一、实验步骤

1.1、实验一

第一步：编写 ItemService 和 ItemAction 的相关代码

ItemService.java 的相关代码

```
public class ItemService {
	public List<Item> getAllItems(){
		List<Item> items = new ArrayList<Item>();
		items.add(new Item("book001", "JAVAEE 技术实验指导教程",
				"WEB 程序设计知识回顾、" + "轻量级 JAVAEE 应用框架、"
					+ "企业级 EJB 组件编程技术、" + "JAVAEE 综合应用开发.",
19.95));
		items.add(new Item("book002", "JAVAEE 技术",
				"Struts 框架、Hibernate 框架、Spring 框架、"
					+ "会话 Bean、实体 Bean、消息驱动 Bean", 29.95));
		return items;
	}
}
```

ItemAction.java 的相关代码

```
public class ItemAction extends ActionSupport {
	private List<Item> items;


	public List<Item> getItems() {
		return items;
	}

	public void setItems(List<Item> items) {
		this.items = items;
	}

	public String getAllItems(){
		ItemService itemService = new ItemService();
		items = itemService.getAllItems();
		System.out.println("ItemAction executed!");
		return SUCCESS;
	}
}
```

第二步：配置 struts.xml 文件

```
<package name="strutsBean" extends="struts-default" namespace="/" strict-method-
invocation="false">
    <action name="login" class="action.UserAction" method="login">
        <result name="success">/loginSuccess.jsp</result>
        <result name="fail">/login.jsp</result>
        <result name="input">/login.jsp</result>
    </action>
    <action name="register" class="action.UserAction" method="register">
        <result name="success">/registerSuccess.jsp</result>
        <result name="fail">/register.jsp</result>
        <result name="input">/register.jsp</result>
    </action>
    <action name="allItems" class="action.ItemAction" method="getAllItems">
        <result name="success">/itemList.jsp</result>
    </action>
</package>
```

第四步：登录及购物车调试

1 查看所有商品信息

编号	名称	说明	单价	数量
book001	JAVAEE 技术实验指导教程	WEB 程序设计知识回顾、轻量级 JAVAEE 应用框架、企业级 EJB 组件编程技术、JAVAEE 综合应用开发.	19.95	
book002	JAVAEE 技术	Struts 框架、Hibernate 框架、Spring 框架、会话 Bean、实体 Bean、消息驱动 Bean	29.95	

图 1 实验 1 运行结果

第五步：创建用户授权拦截器

（1）添加 AuthorityInterceptor.java

```
@Override
public String intercept(ActionInvocation invocation) throws Exception {
    System.out.println("Authority Interceptor executed!");

    ActionContext actionContext = invocation.getInvocationContext();
    Map<String, Object> session = actionContext.getSession();
    String user = (String)session.get("user");
    if (user != null) {
        return invocation.invoke();
    } else {
```

```
            actionContext.put("tip", "您还没有登录，请输入用户名和密码登录系统");
            return Action.LOGIN;
        }
}
```

（2）修改 UserAction.java

```
public String login() {
        ActionContext actionContext = ActionContext.getContext();
        session = actionContext.getSession();
        UserService userService = new UserService();
        if (userService.login(loginUser)) {
            session.put(USER, loginUser.getAccount());
            return SUCCESS;
        }
        return FAIL;
}
```

（3）修改 struts.xml 文件，添加用户授权拦截器

```
<interceptors>
        <interceptor name="authority" class="interceptors.AuthorityInterceptor"/>
</interceptors>



<action name="allItems" class="action.ItemAction" method="getAllItems">
        <result name="login">/login.jsp</result>
        <result name="success">/itemList.jsp</result>
        <!-- 配置系统默认拦截器 -->
        <interceptor-ref name="defaultStack"/>
        <!-- 配置 authority 拦截器 -->
        <interceptor-ref name="authority"/>
</action>
```

第六步：测试 authority interceptor

在未登录的情况下访问连接

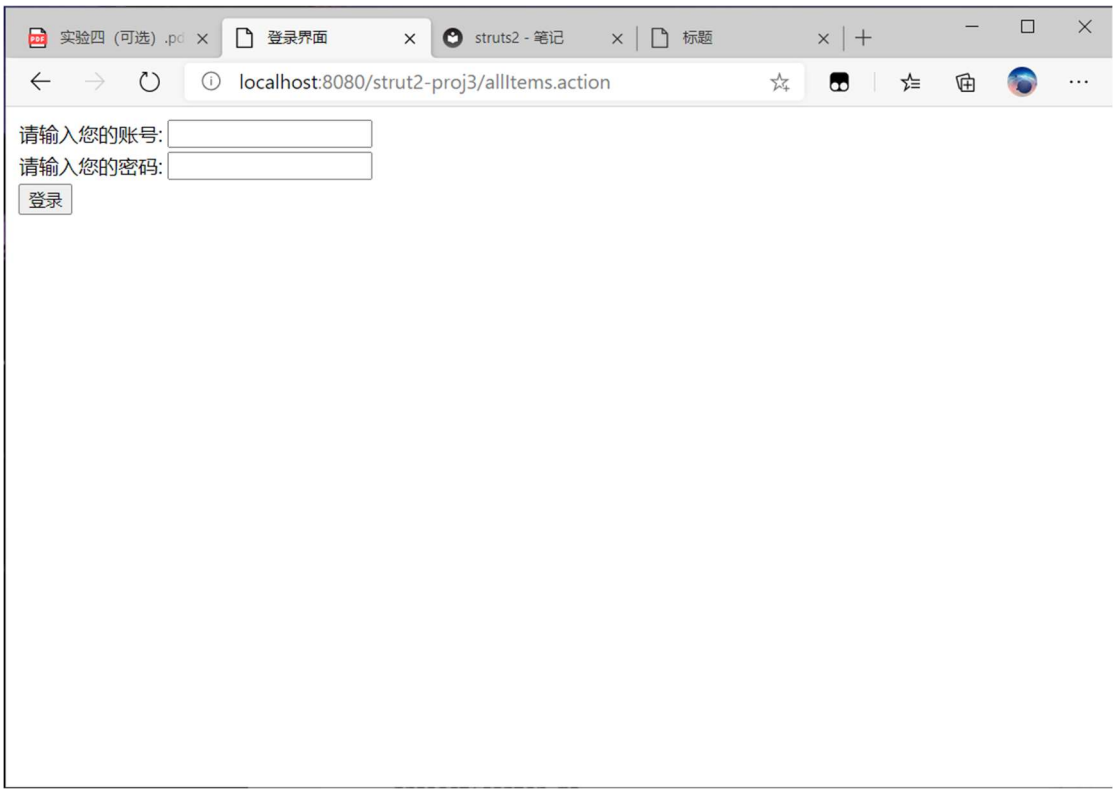


图 2 实验 1 运行结果

在登录的情况下访问链接

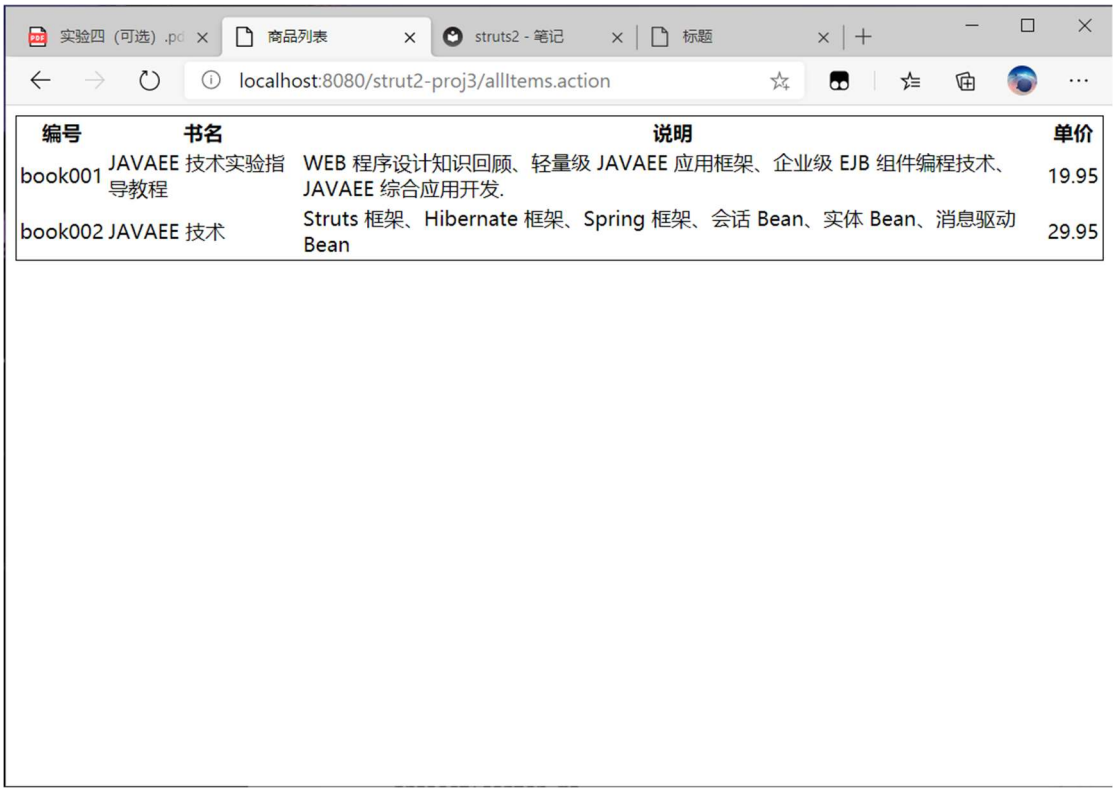


图 3 实验 1 运行结果

第七步：添加 HttpFilter

```
package cn.edu.zjut.filters;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class AccessFilter extends HttpFilter {
    @Override
    public void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain
chain) throws IOException, ServletException {
        System.out.println("Access Filter executed!");
        HttpSession session = request.getSession();
        if (session.getAttribute("user") == null &&
                !request.getRequestURI().contains("login.jsp") &&
                !request.getRequestURI().contains("register.jsp")
        ) {
            response.sendRedirect("login.jsp");
            return;
        }
        chain.doFilter(request, response);
    }
}
```

第八步：测试 HttpFilter

访问 loginSuccess.jsp

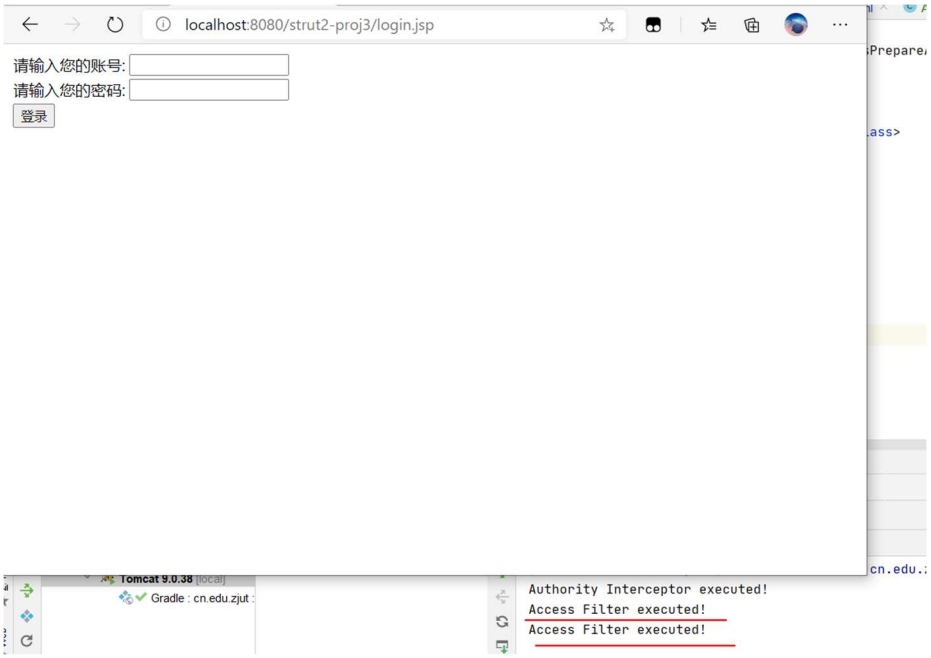


图 4 实验 1 运行结果

访问 itemList.jsp

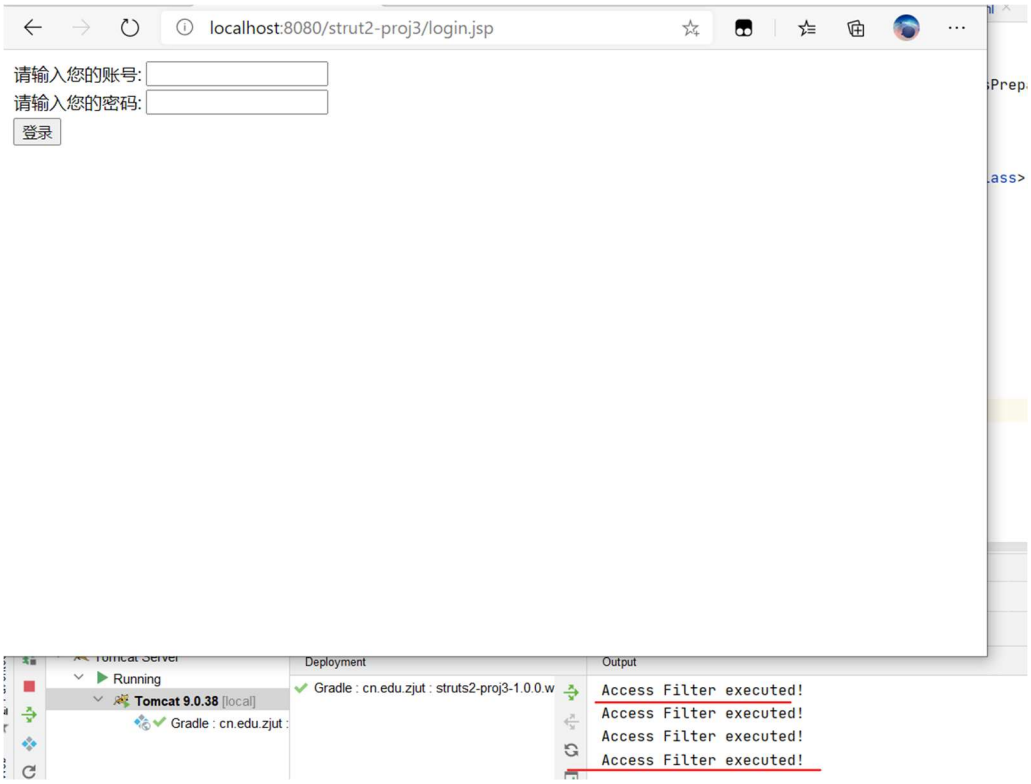


图 5 实验 1 运行结果

登录后访问链接

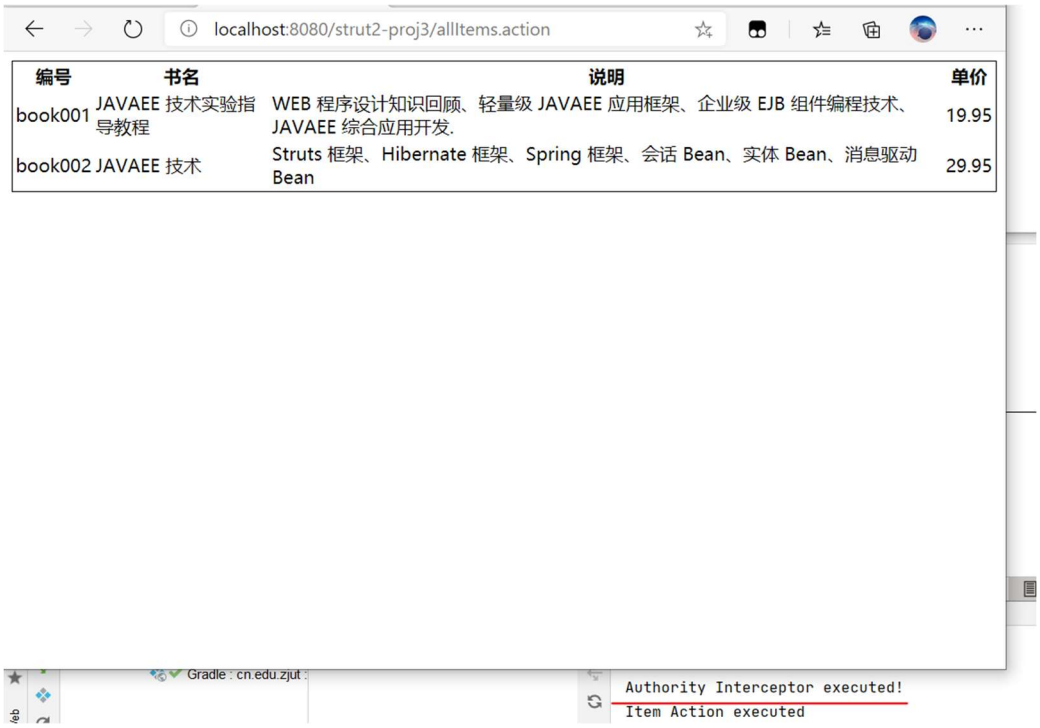


图 6 实验 1 运行结果

1.2、实验二

第一步：使用代码片段输出表格

```
    <%
        ValueStack vs = (ValueStack)request.getAttribute("struts.valueStack");
        String title = vs.findString("tableTitle");
        List<Item> itemList = (List<Item>)vs.findValue("items");
    %>
    <table style="border: 1px solid black">
        <tr>
            <th>编号</th>
            <th>书名</th>
            <th>说明</th>
            <th>单价</th>
        </tr>
<%--        <s:iterator value="items">--%>
<%--            <tr>--%>
<%--                <td><s:property value="itemId"/></td>--%>
<%--                <td><s:property value="name"/></td>--%>
<%--                <td><s:property value="description"/></td>--%>
<%--                <td><s:property value="cost"/></td>--%>
<%--            </tr>--%>
<%--        </s:iterator>--%>
        <%for (Item item : itemList) { %>
          <tr>
            <td><%=item.getItemId()%></td>
            <td><%=item.getName()%></td>
            <td><%=item.getDescription()%></td>
            <td><%=item.getCost()%></td>
          </tr>
        <%} %>
    </table>
```

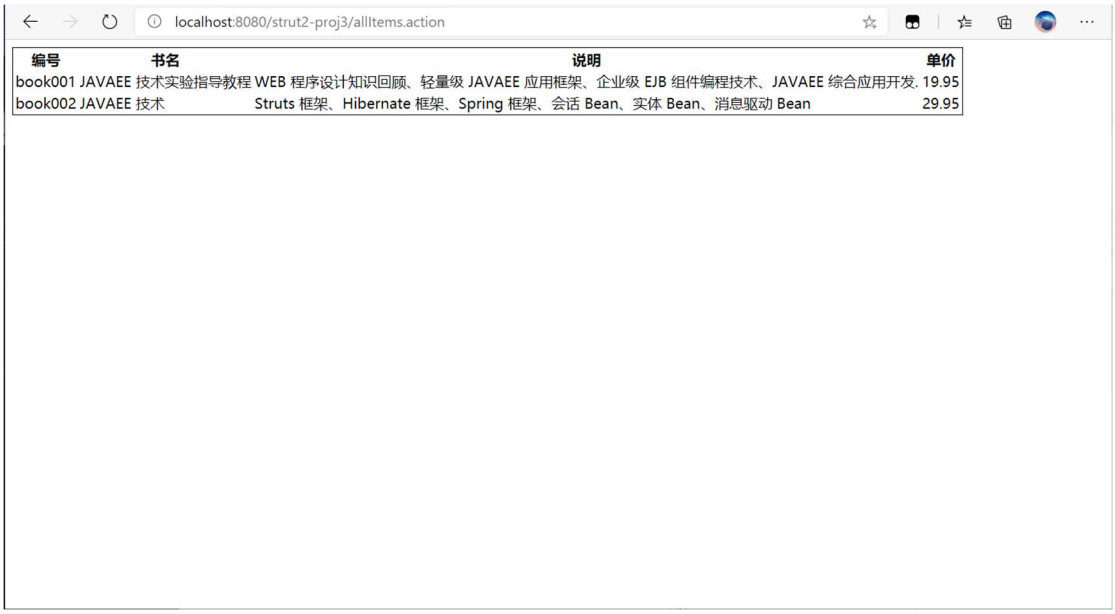


图 7 实验 2 运行结果

第二步：使用 OGNL 输出

图 8 实验 2 运行结果

第三步：使用#和%OGNL 表达式

```
<p>
    商品低于 20 元的商品有：<br/>
    <s:iterator value="items.{?#this.cost<20}">
    <li><s:property value="name"/>:<s:property value="cost"/>元</li>
    </s:iterator>
</p>
<p>
    名称为《JAVAEE  技术实验指导教程》的商品的价格为：
    <s:property value="items.{?#this.name==\"JAVAEE  技术实验指导教程\"}.{cost}[0]"/>元
</p>
```

图 9 实验 2 运行结果

1.3、实验三

第一步：使用异常处理界面（自己处理异常）

UserService 的代码片段

```
public boolean login(UserBean user) throws Exception {
    if (user.getAccount().equalsIgnoreCase("admin")) {
        throw new UserException("用户名不能为 admin");
    }
    if (user.getPassword().toUpperCase().contains(" AND ") ||
        user.getPassword().toUpperCase().contains(" OR ")) {
        throw new java.sql.SQLException("密码不能包括' and '或' or '");
    }

    if (user.getAccount().equals(user.getPassword())){
        return true;
    } else {
        return false;
    }
}
```

UserAction 的代码片段

```
public String login(){
    ActionContext actionContext = ActionContext.getContext();
    session = actionContext.getSession();
    UserService userService = new UserService();

    try {
        if (userService.login(loginUser)) {
            session.put(USER, loginUser.getAccount());
            return SUCCESS;
        }
    } catch (Exception e) {
        e.printStackTrace();
        return "exception";
    }


    return FAIL;
}
```

struts.xml 的代码片段

```
<action name="login" class="cn.edu.zjut.action.UserAction" method="login">
    <result name="success">/loginSuccess.jsp</result>
    <result name="fail">/login.jsp</result>
```

```
    <result name="input">/login.jsp</result>
    <result name="exception">/loginException.jsp</result>
</action>
```

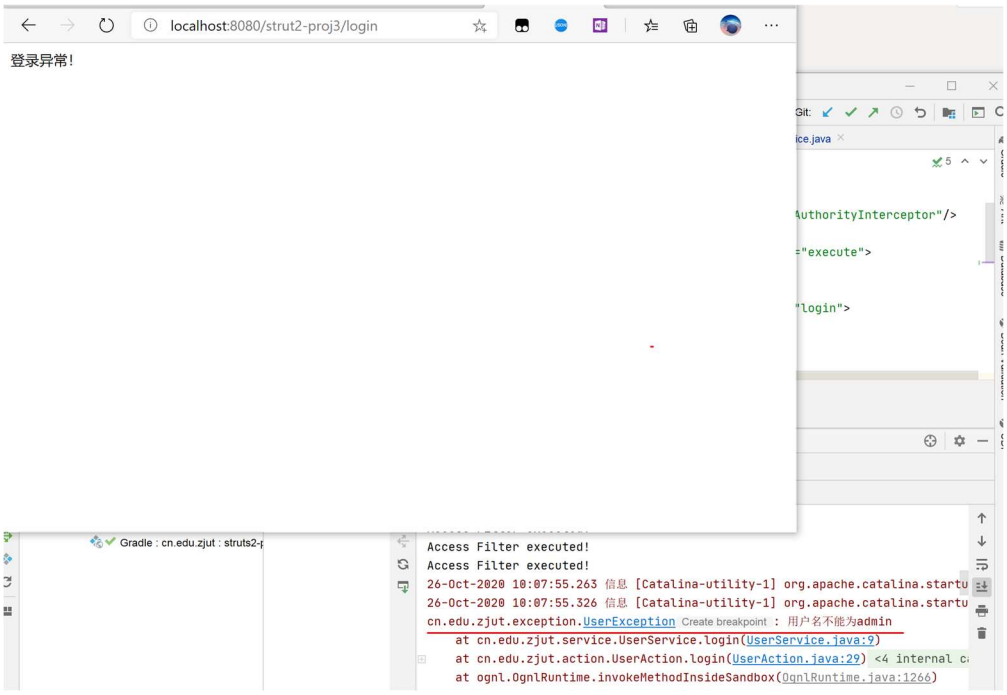


图 10 实验 3 运行结果

第二步：交给框架处理异常

UserAction 的代码片段

```
public String login() throws Exception {
    ActionContext actionContext = ActionContext.getContext();
    session = actionContext.getSession();
    UserService userService = new UserService();

    if (userService.login(loginUser)) {
        session.put(USER, loginUser.getAccount());
        return SUCCESS;
    }

    return FAIL;
}
```

struts.xml 的代码片段

```
<global-results>
    <result name="sqlExcp">/loginException.jsp</result>
</global-results>
<global-exception-mappings>
```

```
        <exception-mapping exception="java.sql.SQLException" result="sqlExcp"/>
</global-exception-mappings>
<action name="index" class="cn.edu.zjut.action.IndexAction" method="execute">
        <result name="success">/index.jsp</result>
</action>
<action name="login" class="cn.edu.zjut.action.UserAction" method="login">
        <exception-mapping exception="cn.edu.zjut.exception.UserException" result="userExcp"/>
        <result name="success">/loginSuccess.jsp</result>
        <result name="fail">/login.jsp</result>
        <result name="input">/login.jsp</result>
        <result name="userExcp">/loginException.jsp</result>
</action>
```

调试结果

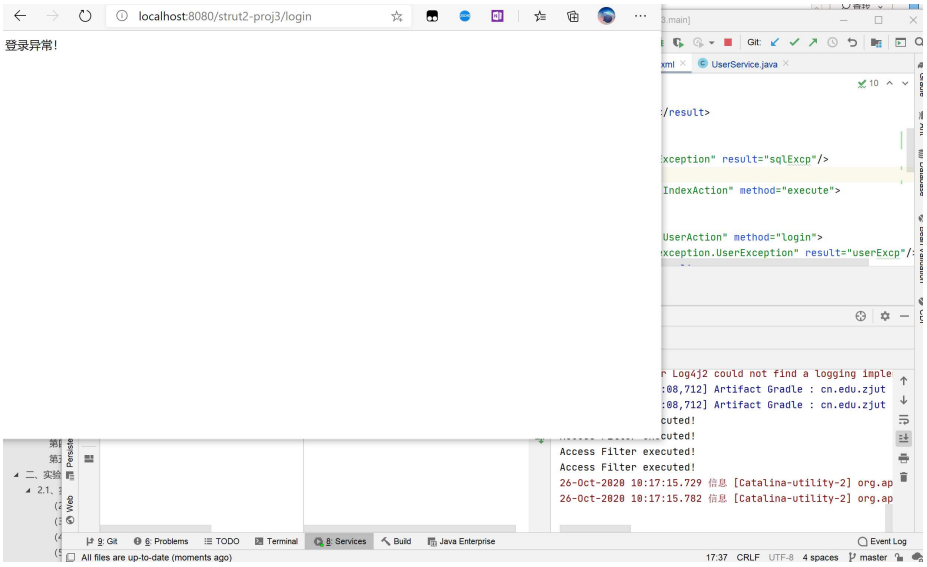


图 11 实验 3 运行结果

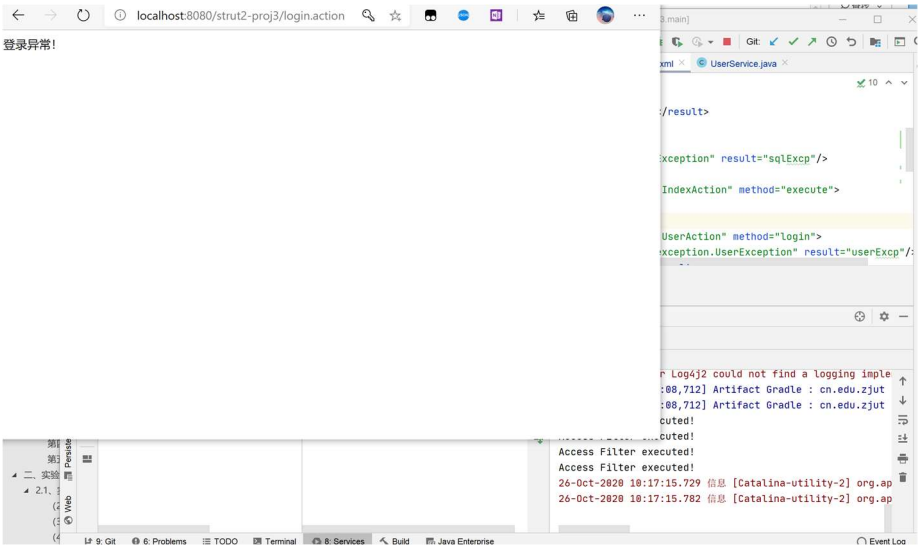


图 12 实验 3 运行结果

二、实验结果及分析

2.1、实验一

（2）自定义拦截器

intercept(ActionInvocation invocation)：拦截器方法。

init()：执行在拦截器创建后，但执行拦截方法前，用于初始化资源。

destory()：执行在拦截器拦截后，用于释放非托管资源。

方法返回值：本次拦截器输出的 result Code，可以被后续的拦截器获取或者覆盖。一般执行 invocation.invoke()表示保留下一级拦截器的 result Code。

（3）自定义拦截器

拦截器应当声明在 package 内，也可以声明一个拦截器栈，示例如下

```
<interceptors>
    <interceptor name="myInterceptor"
class="edu.hust.interceptor.MyInterceptor"></interceptor>
    <interceptor-stack name="myInterceptorStack">
        <interceptor-ref name="defaultStack"></interceptor-ref>
        <interceptor-ref name="myInterceptor"></interceptor-ref>
    </interceptor-stack>
</interceptors>
```

一个拦截器栈内的拦截器将按照声明的顺序依次执行，一个拦截器栈内，可以声明若干个拦截器和拦截器栈。

拦截器的配置应当放在<action>内，表明对该方法启用拦截器（栈），若没有声明任意拦截器（栈），则默认引入 default-stack 拦截器栈。***但是，如果声明了拦截器（栈），则 default-stack 就是失效，需要手动配置***。

另外，可以在 package 内使用<default-interceptor-ref>来声明该包下所有 action 的默认拦截器（栈），但是一旦子 action 设置后，此默认设置失效。

由于 action 内的拦截器（栈）是按顺序依次执行，***因此务必将"defaultStack"放在最前面***。

（4）default-stack

```
<interceptor-stack name="defaultStack">
    <interceptor-ref name="exception"/> 异常处理
    <interceptor-ref name="alias"/> 别名处理
```

```
    <interceptor-ref name="servletConfig"/> servlet 的配置
    <interceptor-ref name="i18n"/> 国家化
    <interceptor-ref name="prepare"/>
    <interceptor-ref name="chain"/> 过滤器链
    <interceptor-ref name="scopedModelDriven"/> 带作用范围的 ModelDriven
    <interceptor-ref name="modelDriven"/>
    <interceptor-ref name="fileUpload"/> 文件上传
    <interceptor-ref name="checkbox"/> 复选框
    <interceptor-ref name="datetime"/> 日期处理
    <interceptor-ref name="multiselect"/> 多选菜单处理
    <interceptor-ref name="staticParams"/> 静态参数
    <interceptor-ref name="actionMappingParams"/> Action 映射参数
    <interceptor-ref name="params"/> 参数装载
    <interceptor-ref name="conversionError"/> 数据转换错误处理
    <interceptor-ref name="validation"> 数据验证
        <param name="excludeMethods">input,back,cancel,browse</param>
    </interceptor-ref>
    <interceptor-ref name="workflow">
        <param name="excludeMethods">input,back,cancel,browse</param>
    </interceptor-ref>
    <interceptor-ref name="debugging"/> 调试
</interceptor-stack>
```

（5）自定义过滤器

自定义过滤器由于属于 Web 容器，因此需要在 web.xml 中进行配置，配置的示例如下：

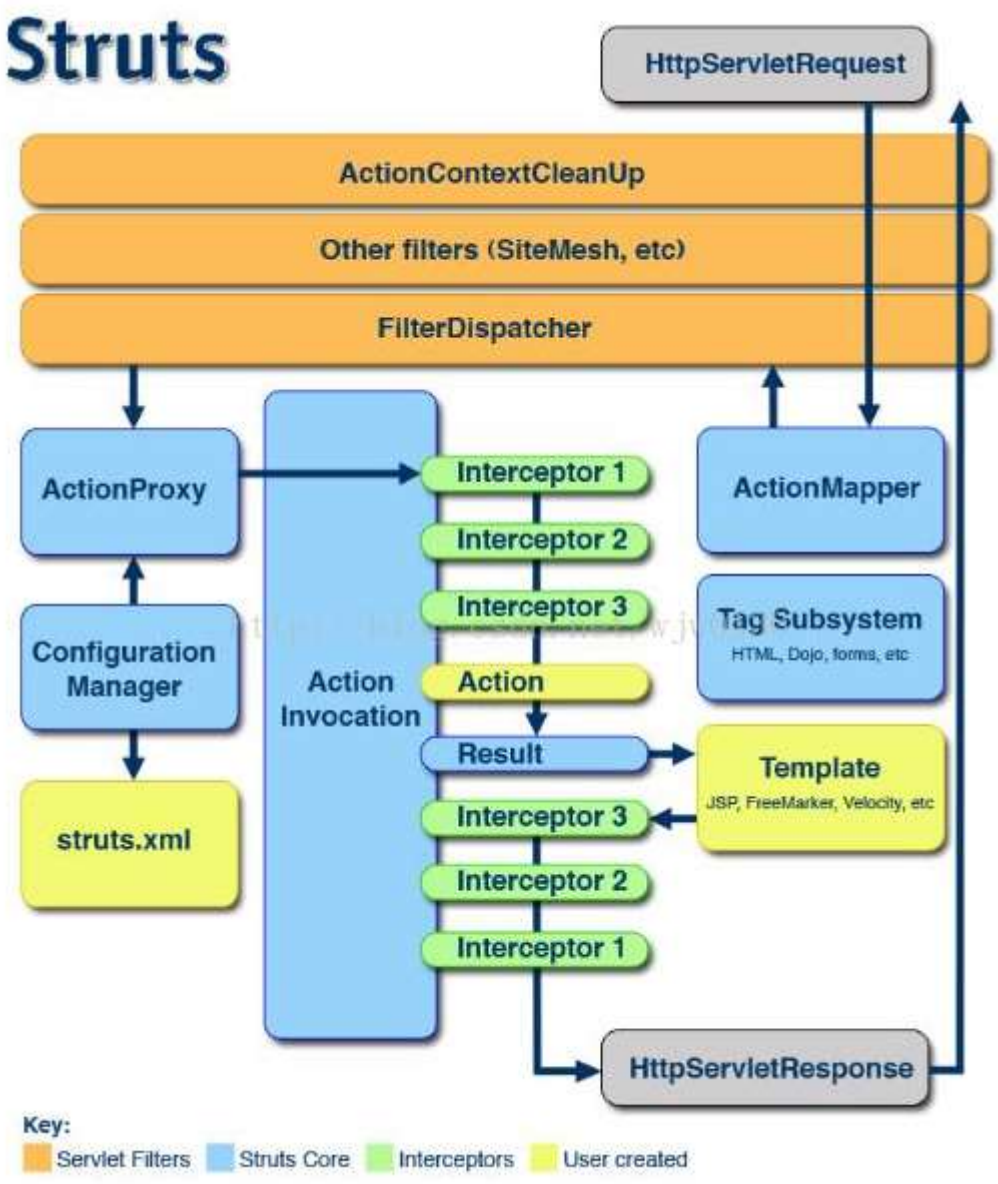
```
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<filter>
    <filter-name>access</filter-name>
    <filter-class>cn.edu.zjut.filters.AccessFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>access</filter-name>
    <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

Filter 的作用于 Action 创建前，适合做全局性的过滤性工作，而 Interceptor

作用域 Action 创建后，适合做局部的过滤性工作，但是两者的功能可以相互替代，过滤器执行于拦截器之前，Filter 的执行顺序和声明顺序是一致的（当然含有通配符的过滤器的优先级更高），要注意过滤器的配置顺序。

（6）Struts 的工作原理和基本工作过程

struts 框架是一个独立于 web 容器的框架，需要 StrutsPrepareAndExecuteFilter 驱动，执行的过程参考下列的图片（也是一张十分经典的图片了）。



（7）实验收获

过滤器和拦截器再次体现了模块化和 AOP 的设计理念，将本来很复杂的逻辑进行拆分然后进行组装，使得整个系统的架构清晰，可阅读性更高。同时也提

高了写代码和架构的效率。

2.2、实验二

（2）ValueStack 的用法

ValueStack 容器是数据的存储对象，其中一部分数据存储于 Map 中，通过 findString 和 findValue 访问，通过 setValue 和 setParameter 或者 set 设置。另一部分数据存在于 Stack 中，其访问的规则遵从堆栈，使用 push，pop，peak 来存取值。当使用 OGNL 表达式时，先依次从值栈中满足查找对应表达式的数据，然后在从字典中查找。

（3）OGNL

1、使用@[类名]@[方法名]来调用静态方法，例如@java.util.Math@random()

2、取值：<s:property value="user.name"/>，当取非根 Map 的对象时，需要加上#，例如#session.user

3、创建迭代器<s:iterator value=""></s:iterator>

4、创建集合，例如"#{'0': '123', '1': '123'}"，可以创建 Map 和 ArrayList 对象。

5、简单 UI 标签

6、设置值<s:set var="var", value="expression"/>用于在 OGNL 值栈中设置值。

7、%表达式，#堆栈中上下文对象，$引用 OGNL 表达式

8、过滤集合表达式，例如 items.{?#this.cost<20}

（5）实验收获

OGNL 表达式和 EL 表达式有异曲同工之妙，都是为了在表示层（Layer）减少 Java 代码的使用频率，因为基于 MVC 的设计原理，在 View 层是不应该出现业务逻辑，而是使用“取值”将存放于上下文的值取出来，而很多时候.jsp 是由前端人员来写的（现在很少了），使用相对简单的表达式语言有利于提高效率，而且学期起来也比较简单。

2.3、实验三

（2）自定义异常类

1、在合适的包下创建 exception 包，然后创建一个 UserException（用户自定义）类继承自 Exception，并写好 Exception 的各种构造器。

2、在用户逻辑处理的方法上声明抛出的方法，当出现异常时，使用 throw 的方法抛出或者不处理内部其他逻辑的异常。

3、在 UserAction 使用 try…catch 块截获或者交由 struts 框架进行处理

（4）structs 异常处理

struts 异常处理在 struts.xml 内部配置和声明。

struts 异常处理可以在 package 或者 action 内部进行配置，首先需要一个<exception-mappings>标签来定义出现异常的结果，还需要一个<result>标签来声明重定向的结果。当配置在 package 内部时，需要使用 global 前缀标识。

（5）实验收获

在很多的实际情况中，一个操作往往会出现异常的情况，而异常往往是需要马上处理的，如果没有异常处理，则会非常频繁地使用到 if…else 分支语句来判断执行结果，会使得整个代码块变得非常臃肿，而且会夹杂很多的不必要的判断语句。因为异常引发时往往会终止当前函数，如果异常在 try…catch 中引发，那么异常会一级级返回，这类机制类似于冒泡，即不需要在每个函数中都进行订阅和处理，只需要拦截即可。

而且，异常虽然对于用户来说非常头痛，但是好的异常处理往往能够快速而精确地定位到程序的错误信息以及堆栈，便于及时地处理和修复异常。

冒泡机制不同于单点的订阅机制，其具有往上冒泡，拦截的作用，在现在很多 UI 的事件处理中往往用到很多的冒泡机制（对应的事件叫做路由事件），冒泡机制依赖于 UI 树的定义与构造，也涉及到命中测试。

总之，异常处理可以规避掉冗余的检测判断代码，是现代代码的一个比较重要的机制，同时，有时候往往拦截器需要配合异常处理来进行判断。