest = **null**;  StringBuilder sb = **new** StringBuilder();  **try** {  messageDigest = MessageDigest.*getInstance*(algorithm);  } **catch** (NoSuchAlgorithmException e) {  **throw new** RuntimeException(e);  }  *// 下面是分批处理的代码* **byte**[] buff = **new byte**[1024 \* 1024 \* 5];  **int** length = 0;  **try** {  **while** ( (length = data.read(buff)) != -1 ) {  messageDigest.update(buff, 0, length); *// 多次调用update方法，没调用一次说明进行分批处理数据，最后由底层自动计算整体的MD5等值* }  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  **byte**[] encByte = messageDigest.digest(); *// 返回资源最终的MD5值。由于数据通过update传递进来了，因此这里digest方法不传递任何参数，得到加密的数据。  // 下面就是转换为十六进制字符串* **for** (**byte** b : encByte) {  **int** i = 0xff & b;  **if** (i < 16) {  sb.append(**"0"**);  }  sb.append(Integer.*toHexString*(i));  }  **return** isUpperCase ? sb.toString().toUpperCase() : sb.toString(); *// 这样写，因为默认十六进制就是小写的* } |

因此我们这时再利用MessageDigestUtils类来获得大文件的MD5值时，只要这样写即可：

|  |
| --- |
| @Test **public void** calcMd5() {  **try** {  FileInputStream fis = **new** FileInputStream(**"L:/服务器/ubuntu-16.04.1-server-amd64.iso"**);  *// 计算MD5。直接传递fis对象即可。* String md5 = MessageDigestUtils.*encrypt*(**"MD5"**, fis, **false**);  System.***out***.println(md5); *// 输出MD5值* fis.close(); *// 关闭fis* } **catch** (IOException e) {  e.printStackTrace();  } } |