

轻量级 J2EE 框架应用

E 3 A Simple Controller with Interceptors

学号: SA16225221

姓名: 欧勇

报告撰写时间: 2016/12/9

1. 主题概述

简要介绍主题的核心内容，作业内容：

1. 定义一个 POJO LogWriter 作为拦截器，在该类中定义方法 log ()，log 方法实现的功能为记录每次客户端请求的 action 名称、类型、访问开始时间、访问结束时间、请求返回结果 result 值，并将信息追加至 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>
```

2. 基于 E2 在 controller.xml 的节点中增加<interceptor>节点作为拦截器定义节点，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"?>
<action-controller>

  <interceptor>
    <name>logWriter</name>
    <class>
      <name>water.servlet.interceptor.LogWriter</name>
      <method>log</method>
    </class>
  </interceptor>

  <!--other interceptors -->

  <action>
    <name>login</name>
    <class>
      <name>water.servlet.action.LoginAction</name>
      <method>login</method>
    </class>
    <interceptor-ref>
      <name>logWriter</name>
    </interceptor-ref>
    <result>
      <name>success</name>
      <type>forward</type>
      <value>pages/success.jsp</value>
    </result>
    <result>
      <name>fail</name>
      <type>redirect</type>
      <value>pages/fail.jsp</value>
    </result>
  </action>
  <action>
    <name>register</name>
    <class>
      <name>water.servlet.action.RegisterAction</name>
      <method>login</method>
    </class>
    <!--other results -->
  </action>
  <!--other actions -->
</action-controller>
```

3. 在<Action>增加<interceptor-ref>节点作为拦截器引用节点，<interceptor-ref>指向<action-controller>中已定义的<interceptor>节点。
4. 当客户端请求某个类型 Action 时，控制器检查该 action 是否配置了 LogWriter 拦截器。如果有配置，在 Action 执行之前，使用 LogWriter 记录 Action 的名称、类型、访问开始时间，并在 Action 执行之后记录 Action 访问结束时间，及返回的结果 result，将以上内容追加到 log.xml。如果无配置，则直接访问 Action。
5. 将任务 3 中的内容通过 Java 的动态代理机制（Java: InvocationHandler, Proxy）实现。即每次在访问 Action 时，先生成该 Action 的代理，在代理中实施请求拦截或日志记录功能。
6. 请分析在 MVC pattern 中，Controller 可以具备哪些功能，并描述是否合理？

2. 假设

主题内容所参照的假设条件，或假定的某故事场景，如调试工具或软硬件环境

开发环境：

Win10

Eclipse kepler

JDK 1.8

Tomcat 7.0

Chrome 浏览器

3. 实现或证明

对主题内容进行实验实现，或例举证明，需描述实现过程及数据。如对 MVC 中 Controller 功能的实现及例证（图示、数据、代码等）

注：本作业报告在第二次作业报告基础上修改添加而成，故测试成功后并没有重新截图，而是采用上次的截图，因为并没有对视图层做任何修改。同时，UserBean 类也没有修改。

流程：

假设用户名为 world，密码为 hello

为了方便查看，采用 get 方式提交，可以通过浏览器 url 看到输入的用户名和密码

（因为若采用 post 方式则无法通过 url 看到用户名和密码，所以采用 get 方式提交）

若登录成功则跳转 login_success.jsp 页面，页面显示 Login Success 的字符串

若登录失败则跳转 login_fail.jsp 页面，页面显示 Login Fail 的字符串

若使用未知的 action 提交，既 `action="unknow.saction"` 则无法找到相应的方法处理，则跳转 error_action.jsp 页面，页面显示“不可识别的 action 请求”提示字符串。

若返回的是未知的处理结果，则返回 error_result.jsp 页面，页面显示“没有请求的资源”提示字符串

图 1：项目目录结构，可以看出项目名称为 SimpleController，src 文件夹下有一个 controller.xml 配置文件，其中记录有 action 和 interceptor 的配置信息，其中 action 标签中会有 result 相关的配置，当启用拦截器时则在 action 中定义 interceptor-ref，否则视为不启用拦截器功能。

同时还有名为 me.king 的包，其下有 LoginController 作为控制层，UserBean 作为模型层，同时定义 LoginInterface 接口，以及动态代理的处理类 LoginActionHandler。

在包 me.king.interceptor 中有负责日志记录的 LogWriter 类。

然后还有 5 个 jsp 页面作为视图层，分别是 login_fail.jsp，login_success.jsp，error_action.jsp，error_result.jsp 和 login.jsp。

注意在 WEB-INF/lib 下需要导入 dom4j 的 jar 包，若仅仅只是将 jar 包放入 Java Resources/Libraries 中，则在编译时能通过，但是却无法完成处理，因为会在执行到语句 `new SAXReader()` 时报如下错误，提示找不到对应的类。

```
root cause
java.lang.NoClassDefFoundError: org/dom4j/io/SAXReader
    me.king.LoginController.doGet(LoginController.java:48)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:728)
    org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)
```

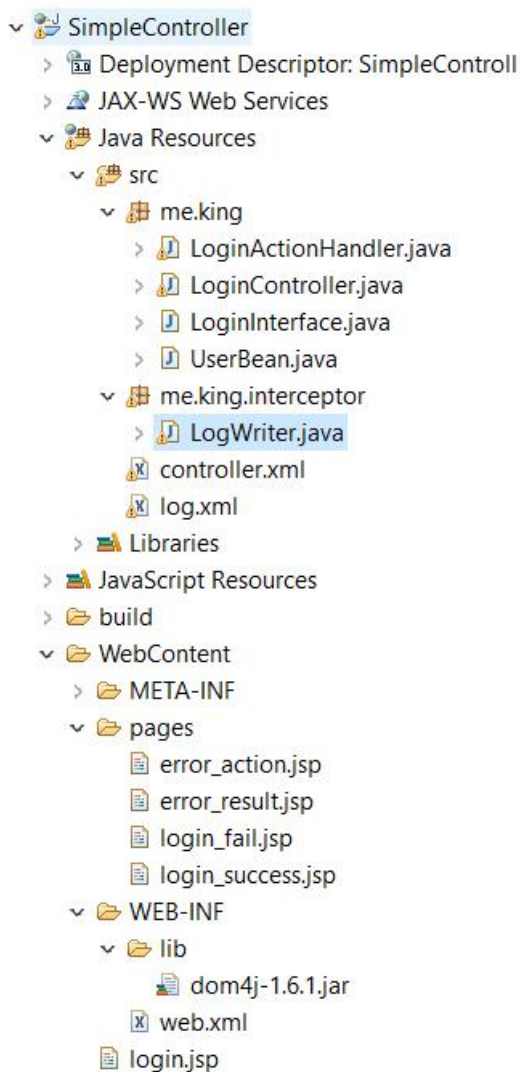


图 1: 项目目录结构

图 2.1: web.xml 配置文件截图

不使用注解的方式告知容器，而是用配置文件的方式配置控制器 LoginController 的映射路径,其中，将 login.jsp 配置为默认页面，将 servlet 控制层类 LoginController 映射名为同类名，同时，对所有以.scaction 结尾的 url 请求进行转发和控制。

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<display-name>SimpleController</display-name>
<welcome-file-list>
  <welcome-file>login.jsp</welcome-file>
</welcome-file-list>
<servlet>
  <servlet-name>LoginController</servlet-name>
  <servlet-class>me.king.LoginController</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>LoginController</servlet-name>
  <url-pattern>*.saction</url-pattern>
</servlet-mapping>
</web-app>

```

图 2.1: web.xml 配置文件截图

图 2.1: LoginController 代码概览截图

在 doGet 方法中进行控制转发，其他采用默认，若前端页面采用 post 方式提交，则在 doPost 方法中也需要进行转发处理，本次采用直接调用 doGet() 方法进行处理。

定义私有 List<Element> 类型变量 actions 和 itcpts，用于保存所有的 action 标签和所有的 interceptor 标签，这两个变量在 init 方法中获取。

同时覆盖 HttpServlet 类中的 init() 方法，用于当有用户首次访问此 servlet 的时候进行配置文件的读取操作，此操作直到本 servlet 销毁之前都会存在，这样后续的其他用户使用此 servlet 时则不会重复读取配置文件 controller.xml 了。

当然这样也无法避免在控制层代码中包含有 dom4j 的解析代码，这时可以写一个 XmlParse 类用于解析 xml 文件，同时返回一个 Action 类和 Interceptor 类用于保存解析的标签信息。

```

2
3* import java.io.IOException;[]
17
18/**
19 * Servlet implementation class LoginController
20 */
21 // @WebServlet("/LoginController")
22 public class LoginController extends HttpServlet {
23     private static final long serialVersionUID = 1L;
24     private List<Element> actions = null; //保存所有的action标签
25     private List<Element> itcpts = null; //保存所有的interceptor标签
26
27     /**
28      * @see HttpServlet#HttpServlet()
29      */
30     public LoginController() {
31         super();
32         // TODO Auto-generated constructor stub
33     }
34
35     public void init() { // 不要重写init(ServletConfig config)方法, []
36
37     }
38
39     protected void doResult(HttpServletRequest request, HttpServletResponse response, Element crtAction, String rstName) throws Exception { []
40
41     }
42
43     /**
44      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
45      * response)
46      */
47     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { []
48
49     }
50
51     /**
52      * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
53      * response)
54      */
55     protected void doPost(HttpServletRequest request,
56         HttpServletResponse response) throws ServletException, IOException {
57         // TODO Auto-generated method stub
58         doGet(request, response);
59     }
60 }

```

图 2.1: LoginController 代码概览截图

图 2.2: LoginController 类 init 方法代码截图

```

35 @Override
36 public void init() { // 不要重写init(ServletConfig config)方法,
37     try { // 获取controller.xml配置文件, 并以流的形式读入
38         InputStream is = this.getClass().getResourceAsStream( "/controller.xml");
39         Document dc = (new SAXReader()).read(is); // 获取文档对象
40         this.actions = dc.getRootElement().elements("action"); // 获取所有的action标签并加入到list中
41         this.itcptps = dc.getRootElement().elements("interceptor"); // 获取所有的interceptor标签并加入到list中
42     } catch (Exception e) {
43         // TODO Auto-generated catch block
44         e.printStackTrace();
45     }
46     System.out.println("初始化读取配置文件controller.xml完成");
47 }

```

图 2.2: LoginController 类 init 方法代码截图

图 2.3: LoginController 类中 doGet 方法代码截图

doGet 中主要控制转发流程为:

1. 获取 request 中的 action 请求名,
2. 遍历 action 标签 list 并判断是否找到需要的 action
3. 若找不到对应 action 请求的时候跳转 error_action 页面并推出 doGet 方法
4. 否则解析当前对应的 action 标签, 获取 name 和 method 标签
5. 判断是否在 action 标签中配置了拦截器 interceptor-ref 标签
6. 若存在 interceptor-ref 标签且非空则去找到对应的 interceptor 标签中设置的 name 和 method
7. 利用反射机制实例化对应的类为 cls 对象
8. 生成处理对象 handler 和代理对象 proxy
9. 获取代理对象中的 LogWriter 类, 使用代理对象调用 method 方法并获取返回结果
10. 当未使用拦截器时则跳过动态代理的部分, 直接使用反射机制获取类和方法并调用。
11. 执行自定义的返回结果给客户端的 doResult 函数

其中, try-catch 中的类型必须为 Exception, 不能为 DocumentException, 否则会在编译的时候报: `Can't load IA 32-bit .dll on a AMD 64-bit platform` 错误

```
java.lang.UnsatisfiedLinkError: D:\Program Files\apache-tomcat-7.0.47\bin\ntnative-1.dll: Can't load IA 32-bit .dll on a AMD 64-bit platform
```

同时, 在最后执行转发操作中, 使用 `response.sendRedirect()` 方法时, 需要使用相对定位, 否则会报: `Path pages/login_success.jsp does not start with a "/" character` 错误, 并且页面显示 404 错误码, 找不到资源。




```

69 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
70     // TODO Auto-generated method stub
71
72     // 获取请求的action
73     String url = request.getRequestURL().toString();
74     1. String actionName = url.substring(url.lastIndexOf('/') + 1, url.lastIndexOf(".")).toString();
75     try { // 已经在init方法中将读入controller.xml的工作完成
76         Element crtAction = null; // 保存当前执行的action标签
77         boolean isFindAction = false;
78         for (Element ele : actions) {
79             2. if (actionName.equals(ele.elementText("name"))) {
80                 isFindAction = true; // 将标志位设置为成功找到该action
81                 crtAction = ele;
82             }
83         }
84         3. if (!isFindAction) { // 当找不到对应action请求的时候跳转error_action页面
85             response.sendRedirect("./pages/error_action.jsp");
86             return; // 退出doGet方法
87         }
88
89         // 则表示找到action, 此时开始解析该action
90         4. String className = crtAction.element("class").elementText("name"),
91             classMethod = crtAction.element("class").elementText("method"), rstName = null, // 用于保存结果字符串
92             userName = request.getParameter("name"), // 用户名
93             userPwd = request.getParameter("pwd"); // 用户密码
94         Class<?> cls = null; // 用于保存反射类
95         Method m = null; // 用于保存反射类的方法
96
97         Element itcpt = crtAction.element("interceptor-ref");
98         5. if (itcpt != null) { // 若存在interceptor-ref标签且非空
99             String itcptCls = null; // 拦截器类
100            String itcptMtd = null; // 拦截器类的方法
101
102            for (Element itcpt : itcpts) { // 找到对应的interceptor的定义
103                6. if (itcpt.elementText("name").equals(itcpt.elementText("name"))) {
104                    itcptCls = itcpt.element("class").elementText("name");
105                    itcptMtd = itcpt.element("class").elementText("method");
106                }
107            }
108            7. cls = Class.forName(className); // 利用反射机制实例化对应的类
109            Object actObj = (LoginInterface) cls.newInstance();
110
111            // 生成处理对象, 传入实例对象, action名, 拦截器类名, 拦截器方法 (此处既logWriter类和log方法)
112            LoginActionHandler handler = new LoginActionHandler(actObj, actionName, itcptCls, itcptMtd);
113            8. Object proxy = Proxy.newProxyInstance( // 生成代理对象
114                actObj.getClass().getClassLoader(), // 传入类加载器,
115                actObj.getClass().getInterfaces(), // 类接口,
116                handler); // 以及真正进行处理的对象
117
118            9. cls = proxy.getClass(); // 此处为LogWriter
119            // 指定获取当前类的classMethod方法, 同时指定参数列表的类型为string, string
120            m = cls.getDeclaredMethod(classMethod, new Class[] {String.class, String.class });
121            rstName = (String) m.invoke(proxy, new Object[] { userName, userPwd });
122        } else { // 未使用拦截器
123            10. cls = Class.forName(className); // 利用反射机制实例化对应的类
124            m = cls.getDeclaredMethod(classMethod, new Class[] { String.class, String.class });
125            rstName = (String) m.invoke(cls.newInstance(), new Object[] { userName, userPwd });
126        }
127
128        11. doResult(request, response, crtAction, rstName); // 将结果返回至客户端
129    } catch (Exception e) {
130        // TODO Auto-generated catch block
131        e.printStackTrace();
132    }
133 }

```

图 2.3: LoginController 类中 doGet 方法代码截图

图 2.4: LoginController 类中 doResult 方法代码截图

在此方法中, 对 result 的类型进行判断并使用不同的方式返回结果视图给客户端
 将返回结果与 result 标签 list 中的对应 name 对比
 获取对应 result 标签中的 value (返回地址), type (跳转还是内部重定向)
 最后执行返回结果, 若返回的字符串类型不匹配, 则返回特定页面

```

49# protected void doResult(HttpServletRequest request, HttpServletResponse response, Element crtAction, String rstName) throws Exception {
50     List<Element> rst = crtAction.elements("result");// 所有的result标签
51     boolean isErrorRst = true;
52     for (Element rstEle : rst) { // 一个rstEle对应一个result节点
53         if (rstName.equals(rstEle.elementText("name"))) { // 当存在对应result返回结果的配置信息的时候
54             isErrorRst = false; // 设置result结果错误标志为false
55             String rstValue = rstEle.elementText("value");
56
57             if (rstEle.elementText("type").equals("forward"))
58                 getServletContext().getRequestDispatcher(rstValue).forward(request, response); // forward为内部重定向
59             else response.sendRedirect('.' + rstValue); // 重定向根据相对地址跳转
60         }
61     }
62     if (isErrorRst) response.sendRedirect("./pages/error_result.jsp"); // 不存在对应result的时候跳转至error_result页面
63 }

```

图 2.4: LoginController 类中 doResult 方法代码截图

```

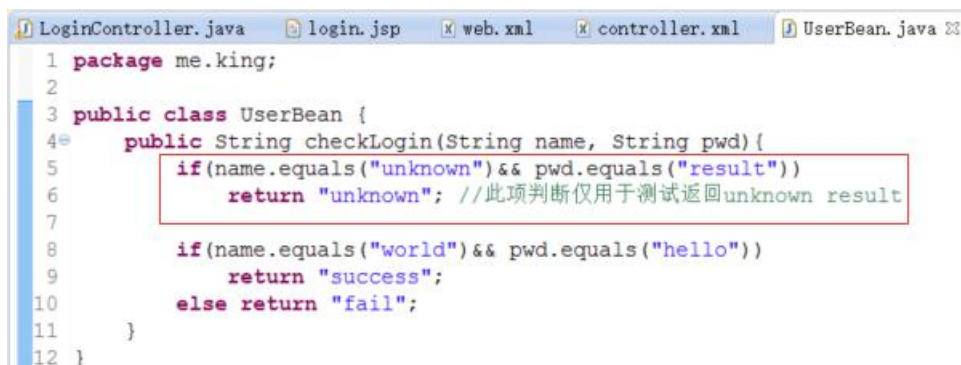
LoginController.java  controller.xml  LogWriter.java
1 <?xml version="1.0" encoding="UTF-8"?>
2 <action-controller>
3   <interceptor>
4     <name>logWriter</name>
5     <class>
6       <name>me.king.interceptor.LogWriter</name>
7       <method>log</method>
8     </class>
9   </interceptor>
10
11  <action>
12    <name>login</name>
13    <class>
14      <name>me.king.UserBean</name>
15      <method>checkLogin</method>
16    </class>
17
18    <interceptor-ref>
19      <name>logWriter</name>
20    </interceptor-ref>
21
22    <result>
23      <name>success</name>
24      <type>forward</type>
25      <value>/pages/login_success.jsp</value>
26    </result>
27    <result>
28      <name>fail</name>
29      <type>redirect</type>
30      <value>/pages/login_fail.jsp</value>
31    </result>
32  </action>
33  <action>
34    <name>register</name>
35  </action>
36 </action-controller>

```

图 2.4: controller.xml 配置文件截图

图 3: UserBean 代码截图，

定义一个简单的 `checkLogin` 方法，参数为 `name` 和 `pwd`，函数体内直接对比两个字符串，若正确则返回“`success`”，否则返回“`fail`”。同时为了测试方便，当 `name` 为 `unknown` 并且 `pwd` 为 `result` 的时候返回“`unknown`”。



```

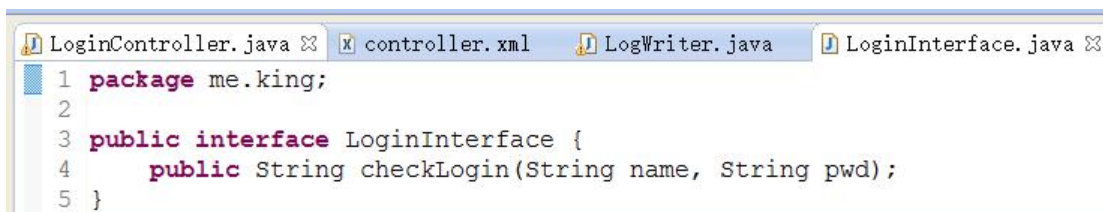
1 package me.king;
2
3 public class UserBean {
4     public String checkLogin(String name, String pwd){
5         if(name.equals("unknown") && pwd.equals("result"))
6             return "unknown"; //此项判断仅用于测试返回unknown result
7
8         if(name.equals("world") && pwd.equals("hello"))
9             return "success";
10        else return "fail";
11    }
12 }

```

图 3: UserBean 代码截图

图 3.1: LoginInterface 接口代码截图

此接口只定义了验证登录操作的方法 `checkLogin`，若使用 JDK 的动态代理机制，则必须采用接口的方式，这样的动态代理的实现方式有缺陷，那就是必须由接口，若由很多不同的类需要采用动态代理的时候，则会使接口爆炸式的增长，不利于维护。这时，可以采用继承的动态代理的实现机制来避免这个问题。具体做法请看参考资料[10]。



```

1 package me.king;
2
3 public interface LoginInterface {
4     public String checkLogin(String name, String pwd);
5 }

```

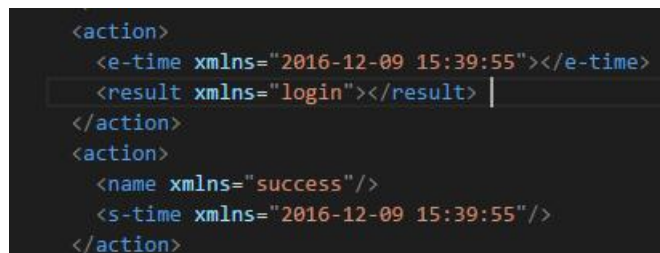
图 3.1: LoginInterface 接口代码截图

图 3.2: LoginActionHandler 代码截图

定义 `LoginActionHandler` 类的构造函数，需要真正执行的类对象，以及用于记录日志的 `action` 名，实际执行 `action` 的类的全名，包括包名，真正需要执行的方法。然后覆盖 `invoke` 方法，需要传入代理类 `proxy` 的实例对象，以及真正执行的方法，以及对应的参数。最后需要将结果返回。

在这里注释有写到：此处不能使用默认的 `act.newInstance()`；否则每次调用 `invoke` 会新建一个 `action` 标签，

即：`actM.invoke(act.newInstance(), this.action, false)`；会出现如下情况：



```

<action>
  <e-time xmlns="2016-12-09 15:39:55"></e-time>
  <result xmlns="login"></result>
</action>
<action>
  <name xmlns="success"/>
  <s-time xmlns="2016-12-09 15:39:55"/>
</action>

```

其中 `name` 和 `result` 分别在一个 `action` 标签中,使用同一个 `obj` 而不是 `actM.newInstance()` 的结果才是正确的在同一个 `action` 标签中:

```
<action>
  <name xmlns="login"/>
  <s-time xmlns="2016-12-09 15:57:17"/>
  <e-time xmlns="2016-12-09 15:57:17"/>
  <result xmlns="unknown"/>
</action>
```

调用处理器的 `handler` 实现类,每次生成动态代理类对象时都需要指定一个实现了 `InvocationHandler` 接口的调用处理器对象。

其中在代理真实对象前,已经在构造函数中初始化了响应的一些对象,包括利用反射初始化的 `LogWriter` 类和 `log` 方法。



```
1 package me.king;
2
3 import java.lang.reflect.InvocationHandler;
4
5 //import me.king.action.LoginAction;
6
7
8
9 //import me.king.action.LoginAction;
10
11 public class LoginActionHandler implements InvocationHandler {
12     private Class<?> act;
13     private Method actM;
14     private Object target; //真正的执行登录验证的对象
15     private String action; //保存执行任务的action名
16     private Object ob; //用于保存logWriter的对象
17
18     public LoginActionHandler(Object target, String action, String actionName, String actionMethod) throws Exception{
19         this.action = action;
20         this.target = target;
21         this.act = Class.forName(actionName); //此处的actionName实际为LogWriter类的全名
22         this.actM = act.getDeclaredMethod(actionMethod, new Class[] {String.class, boolean.class}); //此处为log方法
23         this.ob = act.newInstance();
24     }
25
26     @Override
27     public Object invoke(Object proxy, Method method, Object[] args)
28         throws Throwable {
29         // TODO Auto-generated method stub
30
31         //将action name和传入logWriter,同时true表示是开始记录日志, false表示结束记录日志
32         //同时此处不能使用默认的act.newInstance();否则每次调用invoke会新建一个action标签
33         //actM.invoke(act.newInstance(), this.action, false);
34         actM.invoke(ob, action, true);
35         String result = (String) method.invoke(target, args);
36         actM.invoke(ob, result, false);
37
38         //将真正的执行结果返回给控制器
39         return result;
40     }
41 }
```

图 3.2: LoginActionHandler 代码截图

图 3.3: LogWriter 代码截图

在构造函数中初始化 `log.xml` 的文档对象,以及需要操作的 `action` 标签。在 `log` 方法中执行写入固定格式的日志信息到 `log.xml` 文件中。

在 `tomcat` 内置浏览器测试的时候真正读写的 `log.xml` 的位置并不在 `src` 目录下,而是在一个临时的位置:

```
/E:/DOCUMENT/J2EE/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/SimpleController/WEB-INF/classes/log.xml
```

同时,此处需要注意,当到处为 `war` 包的时候, `Tomcat` 的全目录中,不能存在空格(即 `Tomcat` 不能放在有空格的文件夹名路径下),否则会报错,在 `.../WEB-INF/classes/log.xml` 处找不到文件。

在图 3.4: `log.xml` 截图中,展示了经过一些测试后,日志文件中记录的信息。

分别是:返回未知的 `result` 类型,登录成功,登录失败,返回未知的 `result` 类型。在找

不到 action 的情况下，并不会进入拦截器，而是直接将结果返回客户端了。

```

LoginController.java  controller.xml  LogWriter.java  *LoginActionHandler.java
1  package me.king.interceptor;
2
3  import java.io.File;
17
18  public class LogWriter {
19      private Element crtEle;
20      private Document dc;
21      private String logXml;
22
23      public LogWriter() throws DocumentException { //构造函数, 获取log.xml文件
24          logXml = Thread.currentThread().getContextClassLoader().getResource("").getPath()+"log.xml";
25          //System.out.println(logXml); //测试使用, 输出真正的log.xml的位置
26          dc = (new SAXReader()).read( new File(logXml));
27          crtEle = dc.getRootElement().addElement("action"); //保存当前正在使用的action元素
28      }
29      public void log(String recordStr, boolean isStartTime) throws IOException{
30          SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
31
32          if(isStartTime){ //若是记录开始时间则表示需要创建新的action节点
33              crtEle.addElement("name", recordStr);
34              crtEle.addElement("s-time", sdf.format(new Date()));
35          }else{ //否则操作当前的action节点
36              crtEle.addElement("e-time", sdf.format(new Date()));
37              crtEle.addElement("result", recordStr);
38          }
39          //写入到文件中
40          OutputFormat format = OutputFormat.createPrettyPrint();
41          format.setEncoding("gbk");
42          Writer out = new FileWriter(logXml);
43          XMLWriter writer = new XMLWriter(out, format);
44          writer.write(dc);
45          writer.close();
46          System.out.println("写入logxml完成");
47      }
48  }

```

图 3.3: LogWriter 代码截图

```

log.xml
1  <?xml version="1.0" encoding="gbk"?>
2
3  <log>
4      <action>
5          <name xmlns="login"></name>
6          <s-time xmlns="2016-12-09 16:53:32"></s-time>
7          <e-time xmlns="2016-12-09 16:53:32"></e-time>
8          <result xmlns="unknown"></result>
9      </action>
10     <action>
11         <name xmlns="login"></name>
12         <s-time xmlns="2016-12-09 16:53:56"></s-time>
13         <e-time xmlns="2016-12-09 16:53:56"></e-time>
14         <result xmlns="success"></result>
15     </action>
16     <action>
17         <name xmlns="login"></name>
18         <s-time xmlns="2016-12-09 18:26:30"></s-time>
19         <e-time xmlns="2016-12-09 18:26:30"></e-time>
20         <result xmlns="fail"></result>
21     </action>
22     <action>
23         <name xmlns="login"/>
24         <s-time xmlns="2016-12-09 18:33:38"/>
25         <e-time xmlns="2016-12-09 18:33:38"/>
26         <result xmlns="unknown"/>
27     </action>
28 </log>

```

图 3.4: log.xml 截图

图 4.1: login.jsp 代码截图，
编写 2 个 form 表单，第一个为正常登陆表单 action 设置为“login.scaction”（必须设置为

后缀.scaction，具体配置在 controller.xml 中已经设置完成），设置用户名和密码的 name 属性分别为" name"和"pwd"。

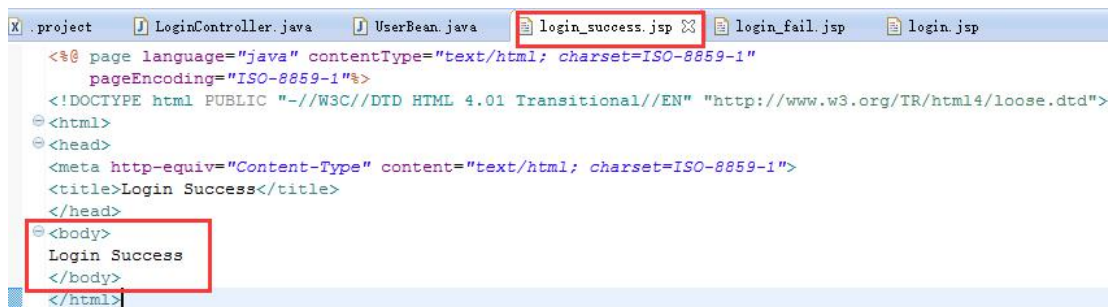
同时为了测试方便，定义一个 action 为"unknown.scaction"的表单，用于测试当 action 为未知的情况。



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9 <body>
10 <form action="login.scaction">
11   <label>NAME:</label><input type="text" name="name"><br>
12   <label>PASSWORD:</label><input type="password" name="pwd"><br>
13   <input type="submit" value="LOGIN">
14 </form>
15 <br><br><br>
16 <form action="unknown.scaction">
17   <label>This is a unknown action request</label><br>
18   <input type="submit" value="unknown">
19 </form>
20
21 </body>
22 </html>
```

图 4.1: login.jsp 代码截图

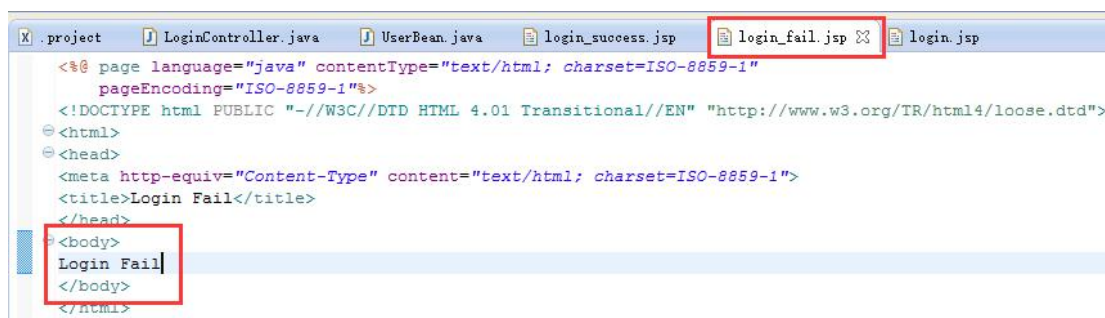
图 4.2: login_success.jsp 代码截图，简单的在 body 中写入 Login Success。



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Login Success</title>
8 </head>
9 <body>
10   Login Success
11 </body>
12 </html>
```

图 4.2: login_success.jsp 代码截图

图 4.3: login_fail.jsp 代码截图，简单的在 body 中写入 Login Fail。



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login Fail</title>
</head>
<body>
Login Fail
</body>
</html>
```

图 4.3: login_fail.jsp 代码截图

图 4.4: error_action.jsp 代码截图

简单的在 body 中写入 Sorry, this is a unrecognized action request.

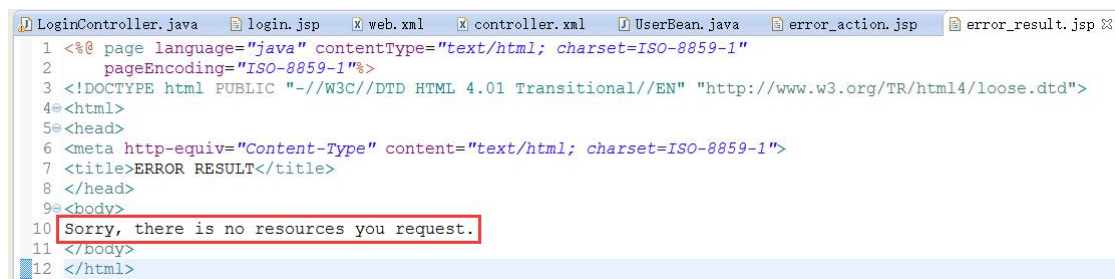


```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERROR ACTION</title>
</head>
<body>
Sorry, this is a unrecognized action request.
</body>
</html>
```

图 4.4: error_action.jsp 代码截图

图 4.5: error_result.jsp 代码截图

简单的在 body 中写入 Sorry, there is no resources you request.



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>ERROR RESULT</title>
8 </head>
9 <body>
10 Sorry, there is no resources you request.
11 </body>
12 </html>
```

图 4.5: error_result.jsp 代码截图

图 5.1: 使用 Chrome 和 Eclipse 内置浏览器测试登录页面截图, 可以看到 url 为: <http://localhost:8080/SimpleController/login.jsp>, 可以在表单中分别填入 NAME 和 PASSWORD, 点击 LOGIN 按钮提交至后台服务器。

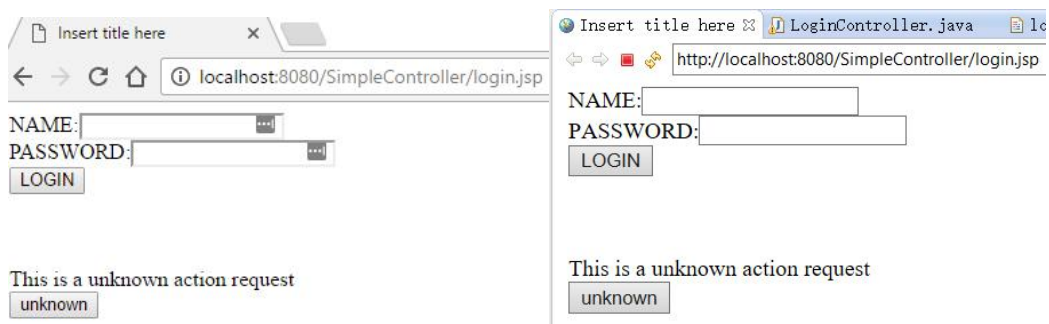


图 5.1：使用 Chrome 和 Eclipse 内置浏览器测试登录页面截图

图 5.2：登录成功截图，

可以通过 url 看到，当输入的用户名 name 为 world，密码 pwd 为 hello 时验证通过时显示 Login Success（虽然页面内容已经变为 login_success.jsp 页面，但此时 url 没有改变为 login_success.jsp 而是在 login.saction 中）



图 5.2：登录成功截图

图 5.3：登录失败截图，

可以通过 url 看到，当输入的用户名 name 或密码 pwd 错误时，验证失败，跳转至 login_fail.jsp 页面，显示 Login Fail

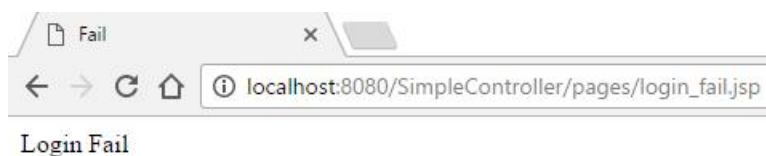


图 5.3：登录失败截图

图 5.4：不可识别的 action 请求截图

当点击图 5.1 中的 unknown 按钮的时候，页面跳转至 error_action.jsp 页面，显示 Sorry, this is a unrecognized action request.

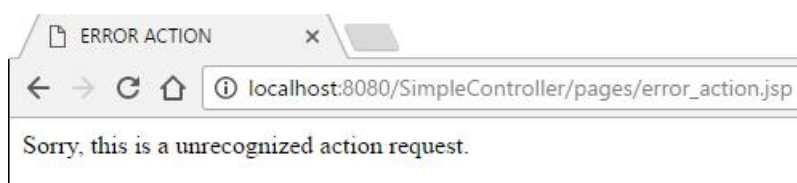


图 5.4：不可识别的 action 请求截图

图 5.5: 没有请求的资源截图

当在登陆表单中输入的用户名 name 为 `unknown`，密码 pwd 为 `result` 时，跳转至 `error_result.jsp` 页面。显示 `Sorry, there is no resources you request.`



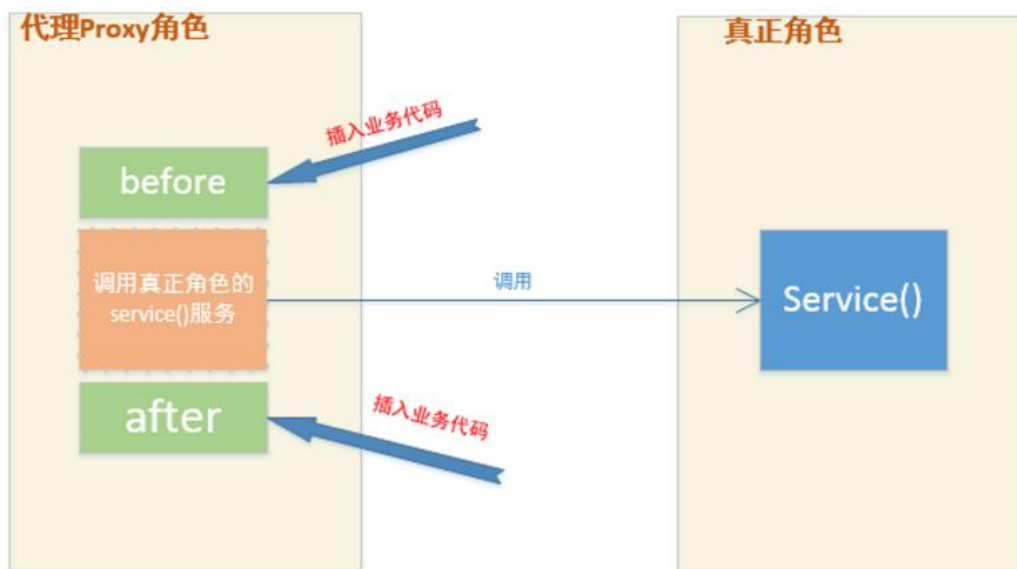
图 5.5: 没有请求的资源截图

4. 结论

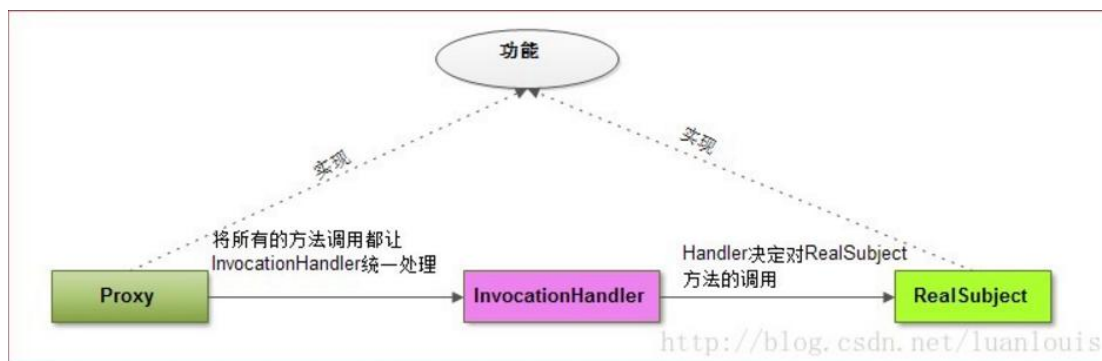
对主题的总结，结果评论，发现的问题，或你的建议和看法。如 MVC 中 Controller 优点与缺点，个人看法（文字、图标、代码辅助等）

关于动态代理，可以通过下面 2 张图对比理解（取自参考资料[10]）：

静态代理示意图：



动态代理示意图：



其中 `InvocationHandler` 是关键的角色，因为有它才从静态代理进化为动态代理。

在静态代理中，代理 `Proxy` 中的方法，都指定了调用了特定的 `realSubject` 中的对应的方法：

在上面的静态代理模式下，`Proxy` 所做的事情，无非是调用在不同的 `request` 时，调用触发 `realSubject` 对应的方法；更抽象点看，`Proxy` 所做的事情；在 Java 中方法（`Method`）也是作为一个对象来看待了，

动态代理工作的基本模式就是将自己的方法功能的实现交给 `InvocationHandler` 角色，外界对 `Proxy` 角色中的每一个方法的调用，`Proxy` 角色都会交给 `InvocationHandler` 来处理，而 `InvocationHandler` 则调用具体对象角色的方法。

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

控制器的作用如下：

主要负责拦截所有用户请求，自定义转发规则，控制处理流程。控制层在服务器端，作为连接视图层（比如用户交互的界面）和模型层（处理业务、提供服务）的纽带，对客户端 request 进行过滤和转发，决定由哪个类来处理请求，或者决定给客户端返回哪个视图，即确定服务器的 response 相对应的视图，将业务和视图分离。

一个 Controller 其实本质上也是一个 servlet，它的实现需要符合 servlet 的标准，即继承 HttpServlet 类，覆写 doGet 或 doPost 等方法，同时可以定义过滤器。

毫无疑问，这对当前的大部分的企业级 Web 服务是合理的，而且是有效的。

但是对于一些超大型的 Web 应用可能分三层就不够了，（比如淘宝，facebook 这样的用户众多，业务及其的 Web 应用）由于业务复杂同时交互对象多，仅仅三层的解耦还不够，因为每一层的业务和代码还是非常的多。这需要对每一层更加细致的细分，即使的控制层也需要继续分为独立的转发，过滤，安全等层。

同时，作为企业级应用，对非功能性的要求也很大，比如可靠性，安全性等，仅仅是三层的 MVC 架构也远远达不到企业要求。

5. 参考文献

以上内容的理论知识点或技术点如果参考了网上或印刷制品，请在这里罗列出来

- [1] Java Reflection : <https://docs.oracle.com/javase/tutorial/reflect/>
- [2] XML Parser SAX : <http://www.saxproject.org/quickstart.html>
- [3] XML Parser DOM : http://www.w3schools.com/dom/dom_parser.asp
- [4] 在 java 中使用 dom4j 解析 xml : <http://www.jb51.net/article/42323.htm>
- [5] 通过 Java 反射调用方法 : <http://blog.csdn.net/ichsonx/article/details/9108173>
- [6] MVC: <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>
- [7] Struts Interceptor : <http://struts.apache.org/release/2.2.x/docs/interceptors.html>
- [8] Java Dynamic Proxy :
<https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html>
- [9] Java JDK 动态代理使用及实现原理分析 :
<http://udn.yyuap.com/thread-128065-1-1.html>
- [10] Java 动态代理机制详解(JDK 和 CGLIB, Javassist, ASM) (清晰, 浅显) :
<http://blog.csdn.net/luanlouis/article/details/24589193>