详解 **Filter** 过滤器

文章首发在CSDN博客，转载请务必注明以下所有链接，否则考虑法律追究责任。

CSDN地址：http://blog.csdn.net/tzs_1041218129/article/details/53345485

个人博客地址：www.54tianzhisheng.cn/Blog/html/filter.html （阅读效果最好）

pdf 版本：www.54tianzhisheng.cn/Blog/pdf/filter.pdf

所有文章干货都在：https://github.com/zhisheng17/zhisheng17.github.io

更多精彩博客还请关注我的微信公众号：猿**blog**

1、简介

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。

它主要用于对用户请求进行预处理，也可以对HttpServletResponse 进行后处理。使用Filter 的完整流程：Filter对用户请求进行预处理，接着将请求交给Servlet 进行处理并生成响应，最后Filter 再对服务器响应进行后处理。

Filter功能：

- 在HttpServletRequest 到达 Servlet 之前，拦截客户的 HttpServletRequest 。 根据需要检查HttpServletRequest ， 也可以修改HttpServletRequest 头和数据。
- 在HttpServletResponse 到达客户端之前，拦截HttpServletResponse 。 根据需要检查 HttpServletResponse ，也可以修改HttpServletResponse头和数据。

2、如何实现拦截

Filter接口中有一个doFilter方法，当开发人员编写好Filter，并配置对哪个web资源进行拦截后，WEB服务器每次在调用web资源的service方法之前，都会先调用一下filter的doFilter方法，因此，在该方法内编写代码可达到如下目的：

1. 调用目标资源之前，让一段代码执行。
2. 是否调用目标资源（即是否让用户访问web资源）。

web服务器在调用doFilter方法时，会传递一个filterChain对象进来，**filterChain**对象是**filter**接口中最重要的一个对象，它也提供了一个doFilter方法，开发人员可以根据需求决定是否调用此方法，调用该方法，则web服务器就会调用web资源的service方法，即web资源就会被访问，否则web资源不会被访问。

3、Filter开发两步走

1. 编写java类实现Filter接口，并实现其doFilter方法。
2. 在 web.xml 文件中使用和元素对编写的filter类进行注册，并设置它所能拦截的资源。

web.xml配置各节点介绍：

```
<filter-name>用于为过滤器指定一个名字，该元素的内容不能为空。
<filter-class>元素用于指定过滤器的完整的限定类名。
<init-param>元素用于为过滤器指定初始化参数，它的子元素<param-name>指定参数的名字，<param-value>指定参数的值。
在过滤器中，可以使用FilterConfig接口对象来访问初始化参数。

<filter-mapping>元素用于设置一个 Filter 所负责拦截的资源。一个Filter拦截的资源可通过两种方式来指定：Servlet 名称和资源访问的请求路径
<filter-name>子元素用于设置filter的注册名称。该值必须是在<filter>元素中声明过的过滤器的名字
<url-pattern>设置 filter 所拦截的请求路径(过滤器关联的URL样式)
<servlet-name>指定过滤器所拦截的Servlet名称。
<dispatcher>指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST。用户可以设置多个<dispatcher> 子元素用来指定 Filter 对资源的多种调用方式进行拦截。

<dispatcher> 子元素可以设置的值及其意义：
REQUEST：当用户直接访问页面时，Web容器将会调用过滤器。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器就不会被调用。
INCLUDE：如果目标资源是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用。除此之外，该过滤器不会被调用。
FORWARD：如果目标资源是通过RequestDispatcher的forward()方法访问时，那么该过滤器将被调用，除此之外，该过滤器不会被调用。
ERROR：如果目标资源是通过声明式异常处理机制调用时，那么该过滤器将被调用。除此之外，过滤器不会被调用。
```

4、Filter链

在一个web应用中，可以开发编写多个Filter，这些Filter组合起来称之为一个Filter链。

web服务器根据**Filter**在**web.xml**文件中的注册顺序，决定先调用哪个**Filter**，当第一个Filter的doFilter方法被调用时，web服务器会创建一个代表Filter链的FilterChain对象传递给该方法。在doFilter方法中，开发人员如果调用了FilterChain对象的doFilter方法，则web服务器会检查FilterChain对象中是否还有filter，如果有，则调用第2个filter，如果没有，则调用目标资源。

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：

```xml
<filter>
    <filter-name>myFilter1</filter-name>
    <filter-class>cn.cloud.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. cloud.filter.MyFilter2</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>myFilter2</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>
```

`MyFilter1`

```java
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...");
    }
}
```

`MyFilter2`

```java
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...");
    }
}
```

```xml
<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...
```

5、Filter的生命周期

```
public void init(FilterConfig filterConfig) throws ServletException;//初始化
```

和我们编写的Servlet程序一样，Filter的创建和销毁由WEB服务器负责。 web 应用程序启动时，web 服务器将创建Filter 的实例对象，并调用其init方法，读取web.xml配置，完成对象的初始化功能，从而为后续的用户请求作好拦截的准备工作（**filter**对象只会创建一次，**init**方法也只会执行一次）。开发人员通过init方法的参数，可获得代表当前filter配置信息的FilterConfig对象。

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException;//拦截请求
```

这个方法完成实际的过滤操作。当客户请求访问与过滤器关联的**URL**（目标资源）的时候，Servlet过滤器将先执行doFilter方法。FilterChain参数用于访问后续过滤器。

```
public void destroy();//销毁
```

服务器在创建Filter对象之后，把Filter放到缓存中一直使用（会驻留在内存），通常不会销毁它，当web应用移除或服务器停止时才销毁Filter对象。在Web容器卸载 Filter 对象之前被调用。该方法在Filter的生命周期中仅执行一次。在这个方法中，可以释放过滤器使用的资源。

6、FilterConfig接口

用户在配置filter时，可以使用为filter配置一些初始化参数，当web容器实例化Filter对象，调用其init方法时，会把封装了filter初始化参数的filterConfig对象传递进来。因此开发人员在编写filter时，通过filterConfig对象的方法，就可获得以下内容：

```
String getFilterName();//得到filter的名称；与<filter-name>元素对应。
String getInitParameter(String name);//返回在部署描述中指定名称的初始化参数的值。如果不存在返回
null，与<init-param>元素对应.
Enumeration getInitParameterNames();//返回过滤器的所有初始化参数的名字的枚举集合。
public ServletContext getServletContext();//返回Servlet上下文对象的引用。
```

7、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)`

doFilter() 方法的放行，让请求流访问目标资源！其实调用该方法的意思是，当前 Filter 放行了，但不代表其他过滤器也放行。一个目标资源上，可能部署了多个过滤器，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。

8、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：

```
<filter-mapping>
       <filter-name>myfilter</filter-name>
       <url-pattern>/*</url-pattern>
    </filter-mapping>
```

特性与Servlet完全相同！通过这一特性，可以在用户访问敏感资源时，执行过滤器，例如：/admin/*，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。

还可以为指定目标资源为某个Servlet，例如：

```
<servlet>
       <servlet-name>myservlet</servlet-name>
       <servlet-class>cn.cloud.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.cloud.filter.MyFilter</filter-class>
    </filter>
    <filter-mapping>
       <filter-name>myfilter</filter-name>
       <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。

9、四种拦截方式

写一个过滤器，指定过滤的资源为b.jsp，然后在浏览器中直接访问b.jsp，会发现过滤器执行了.但是，当在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！

```java
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);
    }
}
```

```xml
<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>
```

```xml
<body>
   <h1>b.jsp</h1>
  </body>
```

```xml
<h1>a.jsp</h1>
    <%
    request.getRequestDispatcher("/b.jsp").forward(request, response);
    %>
  </body>
```

在浏览器输入：

`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！

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、标签都是转发访问
> - INCLUDE：包含访问执行过滤器。包括RequestDispatcher#include()方法、标签都是包含访问
> - ERROR：当目标资源在web.xml中配置为中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。

可以在中添加0~n个子元素，来说明当前访问的拦截方式。

如：

```
 <filter-mapping>
       <filter-name>myfilter</filter-name>
       <url-pattern>/b.jsp</url-pattern>
       <dispatcher>REQUEST</dispatcher>
       <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
```

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，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>
       <dispatcher>ERROR</dispatcher>
    </filter-mapping>
    <error-page>
       <error-code>500</error-code>
       <location>/b.jsp</location>
    </error-page>
```

```
<body>
  <h1>a.jsp</h1>
   <%
   if(true)
   throw new RuntimeException("嘻嘻~");
   %>
  </body>
```

10、Filter使用案例

1、使用Filter验证用户登录安全控制

前段时间参与维护一个项目，用户退出系统后，再去地址栏访问历史，根据url，仍然能够进入系统响应页面。我去检查一下发现对请求未进行过滤验证用户登录。添加一个filter搞定问题！

先在web.xml配置

```
<filter>
    <filter-name>SessionFilter</filter-name>
    <filter-class>com.action.login.SessionFilter</filter-class>
    <init-param>
        <param-name>logonStrings</param-name><!-- 对登录页面不进行过滤 -->
        <param-value>/project/index.jsp;login.do</param-value>
    </init-param>
    <init-param>
        <param-name>includeStrings</param-name><!-- 只对指定过滤参数后缀进行过滤 -->
        <param-value>.do;.jsp</param-value>
    </init-param>
    <init-param>
        <param-name>redirectPath</param-name><!-- 未通过跳转到登录界面 -->
        <param-value>/index.jsp</param-value>
    </init-param>
    <init-param>
        <param-name>disabletestfilter</param-name><!-- Y:过滤无效 -->
        <param-value>N</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>SessionFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

接着编写FilterServlet：

```java
package com.action.login;

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 javax.servlet.http.HttpServletResponseWrapper;

/**
 *    判断用户是否登录,未登录则退出系统
 */
public class SessionFilter implements Filter {

    public FilterConfig config;

    public void destroy() {
        this.config = null;
    }

    public static boolean isContains(String container, String[] regx) {
        boolean result = false;

        for (int i = 0; i < regx.length; i++) {
            if (container.indexOf(regx[i]) != -1) {
                return true;
            }
        }
        return result;
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest hrequest = (HttpServletRequest)request;
        HttpServletResponseWrapper wrapper = new HttpServletResponseWrapper((HttpServletResponse) response);

        String logonStrings = config.getInitParameter("logonStrings");        // 登录登陆页面
        String includeStrings = config.getInitParameter("includeStrings");    // 过滤资源后缀参数
        String redirectPath = hrequest.getContextPath() + config.getInitParameter("redirectPath");// 没有登陆转向页面
        String disabletestfilter = config.getInitParameter("disabletestfilter");// 过滤器是否有效

        if (disabletestfilter.toUpperCase().equals("Y")) {    // 过滤无效
            chain.doFilter(request, response);
```

```
            return;
        }
        String[] logonList = logonStrings.split(";");
        String[] includeList = includeStrings.split(";");

        if (!this.isContains(hrequest.getRequestURI(), includeList)) {// 只对指定过滤参数后缀进行过滤
            chain.doFilter(request, response);
            return;
        }

        if (this.isContains(hrequest.getRequestURI(), logonList)) {// 对登录页面不进行过滤
            chain.doFilter(request, response);
            return;
        }

        String user = ( String ) hrequest.getSession().getAttribute("useronly");//判断用户是否登录
        if (user == null) {
            wrapper.sendRedirect(redirectPath);
            return;
        }else {
            chain.doFilter(request, response);
            return;
        }
    }

    public void init(FilterConfig filterConfig) throws ServletException {
        config = filterConfig;
    }
}
```

这样既可完成对用户所有请求，均要经过这个Filter进行验证用户登录。

2、防止中文乱码过滤器

项目使用spring框架时。当前台JSP页面和JAVA代码中使用了不同的字符集进行编码的时候就会出现表单提交的数据或者上传/下载中文名称文件出现乱码的问题，那就可以使用这个过滤器。

```
<filter>
    <filter-name>encoding</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name><!--用来指定一个具体的字符集-->
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name><!--true：无论request是否指定了字符集，都是用encoding；false：如果request已指定一个字符集，则不使用encoding-->
        <param-value>false</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

3、Spring+Hibernate的OpenSessionInViewFilter控制session的开关

当 hibernate+spring 配合使用的时候，如果设置了lazy=true（延迟加载）,那么在读取数据的时候，当读取了父数据后，hibernate 会自动关闭 session，这样，当要使用与之关联数据、子数据的时候，系统会抛出lazyinit的错误，这时就需要使用 spring 提供的 OpenSessionInViewFilter 过滤器。

OpenSessionInViewFilter主要是保持 Session 状态直到 request 将全部页面发送到客户端，直到请求结束后才关闭 session，这样就可以解决延迟加载带来的问题。

注意：OpenSessionInViewFilter 配置要写在struts2的配置前面。因为 tomcat 容器在加载过滤器的时候是按照顺序加载的，如果配置文件先写的是 struts2 的过滤器配置，然后才是 OpenSessionInViewFilter 过滤器配置，所以加载的顺序导致，action 在获得数据的时候 session 并没有被 spring 管理。

```
<!-- lazy loading enabled in spring -->
<filter>
    <filter-name>OpenSessionInViewFilter</filter-name>
    <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
    <init-param>
        <param-name>sessionFactoryBeanName</param-name><!-- 可缺省。默认是从spring容器中找id为sessionFactory的bean，如果id不为sessionFactory，则需要配置如下，此处SessionFactory为spring容器中的bean。 -->
        <param-value>sessionFactory</param-value>
    </init-param>
    <init-param>
        <param-name>singleSession</param-name><!-- singleSession默认为true,若设为false则等于没用OpenSessionInView -->
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>OpenSessionInViewFilter</filter-name>
    <url-pattern>*.do</url-pattern>
</filter-mapping>
```

4、Struts2的web.xml配置

项目中使用Struts2同样需要在web.xml配置过滤器，用来截取请求，转到Struts2的Action进行处理。

注意：如果在2.1.3以前的Struts2版本，过滤器使用`org.apache.struts2.dispatcher.FilterDispatcher`。否则使用`org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter`。从Struts2.1.3开始，将废弃`ActionContextCleanUp`过滤器，而在`StrutsPrepareAndExecuteFilter`过滤器中包含相应的功能。

三个初始化参数配置：

1. config参数：指定要加载的配置文件。逗号分割。
2. actionPackages参数：指定Action类所在的包空间。逗号分割。
3. configProviders参数：自定义配置文件提供者，需要实现ConfigurationProvider接口类。逗号分割。

```
<!-- struts 2.x filter -->
<filter>
    <filter-name>struts2</filter-name>
    <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>*.do</url-pattern>
</filter-mapping>
```