

day33-filter_listener

学习目标

1. 能够说出过滤器的作用
2. 能够编写过滤器
3. 能够说出过滤器生命周期相关方法
4. 能够根据过滤路径判断指定的过滤器是否起作用
5. 能够说出什么是过滤器链
6. 能够编写过滤器解决全局乱码

案例一:统一全网站中文乱码的处理

一,需求分析

在整个网站中,可能会有get请求或post请求向服务器提交参数.参数中往往有中文信息.在后台每个Servlet中都需要去处理乱码.

我们想做的是: 请求到达Servlet中.就可以直接调用getParameter方法获得请求参数,请求参数已经没有乱码了.

二,技术分析

1.Filter概述

1.1什么是filter

Filter: 一个实现了特殊接口(Filter)的Java类.实现对请求资源(jsp,servlet,html,)的过滤的功能.

过滤器是一个运行在服务器的程序,优先于请求资源(Servlet或者jsp,html)之前执行. 过滤器是javaweb技术中最实用的技术.

1.2过滤器的作用

对目标资源(Servlet,jsp)进行过滤.

应用场景:登录权限检查,解决网站乱码,过滤敏感字符 ...

2. filter入门

2.1 通过xml配置方式

- 创建一个类实现Filter接口
- 在web.xml对过滤器进行配置

```

/**
 * - 创建一个类实现Filter接口
 * - 在web.xml对过滤器进行配置
 */
public class FilterDemo01 implements Filter {
    @Override
    //过滤的方法
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
        System.out.println("FilterDemo01收到了请求...");
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void destroy() {

    }
}

<!--注册Filter-->
<filter>
    <filter-name>FilterDemo01</filter-name>
    <filter-class>com.itheima.web.filter.FilterDemo01</filter-class>
</filter>
<!--配置Filter过滤路径-->
<filter-mapping>
    <filter-name>FilterDemo01</filter-name>
    <url-pattern>/demo01</url-pattern>
</filter-mapping>

```

2.2通过注解方式

- 创建一个类实现Filter接口
- 直接在这个类上面添加注解进行配置

```
@WebFilter("/demo02")
public class FilterDemo02 implements Filter{

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
        System.out.println("FilterDemo02... 收到了请求");
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void destroy() {

    }
}
```

三,思路分析

四,代码实现

- CodingFilter

```

@WebFilter("/*")
public class CodingFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
        FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;

        HttpServletRequest proxyRequest = (HttpServletRequest)
        Proxy.newProxyInstance(request.getClass().getClassLoader(),
            request.getClass().getInterfaces(), new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws
                Throwable {
                    if("getParameter".equals(method.getName())){
                        String methodStr = request.getMethod();
                        if("post".equalsIgnoreCase(methodStr)){
                            request.setCharacterEncoding("utf-8");
                            return method.invoke(request,args);
                        }
                    }

                    return method.invoke(request,args);
                }
            });

        filterChain.doFilter(proxyRequest,response);
    }

    @Override
    public void destroy() {

    }
}

```

五,总结

1.Filter的生命周期

1.1Filter生命周期

过滤器从创建到销毁的过程

1.2生命周期方法

init(FilterConfig):初始化

doFilter(ServletReqeust req,ServletResponse resp,FilterChain chain):执行过滤的方法

destroy():销毁

1.3Filter生命周期描述

1. 当服务器启动的时候，过滤器就被初始化了，执行过滤器的init方法
2. 每当一个请求的路径是满足过滤器的配置路径，那么就会执行一次过滤器的doFilter方法
3. 当服务器停止的时候，销毁过滤器，执行过滤器的destory方法

2.映射路径

假设有一个管理员权限的过滤器，它应该对用户发出的管理员功能的请求进行条件的过滤。但是当用户发出登录、注册等请求的时候，不应该进行过滤。所以我们过滤器，应该有选择的过滤器请求。这就需要学习配置过滤器不同的映射路径，从而让过滤器过滤希望过滤器的请求。

- 完全路径匹配, 以"/"开始

```
/demo01 ---> 过滤器只能拦截路径/demo01; /aa/a.jsp ---> 只能拦截 aa目录下的a.jsp
```

- 目录匹配:以"/"开始 以 *结束 .

```
/* --->当前项目下的所有的路径都可以拦截; /aa/* ---> 可以拦截 /aa/bb, /aa/bb/cc
```

- 扩展名匹配:以"*"开始 例如: *.jsp *.do

```
*.do--->可以拦截路径的后缀是 do的 ; *.jsp--->只能拦截JSP(所有的)
```

3.拦截方式

有了上面学习的映射路径，我们可以控制过滤器过滤指定的内容，但是我们在访问资源的时候，并不是每次都是之间访问，有时是以转发的方式访问的，这就需要我们让过滤器可以区分不同的访问资源的方式，有不同的拦截方式。是通过

DispatcherType 来指定的.

3.1DispatcherType.REQUEST

默认值,过滤从浏览器发送过来的请求和重定向 不过滤转发

3.2 DispatcherType.FORWARD

只过滤转发过来的请求

```

/*只拦截转发*/
/*@WebFilter(urlPatterns = "/demo05",dispatcherTypes={DispatcherType.FORWARD})*

/*转发,重定向,浏览器的直接请求都会拦截了*/
@WebFilter(urlPatterns = "/demo05",dispatcherTypes=
{DispatcherType.FORWARD,DispatcherType.REQUEST})
public class FilterDemo05 implements Filter {

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
ServletException, IOException {
        System.out.println("FilterDemo05() 收到了请求...");
        chain.doFilter(req, resp);
    }
    public void destroy() {

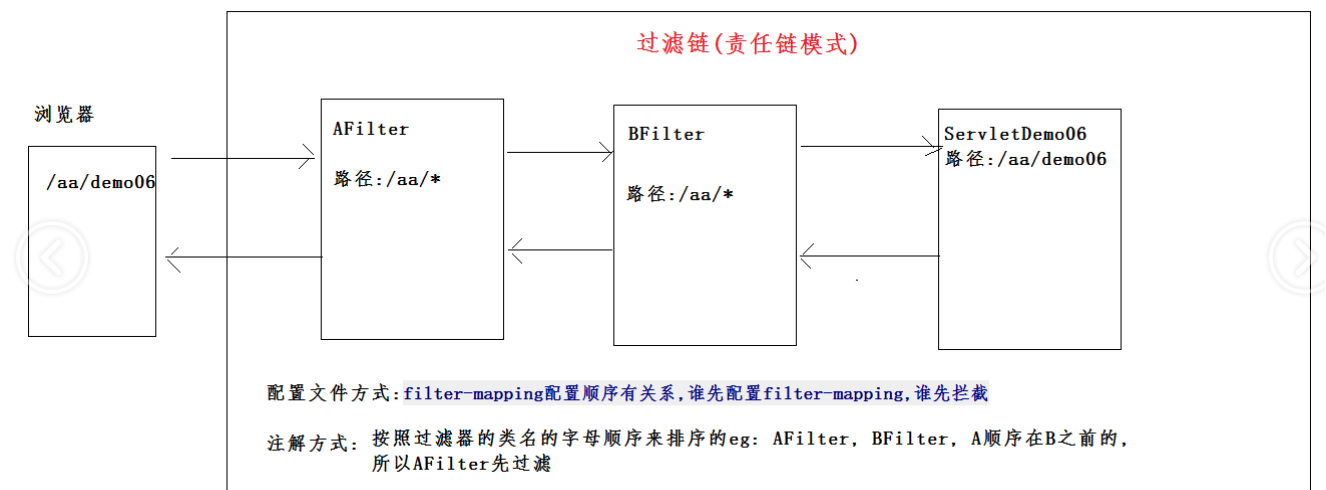
    }
    public void init(FilterConfig config) throws ServletException {

    }
}

```

4.过滤器链

过滤器链作用：当一个filter收到请求的时候,调用chain.doFilter才可以访问下一个匹配的filter,若当前的filter是最后一个filter,调用chain.doFilter才能访问目标资源



案例二:非法字符

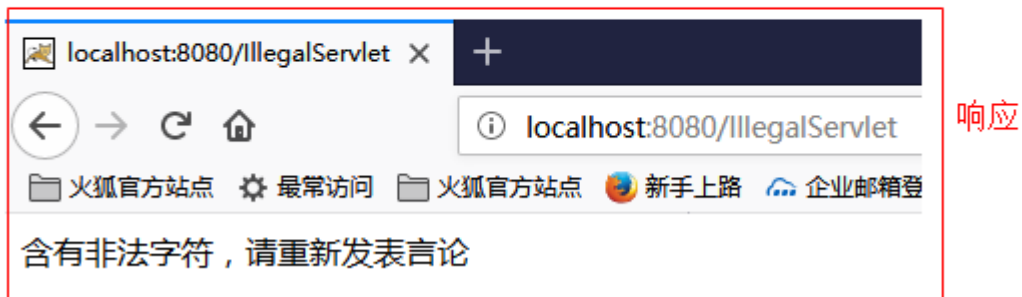
一,需求分析

当用户发出非法言论的时候,提示用户言论非法。

效果:

请发表你的言论：

页面输入



二,思路分析

- 1.创建一个表单用于发表言论。
- 2.创建一个txt文件，其中存入非法字符。
- 3.创建一个Filter，拦截请求。在init方法中将txt文件中的非法字符读取到内存中。
- 4.获取请求中的参数，对请求的参数进行非法字符的校验。
- 5.如果言论中不含有非法字符，就放行。
- 6.如果言论中含有非法字符，就拦截，并且提示用户非法言论。

三,代码实现

- IllegalFilter

```

@WebFilter(filterName = "IllegalFilter",urlPatterns = "/*")
public class IllegalFilter implements Filter {
    private List<String> illegalWords = new ArrayList<String>();

    public void init(FilterConfig config) throws ServletException {
        try {
            InputStream is = config.getServletContext().getResourceAsStream("/IllegalWords.txt");
            BufferedReader reader = new BufferedReader(new InputStreamReader(is,"utf-8"));
            String line = null;
            while((line = reader.readLine()) != null){
                illegalWords.add(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
    ServletException, IOException {
        //将ServletRequest和ServletResponse强转成http协议的请求和响应对象

        HttpServletRequest request = (HttpServletRequest)req;
        HttpServletResponse response = (HttpServletResponse)resp;
        response.setContentType("text/html;charset=utf-8");
        request.setCharacterEncoding("utf-8");
        String message = request.getParameter("message");
        for (String illegalWord : illegalWords) {
            if(message != null && message.trim().contains(illegalWord)){
                response.getWriter().write("含有非法字符, 请重新发表言论");
                return;
            }
        }

        chain.doFilter(request, response);
    }

    public void destroy() {
    }
}

```

案例三:监听ServletContext的创建和销毁

一,案例需求

监听ServletContext域对象的创建和销毁.

问题: ServletContext什么时候创建,什么时候销毁?

服务器启动的时候创建

服务器正常关闭的时候销毁

我这个案例表面上是监听ServletContext的创建和销毁, 实际上是监听服务器启动和关闭的

二, 技术分析

1. Listener的概述

1.1 什么是Listener

监听器就是一个Java类, 用来监听其他的JavaBean的变化

在javaweb中监听器就是监听三个域对象的状态的。request, session, servletContext(application)

1.2 监听器的应用

主要在Swing编程

在Android大量应用

JS里面的事件

1.3 监听器的术语

eg: 一个狗仔拍明星出轨

事件源: 被监听的对象.(目标对象)

明星

监听器对象: 监听的对象.

狗仔

事件: 事件源行为的称呼.

出轨

注册(绑定): 将"监听器"对象注册给"事件源".

狗仔和明星进行绑定(去楼下蹲点, 跟踪..)

事件对象: 在"监听器对象"中获得"事件源"

在狗仔的镜头下可以找到出轨证据

2. ServletContextListener

2.1 概述

ServletContextListener是监听ServletContext对象的创建和销毁的

问题: ServletContext对象何时创建和销毁:

创建: 服务器启动时候(前提是这个项目在这个服务器里面). 服务器为每个WEB应用创建一个单独的ServletContext.

销毁: 服务器关闭的时候, 或者项目从服务器中移除.

2.2 ServletContextListener的使用

2.2.1 编写步骤

1. 编写一个类, 实现ServletContextListener接口

2. 在web.xml里面注册注册listener

2.2.2 代码实现

- JAVA代码

```

public class ServletContextLis implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        //相当于服务器启动:加载配置 Spring
        System.out.println("ServletContext 创建了... 555555");
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("ServletContext 销毁了... 444444");
    }

}

```

- 配置(web.xml)

```

<listener>
    <!-- 全限定名(带包名) -->
    <listener-class>com.itheima.web.listener.ServletContextLis</listener-class>
</listener>

```

2.3 ServletContextListener 企业中应用

初始化工作.

加载配置文件:Spring框架,ContextLoaderListener

常见设计模式:

- 单例模式
- 装饰者
- 代理(动态代理)
- 工厂模式
- 构建模式
- 责任链
- 类模板模式(模板方法)
- 观察者