

# 轻量级 J2EE 框架应用

## E 4 A Simple Controller with handling result view

学号：SA18225170

姓名：李朝喜

报告撰写时间：2018/12/22

## 文档目录

### 1. 主题概述

- 1.1 Result View
- 1.2 XSLT

### 2. 假设

- 2.1 知识背景
- 2.2 环境背景

### 3. 实现或证明

- 3.1 项目包结构及简要描述
- 3.2 UML 类图
- 3.3 请求处理流程图
- 3.4 success\_view.xml
- 3.5 success\_view.xsl
- 3.6 xml+xsl 转为 html
- 3.8 handleResult（结果分发）
- 3.7 测试结果
- 3.8 Struts 框架中试图组件的工作方式

### 4. 结论

- 4.1 总结
- 4.2 问题及看法

### 5. 参考文献

# 1. 主题概述

## 1.1 Result View

**Result View**，可简单理解为浏览器发送请求后对方响应的页面。在 **JavaWeb** 开发中，通常是后端根据请求处理结果来决定要返回的视图（转发或重定向）。返回的视图可能是已经编好的静态页面，或者是基于某种规则动态生成的页面。在 **MVC** 中，**Result View** 属于其中的 **V**（视图层）部分。

## 1.2 XSLT

**XSL** 称为：扩展样式表达语言（**EXtensible Stylesheet Language**），而 **XSLT** 则是指 **XSL** 转换。利用该技术可将 **XML** 数据文档转换为其他格式，如：**HTML** 页面、纯文字内容等。

**XSL** 不仅仅是样式表达语言，其包括三部分：

- **XSLT**：一种用于转换 **XML** 文档的语言；
- **XPath**：一种用于在 **XML** 文档中导航的语言；
- **XSL-FO**：一种用于格式化 **XML** 文档的语言；

本次实验中，主要用到了 **XPath** 和 **XSLT**。具体内容可参考：

- **XPath**：<http://www.w3school.com.cn/xpath/index.asp>
- **XSLT**：<http://www.w3school.com.cn/xsl/index.asp>

## 2. 假设

需要注意，实验 E4 是在 E3 基础上进行的，所以在做 E4 前请先完成 E3，否则可能对项目上下文等内容不清楚。

### 2.1 知识背景

根据我的实际情况，我认为在至少具备以下知识背景的前提下进行实验，更有助于较好的完成实验。

- 理解 Java 基础知识，并且具备基本的 Java 编程能力，如：理解 Java 中对象封装、继承等概念，以及具备 JavaSE 基本开发能力；
- 了解代理模式，能够使用 Java 实现静态和动态代理方式；
- 了解 Java 反射机制，能够利用 Java 反射来调用某个类中的方法；
- 了解 XSLT 相关技术，如：XPath、XSL、XSLT 等；

### 2.2 环境背景

本次实验是在 Windows 10 系统下进行的，部分操作可能对其他系统并不适用。此外，请确保你的电脑至少有 2G 的内存，否则在同时运行多款软件时，可能会出现卡顿现象。

具体要求：

- 已安装并且配置好 Java 环境：[Java SE Development Kit 11.0.1](#)
- 已安装好 Eclipse：[eclipse-jee-2018-09-win32-x86\\_64.zip](#)
- 已安装好 Tomcat：<https://tomcat.apache.org/download-90.cgi>
- 已安装好 Maven：<https://maven.apache.org/download.cgi>

注意：软件安装以及相关配置不在本文档涉及范围内，对于如：Java 环境变量配置、Maven 修改镜像源、Eclipse 中配置 Maven 等问题，请读者自行查阅资料。

### 3. 实现或证明

由于之前想法有误，所以在 E4 中重构了 E3 的不少代码（大约 60%），现在 E4 中的包、类结构与 E3 的有很大区别。重构的内容主要包括：

#### SimpleController 工程：

- 1) 添加了 action 包，定义了 Actionface（接口）和 ActionSupport（父类），让 UseSC 中的 Action 实现或继承该包中的内容，将两个工程中的 Action 归为统一体系；
- 2) 将 bean 包放到 config 包下，表示其是与配置（controller.xml）有关的 bean；
- 3) 删除了基于原生 DOM 实现的 DOMControllerFactory；并且将最高抽象 Factory<T>提出来单独放到 factory 包中；
- 4) 删除了 LogInvocationHandler 类，改为使用 cglib 的方式来实现动态代理；
- 5) 添加了 CommonUtil 类，将 FileUtil 中的 CLASS\_PATH 属性放到 CommonUtil 下；
- 6) 删除了 servlet 包，修改了 web.xml，控制器统一由 SimpleController 类来承担；
- 7) 删除了 service 包，将业务层的内容（service）放 UseSC 工程中实现；

#### UseSC 工程：

- 1) 添加了 bean 包，定义了 User 类，表示 PO 类；
- 2) 添加了 service 包及子包 impl，用于业务逻辑的处理，UserService 使用单例实现；
- 3) 修改了 LoginAction 和 RegisterAction 类的实现，将这两个类定义为 ActionBean，通过调用 service 来实现业务逻辑处理；

#### 3.1 项目包结构及简要描述

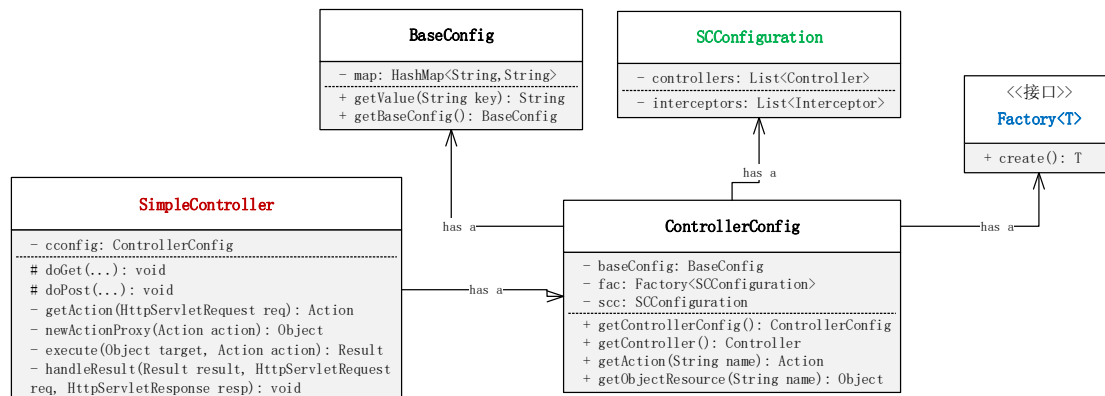
##### SimpleController

- src/main/java	# 包根目录
- southday.j2eework.sc.ustc.controller	# Servlet, 用于处理*.sc的请求
- SimpleController.java	# action包, 里面包含了Action接口定义及默认实现类等
- action	# Action接口
- Actionface.java	# Action的默认实现类, 框架使用者可以继承该类
- ActionSupport.java	# 默认Action, 当未找到匹配的action时, 就返回DefaultAction
- DefaultAction.java	# Config包, 里面包含了项目所用的公共资源对象类, 通常是单例实现
- config	# 基本资源配置类, 包括UseSC中资源文件(jsp、.xml)的位置, 以及一些公共使用的配置属性
- BaseConfig.java	# Controller对象配置类, 通过扫描UseSC中的controller.xml文件, 来获得全局共享的Controller对象
- ControllerConfig.java	# bean包, 包含与UseSC中controller.xml文件里定义内容相对应的JavaBean对象类
- bean	# 与controller.xml中定义的<controller>标签相对应
- Controller.java	# 与controller.xml中定义的<action>标签相对应
- Action.java	# 与controller.xml中定义的<result>标签相对应
- Result.java	# 与controller.xml中定义的<interceptor>标签相对应
- Interceptor.java	# 与controller.xml中定义的<interceptor-ref>标签相对应
- InterceptorRef.java	# 与controller.xml中定义的<sc-configuration>标签相对应
- SCConfiguration.java	# factory包, 工厂模式, 里面包含了用于创建对象的工厂类
- factory	# 用于生成Action对象的工厂
- ActionFactory.java	# 用于生成Result对象的工厂
- ResultFactory.java	# 用于生成Controller对象的工厂
- ControllerFactory.java	# 用于生成SCConfiguration对象的工厂
- SCConfigurationFactory.java	# dom4j包, 基于DOM4J的XML解析来实现SCConfiguration对象创建的工厂
- dom4j	# 基于DOM4J的XML解析来创建SCConfiguration对象的具体实现类
- DOM4JSCConfigurationFactory.java	# factory包, 目前包含工厂的抽象定义(接口Factory)
- factory	# 抽象工厂, 提供接口: T create() throws Exception;
- Factory.java	# interceptor包, 用于存放拦截器的相关类、接口等
- interceptor	# 参数拦截器, 目前仅用于给Action代理对象(动态)填充参数
- ParameterInterceptor.java	# proxy包, 包含与代理机制相关的类
- proxy	# 代理工厂, 用于生成代理类对象
- ProxyFactory.java	# cglib包, 关于Action的动态代理, 使用cglib技术来实现
- cglib	# 针对日志记录的Action代理拦截器
- LogActionProxy.java	# transformer包, 包含各类转换器
- transformer	# 将xml转为html的转换器, 目前使用xslt技术实现
- XML2HTMLTransformer.java	# util包, 包含项目工具类
- util	# 包含普遍被使用的、公用的方法, 如: 获取类加载路径、检查String类型参数等
- CommonUtil.java	# 包含与文件资源处理相关的常用方法, 如: 关闭资源, 加载properties配置等
- FileUtil.java	# 包含与反射处理相关的常用方法
- ReflectUtil.java	

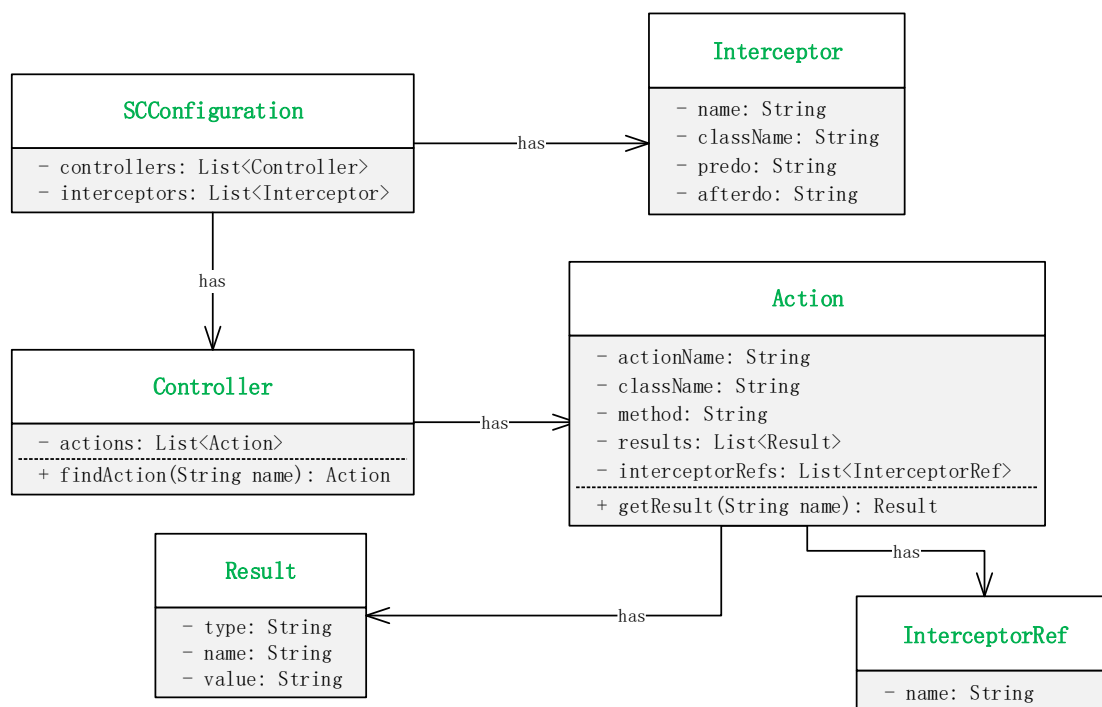
UseSC	
- src/main/java	
- southdayj2eework.water.ustc	
- action	# 包根目录
- BaseAction.java	# action包, 包含了具体业务对应的action类
- LoginAction.java	# 基Action, 包含了一些Action共用的资源, 是其他Action的基类
- RegisterAction.java	# 登陆业务对应的Action
- LogoutAction.java	# 注册业务对应的Action
- bean	# 退出登陆对应的Action
- User.java	# bean包
- db	# UserBean
- SimpleUserDB.java	# db包, 服务端管理数据库资源的包
- interceptor	# 自制的简易User数据库, 内部使用ConcurrentHashMap实现, 单例模式
- LogInterceptor.java	# interceptor包, 包含拦截器相关类
- log	# 日志记录拦截器
- LogProperties.java	# log包, 包含与日志记录操作相关的类
- LogWriter.java	# 用于加载和保存log.properties配置文件中的信息
- bean	# 用于写日志
- Log.java	# bean包, 包含日志文件xml中相对应的bean对象类
- ActionLog.java	# 与log.xml中定义的<log>标签相对应
- service	# 与log.xml中定义的<action>标签相对应
- UserService.java	# service包, 包含模型层(业务逻辑处理)的相关类
- impl	# 接口, 定义了与User有关的相关业务逻辑的抽象方法
- UserServiceImpl.java	# impl包, 包含了针对模型层接口的具体实现类
- src/main/resources	# UserService接口的具体实现类
- config	# Web项目的资源文件包
- controller.xml	# config包, 包含了开发者自定义的配置文件
- files-locations.properties	# controller.xml, 里面配置了Action的执行策略、结果等信息, 会被SimpleController项目扫描使用
- log.properties	# files-locations.properties, 里面包含了各资源文件的存放位置信息, 会被SimpleController项目扫描使用
- src/main/webapp	# log.properties, 里面包含了关于日志记录的一些配置信息, 比如: 日志文件保存位置等
- welcome.jsp	# Web项目部署包
- WEB-INF	# 项目首页.jsp
- web.xml	# Web配置包
- pages	# Web项目配置文件
- failure.jsp	# pages包, 存放返回结果页面
- no-req-resource.jsp	# "请求失败" 页面
- unknown-action.jsp	# "为找到请求资源" 页面
- welcome.jsp	# "未知Action" 页面
- success_view.xml	# "欢迎" 页面(用户已登陆)
- success_view.xml	# "登陆成功" 页面的XML配置
	# "登陆成功" 页面的XSL配置

## 3.2 UML 类图

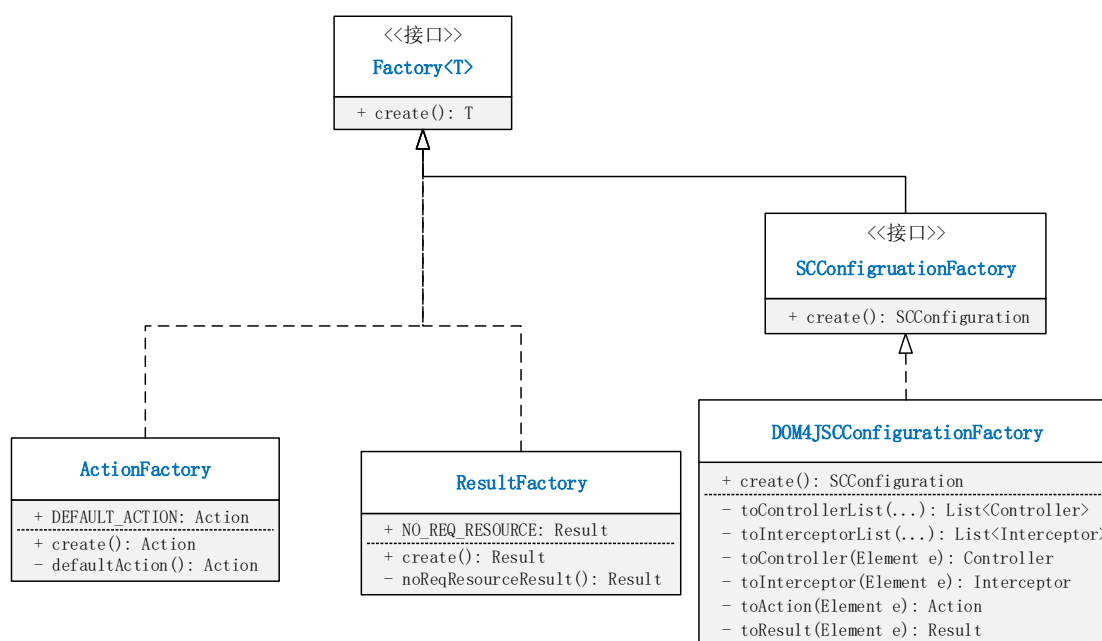
### 3.2.1 SimpleController UML 类图



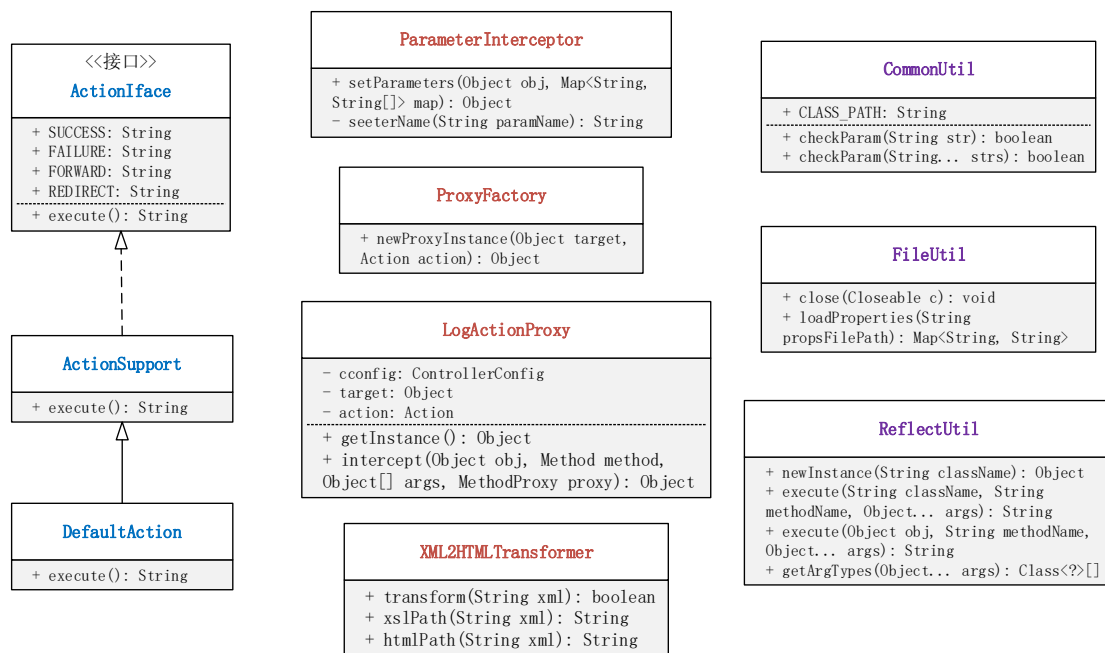
Servlet 相关类-UML 类图



contoller.xml 对应的相关 Bean 类-UML 类图

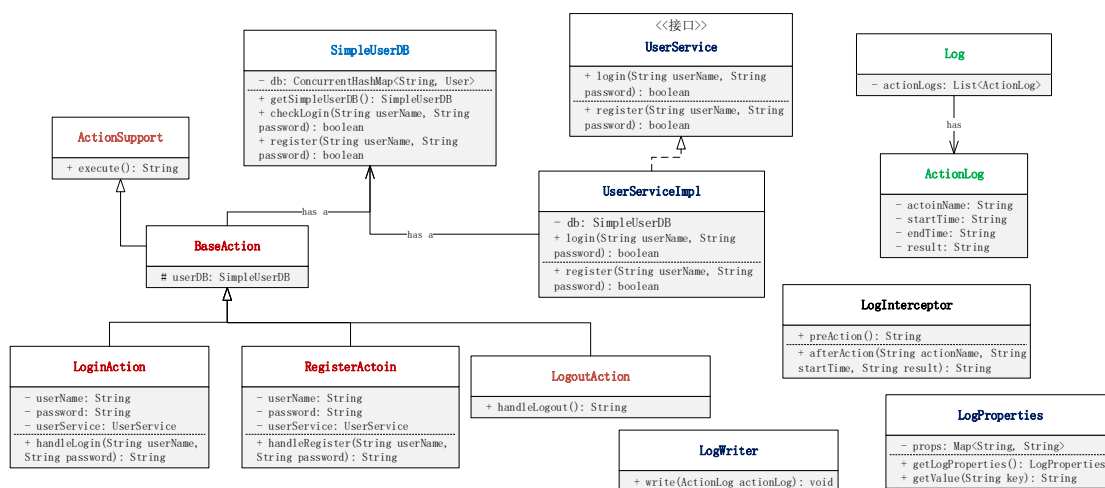


Factory-UML 类图



其他相关类-UML 类图

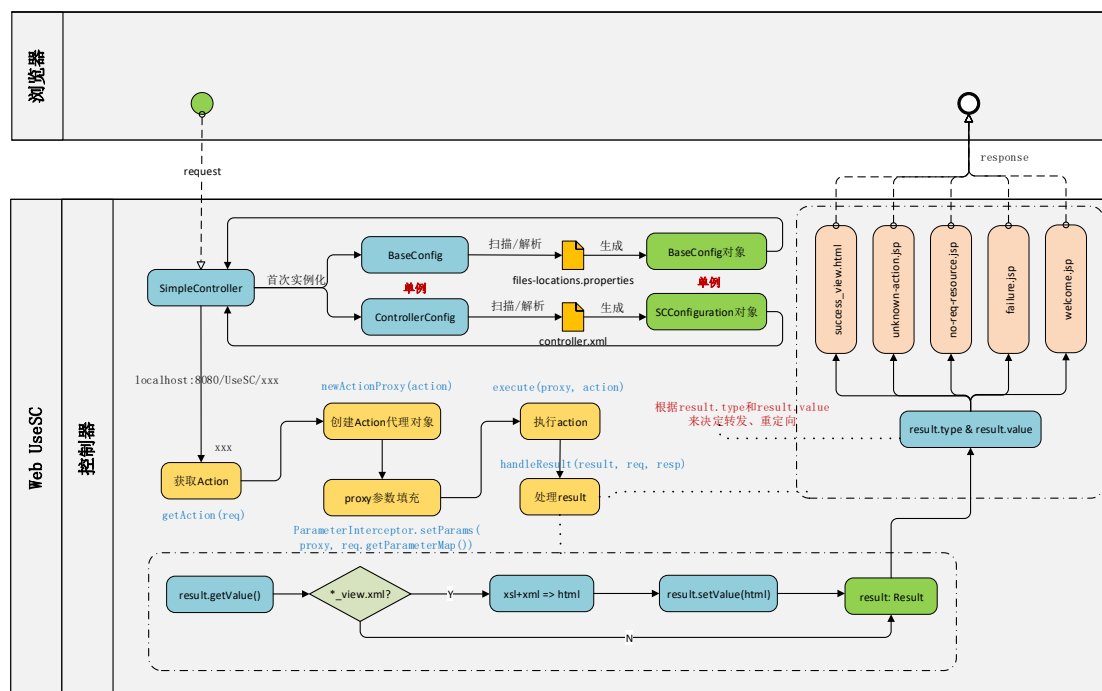
### 3.2.2 UseSC UML 类图



### 3.3 请求处理流程图

由于业务逻辑越来越多，代码实现也越来越复杂，要把所有细节都画出来可行性低。所以我决定从 E4 开始，流程图的细节部分都只针对当前实验所涉及的具体内容，其他部分将简易带过。如果有哪部分流程不清楚的，可以看之前实验报告的流程图。





### 3.4 success\_view.xml

1) 在 controller.xml 中配置 success\_view.xml，如下：

```
<action name="login" class="southday.j2eework.water.ustc.action.LoginAction" method="handleLogin">
  <interceptor-ref name="log"></interceptor-ref>
  <result name="success" type="redirect" value="/pages/success_view.xml"></result>
  <result name="failure" type="redirect" value="/pages/failure.jsp"></result>
</action>
```

2) 编写 success\_view.xml 文件，并保存到 webapp/pages/目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xml>

<view>
  <header>
    <title>A view in MVC - southday</title>
  </header>
  <body>
    <form>
      <name>logoutForm</name>
      <action>logout.sc</action>
      <method>post</method>
      <textview>
        <name>userName</name>
        <label>用户名</label>
        <value>lcx</value>
      </textview>
      <textview>
```

```
<name>userAge</name>
<label>年龄</label>
<value>24</value>
</textView>
<checkBoxView>
  <name>sex</name>
  <label>性别</label>
  <type>radio</type>
  <box>
    <value>male</value>
    <text>男</text>
    <checked>1</checked>
  </box>
  <box>
    <value>female</value>
    <text>女</text>
  </box>
</checkBoxView>
<selectView>
  <name>education</name>
  <label>学历</label>
  <option>
    <value>undergraduate</value>
    <text>本科生</text>
  </option>
  <option>
    <value>postgraduate</value>
    <text>研究生</text>
  </option>
  <option>
    <value>doctoral</value>
    <text>博士生</text>
  </option>
</selectView>
<buttonView>
  <name>logoutButton</name>
  <label>退出</label>
</buttonView>
</form>
</body>
</view>
```

### 3.5 success\_view.xml

编写 success\_view.xml 文件，并保存到 webapp/pages/目录下。

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
<head>
<title>
<xsl:value-of select="view/header/title"/>
</title>
</head>
<body>
<form>
<xsl:attribute name="name"><xsl:value-of select="view/body/form/name" /></xsl:attribute>
<xsl:attribute name="action"><xsl:value-of select="view/body/form/action" /></xsl:attribute>
<xsl:attribute name="method"><xsl:value-of select="view/body/form/method" /></xsl:attribute>
<xsl:for-each select="view/body/form/textView">
<xsl:value-of select="label" />: <xsl:value-of select="value" /><br/>
</xsl:for-each>
<xsl:value-of select="view/body/form/checkboxView/label" />:
<xsl:for-each select="view/body/form/checkboxView/box">
<input>
<xsl:attribute name="type"><xsl:value-of select="../type" /></xsl:attribute>
<xsl:attribute name="name"><xsl:value-of select="../name" /></xsl:attribute>
<xsl:attribute name="value"><xsl:value-of select="value" /></xsl:attribute>
<xsl:if test="checked = 1">
<xsl:attribute name="checked">checked</xsl:attribute>
</xsl:if>
</input>
<xsl:value-of select="text" />
</xsl:for-each>
<br/>
<xsl:value-of select="view/body/form/selectView/label" />:
<select>
<xsl:attribute name="name"><xsl:value-of select="view/body/form/selectView/name" /></xsl:attribute>
<xsl:for-each select="view/body/form/selectView/option">
<option>
<xsl:attribute name="value"><xsl:value-of select="value" /></xsl:attribute>
<xsl:value-of select="text" />
</option>
</xsl:for-each>
</select>
<br/>
<input>
<xsl:attribute name="name"><xsl:value-of select="view/body/form/buttonView/name" /></xsl:attribute>
<xsl:attribute name="type">submit</xsl:attribute>
<xsl:attribute name="value"><xsl:value-of select="view/body/form/buttonView/label" /></xsl:attribute>
</input>
</form>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

### 3.6 xml+xsl 转为 html

1) 在 SimpleController 工程中创建 XML2HTMLTransformer 类，通过 xsl 与 xml 结合，将 xml 转为 html。XML2HTMLTransformer.java 代码如下：

```
public class XML2HTMLTransformer {
    private static TransformerFactory tFactory = TransformerFactory.newInstance();

    public static boolean transform(String xml) {
        if (!CommonUtil.checkParam(xml))
            return false;
        String xsl = xslPath(xml);
        String html = htmlPath(xml);
        try {
            Transformer transformer = tFactory.newTransformer(new StreamSource(xsl));
            transformer.transform(new StreamSource(xml), new StreamResult(new FileOutputStream(html)));
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }

    public static String xslPath(String xml) {
        if (!CommonUtil.checkParam(xml))
            return null;
        return xml.replace(".xml", ".xsl");
    }

    public static String htmlPath(String xml) {
        if (!CommonUtil.checkParam(xml))
            return null;
        return xml.replace(".xml", ".html");
    }
}
```

2) 生成 success\_view.html，页面显示如下：

### 3.7 handleResult（结果分发）

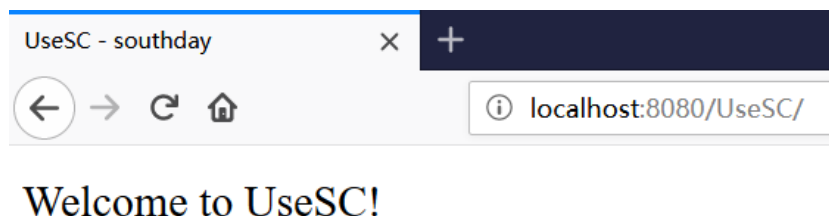
1) 根据 execute(proxy, action) 执行后返回的结果 result 进行结果分发：如果 result.getValue() 以 “\_view.xml” 结尾，则将对应的 xml 文件转为 html 格式，然后更新 result 中 value 的值。

2) 根据 result 的 type 和 value 属性来处理结果：请求转发或重定向。handleResult 方法代码片段如下：

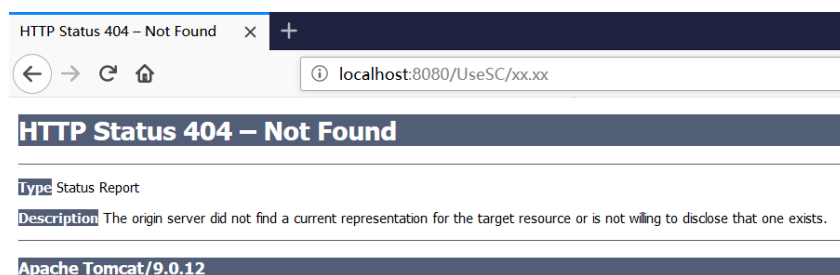
```
private void handleResult(Result result, HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String value = result.getValue();
    if (value.lastIndexOf("_view.xml") != -1) {
        String xmlPath = realRootPath(req) + value;
        boolean success = XML2HTMLTransformer.transform(xmlPath);
        if (success)
            result.setValue(XML2HTMLTransformer.htmlPath(value));
        else
            result = ResultFactory.NO_REQ_RESOURCES;
    }
    if (ActionInterface.FORWARDED.equals(result.getType())) {
        req.getRequestDispatcher(result.getValue()).forward(req, resp);
    } else if (ActionInterface.REDIRECT.equals(result.getType())) {
        resp.sendRedirect(req.getContextPath() + result.getValue());
    }
}
```

### 3.8 测试结果

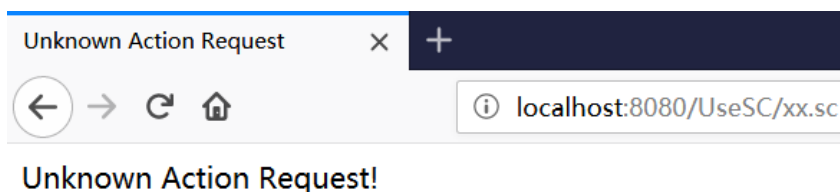
1) 访问主页：<http://localhost:8080/UseSC/>，成功访问主页；



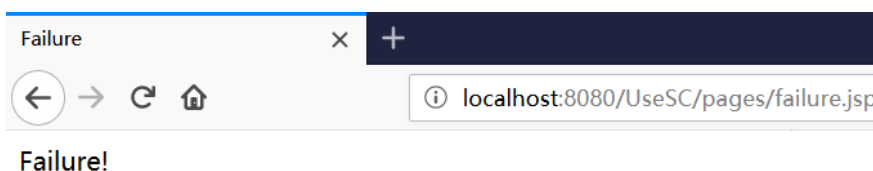
2) 非法 URL：<http://localhost:8080/UseSC/xx.xx>，404-Not found；



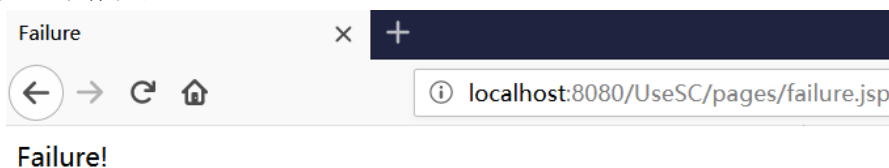
3) 非法请求: <http://localhost:8080/UseSC/xx.sc>, 无法识别的 action;



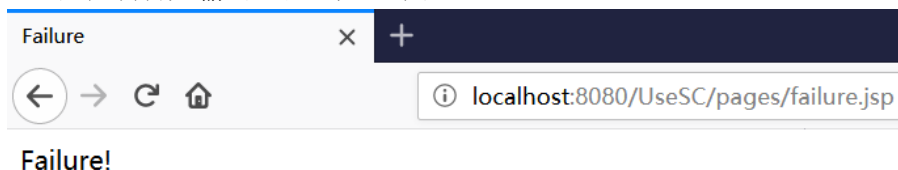
4) 测试登陆: <http://localhost:8080/UseSC/login.sc?userName=lcx>, 失败, 因为缺少参数: password;



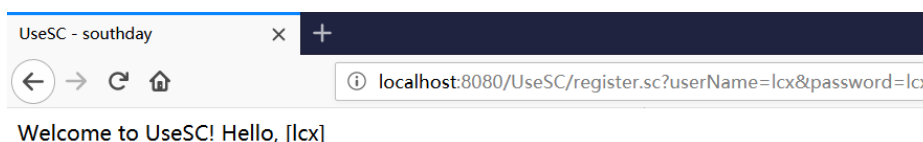
5) 测试登陆: <http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx>, 失败, 用户账号 lcx 不存在;



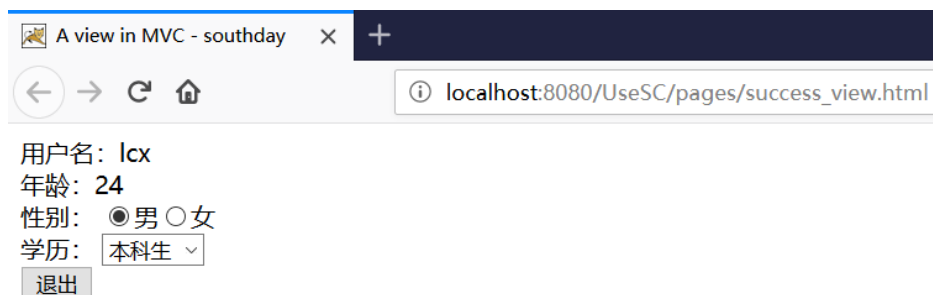
6) 测试注册: <http://localhost:8080/UseSC/register.sc?password=lcx>, 失败, 因为缺少参数 userName; 控制台无输出, 日志无记录;



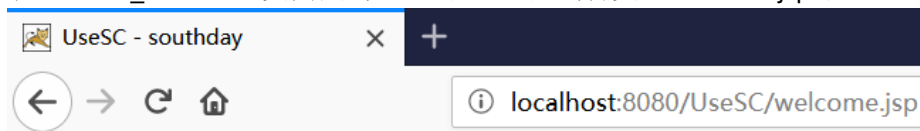
7) 测试注册: <http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx>, 注册成功, 跳转到 welcome 页面;



8) 注册成功后, 用 lcx 账号登陆: <http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx>, 登陆成功, 跳转到 success\_view.html 页面;

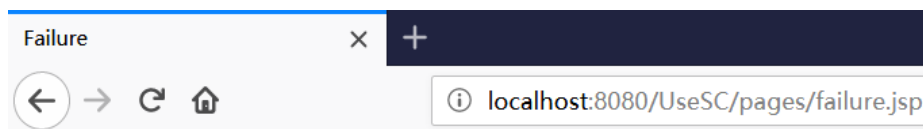


- 9) 在 success\_view.html 页面点击“退出”，返回到首页（welcome.jsp）：



## Welcome to UseSC!

- 10) 注册成功后，使用 lcx 账号登陆，填写错误密码 xcl，<http://localhost:8080/UseSC/login.sc?userName=lcx&password=xcl>，登陆失败，因为密码错误；



Failure!

### 3.9 Struts 框架中视图组件的工作方式

根据我的学习，我对 Struts 框架中视图组件工作方式的理解如下：

#### 1) Struts 框架视图组件的重要组成部分：

A) JSP (JavaServer Pages)，是由 Sun Microsystems 公司主导建立的一种动态网页技术标准。JSP 部署于网络服务器上，可以响应客户端发送的请求，并根据请求内容动态地生成 HTML、XML 或其他格式文档的 Web 网页，然后返回给请求者。

B) OGNL (Object Graph Navigation Language)，是一种功能强大的表达式语言，通过它简单一致的表达式语法，可以存取对象的任意属性，调用对象的方法，遍历整个对象的结构图，实现字段类型转化等功能。

C) Value Stack，是一个接口，在 struts2 中使用 OGNL (Object-Graph Navigation Language) 表达式实际上是使用实现了 ValueStack 接口的类 OgnlValueStack。ValueStack 贯穿整个 action 的生命周期，每一个 action 实例都拥有一个 ValueStack 对象，其中保存了当前 action 对象和其他相关对象。

D) Struts Tags，是 Struts 框架提供的一个标签库，其允许开发人员通过指定操作名称和可选名称空间，直接从 JSP 页面调用操作，标记的主体内容用于呈现操作的结果。

- 2) 在使用 Struts 框架编写视图时，并非是单纯的 html 或者标准的 jsp，而是通过在 jsp 中内嵌 Struts Tags，并且由 OGNL 和 Value Stack 来访问对象实现的。例如：

### Form UI tags Demo

```

private String name, password, address;
private boolean male;
private String[] favorites = new String[] { "sport", "music",
selectedFavorites;
private String[] leftList = new String[] { "a", "b", "c" },
rightList = new String[] { "e", "g" },
leftResult,
rightResult;

```

```

<:s:form action="welcome">
  <s:textfield name="name" label="Name"></s:textfield>
  <s:password name="password" label="password"></s:password>
  <s:textarea name="address" label="address"></s:textarea>
  <s:checkbox name="male" label="Gender"></s:checkbox>
  <s:checkboxlist list="favorites" name="selectedFavorites"
    label="Favorites" value="selectedFavorites"></s:checkboxlist>

  <s:optiontransferselect name="LeftResult" doubleList="rightList"
    list="LeftList" doubleName="rightResult" value="LeftResult"
    doubleValue="rightResult"></s:optiontransferselect>
  <s:submit value="Register"></s:submit>
</s:form>

```

- A) 页面中使用了 Struts Tags，并且配置了各类属性；
- B) ValueStack 中保存了当前 Action 及其相关对象实例；
- C) OGNL 作为页面与对象的桥梁，将页面中的表达式转为对 Java 对象的存取操作；

## 4. 结论

### 4.1 总结

通过本次实验，我学习了如何使用 XSLT 技术将 XML 文件转为 HTML，算是增长了我的见识。而在做 E4 实验的过程中，更大的收获来自于对 E3 代码的重构，由于之前理解的错误，导致了设计的错误，最终重构了大约 60% 的代码。虽然花了不少时间，但是挺值的，毕竟只有亲身经历过，才会印象深刻。

### 4.2 问题及看法

在 XML 转 HTML 这个技术点上，我并没有遇到问题。我遇到的问题主要来自对 E3 代码重构的过程，以及对 E4 实验内容的困惑。

#### 4.2.1 由 JDK 动态代理转为 CGLIB 动态代理

本来我是想直接使用 JDK 动态代理方式来实现的，但无奈这种方式必须要实现接口，而我的 ActionBean 并不想实现某个接口。后来查到 CGLIB 可以在不实现接口的情况下来实现动态代理，虽然也有一定局限性（不能是 final 类），但满足我目前的需求。

#### 4.2.2 创建代理对象与参数设置的顺序问题

最初的代码逻辑：生成 Action 实例 -> 填充参数 -> 生成 Action 代理对象；如下：

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    Action action = getAction(req);
    Object target = newTargetAction(action.getClassName());
    ParameterInterceptor.setParameters(target, req.getParameterMap());
    Object proxy = ProxyFactory.newProxyInstance(target, action);
    Result result = execute(proxy, action);
    handleResult(result, req, resp);
}
```

但是我发现生成的 proxy 中一直无法拿到之前 set 进去的参数值，而 target 中是可以获取到的。后来我猜到可能 proxy 中封装的并不是原先的 target 对象，而是重新生成的实例，所以我就改了一下顺序：生成 Action 实例 -> 生成 Action 代理对象 -> 填充参数；发现这样就能正确获取到参数值；如下：

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    Action action = getAction(req);
    Object proxy = newActionProxy(action);
    ParameterInterceptor.setParameters(proxy, req.getParameterMap());
    Result result = execute(proxy, action);
    handleResult(result, req, resp);
}
```

#### 4.2.3 反射实现的局限性：未知类型与 null 参数

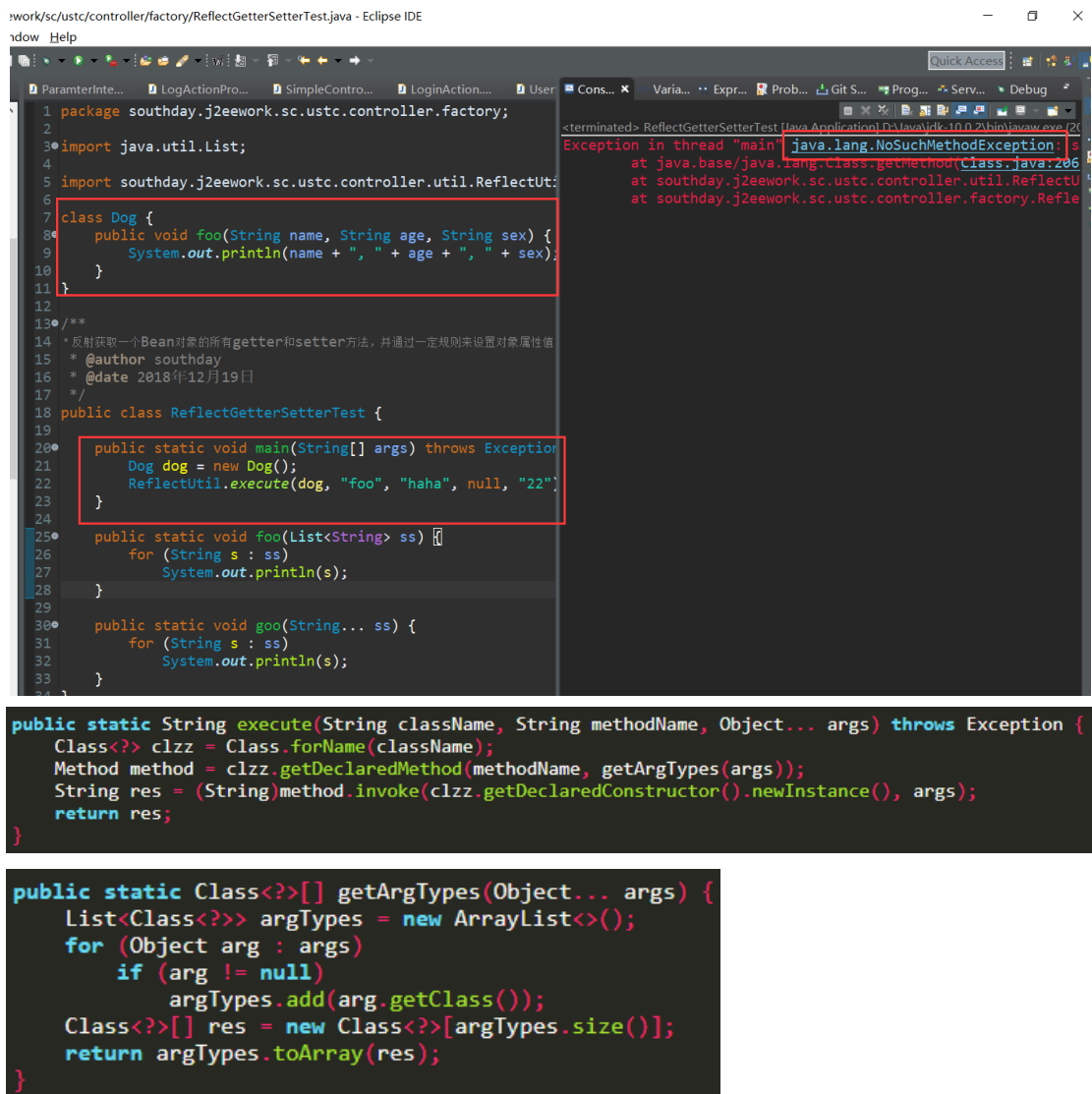
通常我们要使用反射来获取类中的某个方法时，要提供：方法名、参数类型。而在作业中，除非你对相关方法信息进行了配置，然后 SimpleController 扫描该配置以获得方法的参数类型，否则你是无法提前知道方法参数类型的。



在这种情况下想要获得方法的参数类型，就只能通过前端提交来的数据来进行猜测，即获取 `param.getClass()`。而这种做法就存在两个问题：

1) `req.getParameter("key")`中返回的是 `String` 类型，那我们获取的都是 `java.lang.String`，而实际方法中的参数类型可能是 `int`、`long`、`float` 等。所以就需要一个转换器来将前端传来的数据转为后端需要的数据类型。

2) `param.getClass()`，这个 `param` 可能为 `null`，如果考虑不当就报 `NullPointerException`，考虑周全的话，把 `null` 的参数过滤掉，但最后获取方法时也会抛出 `NoSuchMethodException`，所以也是挺低效的方法。针对（2），测试代码如下：



```

1 package southday.j2eework.sc.ustc.controller.factory;
2
3 import java.util.List;
4
5 import southday.j2eework.sc.ustc.controller.util.ReflectUtil;
6
7 class Dog {
8     public void foo(String name, String age, String sex) {
9         System.out.println(name + ", " + age + ", " + sex);
10    }
11 }
12
13 /**
14  * 反射获取一个Bean对象的所有getter和setter方法，并通过一定规则来设置对象属性值
15  * @author southday
16  * @date 2018年12月19日
17  */
18 public class ReflectGetterSetterTest {
19
20     public static void main(String[] args) throws Exception {
21         Dog dog = new Dog();
22         ReflectUtil.execute(dog, "foo", "haha", null, "22");
23     }
24
25     public static void foo(List<String> ss) {
26         for (String s : ss)
27             System.out.println(s);
28     }
29
30     public static void goo(String... ss) {
31         for (String s : ss)
32             System.out.println(s);
33     }
34 }

```

```

public static String execute(String className, String methodName, Object... args) throws Exception {
    Class<?> clzz = Class.forName(className);
    Method method = clzz.getDeclaredMethod(methodName, getArgTypes(args));
    String res = (String)method.invoke(clzz.getDeclaredConstructor().newInstance(), args);
    return res;
}

```

```

public static Class<?>[] getArgTypes(Object... args) {
    List<Class<?>> argTypes = new ArrayList<>();
    for (Object arg : args)
        if (arg != null)
            argTypes.add(arg.getClass());
    Class<?>[] res = new Class<?>[argTypes.size()];
    return argTypes.toArray(res);
}

```

#### 4.2.4 对于 E4 实验内容的困惑

1) 根据'e4\_20181124171032339.pdf'中对实验内容的描述，我想到的解法是：

- A) 创建 demo.xml;
- B) 创建 demo.xsl;
- C) 通过 java 用 demo.xsl 来转换 demo.xml，返回给前端显示；

2) 实验内容中的"success\_view.xml"，这个<value>应该是动态显示的，即：哪个用户登陆了，返回的页面中应该是显示对应的用户名（动态效果）；但是这个 xml 文件中<value>字段是硬编码的，也就说最后 xsl 转换 xml 后返回的页面也是静态的内容；

```
<?xml version="1.0" encoding="UTF-8"?>
<view>
  <header>
    <title>a viewer in MVC</title>
  </header>
  <body>
    <form>
      <name>logoutForm</name>
      <action>logout.action</action>
      <method>post</method>
      <textview>
        <name>userName</name>
        <label>Login Name:</label>
        <value>Water</value>
      </textView>
    </form>
  </body>
</view>
```

3) 如果 xml 是静态的, xls 也是静态的, 最后转换成的 xhtml 也是静态的; 那么让框架使用者开发页面: 要编写一个 xml, 还要编写一个 xsl, 两倍的开发工作量, 途中还要经过转换, 最后效果和开发一个 html 一样 (而且 xsl 的编写本来就很像 html), 那为何要实现这种 xml+xsl 方式?

4) E4 是要我们实现动态的效果, 还是根据静态 xml 转为 html? 如果要实现动态效果, 为何要通过 xml 转 html 这种方式?

5) 我个人认为较为合理的做法是: 用户只用开发一个页面 (不管 xml 格式还是其他格式), 最后视图的生成由框架来解决: struts 中使用 jsp, ognl 等技术来实现这个效果。

## 5. 参考文献

- [1] XPath 语法: [http://www.w3school.com.cn/xpath/xpath\\_syntax.asp](http://www.w3school.com.cn/xpath/xpath_syntax.asp)
- [2] XSLT 元素参考手册: [http://www.w3school.com.cn/xsl/xsl\\_w3celementref.asp](http://www.w3school.com.cn/xsl/xsl_w3celementref.asp)
- [3] java 使用 xslt 将 xml 文件转换成 html:  
<http://outofmemory.cn/code-snippet/2191/java-usage-xslt-jiang-xml-file-turn-huancheng-html>
- [4] 再读 Struts-1.3.5 User Guide 2 — Building View Components:  
<https://www.xuebuyuan.com/415086.html>
- [5] Java--获取 request 中所有参数的方法:  
<https://www.cnblogs.com/renxiaoren/p/5512022.html>
- [6] How To Create A Struts 2 Web Application:  
<https://struts.apache.org/getting-started/how-to-create-a-struts2-web-application.html>
- [7] Form Tags: <https://struts.apache.org/getting-started/form-tags.html#struts-2-select-tag>
- [8] Struts User's Guide:  
[http://svn.apache.org/repos/asf/struts/struts1/tags/STRUTS\\_0\\_5/web/documentation/users\\_guide.html](http://svn.apache.org/repos/asf/struts/struts1/tags/STRUTS_0_5/web/documentation/users_guide.html)
- [9] 走进 Struts2（一） — Struts2 的执行流程及其工作原理:  
<https://www.cnblogs.com/mfmdaoyou/p/7189578.html>
- [10] Struts2 中 Action 接收参数的方法主要有以下三种:  
<https://www.cnblogs.com/haimishasha/p/6193235.html>
- [11] Struts2 Action 类的四种实现方式:  
[https://blog.csdn.net/qg\\_33800083/article/details/80251112](https://blog.csdn.net/qg_33800083/article/details/80251112)
- [12] 【Struts2 进阶】Struts2 深度解析 ModelDriven 原理:  
<https://blog.csdn.net/u010028869/article/details/50850103>
- [13] Java 动态代理之 JDK 实现和 CGLib 实现（简单易懂）:  
<https://www.cnblogs.com/ygj0930/p/6542259.html>