轻量级 J2EE 框架应用

E 3　A Simple Controller with Interceptors

学号：SA18225433

姓名：杨帆

报告撰写时间：2018/11/29

1.主题概述

1、 拦截器 Interceptor 概念。

拦截器是动态拦截 Action 调用的对象，它提供了一种机制可以使开发者在一个 Action 执行的前后执行一段代码，也可以在一个 Action 执行前阻止其执行，同时也提供了一种可以提取 Action 中可重用部分代码的方式。在 AOP 中，拦截器用于在某个方法或者字段被访问之前，进行拦截然后再之前或者之后加入某些操作。

2、 拦截器 Interceptor 原理。

当请求到达 Struts2 的 ServletDispatcher 时，Struts2 会查找配置文件，并根据配置实例化相对的拦截器对象，然后串成一个列表（List），最后一个一个的调用列表中的拦截器。Struts2 的拦截器是可插拔的，拦截器是 AOP 的一个实现。Struts2 拦截器栈就是将拦截器按一定的顺序连接成一条链。在访问被拦截的方法或者字段时，Struts2 拦截器链中的拦截器就会按照之前定义的顺序进行调用。

3、 动态代理。

Java 中的代理就是不直接操作目标类，而是通过一个代理类去间接的使用目标类中的方法，通过此种方法，在操作和目标类中间添加了一层中间层，能有效控制对目标类对象的直接访问，也可以很好的隐藏和保护目标类对象，同时在代理类中能给目标类实施多种控制策略，提高了设计上的灵活性，我们使用 jdk 动态代理，包括一个类 Proxy 和一个接口 InvocationHandler，我们创建动态代理类必须继承 InvocationHandler 接口并实行 invoke 方法，当使用 Proxy 调用 newProxyInstance 方法后即创建了目标类的动态代理对象，此时通过代理对象调用目标对象方法时，这个方法就会被转发由 InvocationHandler 接口的 invoke 方法来进行调用。

4、 请分析在 MVC pattern 中，Controller 可以具备哪些功能，并描述是否合理？

Controller 可以决定要显示哪一个 View。

Controller 负责定义和调用 Model。

控制器接受用户的输入并调用模型和视图去完成用户的需求。当 web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何东西和做任何处理。它只接受请求并决定调用哪个模型构件去处理请求，然后决定用哪个视图来显示模型处理返回的数据。

2.假设

本次作业能够实现使用 IDEA 实现一个具有拦截器的 Action 处理请求的过程，Servlet 监听请求，当接受到一个能够匹配的请求，获得请求名，创建 Action 接口，创建 Action 接口的实现类并用执行方法封装对 Xml 文件的解析，对于解析出来类路径与方法名使用反射调用对应的 action 方法，使用动态代理代理这个 Action 实现类对象，在动态代理 invoke 方法中添加拦截器，当判断该 action 确实有被此拦截器拦截则在 action 执行开始前和结束后调用拦截器方法在控制台打印信息，并将信息写到 log.xml 文件中。

3.实现或证明

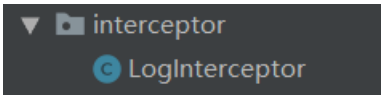
1. 实现成果

e3： https://github.com/saaaaaail/J2eee3

2.在工程 UseSC 中定义源码包 water.ustc.interceptor，在该包中定义一个 POJO 类 LogInterceptor 作为拦截器，在该类中定义方法 preAction（）和 afterAction()，分别实现功能为：记录每次客户端请求的 action 名称<name>、访问开始时间<s-time>，访问结束时间<e-time>、请求返回结果<result>，并将信息追加至日志文件 log.xml， 保存在 PC 磁盘上。log.xml 格式可参考如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<log>
    <action>
        <name>login</name>
        <s-time>2013-12-04 14:20:56</s-time>
        <e-time>2013-12-04 14:20:59</e-time>
        <result>success</result>
    </action>
    <!-- other actions -->
</log>
```

定义 water.ustc.interceptor 包，并定义 LogInterceptor 为拦截器类，



在类中定义 preAction()和 afterAction()方法，在 preAction()方法中打印开始时间，在 afterAction 中打印结束时间并将 action 名称，开始时间，结束时间，返回结果写到“log.xml”文件中去，

```
public void preAction(String actionName) {
    Date date = new Date();
    startTime = sdf.format(date);
    name = actionName;
    System.out.println("actionName: "+name+" "+"startTime: "+startTime);
}
```

```
public void afterAction(String resultString) {
    Date date = new Date();
    endTime = sdf.format(date);
    String[] tmp = resultString.split( regex: ",");
    result = tmp[2];
    System.out.println("actionName: "+name+" "+"result: "+result+" "
            +"startTime: "+startTime+" "+"endTime: "+endTime);
    XmlUtil xmlUtil = XmlUtil.getInstance();
    String file = this.getClass().getClassLoader().getResource( name: "log.xml").getPath();
    System.out.println(file);
    xmlUtil.writeToXml(file,name,result,startTime,endTime);
}
```

在 SimpleController 中的 XmlUtil 工具类中编写 writeToXml 方法，并事先在资源 resourse

目录下创建“log.xml”文件，在 writeToXml 方法中读取 log 标签保存为 Node，若没有 log 标签，则创建 log 标签添加为 DOM 树中的一个 Node，分别创建 action、name、start_time、end_time、result 标签节点，给 name、start_time、end_time、result 标签写入值，将这四个标签添加到 action 标签中，最后使用 log 标签添加该 action 标签，最后使用 Transformer 对象将该 DOM 树转化为 xml 文件，

```
NodeList logs = doc.getElementsByTagName("log");
Node logElement = logs.item( index: 0);
if(logElement==null){
    logElement = doc.createElement( tagName: "log");
    doc.appendChild(logElement);
}
```

```
Element actionElement = doc.createElement( tagName: "action");
Element nameElement = doc.createElement( tagName: "name");
Element startElement = doc.createElement( tagName: "start_time");
Element endElement = doc.createElement( tagName: "end_time");
Element resultElement = doc.createElement( tagName: "result");

nameElement.setTextContent(name);
startElement.setTextContent(startTime);
endElement.setTextContent(endTime);
resultElement.setTextContent(result);

actionElement.appendChild(nameElement);
actionElement.appendChild(startElement);
actionElement.appendChild(endElement);
actionElement.appendChild(resultElement);

logElement.appendChild(actionElement);

// 创建TransformerFactory对象
TransformerFactory tff = TransformerFactory.newInstance();
// 创建Transformer对象
Transformer tf = tff.newTransformer();
// 设置输出数据时换行
tf.setOutputProperty(OutputKeys.INDENT, value: "yes");
// 使用Transformer的transform()方法将DOM树转换成XML
tf.transform(new DOMSource(doc), new StreamResult(file));
```

3.基于 E2 ，在 UseSC 工程的 controller.xml 的中增加<interceptor>节点作为拦截器节点，该节点指明拦截器定义类型及拦截方法。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<sc-configuration>

    <interceptor name="log" class="water.ustc.inteceptor.LogInterceptor"
        predo="preAction" afterdo="afterAction">
    </interceptor>
    <!-- some interceptors -->

    <controller>
        <action name="login" class="water.ustc.action.LoginAction"
            method="handleLogin">

            <interceptro-ref name="log"></interceptro-ref>

            <result name="success" type="forward" value="pages/welcome.jsp"></result>
            <result name="failure" type="redirect" value="pages/failure.jsp"></result>
            <!-- some results -->
        </action>
        <action name="register" class="water.ustc.action.RegisterAction"
            method="handleRegister">

            <interceptro-ref name="log"></interceptro-ref>

            <result name="success" type="forward" value="pages/welcome.jsp"></result>
            <!-- some results -->
        </action>
        <!-- some actions -->
    </controller>

</sc-configuration>
```

修改 controller.xml 文件添加拦截器如上图所示，

```
<interceptor name="log" class="water.ustc.interceptor.LogInterceptor" predo="preAction" afterdo="afterAction">
</interceptor>
```

4.在<action>增加<interceptor-ref>节点作为拦截器引用节点，<interceptor-ref>指向 controller.xml 中已定义的<interceptor>节点 name 属性。

给 login action 添加拦截器引用结点，

```
<action name="login" class="water.ustc.action.LoginAction" method="handleLogin">
    <interceptor-ref name="log"></interceptor-ref>
    <result name="success" type="forward" value="pages/welcome.jsp"></result>
    <result name="failure" type="redirect" value="pages/failure.jsp"></result>
</action>
```

5.修改 SimpleController 工程的控制器代码，当 http 请求某个类型 action 时，控制器检查该 action 是否配置了拦截器。如果有配置， 在 action 执行之前，执行拦截器的 predo()方法，并在 action 执行之后执行拦截器的 afterdo()方法。如果没有配置拦截器，则直接访问目标 action。

在 XmlUtil 工具类中编写 analyzeAction 方法，解析 controller.xml 文件中的每个结点，首先获得所有标签为 action 的结点 list actions，与所有标签为 interceptor 的结点 list interceptors，

```
NodeList actions = doc.getElementsByTagName("action");
NodeList interceptors = doc.getElementsByTagName("interceptor");
```

在 actions 中查找 actionName 结点，若找到保存该结点的所有属性，

```
NamedNodeMap actionNodeMap = actionNode.getAttributes();
String nameString = actionNodeMap.getNamedItem("name").getNodeValue();
String methodString = actionNodeMap.getNamedItem("method").getNodeValue();
String classString = actionNodeMap.getNamedItem("class").getNodeValue();
```

并获得该结点的所有孩子结点 list childs，

```
//get单个节点中的子节点list
NodeList actionChildNodes = actionNode.getChildNodes();
```

在 childs 找到标签为 interceptor-ref 的结点使用其 name 与 interceptors 中的拦截器 name 比较，若找的匹配项，则获得对应拦截器的所有属性，

```
if(actionChildNodes.item(j).getNodeName().equals("interceptor-ref")){
    NamedNodeMap interRefMap = actionChildNodes.item(j).getAttributes();
    String interRefName = interRefMap.getNamedItem("name").getNodeValue();
    for(int k=0;k<interLength;k++){
        Node interNode = interceptors.item(i);
        NamedNodeMap interMap = interNode.getAttributes();
        String interNameString = interMap.getNamedItem("name").getNodeValue();
        String interPreString = interMap.getNamedItem("predo").getNodeValue();
        String interAfterString = interMap.getNamedItem("afterdo").getNodeValue();
        String interClassString = interMap.getNamedItem("class").getNodeValue();

        //拦截器查找成功
        if(interNameString.equals(interRefName)){
```

然后通过 action 的属性获得 action 的 Class 对象与 method 对象，通过拦截器的属性获得其 Class 对象、predo 方法对象与 afterdo 方法对象，先使用反射执行拦截器的 predo 方法，然后使用反射完成对应 action 方法的执行，最后执行拦截器的 afterdo 方法，完成模拟拦截器的实现，最后获得 action 的返回结果，

```
//拦截器
Class interClass = Class.forName(interClassString);
Method interPre = interClass.getMethod(interPreString,String.class);
Method interAfter = interClass.getMethod(interAfterString,String.class);

//Action
Class clazz = Class.forName(classString);
Method method = clazz.getMethod(methodString);

//predo action afterdo
interPre.invoke(interClass.newInstance(),nameString);
Object obj = method.invoke(clazz.newInstance());
resultString = (String)obj;
interAfter.invoke(interClass.newInstance(),resultString);
```

使用结果 result 与其所有 result 子标签的 name 进行比较，若匹配，则获得 result 的标签的所有属性并返回去解析，最后跳转到对应 jsp 页面，

```
for (int j = 0; j < actionChildNodes.getLength(); j++) {
    if (actionChildNodes.item(j).getNodeType() == Node.ELEMENT_NODE) {
        if (actionChildNodes.item(j).getNodeName().toString().equals("result")) {
            NamedNodeMap resultMap = actionChildNodes.item(j).getAttributes();
            String resultName = resultMap.getNamedItem("name").getNodeValue();
            String resultType = resultMap.getNamedItem("type").getNodeValue();
            String resultValue = resultMap.getNamedItem("value").getNodeValue();
            if (resultName.equals(resultString)) {
                return resultType + "," + resultValue;
            }
        }
    }
}
//result不匹配
return "result:failure";
```

6.将任务 5 中的内容通过 Java 的动态代理机制（Java：InvocationHandler，Proxy，cglib）实现。即每次在访问目标 action 时， 先生成该 action 的代理，在代理中实施拦截功能。

使用动态代理机制实现拦截器，由于动态代理只能代理接口，而我们需要调用的 action 方法在 SimpleController 类中只能通过反射完成对 UseSC 的 action 方法的调用，因此需要在 SimpleController 中构建 Action 反射调用方法的封装，定义 Action 接口，

```
public interface Action {
    public String doAction(String s);
}
```

定义 ActionPackage 为 Action 接口的实现类，同时实现 doAction 方法，其中完成反射调用 UseSC 中的 action 方法的功能，

```
@Override
public String doAction(String name) {
    try {
        String fileString = this.getClass().getClassLoader().getResource( name: "controller.xml").getPath();
        XmlUtil xmlUtil = XmlUtil.getInstance();
        String actionMess = xmlUtil.parseXml(fileString, field: "action",name);
        String[] mess = actionMess.split( regex: ",");
        System.out.println("doAction: "+actionMess);
        System.out.println("doAction: mess[0]:"+mess[0]);
        if(mess[0].equals("action:failure")){
            return mess[0];
        }
        Class actionClass = Class.forName(mess[0]);
        Method method = actionClass.getMethod(mess[1]);
        Object obj = method.invoke(actionClass.newInstance());
        System.out.println((String)obj);
        String result = xmlUtil.parseChild(fileString,name,(String)obj);
        return result;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

同时还需要构建动态代理类实现 InvocationHandler 接口，实现其中 invoke 方法，在方法

中解析 xml 判断是否给 action 添加了拦截器，若没有添加则返回原方法，

```
result = method.invoke(target,args);
```

若添加了，则在原调用方法前后添加拦截器的反射调用 UseSC 中拦截器方法的封装方法 preAction 与 afterAction，

```
//检查当前action是否有拦截器
String interName = xmlUtil.parseInterceptor(fileString,actionName);
if(interName.equals("log")){
    //调用前置方法
    preAction();

    //target方法
    result = method.invoke(target,args);
    resultString = (String)result;

    //调用后置方法
    afterAction();
```

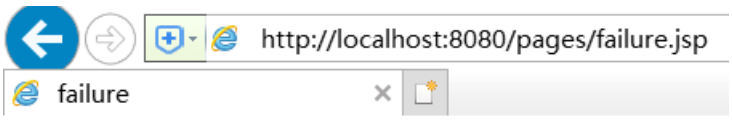
在 preAction 与 afterAction 方法中解析 controller.xml 文件，获得类地址，方法名，构建 Class 对象，构建 method 对象，调用该方法，完成对 UseSC 中的拦截器方法的调用，

```
public void preAction(){
    try {
        XmlUtil xmlUtil = XmlUtil.getInstance();
        String methodString = xmlUtil.parseXml(fileString, field: "interceptor", name: "log", attr: "predo");
        String classString = xmlUtil.parseXml(fileString, field: "interceptor", name: "log", attr: "class");
        System.out.println("preAction: "+classString);
        Class interPreClass = Class.forName(classString);
        Method interPre = interPreClass.getMethod(methodString,String.class);
        interPre.invoke(interPreClass.newInstance(),actionName);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void afterAction(){
    try {
        XmlUtil xmlUtil = XmlUtil.getInstance();
        String methodString = xmlUtil.parseXml(fileString, field: "interceptor", name: "log", attr: "afterdo");
        String classString = xmlUtil.parseXml(fileString, field: "interceptor", name: "log", attr: "class");
        System.out.println("afterAction: "+classString);
        Class interAfterClass = Class.forName(classString);
        Method interPre = interAfterClass.getMethod(methodString,String.class);
        interPre.invoke(interAfterClass.newInstance(),resultString);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

7.自主添加辅助类，以使控制器代码简洁易读。重新打包输出 simple-controller.jar，并测试 UseSC 工程拦截器任务输出结果，直到调试正确。

请求 http://localhost:8080/login.sc



失败

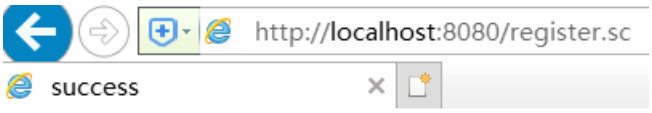
返回 failure，于是重定向到 failure.jsp 中，控制台打印输出，

```
actionName: login startTime: 2018-11-30 21:38:19
执行handleLogin...
actionName: login result: failure startTime: 2018-11-30 21:38:19 endTime: 2018-11-30 21:38:19
```

同时在 log.xml 文件添加 action 标签，

```
<action>
<name>login</name>
<start_time>2018-11-30 21:38:19</start_time>
<end_time>2018-11-30 21:38:19</end_time>
<result>failure</result>
</action>
```

请求 http://localhost:8080/register.sc



成功

返回 success，控制台打印输出，

```
actionName: register startTime: 2018-11-30 21:40:19
执行handleRegister...
actionName: register result: success startTime: 2018-11-30 21:40:19 endTime: 2018-11-30 21:40:19
```

同时在 log.xml 文件中添加 action 标签，

```
<action>
<name>register</name>
<start_time>2018-11-30 21:40:19</start_time>
<end_time>2018-11-30 21:40:19</end_time>
<result>success</result>
</action>
```

5.结论

对主题的总结，结果评论，发现的问题，或你的建议和看法。

本次作业学习了什么是拦截器 Interceptor，学习了拦截器的概念，学习了拦截器的原理，并在 xml 解析过程中模拟了拦截器的过程，同时学习了什么是动态代理，学习了动态代理的概念，学习了如何使用 InvocationHandler 接口与 Proxy 类实现一个动态代理的思路，并查阅资料回答在 MVC pattern 中，Controller 可以具备哪些功能，并描述是否合理？的问题。

第一个问题，如何使用动态代理代理反射获得的对象：

由于使用动态代理代理需要在 SimpleController 中获得具体的类对象，而需要调用的类在 UseSC 中，因此考虑对 SimpleController 中的反射调用过程进行封装，封装成一个类，用来进行动态代理，

```
public class ActionPackage
```

第二个问题，动态代理只能代理接口：

在需要继承 proxy 类获得有关方法和 InvocationHandler 构造方法传参的同时,java 不能同时继承两个类，我们需要和想要代理的类建立联系，只能实现一个接口，让代理类去实现我们所创建的接口，

```
public interface Action {
    public String doAction(String s);
}
```

6.参考文献

1. 参阅 github 学长代码编写风格
2.Java——DOM 方式生成 XML
3.Java 三大器之拦截器(Interceptor)的实现原理及代码示例
4.Java 通过动态代理实现一个简单的拦截器
5.java 的动态代理机制详解
6.Java 设计模式之动态代理（拦截器的应用）
7.java 经典讲解-静态代理和动态代理的区别
8.MVC 中的 Controller 都有哪些作用？
9.jdk 的动态代理及为什么需要接口