

轻量级 J2EE 框架应用

E 3 A Simple Controller with Interceptors

学号：SA17225052

姓名：戴赛

报告撰写时间：2017/12/17

1.主题概述

思路整理:

使用 DOM4J 读取 XML 配置文件并实现 Action 和 Result,具体步骤与上 E2 相同

但是在 Action 之前会读取到 Interceptor 的声明信息,读取到这些信息后建立一个存放这些信息的 List 表

之后执行 Action 操作,得到 Action 的 Result 结果,但是不执行相应 Result 操作.接下来再读取该 Action 下的拦截器信息,将拦截器信息与之前 List 中的拦截器做比对,执行相应的拦截器操作.最后再根据 Result 结果执行 Result 操作.

2.假设

上次作业是使用的 SAX 接口读取的 XML 文件,代码较为繁琐.所以这次改用 DOM4J,重写了 E2 的作业.

3.实现或证明

1.

新建 LogInterceptor 类,实现 preAction()和 afterAction()方法

声明 4 个私有变量,保存拦截器信息

```
public class LogInterceptor {  
    private String name;  
    private String s_time;  
    private String e_time;  
    private String result;
```

preAction()方法,记录 Action 的 name 和开始执行时间

```
public void preAction(String input_name,String input_time)  
{  
    // Action执行前执行,记录name和time  
    // 此时的时间为访问开始时间  
    name = input_name;  
    s_time = input_time;  
    System.out.println("拦截器preAction执行完毕");  
}
```

afterAction()方法,记录 Action 的 result 和结束时间,并将拦截器信息保存下来

```
public void afterAction(String input_result,String input_time)  
{  
    // Action执行后执行,记录time和result  
    // 此时的时间为访问结束时间  
    // 同时要写入log  
    result = input_result;  
    e_time = input_time;  
    writeLog();  
    System.out.println("拦截器afterAction执行完毕");  
}
```

信息保存方法 writeLog()

```

private void writeLog()
{
    //      <log>---(0)根节点
    //      <action>-----(1)一级节点
    //      <name>login</name>------(2)二级节点
    //      <s-time>*****</s-time>------(2)
    //      <e-time>####</e-time>------(2)
    //      <result>success</result>------(2)
    //      </action>
    //      </log>
    // 不用加属性
    System.out.println("在写入log前检查四个属性");
    System.out.println("name"+name);
    System.out.println("进入时间"+s_time);
    System.out.println("退出时间"+e_time);
    System.out.println("返回结果"+result);

    System.out.println("开始写入Log文件");
    try {

        Document document = DocumentHelper.createDocument();

        // 根节点<log>,啥都不用加 空的
        Element logElement = document.addElement( s: "log");

        // <action>节点,依然是空的
        Element actionElement = logElement.addElement( s: "action");

        // 添加四个节点,name s_time e_time result,里面有内容(就是加个名字)
        Element nameElement = actionElement.addElement( s: "name");
        nameElement.setText(name);
        Element s_timeElement = actionElement.addElement( s: "s_time");
        s_timeElement.setText(s_time);
        Element e_timeElement = actionElement.addElement( s: "e_time");
        e_timeElement.setText(e_time);
        Element resultElement = actionElement.addElement( s: "result");
        resultElement.setText(result);

        // 格式化
        OutputFormat format = OutputFormat.createPrettyPrint();
        format.setEncoding("UTF-8");

        Writer fileWriter = new FileWriter( fileName: "f:\\\\log.xml");
        // 写入文件的对象XMLWriter
        // XMLWriter xmlWriter = new XMLWriter(fileWriter);
        XMLWriter xmlWriter = new XMLWriter(fileWriter,format);
        xmlWriter.write(document);
        xmlWriter.flush();
        xmlWriter.close();
        System.out.println("xml文档添加成功!");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

2.和 3.

controller.xml 的修改

```
<?xml version='1.0' encoding='UTF-8'?>
<sc-configuration xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
  <interceptor name="log" class="water.ustc.interceptor.LogInterceptor" predo="preAction" afterdo="afterAction"></interceptor>
  <controller>
    <action name="loginAction.sc" class="water.ustc.action.LoginAction" method="handleLogin">
      <interceptor-ref name="log"></interceptor-ref>
      <result name="success" type="forward" value="/welcome.jsp"></result>
      <result name="failure" type="redirect" value="/failure.jsp"></result>
    </action>
    <action name="registerAction.sc" class="water.ustc.action.RegisterAction" method="handleRegister">
      <interceptor-ref name="log"></interceptor-ref>
      <result name="success" type="forward" value="/login.jsp"></result>
    </action>
  </controller>
</sc-configuration>
```

4.

SimpleController 中不使用动态代理机制实现拦截器效果

获得位于根节点下的<interceptor>节点信息,并将其中的拦截器信息存储到单独的一个类中,考虑到可能会有多个拦截器,所以这里使用了 List

```
List<Element> interceptorList = rootElement.elements( $: "interceptor");
// System.out.println("*****");
List<InterceptorAttribute> interceptorAttributeList = new ArrayList<InterceptorAttribute>(); // 存放过滤器信息
System.out.println("    (1)层某单一节点名称为:interceptor");
System.out.println("    其总数为:"+interceptorAttributeList.size());
for (Element interceptor:interceptorList)
{
    System.out.println("***");
    InterceptorAttribute interceptorAttribute_temp = new InterceptorAttribute();
    interceptorAttribute_temp.setName(interceptor.attributeValue( $: "name"));
    // System.out.println("存放过滤器信息:<"+interceptor.attributeValue( "name")+><name>属性完毕");
    interceptorAttribute_temp.setClazz(interceptor.attributeValue( $: "class"));
    // System.out.println("存放过滤器信息:<"+interceptor.attributeValue( "name")+><class>属性完毕");
    interceptorAttribute_temp.setPredo(interceptor.attributeValue( $: "predo"));
    // System.out.println("存放过滤器信息:<"+interceptor.attributeValue( "name")+><predo>属性完毕");
    interceptorAttribute_temp.setAfterdo(interceptor.attributeValue( $: "afterdo"));
    // System.out.println("存放过滤器信息:<"+interceptor.attributeValue( "name")+><afterdo>属性完毕");

    interceptorAttributeList.add(interceptorAttribute_temp);
    System.out.println("    存放过滤器信息:<"+interceptor.attributeValue( $: "name")+>完毕");
    System.out.println("    其属性个数为:"+interceptor.attributeCount());
    System.out.println("***");
}
```

存放拦截器信息的 `InterceptorAttribute` 类

```
public class InterceptorAttribute {  
    private String name;  
    private String clazz;  
    private String predo;  
    private String afterdo;  
  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public String getClazz() { return clazz; }  
    public void setClazz(String clazz) { this.clazz = clazz; }  
    public String getPredo() { return predo; }  
    public void setPredo(String predo) { this.pred = predo; }  
    public String getAfterdo() { return afterdo; }  
    public void setAfterdo(String afterdo) { this.afterdo = afterdo; }  
}
```


在获得 Aciton 的 result 后,先判断该 Action 下是否有拦截器,若有拦截器则将拦截器其与之前 List 中的拦截器信息进行匹配,匹配成功则顺序执行拦截器 preAction,执行完拦截器后再执行 result

```
// 反射后,获取执行对应类的对应方法的结果
String returnResult = (String) actionMethod.invoke(actionObject, username, pwd);
System.out.println("反射后,获取执行对应类的对应方法的结果为"+returnResult);

// 虽然已经返回了Action结果,但是还不执行,这里开始执行拦截器进入pre方法
List<String> interceptorNameList = hasInterceptor(action);
if (!interceptorNameList.isEmpty())
{
    // 在result前执行拦截器
    for (String interceptorName : interceptorNameList) {
        // 从之前的过滤器信息List中找出该Action下的拦截器信息
        // System.out.println("@@@@@@@暂存列表中的拦截器名称"+interceptorName);
        for (InterceptorAttribute interceptor : interceptorAttributeList) {
            if (interceptorName.equals(interceptor.getName()))
            {
                // 获取当前时间并转化成字符串
                Date currDate = Calendar.getInstance().getTime();
                SimpleDateFormat sdf = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");
                String dateTime = sdf.format(currDate);

                System.out.println("现在执行拦截器preDo方法");
                System.out.println("此时时间为"+dateTime);
                System.out.println(interceptorName+"匹配成功");
                // System.out.println("假装执行拦截器~~~~~");

                // 对拦截器进行反射
                Class interClass = Class.forName(interceptor.getClazz());
                interObject = interClass.newInstance(); // 使用全局变量
                Method preDoMethod = interClass.getDeclaredMethod(interceptor.getPreDo(), String.class, String.class);
                preDoMethod.invoke(interObject,interceptor.getName(),dateTime);

                // System.out.println("这里做反射,找到拦截器的class" + interceptor.getClazz());
                // System.out.println("这里先执行preDo方法"+interceptor.getPreDo());
            }
        }
    }
    System.out.println("~~~~~下面是result~~~~~");
}
```

判断是否有拦截器的方法 hasInterceptor(),考虑到可能有多个拦截器,故其返回结果是一个 List,若无拦截器则返回一个空 List

```
private List<String> hasInterceptor(Element action)
{
    // 判断该action下是否有有拦截器,并返回拦截器名称List
    List<String> returnList = new ArrayList<String>();
    if (action.element("interceptro-ref")!= null)
    {
        System.out.println("!!!!!!");
        System.out.println(action.attributeValue( $: "name")+ "下具有拦截器:");
        List<Element> inter_refList = action.elements( $: "interceptro-ref");
        for (Element inter_ref:inter_refList)
        {
            System.out.println("其名称为:"+inter_ref.attributeValue( $: "name"));
            String inter_refName = inter_ref.attributeValue( $: "name");
            returnList.add(inter_refName);
        }
    }
    else
    {
        System.out.println(action.attributeValue( $: "name")+ "下没有拦截器");
    }
    // System.out.println(action.attributeValue("name")+ "!!!!!!判断拦截器函数返回结果为:"+returnList);
    // System.out.println("%%%%%%%%%%");
    System.out.println("!!!!!!");
    return returnList;
}
```

之后执行相应的 result

```
// 获得(3)层节点<result>的List
List<Element> resultList = action.elements( $: "result");
doResult(resultList,xmlActionName,returnResult,request,response);

// result执行完毕 逆序执行过滤器
```


result 执行方法 doResult()

```

private void doResult(List<Element> resultList,String xmlActionName,String returnResult,HttpServletRequest request, HttpServletResponse response)
{
    try {
        boolean resultNotInXML = true; // 是否有在xml中找到result
        for (Element result:resultList)
        {
            String resultName = result.attributeValue( "name");
            System.out.println("XML中+xmlActionName+Action的结果有"+resultName);
            if (resultName.equals(returnResult))
            {
                // 有找到result
                resultNotInXML = false;
                String resultType = result.attributeValue( "type");
                System.out.println("有找到Result!!!!!!xml中result的type为:"+resultType);
                String resultValue = result.attributeValue( "value");
                System.out.println("其value为:"+resultValue);
                if (resultType.equals("forward"))
                {
                    // 转发
                    request.getRequestDispatcher(resultValue).forward(request,response);
                }
                else if(resultType.equals("redirect"))
                {
                    // 重定向
                    response.sendRedirect( " " request.getContextPath()+resultValue);
                }
                else
                {
                    System.out.println("未查到相关的result类型");
                }
            }
            else
            {
                System.out.println("未在xml中找到对应的result结果,继续查询result");
                System.out.println("-----继续查询Result-----");
                // response.sendError(501,"没有请求的资源");
            }
        }
        if (resultNotInXML) response.sendError( " 501, " "没有请求的资源");
    } catch (ServletException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Result 执行完毕后,在执行拦截器的 afterAction(),因为 result 结束后拦截器的执行顺序是先执行的后结束,所以逆序执行

```

// result执行完毕,逆序执行过滤器
if (!interceptorNameList.isEmpty())
{
    // 在result后执行拦截器
    // System.out.println("gggggggggggg可执行过滤器列表长度为"+interceptorNameList.size());
    for (int interNum=(interceptorNameList.size()-1);interNum<interceptorNameList.size();&&interNum>=0;interNum--) {
        // 从之前的过滤器信息List中找出该Action下的拦截器信息
        // System.out.println(interNum);
        String interName_temp = interceptorNameList.get(interNum);
        System.out.println("action退出时的拦截器为:"+interName_temp);
        for (InterceptorAttribute interceptor : interceptorAttributelist) {
            if (interName_temp.equals(interceptor.getName()))
            {
                Date currDate = Calendar.getInstance().getTime();
                SimpleDateFormat sdf = new SimpleDateFormat( "pattern: yyyy-MM-dd HH:mm:ss");
                String dateTime = sdf.format(currDate);
                System.out.println("现在执行拦截器afterDo方法");
                System.out.println("此时时间为"+dateTime);
                // 对拦截器进行反射
                Class interClass = Class.forName(interceptor.getClazz());
                // Object interObject = interClass.newInstance();
                Method preDoMethod = interClass.getDeclaredMethod(interceptor.getAfterdo(), String.class, String.class);
                preDoMethod.invoke(interObject,returnResult,dateTime);
                System.out.println("拦截器执行完毕");
            }
        }
    }
}

```

最后结果

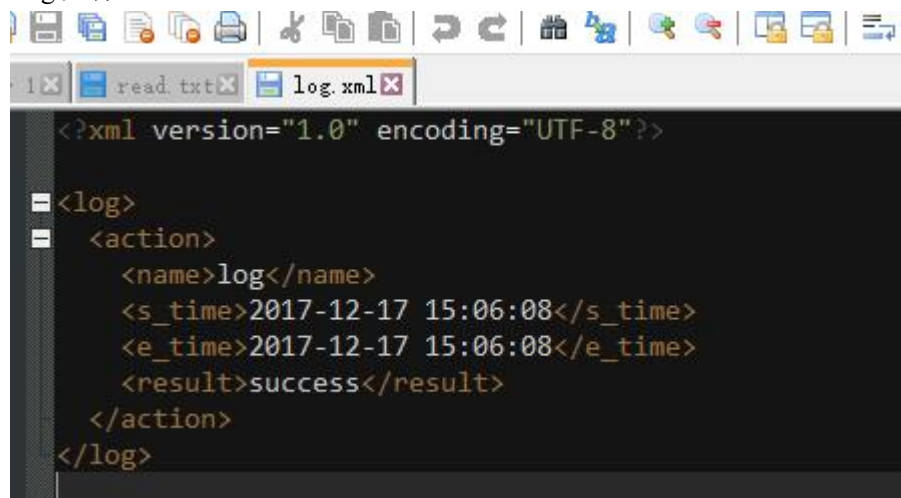
原 Action 正常执行

登陆成功！欢迎您！

拦截器和 Result 正常执行

```
-----继续查询Result-----  
action退出时的拦截器为:log  
现在执行拦截器afterDo方法  
此时时间为2017-12-17 15:06:08  
在写入log前检查四个属性  
namelog  
进入时间2017-12-17 15:06:08  
退出时间2017-12-17 15:06:08  
返回结果success  
开始写入Log文件  
xml文档添加成功！  
拦截器afterAction执行完毕  
~~~~~拦截器执行完毕~~~~~
```

Log 文件



The screenshot shows a web browser window with a toolbar at the top. Below the toolbar, there are three tabs: '1x', 'read.txt', and 'log.xml'. The 'log.xml' tab is active, displaying the following XML content:

```
<?xml version="1.0" encoding="UTF-8"?>  
  
=<log>  
=<action>  
  <name>log</name>  
  <s_time>2017-12-17 15:06:08</s_time>  
  <e_time>2017-12-17 15:06:08</e_time>  
  <result>success</result>  
  </action>  
</log>
```

5.

使用动态代理的方式实现拦截器

这里选用 `InvocationHandler` 接口

创建一个接口 `ITask`

```
public interface ITask {
    void doSomething(List<Element> resultList, String xmlActionName, String returnResult, HttpServletRequest request, HttpServletResponse response);
}
```

创建类 `IResult` 实现 `ITask` 接口(`IResult` 的 `doSomething()` 方法主要内容与需求 4 中的 `doResult()` 基本相同)

```
public class IResult implements ITask {
    @Override
    public void doSomething(List<Element> resultList, String xmlActionName, String returnResult, HttpServletRequest request, HttpServletResponse response) {
        try {
            boolean resultNotInXML = true; // 是否有在xml中找到result
            for (Element result:resultList) {
                String resultName = result.attributeValue("name");
                System.out.println("XML中"+xmlActionName+"Action的结果有"+resultName);
                if (resultName.equals(returnResult)) {
                    // 有找到result
                    resultNotInXML = false;
                    String resultType = result.attributeValue("type");
                    System.out.println("有找到Result!!!!!!xml中result的type为:"+resultType);
                    String resultValue = result.attributeValue("value");
                    System.out.println("其value为:"+resultValue);
                    if (resultType.equals("forward")) {
                        // 转发
                        request.getRequestDispatcher(resultValue).forward(request, response);
                    } else if (resultType.equals("redirect")) {
                        // 重定向
                        response.sendRedirect(request.getContextPath()+resultValue);
                    } else {
                        System.out.println("未查到相关的result类型");
                    }
                } else {
                    System.out.println("未在xml中找到对应的result结果,继续查询result");
                    System.out.println("-----继续查询Result-----");
                    // response.sendError(501, "没有请求的资源");
                }
            }
            if (resultNotInXML) response.sendError(501, "没有请求的资源");
        } catch (ServletException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

创建类 `InterDynProxy` 实现 `InvocationHandler` (写的仓促,许多代码是需求 4 中复制来的,代码比较乱)

```
public class InterDynProxy implements InvocationHandler {
    private Object obj;
    private static InterceptorAttribute interceptor;
    private static String returnResult;

    private InterDynProxy(Object object) { obj = object; }

    public static Object newInstance(Object obj, InterceptorAttribute inter, String result) {
        interceptor = inter; // 拦截器属性信息赋值
        returnResult = result; // 将要执行的Result方法

        return Proxy.newProxyInstance(obj.getClass().getClassLoader(), obj.getClass().getInterfaces(), new InterDynProxy(obj));
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 这里可以写拦截逻辑
        return method.invoke(obj, args);
    }
}
```

重写 invoke()方法

```

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object result;

    // 创建拦截器类
    Object interObject = null;
    try {
        // 可以将所有方法相同的一些操作放在这里处理
        System.out.println("--before method " + method.getName());
        // preAction内容
        Date currDate = Calendar.getInstance().getTime();
        SimpleDateFormat sdf = new SimpleDateFormat( "pattern: \"yyyy-MM-dd HH:mm:ss\"");
        String dateTime = sdf.format(currDate);

        System.out.println("现在执行拦截器preDo方法");
        System.out.println("此时时间为"+dateTime);
        // System.out.println(interceptorName+"匹配成功");
        // System.out.println("假装执行拦截器~~~~~");

        // 对拦截器进行反射
        Class interClass = Class.forName(interceptor.getClazz());
        interObject = interClass.newInstance();
        Method preDoMethod = interClass.getDeclaredMethod(interceptor.getPreDo(), String.class, String.class);
        preDoMethod.invoke(interObject, interceptor.getName(), dateTime);

        result = method.invoke(obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    } catch (Exception e) {
        throw new RuntimeException("unexpected invocation exception: " + e.getMessage());
    } finally {
        System.out.println("--after method " + method.getName());
        // afterAction内容
        Date currDate = Calendar.getInstance().getTime();
        SimpleDateFormat sdf = new SimpleDateFormat( "pattern: \"yyyy-MM-dd HH:mm:ss\"");
        String dateTime = sdf.format(currDate);
        System.out.println("现在执行拦截器afterDo方法");
        System.out.println("此时时间为"+dateTime);
        // 对拦截器进行反射
        Class interClass = Class.forName(interceptor.getClazz());
        // Object interObject = interClass.newInstance();
        Method preDoMethod = interClass.getDeclaredMethod(interceptor.getAfterdo(), String.class, String.class);
        preDoMethod.invoke(interObject, returnResult, dateTime);
        System.out.println("~~~~~拦截器执行完毕~~~~~");
    }

    return result;
}

```

SimpleController 中实现动态代理

前面代码与 E4 相同,构建两个列表保存拦截器信息,之后在比对,不过执行是只用生成代理就可以了,不用再写两个循环

```

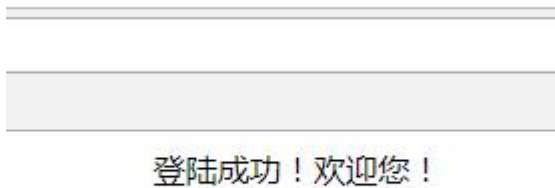
// 获得(3)层节点<result>的List
List<Element> resultList = action.elements( 3, "result");

// 虽然已经返回了Action结果,但是还不执行,这里开始执行拦截器进入pre方法
List<String> interceptorNameList = hasInterceptor(action);
if (!interceptorNameList.isEmpty())
{
    // 在result前执行拦截器
    for (String interceptorName : interceptorNameList) {
        // 从之前的过滤器信息List中找出该Action下的拦截器信息
        // System.out.println("@@@@@暂存列表中的拦截器名称"+interceptorName);
        for (InterceptorAttribute interceptor : interceptorAttributeList) {
            if (interceptorName.equals(interceptor.getName()))
            {
                ITask task = (ITask) InterDynProxy.newInstance(new IResult(), interceptor, returnResult); // 生成动态代理对象,并传入相关参数
                task.doSomething(resultList, xmlActionName, returnResult, request, response); // 执行result方法
            }
        }
    }
    // System.out.println("~~~~~下面是result~~~~~");
}
}

```

最后结果:

页面跳转正常执行



输出结果(输出有些乱)

```
现在执行拦截器preDo方法
此时时间为2017-12-17 15:29:31
preAction执行完毕
XML中loginAction.scAction的结果有success
有找到Result!!!!!!xml中result的type为:forward
其value为:/welcome.jsp
XML中loginAction.scAction的结果有failure
未在xml中找到对应的result结果,继续查询result
-----继续查询Result-----
--after method doSomething
现在执行拦截器afterDo方法
此时时间为2017-12-17 15:29:31
开始写入log文件
```

Log

```
<?xml version="1.0" encoding="UTF-8"?>
<log>
  <action>
    <name>log</name>
    <s_time>2017-12-17 15:29:31</s_time>
    <e_time>2017-12-17 15:29:31</e_time>
    <result>success</result>
  </action>
</log>
```

7.

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

答:

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 骑士本质上也是个 servlet,它的实现需要符合 servlet 的标准,即 HttpServlet 类,复写 doGet 和 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.

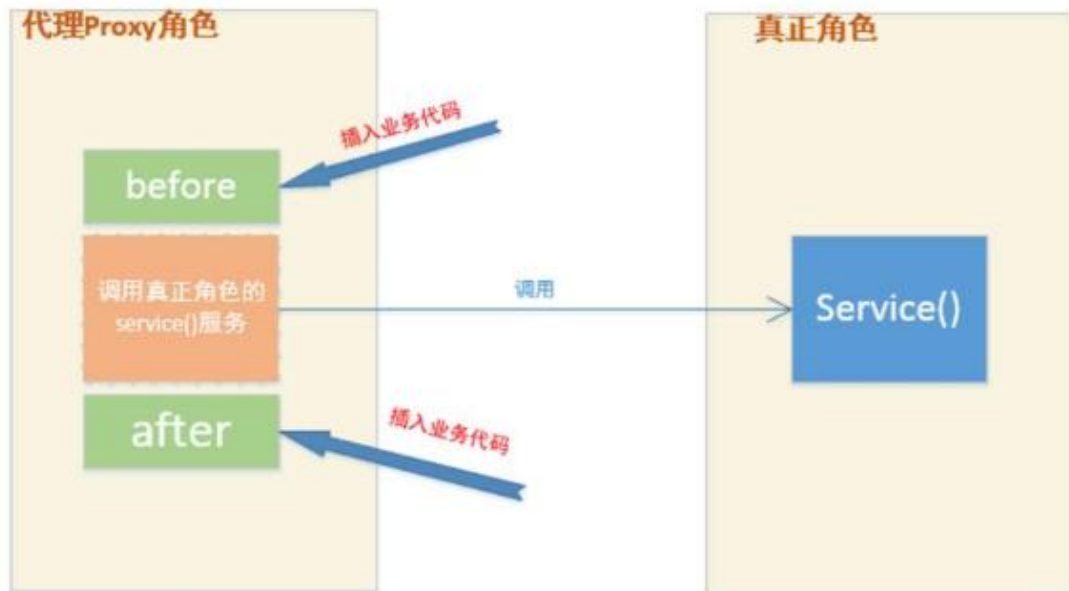
但是对于一些超大型的 Web 应用可能分三层就不够了,(比如淘宝,亚马逊这样的用户众多,业务及其 Web 应用)由于业务复杂同时交互对象多,仅仅三层解耦还不够,因为每一次的业务和代码还是很多的.这需要对每一层的业务和代码还是非常多的.这需要对每一层更加细致的细分,即使的控制层也需要继续分为独立的转发,过滤,安全等层

同时,作业企业级应用,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.

4. 结论

这次作业的难点是动态代理部分,其具体解释如下图所示

静态代理示意图



动态代理示意图



)

5.参考文献

Dom4j 解析和生成 XML 文档

<http://blog.csdn.net/chenghui0317/article/details/11486271>

关于 InvocationHandler 动态代理

<http://blog.csdn.net/xiaoyg830/article/details/51842562>

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