今日内容

1. Filter：过滤器
2. Listener：监听器

Filter：过滤器

概念：

- 生活中的过滤器：净水器,空气净化器，土匪、
- web中的过滤器：当访问服务器的资源时，过滤器可以将请求拦截下来，完成一些特殊的功能。
- 过滤器的作用：
    - 一般用于完成通用的操作。如：登录验证、统一编码处理、敏感字符过滤...

快速入门：

步骤：

1. 定义一个类，实现接口Filter（import javax.servlet.Filter）
2. 复写方法
3. 配置拦截路径
    1. web.xml

```
<filter>
    <filter-name>demo1</filter-name>
    <filter-class>cn.itcast.web.filter.FilterDemo1</filter-class>
</filter>

<filter-mapping>
    <filter-name>demo1</filter-name>
    <!-- 拦截路径 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

    2. 注解：`@WebFilter("/*")`, `@WebFilter("/*demo.jsp")`

代码：

```
@WebFilter("/*")//访问所有资源之前，都会执行该过滤器
public class FilterDemo1 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
```

```
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {

        System.out.println("filterDemo1被执行了....");
        //放行
        filterChain.doFilter(servletRequest,servletResponse);

    }

    @Override
    public void destroy() {

    }
}
```

过滤器细节：

web.xml配置

```
<filter>
    <filter-name>demo1</filter-name>
    <filter-class>cn.itcast.web.filter.FilterDemo1</filter-class>
</filter>

<filter-mapping>
    <filter-name>demo1</filter-name>
    <!-- 拦截路径 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器执行流程

1. 执行过滤器
2. 执行放行后的资源
3. 回来执行过滤器放行代码下边的代码

过滤器生命周期方法

1. init:在服务器启动后，会创建Filter对象，然后调用init方法。只执行一次。用于加载资源
2. doFilter:每一次请求被拦截资源时，会执行。执行多次
3. destroy:在服务器关闭后，Filter对象被销毁。如果服务器是正常关闭，则会执行destroy方法。只执行一次。用于释放资源

过滤器配置详解

- 拦截路径配置：

1. 具体资源路径： /index.jsp 只有访问index.jsp资源时，过滤器才会被执行
2. 拦截目录： /user/* 访问/user下的所有资源时，过滤器都会被执行
3. 后缀名拦截： *.jsp 访问所有后缀名为jsp资源时，过滤器都会被执行
4. 拦截所有资源：/* 访问所有资源时，过滤器都会被执行

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;

//@WebFilter("/*")
//@WebFilter("/user/updateServlet*")
//@WebFilter("*.jsp")
//@WebFilter("/index.jsp")
public class FilterDemo1 implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain chain) throws ServletException, IOException {


        chain.doFilter(req, resp);
    }

    public void init(FilterConfig config) throws ServletException {

    }

}
```

- 拦截方式配置：按照资源被访问的方式拦截，当资源从这种方式请求时会被拦截
  - 注解配置：
    - 设置dispatcherTypes属性，注意这是个数组，所以可以设置多个值，但其实这五个值本质上是枚举
      1. REQUEST：默认值。浏览器直接请求资源
      2. FORWARD：转发访问资源
      3. INCLUDE：包含访问资源
      4. ERROR：错误跳转资源
      5. ASYNC：异步访问资源

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;

//@WebFilter(value = "/*",dispatcherTypes = DispatcherType.ASYNC)
//@WebFilter(value = "/*",dispatcherTypes = DispatcherType.ERROR)
//@WebFilter(value = "/*",dispatcherTypes = DispatcherType.FORWARD)
//@WebFilter(value = "/*",dispatcherTypes = DispatcherType.INCLUDE)
```

```
@WebFilter(value = "/*",dispatcherTypes =
{DispatcherType.REQUEST,DispatcherType.ASYNC})


public class FilterDemo1 implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain chain) throws ServletException, IOException {


        chain.doFilter(req, resp);
    }

    public void init(FilterConfig config) throws ServletException {

    }

}
```

- web.xml配置
  - 设置<dispatcher></dispatcher>标签即可

```
<filter>
    <filter-name>demo1</filter-name>
    <filter-class>cn.itcast.web.filter.FilterDemo1</filter-class>
</filter>

<filter-mapping>
    <filter-name>demo1</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

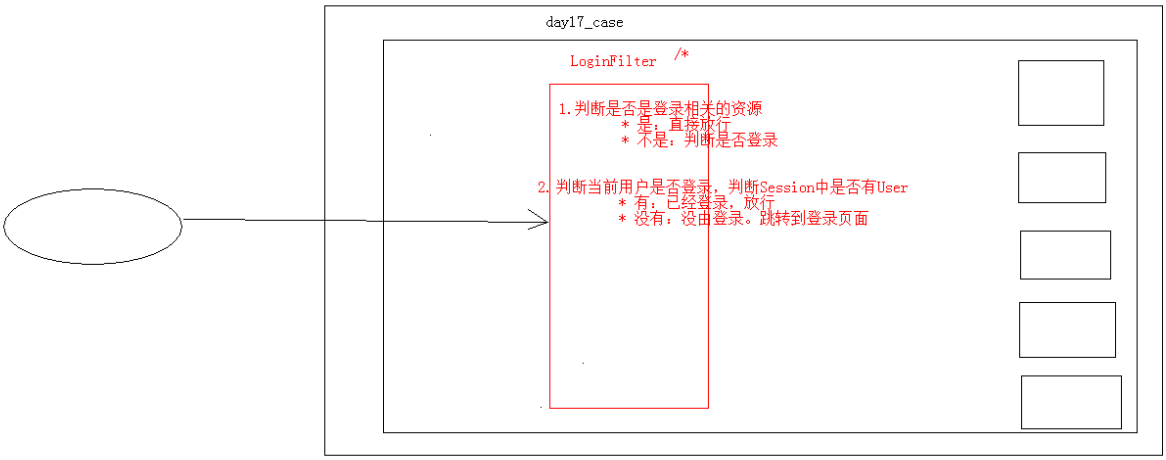
过滤器链(配置多个过滤器)

- 过滤器的访问和返回的执行顺序：如果有两个过滤器：过滤器1和过滤器2
  - 那么执行的顺序为：
    - 访问资源：过滤器1-->过滤器2-->访问到了资源
    - 从资源处返回：过滤器2<--过滤器1<--资源
- 过滤器执行的先后顺序问题：
  1. 注解配置：按照类名的字符串比较规则比较，值小的先执行
     - 如： AFilterDemo1 和 BFilterDemo2，AFilter就先执行了。
  2. web.xml配置： <filter-mapping>谁定义在上边，谁先执行

案例：

案例1_登录验证

- 需求：
    1. 访问day17_case案例的资源。验证其是否登录
    2. 如果登录了，则直接放行。
    3. 如果没有登录，则跳转到登录页面，提示"您尚未登录，请先登录"。



> *注意，在登陆的时候除了login.jsp和loginServlet这些资源之外，还有验证码，css，js这些资源也要放行，否则就会出现布局错误，验证码刷不出来的情况*

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import java.io.IOException;



@WebFilter(value = "/*")
public class FilterDemo1 implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        /**
         * 业务分析：
         *      目的：判断是否登录，如果没有登录则转发到登录界面去
         *      问题一：不能把所有的请求界面都弄到登录界面去，假如用户本来就需要登录那就不必
到登录界面去了，否则会出现一个死循环的问题
```

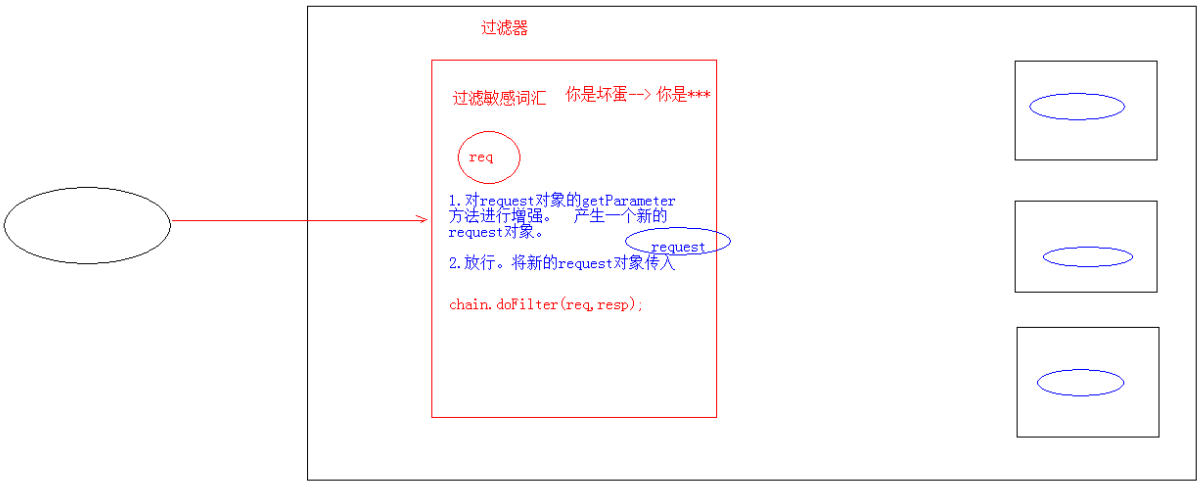
```
         *            解决办法：使用HttpServletRequest来获取请求资源路径，判断是不是从登录界面出来的
         *          问题二：除了login.jsp和loginServlet之外，还有css、js、font、image、验证码等数据也要被放行
         *          问题三：这里是ServletRequest，我们需要的是HttpServletRequest
         *              解决办法：强制转换
         */
        HttpServletRequest request = (HttpServletRequest) req;

        String requestURI = request.getRequestURI();


 if(requestURI.contains("/login.jsp")||requestURI.contains("/loginServlet")||requestURI.contains("/checkCodeServlet")
 ||requestURI.contains("/css/")||requestURI.contains("/js/")||requestURI.contains("/font")||requestURI.contains("/img/")){


            /*这里表明了用户就是想要登录，放行*/

            chain.doFilter(req,resp);
        }else {

            /*表明了用户要访问其他的界面，判断是否有登陆的信息，
            这个可以利用Session来进行处理，如果登陆了那么就会有一个Session，如果没有登录那么就没有sessio信息*/
            HttpSession session = request.getSession();
            Object user = session.getAttribute("user");

            /*如果Session!=null说明了登陆了，假如session为空说明没有登录*/
            if (user!=null){
                chain.doFilter(req,resp);
            }else{

 request.getRequestDispatcher("/login.jsp").forward(request,resp);
            }
        }
    }

    public void init(FilterConfig config) throws ServletException {

    }

}
```

案例2_敏感词汇过滤

- 需求：

1. 对day17_case案例录入的数据进行敏感词汇过滤
2. 敏感词汇参考《敏感词汇.txt》
3. 如果是敏感词汇，替换为 ***

- 分析：
  1. 对request对象进行增强。增强获取参数相关方法
  2. 放行。传递代理对象



> *在实现案例之前要讲解一下如何对对象功能进行增强*

增强对象的功能：

- 设计模式：一些通用的解决固定问题的方式，一共有23种设计模式，现在来讲解1种,其实有两种设计模式可以进行增强：
  - 装饰模式
  - 代理模式，讲代理模式

代理模式

- 概念：
  1. 真实对象：被代理的对象
  2. 代理对象：代理真实对象
  3. 代理模式：代理对象代理真实对象，达到增强真实对象功能的目的
- 实现方式：
  1. 静态代理：有一个类文件描述代理模式
  2. 动态代理：在内存中形成代理类
     - 实现步骤：
       1. 代理对象和真实对象实现相同的接口
       2. 代理对象 = Proxy.newProxyInstance();
          - 三个参数：
            - 类加载器：真实对象.getClass().getClassLoader()

- 接口数组：真实对象.getClass().getInterfaces()
  - 处理器：new InvocationHandler(){实现接口}
    - 接口的三个参数：
      -

3. 使用代理对象调用方法。

```
package proxy;

public interface Person {

    String eat(String name);
}
```

```
package proxy.Impl;

import proxy.Person;

public class PersonImpl implements Person {

    @Override
    public String eat(String name) {

        System.out.println("PersonImol_eat"+name+"...");

        return "eat";
    }
}
```

```
package proxy.proxy;

import proxy.Impl.PersonImpl;
import proxy.Person;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyTest {

    public static void main(String[] args) {

        PersonImpl person = new PersonImpl();

        /*动态代理增强方法Proxy.newProxyInstance()
        *    三个参数，固定写法：
        *        1，类加载器：真实对象.getClass().getClassLoader()
        *        2，接口数组：真实对象.getClass().getInterfaces()
```

```
         *       3，处理器：new InvocationHandler(){实现接口}
         * 其实如果返回的话是一个Object，但是要强制转换为Person
         * */

        Person person_proxy =
(Person)Proxy.newProxyInstance(person.getClass().getClassLoader(),
person.getClass().getInterfaces(), new InvocationHandler() {

            /*
            * 无论调用这个类下面的什么方法，都会调用一次invoke方法
            * 三个参数：
            *   1，Object proxy：就是proxy本体
            *   2. Method method：调用的什么方法
            *   3. Object[] args当方法调用的时候传进来的值，以数组形式呈现
            * */
            @Override
            public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {

                System.out.println(method.getName());//eat
                System.out.println(args[0]);//苹果

                return null;
            }
        });


        /*就像平常那样调用方法，但是是使用person1_proxy来调用
        * 但是有一点要注意，就是无论调用这个类下面的什么方法，都会调用一次invoke
方法
        String eat = person_proxy.eat("苹果");

    }

}
```

> *有一点需要注意，就是如果使用person_proxy来执行方法，那么PersonImpl的方法不会被执行，也就是说System.out.println("PersonImol_eat"+name+"...");不会被执行，那么就需要我们在invoke里面进行调用*

```
package proxy.proxy;

import proxy.Impl.PersonImpl;
import proxy.Person;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
```

```
public class ProxyTest {

    public static void main(String[] args) {

        PersonImpl person = new PersonImpl();

        Person person_proxy =
(Person)Proxy.newProxyInstance(person.getClass().getClassLoader(),
person.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {

                /*为了避免真实对象无法执行方法的缺点，在这里使用真实对象来调用方
法
                * 因为method就是真实对象，那么就使用method也就是真实对象来调
用方法
                *    两个参数:
                *        1.Object obj:             真实对象名
                *        2.Object[] ... args:      参数数组
                * */
                Object invoke = method.invoke(person,args);
                //PersonImol_eat苹果...

                /*然后把invoke返回*/
                return invoke;
            }
        });

        String eat = person_proxy.eat("苹果");

        System.out.println(eat);
        //eat，注意这个eat是调用PersonImpl.eat(String name)的返回值

    }

}
```

> *现在实现了代理调用和真是对象调用相同的效果，那么现在就看如何将方法进行增强，*
>
> *我们知道方法的三要素：方法参数，名称，返回值*
>
> *那么方法执行的三要素是：方法参数，返回值，具体的实现逻辑代码*
>
> *我们从方法执行的三要素进行处理*

4. 增强方法

增强方式：

- 增强参数列表

```
package proxy.proxy;

import proxy.Impl.PersonImpl;
import proxy.Person;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyTest {

    public static void main(String[] args) {

        PersonImpl person = new PersonImpl();

        Person person_proxy =
(Person)Proxy.newProxyInstance(person.getClass().getClassLoader(),
person.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

//                增强参数
                /*首先判断是否是有参数的方法，有就可以增强，没有就不必增强，在这里有参数的方
法是eat方法*/
                if (method.getName().equals("eat")){
                    /*获取*/
                    Double d = (Double)args[0];
                    /*增强*/
                    d= d*100;
                    /*传递不是传递args了，是传递增强后的结果*/
                    Object invoke = method.invoke(person,d);
                    return invoke;
                }else {
                    /*如果不是，该咋执行咋执行*/
                    Object invoke = method.invoke(person, args);
                    return invoke;
                }
            }
        });

        double live = person_proxy.live(5.0);

        System.out.println(live);

    }

}
```

- 增强返回值类型

```
package proxy.proxy;

import proxy.Impl.PersonImpl;
import proxy.Person;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyTest {

    public static void main(String[] args) {

        PersonImpl person = new PersonImpl();

        Person person_proxy =
(Person)Proxy.newProxyInstance(person.getClass().getClassLoader(),
person.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

//                增强返回值


                Double invoke = (Double)method.invoke(person,args);

                return invoke+1;
            }
        });

        Double live = person_proxy.live(5.0);

        System.out.println(live);

    }

}
```

- 增强方法体执行逻辑

```
package proxy.proxy;

import proxy.Impl.PersonImpl;
import proxy.Person;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
```

```
public class ProxyTest {

    public static void main(String[] args) {

        PersonImpl person = new PersonImpl();

        Person person_proxy =
(Person)Proxy.newProxyInstance(person.getClass().getClassLoader(),
person.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

//                在代码体中进行增强

                System.out.println("假装增强了。。。。");

                Double invoke = (Double)method.invoke(person,args);

                return invoke+1;
            }
        });

        Double live = person_proxy.live(5.0);

        System.out.println(live);

    }

}
```

实现案例二

敏感词汇:
笨蛋
坏蛋

```
package filter;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.List;
```

```
/*
* 敏感词汇过滤器
* */
@WebFilter("/*")
public class SensitiveWordsFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        /*创建代理对象，增强getParameter方法
        * 会返回一个ServletRequest
        * */
        ServletRequest proxy_req =
(ServletRequest)Proxy.newProxyInstance(req.getClass().getClassLoader(),
req.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
//                增强getParameter方法
                /*获取是否是getParameter方法*/
                if (method.getName().equals("getParameter")){
                    /*增强返回值*/
                    /*首先获取返回值*/
                    String value = (String)method.invoke(req,args);
                    /*然后找到敏感词汇，敏感词汇专门有一个配置文件，加载配置文件在init中加
载*/
                    if (value!=null){
                        for (String s : list) {
                            if (value.contains(s)){
                                value = value.replaceAll(s,"***");
                            }
                        }
                    }
                }

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


        chain.doFilter(req, resp);
    }


    /*命案词汇的List集合*/
    private List<String> list = new ArrayList<String>();
    public void init(FilterConfig config) throws ServletException {
```

```
//          加载文件

        try {
            /*获取文件的真实路径*/
            ServletContext servletContext = config.getServletContext();
            String realPath = servletContext.getRealPath("/WEB-INF/classes/敏感词
汇.txt");
            /*读取文件*/
            BufferedReader bufferedReader = new BufferedReader(new
FileReader(realPath));
            /*将文件的每一行都添加到list中*/
            String line = null;
            while ((line=bufferedReader.readLine())!=null){
                list.add(line);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Listener：监听器

概念：web的三大组件之一。

事件监听机制

- 事件：一件事情
- 事件源：事件发生的地方
- 监听器：一个对象
- 注册监听：将事件、事件源、监听器绑定在一起。 当事件源上发生某个事件后，执行监听器代码

ServletContextListener:监听ServletContext对象的创建和销毁

方法：

- void contextDestroyed(ServletContextEvent sce)： ServletContext对象被销毁之前会调用该方法
- void contextInitialized(ServletContextEvent sce)： ServletContext对象创建后会调用该方法

步骤：

1. 定义一个类，实现ServletContextListener接口
2. 复写方法，一般用于实现资源加载，资源卸载啥的

```java
package listener;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ContextLoaderListener  implements ServletContextListener{

    /*
    * 监听ServletContext而创建的，ServletContext对象服务器启动后自动创建
    * 在服务器启动之后会自动调用
    * */
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
//          加载资源文件

        /*获取ServletContext对象，使用这个来加载资源文件*/
        ServletContext servletContext = servletContextEvent.getServletContext();

        /*加载资源文件*/
        String initParameter =
servletContext.getInitParameter("contextConfigLocation");

        /*获取真实路径*/
        String realPath = servletContext.getRealPath(initParameter);

        /*加载进内存*/
        try {
            FileInputStream fileInputStream = new FileInputStream(realPath);

 System.out.println(fileInputStream);//java.io.FileInputStream@388e004b
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }


    /*
    * 服务器关闭之后，ServletContext对象被销毁，当服务器正常关闭之后该方法被调用
    * */
    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {

    }
}
```

3. 配置

1. web.xml

```
<?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">


    <!--配置监听器-->
    <listener>
        <listener-class>listener.ContextLoaderListener</listener-
class>
    </listener>

    <!--指定初始化参数，注意这个xml文件是在src下的，但是如果要放到服务器上就是--
>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/appContext.xml</param-value>
    </context-param>
</web-app>
```

- 指定初始化参数<context-param>

2. 注解：

- @WebListener

```
package listener;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
import java.io.FileInputStream;
import java.io.FileNotFoundException;




@WebListener
public class ContextLoaderListener  implements
ServletContextListener{

    /*
    * 监听ServletContext而创建的，ServletContext对象服务器启动后自动创
建
    * 在服务器启动之后会自动调用
    * */
    @Override
```

```
    public void contextInitialized(ServletContextEvent
servletContextEvent) {
//         加载资源文件


        System.out.println("创建成功");

        ServletContext servletContext =
servletContextEvent.getServletContext();

        String initParameter =
servletContext.getInitParameter("contextConfigLocation");

        String realPath =
servletContext.getRealPath(initParameter);

        try {
            FileInputStream fileInputStream = new
FileInputStream(realPath);
            System.out.println(fileInputStream);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }



    /*
    * 服务器关闭之后，ServletContext对象被销毁，当服务器正常关闭之后该方
法被调用
    * */
    @Override
    public void contextDestroyed(ServletContextEvent
servletContextEvent) {

    }
}
```

> *但是注意了，虽然这个方法不需要配置监听器，但是初始化参数还是要通过xml指定的*