

# 15-Filter 过滤器

讲师：王振国

## 今日任务

## 1、Filter 什么是过滤器

- 1、Filter 过滤器它是 JavaWeb 的三大组件之一。三大组件分别是：Servlet 程序、Listener 监听器、Filter 过滤器
- 2、Filter 过滤器它是 JavaEE 的规范。也就是接口
- 3、Filter 过滤器它的作用是：**拦截请求**，过滤响应。

拦截请求常见的应用场景有：

- 1、权限检查
- 2、日记操作
- 3、事务管理
- .....等等

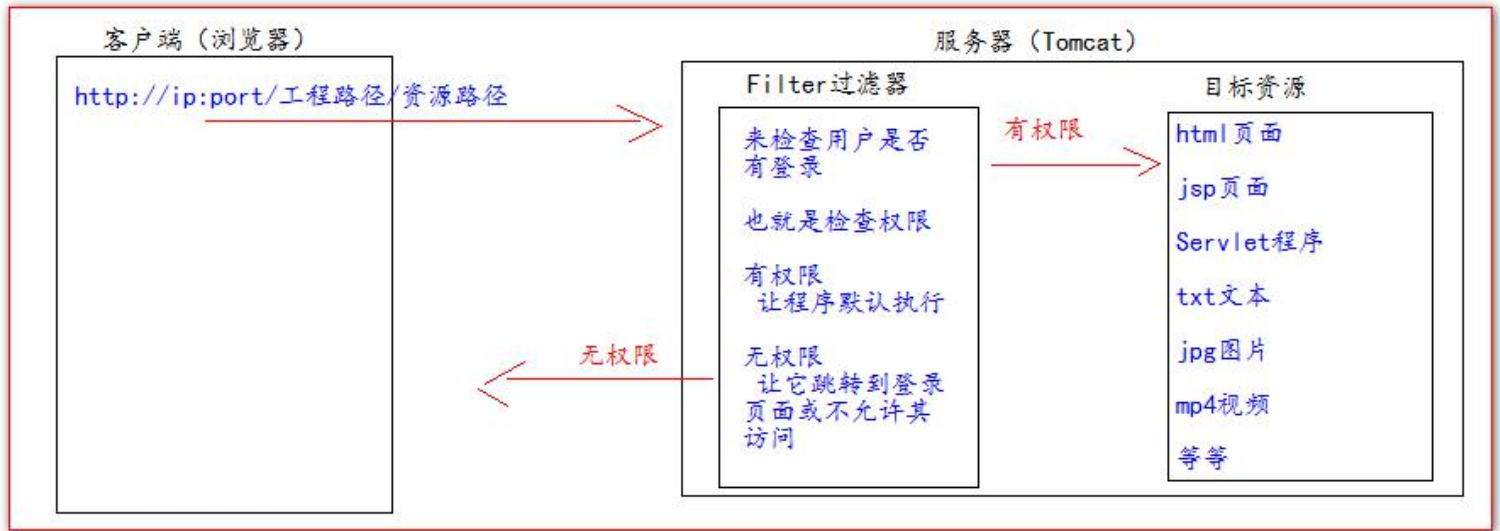
## 2、Filter 的初体验

要求：在你的 web 工程下，有一个 admin 目录。这个 admin 目录下的所有资源（html 页面、jpg 图片、jsp 文件、等等）都必须是在用户登录之后才允许访问。

思考：根据之前我们学过内容。我们知道，用户登录之后都会把用户登录的信息保存到 Session 域中。所以要检查用户是否登录，可以判断 Session 中是否包含有用户登录的信息即可！！！！

```
<%  
    Object user = session.getAttribute("user");  
    // 如果等于 null，说明还没有登录  
    if (user == null) {  
        request.getRequestDispatcher("/login.jsp").forward(request, response);  
        return;  
    }  
%>
```

Filter 的工作流程图：



Filter 的代码:

```
public class AdminFilter implements Filter {
    /**
     * doFilter 方法, 专门用于拦截请求。可以做权限检查
     */
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain) throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;

        HttpSession session = httpServletRequest.getSession();
        Object user = session.getAttribute("user");
        // 如果等于 null, 说明还没有登录
        if (user == null) {
            servletRequest.getRequestDispatcher("/login.jsp").forward(servletRequest, servletResponse);
            return;
        } else {
            // 让程序继续往下访问用户的目标资源
            filterChain.doFilter(servletRequest, servletResponse);
        }
    }
}
```

web.xml 中的配置:

```
<!--filter 标签用于配置一个Filter 过滤器-->
<filter>
    <!--给filter 起一个别名-->
    <filter-name>AdminFilter</filter-name>
    <!--配置filter 的全类名-->
    <filter-class>com.atguigu.filter.AdminFilter</filter-class>
</filter>
```

```
<!--filter-mapping 配置Filter 过滤器的拦截路径-->
<filter-mapping>
    <!--filter-name 表示当前的拦截路径给哪个filter 使用-->
    <filter-name>AdminFilter</filter-name>
    <!--url-pattern 配置拦截路径
        / 表示请求地址为: http://ip:port/工程路径/ 映射到IDEA 的 web 目录
        /admin/* 表示请求地址为: http://ip:port/工程路径/admin/*
    -->
    <url-pattern>/admin/*</url-pattern>
</filter-mapping>
```

Filter 过滤器的使用步骤:

- 1、编写一个类去实现 Filter 接口
- 2、实现过滤方法 doFilter()
- 3、到 web.xml 中去配置 Filter 的拦截路径

## 完整的用户登录

login.jsp 页面 == 登录表单

```
这是登录页面。login.jsp 页面 <br>
<form action="http://localhost:8080/15_filter/loginServlet" method="get">
    用户名: <input type="text" name="username"/> <br>
    密 码: <input type="password" name="password"/> <br>
    <input type="submit" />
</form>
```

LoginServlet 程序

```
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        resp.setContentType("text/html; charset=UTF-8");
        String username = req.getParameter("username");
        String password = req.getParameter("password");

        if ("wzg168".equals(username) && "123456".equals(password)) {
            req.getSession().setAttribute("user",username);
            resp.getWriter().write("登录 成功!!!");
        } else {
            req.getRequestDispatcher("/login.jsp").forward(req,resp);
        }
    }
}
```

## 3、Filter 的生命周期

Filter 的生命周期包含几个方法

- 1、构造器方法
- 2、init 初始化方法  
第 1, 2 步, 在 web 工程启动的时候执行 (Filter 已经创建)
- 3、doFilter 过滤方法  
第 3 步, 每次拦截到请求, 就会执行
- 4、destroy 销毁  
第 4 步, 停止 web 工程的时候, 就会执行 (停止 web 工程, 也会销毁 Filter 过滤器)

## 4、FilterConfig 类

FilterConfig 类见名知义, 它是 Filter 过滤器的配置文件类。

Tomcat 每次创建 Filter 的时候, 也会同时创建一个 FilterConfig 类, 这里包含了 Filter 配置文件的配置信息。

FilterConfig 类的作用是获取 filter 过滤器的配置内容

- 1、获取 Filter 的名称 filter-name 的内容
- 2、获取在 Filter 中配置的 init-param 初始化参数
- 3、获取 ServletContext 对象

java 代码:

```
@Override
public void init(FilterConfig filterConfig) throws ServletException {
    System.out.println("2.Filter 的 init(FilterConfig filterConfig)初始化");

    //    1、获取Filter 的名称 filter-name 的内容
    System.out.println("filter-name 的值是: " + filterConfig.getFilterName());
    //    2、获取在web.xml 中配置的init-param 初始化参数
    System.out.println("初始化参数 username 的值是: " + filterConfig.getInitParameter("username"));
    System.out.println("初始化参数 url 的值是: " + filterConfig.getInitParameter("url"));
    //    3、获取ServletContext 对象
    System.out.println(filterConfig.getServletContext());
}
```

web.xml 配置:

```
<!--filter 标签用于配置一个Filter 过滤器-->
<filter>
    <!--给filter 起一个别名-->
    <filter-name>AdminFilter</filter-name>
    <!--配置filter 的全类名-->
```

```
<filter-class>com.atguigu.filter.AdminFilter</filter-class>

<init-param>
  <param-name>username</param-name>
  <param-value>root</param-value>
</init-param>

<init-param>
  <param-name>url</param-name>
  <param-value>jdbc:mysql://localhost3306/test</param-value>
</init-param>

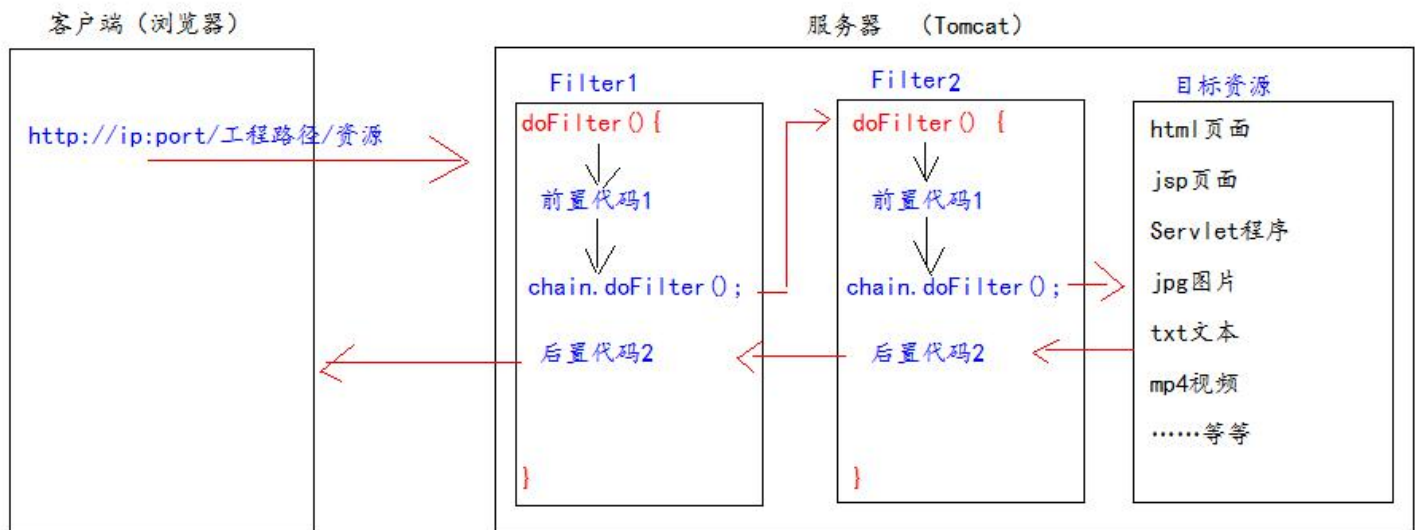
</filter>
```

## 5、FilterChain 过滤器链

Filter 过滤器

Chain 链，链条

FilterChain 就是过滤器链（多个过滤器如何一起工作）



多个Filter过滤器执行的特点：

- 1、所有filter和目标资源默认都执行在同一个线程中
- 2、多个Filter共同执行的时候，它们都使用同一个Request对象。

FilterChain.doFilter()方法的作用

- 1、执行下一个Filter过滤器（如果有Filter）
- 2、执行目标资源（没有Filter）

在多个Filter过滤器执行的时候，它们执行的优先顺序是由他们在web.xml中从上到下配置的顺序决定！！

## 6、Filter 的拦截路径

### --精确匹配

```
<url-pattern>/target.jsp</url-pattern>
```

以上配置的路径，表示请求地址必须为：<http://ip:port/工程路径/target.jsp>

### --目录匹配

```
<url-pattern>/admin/*</url-pattern>
```

以上配置的路径，表示请求地址必须为：[http://ip:port/工程路径/admin/\\*](http://ip:port/工程路径/admin/*)

### --后缀名匹配

```
<url-pattern>*.html</url-pattern>
```

以上配置的路径，表示请求地址必须以.html 结尾才会拦截到

```
<url-pattern>*.do</url-pattern>
```

以上配置的路径，表示请求地址必须以.do 结尾才会拦截到

```
<url-pattern>*.action</url-pattern>
```

以上配置的路径，表示请求地址必须以.action 结尾才会拦截到

后缀名匹配不要/打头

Filter 过滤器它只关心请求的地址是否匹配，不关心请求的资源是否存在！！！！

## 7、书城第八阶段：

### 1、使用 Filter 过滤器拦截/pages/manager/所有内容，实现权限检查

Filter 代码：

```
public class ManagerFilter implements Filter {

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

    }

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

        Object user = httpRequest.getSession().getAttribute("user");

        if (user == null) {
            httpRequest.getRequestDispatcher("/pages/user/login.jsp").forward(servletRequest, servletRes
ponse);
        } else {
            filterChain.doFilter(servletRequest, servletResponse);
        }
    }

    @Override
    public void destroy() {

    }
}
```

web.xml 中的配置：

```
<filter>
    <filter-name>ManagerFilter</filter-name>
    <filter-class>com.atguigu.filter.ManagerFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ManagerFilter</filter-name>
    <url-pattern>/pages/manager/*</url-pattern>
```



```
<url-pattern>/manager/bookServlet</url-pattern>
</filter-mapping>
```

## 2、ThreadLocal 的使用

ThreadLocal 的作用，它可以解决多线程的数据安全问题。

ThreadLocal 它可以给当前线程关联一个数据（可以是普通变量，可以是对象，也可以是数组，集合）

ThreadLocal 的特点：

- 1、ThreadLocal 可以为当前线程关联一个数据。（它可以像 Map 一样存取数据，key 为当前线程）
- 2、每一个 ThreadLocal 对象，只能为当前线程关联一个数据，如果要为当前线程关联多个数据，就需要使用多个 ThreadLocal 对象实例。
- 3、每个 ThreadLocal 对象实例定义的时候，一般都是 static 类型
- 4、ThreadLocal 中保存数据，在线程销毁后。会由 JVM 虚拟自动释放。

测试类：

```
public class OrderService {

    public void createOrder(){
        String name = Thread.currentThread().getName();
        System.out.println("OrderService 当前线程[" + name + "]中保存的数据是：" +
ThreadLocalTest.threadLocal.get());
        new OrderDao().saveOrder();
    }
}

public class OrderDao {

    public void saveOrder(){
        String name = Thread.currentThread().getName();
        System.out.println("OrderDao 当前线程[" + name + "]中保存的数据是：" +
ThreadLocalTest.threadLocal.get());
    }
}

public class ThreadLocalTest {

//    public static Map<String, Object> data = new Hashtable<String, Object>();
    public static ThreadLocal<Object> threadLocal = new ThreadLocal<Object>();

    private static Random random = new Random();
```



```
public static class Task implements Runnable {
    @Override
    public void run() {
        // 在Run方法中，随机生成一个变量（线程要关联的数据），然后以当前线程名为key保存到map中
        Integer i = random.nextInt(1000);
        // 获取当前线程名
        String name = Thread.currentThread().getName();
        System.out.println("线程[" + name + "]生成的随机数是：" + i);
        // data.put(name, i);
        threadLocal.set(i);

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        new OrderService().createOrder();

        // 在Run方法结束之前，以当前线程名获取出数据并打印。查看是否可以取出操作
        // Object o = data.get(name);
        Object o = threadLocal.get();
        System.out.println("在线程[" + name + "]快结束时取出关联的数据是：" + o);
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 3; i++) {
        new Thread(new Task()).start();
    }
}
```

## 3、使用 Filter 和 ThreadLocal 组合管理事务

### 3.1、使用 ThreadLocal 来确保所有 dao 操作都在同一个 Connection 连接对象中完成

原理分析图：

回顾Jdbc的数据库事务管理

```

ThreadLocal<Connection> conn = new ThreadLocal<Connection>();
Connection conn = JdbcUtils.getConnection();
try {
    conn.setAutoCommit(false); // 设置为手动管理事务
    执行一系列的jdbc操作。OrderService.createOrder()
    conn.commit(); // 手动提交事务
} catch (Exception e) {
    conn.rollback(); // 回滚事务
} finally {
    JdbcUtils.close(conn);
}
    
```

ThreadLocal<Connection> conn = new ThreadLocal<Connection>();  
 conn.set(conn); 保存从数据库连接池中获取的连接对象  
 conn.get(); 得到前面保存的Connection连接对象  
 conn.get(); 得到前面保存的Connection连接对象  
 conn.get(); 得到前面保存的Connection连接对象  
 conn.get(); 得到前面保存的Connection连接对象

要确保所有操作要么都成功。要么都失败，就必须使用数据库的事务。

要确保所有操作都在一个事务内，就必须确保，所有操作都使用同一个Connection连接对象。

如何确保所有操作都使用同一个Connection连接对象？

我们可以使用ThreadLocal对象。来确保所有操作都使用同一个Connection对象

ThreadLocal要确保所有操作都使用同一个Connection连接对象。  
那么操作的前提条件是所有操作都必须在同一个线程中完成!!!

JdbcUtils 工具类的修改：

```

public class JdbcUtils {

    private static DruidDataSource dataSource;
    private static ThreadLocal<Connection> conns = new ThreadLocal<Connection>();

    static {
        try {
            Properties properties = new Properties();
            // 读取 jdbc.properties 属性配置文件
            InputStream inputStream =
JdbcUtils.class.getClassLoader().getResourceAsStream("jdbc.properties");
            // 从流中加载数据
            properties.load(inputStream);
            // 创建 数据库连接 池
            dataSource = (DruidDataSource) DruidDataSourceFactory.createDataSource(properties);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 获取数据库连接池中的连接
     * @return 如果返回null, 说明获取连接失败<br/>有值就是获取连接成功
     */
    public static Connection getConnection(){
        Connection conn = conns.get();
    }
}
    
```

```
if (conn == null) {
    try {
        conn = dataSource.getConnection();//从数据库连接池中获取连接
        conns.set(conn); // 保存到ThreadLocal 对象中, 供后面的jdbc 操作使用
        conn.setAutoCommit(false); // 设置为手动管理事务
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
return conn;
}

/**
 * 提交事务, 并关闭释放连接
 */
public static void commitAndClose(){
    Connection connection = conns.get();
    if (connection != null) { // 如果不等于null, 说明 之前使用过连接, 操作过数据库
        try {
            connection.commit(); // 提交 事务
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                connection.close(); // 关闭连接, 资源资源
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    // 一定要执行remove 操作, 否则就会出错。(因为Tomcat 服务器底层使用了线程池技术)
    conns.remove();
}

/**
 * 回滚事务, 并关闭释放连接
 */
public static void rollbackAndClose(){
    Connection connection = conns.get();
    if (connection != null) { // 如果不等于null, 说明 之前使用过连接, 操作过数据库
        try {
            connection.rollback();//回滚事务
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                connection.close(); // 关闭连接, 资源资源
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }  
    }  
}  
// 一定要执行remove 操作, 否则就会出错。(因为Tomcat 服务器底层使用了线程池技术)  
conns.remove();  
}  
  
/**  
 * 关闭连接, 放回数据库连接池  
 * @param conn  
 */  
public static void close(Connection conn){  
    if (conn != null) {  
        try {  
            conn.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}  
} */  
}
```

修改 BaseDao

```
public abstract class BaseDao {  
  
    //使用 DbUtils 操作数据库  
    private QueryRunner queryRunner = new QueryRunner();  
  
    /**  
     * update() 方法用来执行: Insert\Update\Delete 语句  
     *  
     * @return 如果返回-1, 说明执行失败<br/>返回其他表示影响的行数  
     */  
    public int update(String sql, Object... args) {  
  
        System.out.println(" BaseDao 程序在[" + Thread.currentThread().getName() + "]中");  
  
        Connection connection = JdbcUtils.getConnection();  
        try {  
            return queryRunner.update(connection, sql, args);  
        } catch (SQLException e) {  
            e.printStackTrace();  
            throw new RuntimeException(e);  
        }  
    }  
}  
  
/**
```

```
* 查询返回一个 javaBean 的 sql 语句
*
* @param type 返回的对象类型
* @param sql 执行的 sql 语句
* @param args sql 对应的参数值
* @param <T> 返回的类型的泛型
* @return
*/
public <T> T queryForOne(Class<T> type, String sql, Object... args) {
    Connection con = JdbcUtils.getConnection();
    try {
        return queryRunner.query(con, sql, new BeanHandler<T>(type), args);
    } catch (SQLException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}

/**
* 查询返回多个 javaBean 的 sql 语句
*
* @param type 返回的对象类型
* @param sql 执行的 sql 语句
* @param args sql 对应的参数值
* @param <T> 返回的类型的泛型
* @return
*/
public <T> List<T> queryForList(Class<T> type, String sql, Object... args) {
    Connection con = JdbcUtils.getConnection();
    try {
        return queryRunner.query(con, sql, new BeanListHandler<T>(type), args);
    } catch (SQLException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}

/**
* 执行返回一行一列的 sql 语句
* @param sql 执行的 sql 语句
* @param args sql 对应的参数值
* @return
*/
public Object queryForSingleValue(String sql, Object... args){

    Connection conn = JdbcUtils.getConnection();

    try {
        return queryRunner.query(conn, sql, new ScalarHandler(), args);
    }
```

```

    } catch (SQLException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }

}
}

```

### 3.2、使用 Filter 过滤器统一给所有的 Service 方法都加上 try-catch。来进行实现的管理。

原理分析图：

现在有一个TransactionFilter事务的Filter过滤器。

```
public void doFilter(request, response, filterChain) {
```

```
try{
```

filterChain.doFilter();      间接调用Servlet程序中的业务方法      OrderServlet.createOrder();  
提交事务      Filter里间接调用了      OrderService.createOrder();      直接调用

```
} catch (Exception e) {
```

```
    回滚事务
```

```
}
```

按照上面的分析示意，那么就可以使用一个Filter一次性，统一地给所有的XxxService.xxxx()方法都统一加上 try-catch() 来实现事务的管理

filterChain.doFilter()方法的使用是

- 1、调用下一个filter过滤器
- 2、调用目标资源
  - html页面
  - jsp页面
  - txt文本
  - jpg图片
  - Servlet程序

Filter 类代码：

```

public class TransactionFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain) throws IOException, ServletException {
        try {
            filterChain.doFilter(servletRequest, servletResponse);
            JdbcUtils.commitAndClose(); // 提交事务
        } catch (Exception e) {
            JdbcUtils.rollbackAndClose(); // 回滚事务
            e.printStackTrace();
        }
    }
}

```

```
}
```

在 web.xml 中的配置:

```
<filter>
  <filter-name>TransactionFilter</filter-name>
  <filter-class>com.atguigu.filter.TransactionFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>TransactionFilter</filter-name>
  <!-- /* 表示当前工程下所有请求 -->
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

一定要记得把 BaseServlet 中的异常往外抛给 Filter 过滤器

```
public abstract class BaseServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        doPost(req, resp);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        // 解决 post 请求中文乱码问题
        // 一定要在获取请求参数之前调用才有效
        req.setCharacterEncoding("UTF-8");

        String action = req.getParameter("action");
        try {
            // 获取 action 业务鉴别字符串, 获取相应的业务 方法反射对象
            Method method = this.getClass().getDeclaredMethod(action, HttpServletRequest.class,
HttpServletResponse.class);
            // System.out.println(method);
            // 调用目标业务 方法
            method.invoke(this, req, resp);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e); // 把异常抛给 Filter 过滤器
        }
    }
}
```

### 3.3、将所有异常都统一交给 Tomcat, 让 Tomcat 展示友好的错误信息页面。

在 web.xml 中我们可以通过错误页面配置来进行管理。



<!--error-page 标签配置，服务器出错之后，自动跳转的页面-->

<error-page>

<!--error-code 是错误类型-->

<error-code>500</error-code>

<!--location 标签表示。要跳转去的页面路径-->

<location>/pages/error/error500.jsp</location>

</error-page>

<!--error-page 标签配置，服务器出错之后，自动跳转的页面-->

<error-page>

<!--error-code 是错误类型-->

<error-code>404</error-code>

<!--location 标签表示。要跳转去的页面路径-->

<location>/pages/error/error404.jsp</location>

</error-page>



