轻量级 J2EE 框架应用

E 2 A Simple Controller Based on Configuration File

学号：SA18225170
姓名：李朝喜

报告撰写时间：2018/12/03

文档目录

1. 主题概述

根据我的实现，我认为 E2 中涉及的主要内容有 4 部分：Java XML 解析、Java 反射机制、设计模式（单例、工厂），Servlet 请求转发与重定向。

1.1 Java XML 解析

XML 是一种通用的数据交换格式，它的平台无关性、语言无关性、系统无关性、给数据集成与交互带来了极大的方便。XML 在不同的语言环境中解析方式都是一样的，只不过实现的语法不同而已。

XML 的解析方式分为四种：DOM 解析、SAX 解析、JDOM 解析、DOM4J 解析。其中前两种属于基础方法，是官方提供的平台无关的解析方式；后两种属于扩展方法，它们是在基础的方法上扩展出来的，只适用于 java 平台。

1.1.1 DOM 解析

DOM 的全称是 Document Object Model，也即文档对象模型。在应用程序中，基于 DOM 的 XML 分析器将一个 XML 文档转换成一个对象模型的集合（通常称 DOM 树），应用程序正是通过对这个对象模型的操作，来实现对 XML 文档数据的操作。通过 DOM 接口，应用程序可以在任何时候访问 XML 文档中的任何一部分数据，因此，这种利用 DOM 接口的机制也被称作随机访问机制。

优点：

- 形成了树结构，有助于更好的理解、掌握，且代码容易编写。
- 解析过程中，树结构保存在内存中，方便修改。

缺点：

- 由于文件是一次性读取，所以对内存的耗费比较大。
- 如果 XML 文件比较大，容易影响解析性能且可能会造成内存溢出。

1.1.2 SAX 解析

SAX 的全称是 Simple APIs for XML，即 XML 简单应用程序接口。与 DOM 不同，SAX 提供的访问模式是一种顺序模式，这是一种快速读写 XML 数据的方式。当使用 SAX 分析器对 XML 文档进行分析时，会触发一系列事件，并激活相应的事件处理函数，应用程序通过这些事件处理函数实现对 XML 文档的访问，因而 SAX 接口也被称作事件驱动接口。

优点：

- 采用事件驱动模式，对内存耗费比较小。
- 适用于只处理 XML 文件中的数据时。

缺点：

- 编码比较麻烦。
- 很难同时访问 XML 文件中的多处不同数据。

1.1.3 JDOM 解析

- 仅使用具体类，而不使用接口。
- API 大量使用了 Collections 类。

1.1.4 DOM4J 解析

- JDOM 的一种智能分支，它合并了许多超出基本 XML 文档表示的功能。
- 它使用接口和抽象基本类方法。
- 具有性能优异、灵活性好、功能强大和极端易用的特点。
- 是一个开放源码的工具。

1.2 Java 反射机制

Java 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 Java 语言的反射机制。

Java 反射涉及的类，除了 Class 类之外，基本上都在 java.lang.reflect 包里面，常用的类有 Constructor、Field、Method 类等，AccessibleObject 类是前面三个类的基类，主要包含设置安全性检查等方法。

1.3 设计模式（单例、工厂）

1.3.1 单例模式

单例模式，是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中，应用该模式的类一个类只有一个实例。即一个类只有一个对象实例。

单例模式在 Java 语言中，有两种构建方式：

- 懒汉方式：指全局的单例实例在第一次被使用时构建。
- 饿汉方式：指全局的单例实例在类装载时构建。

1.3.2 工厂模式

工厂模式是常用的实例化对象模式，是用工厂方法代替 new 操作的一种模式。使用工厂模式可能会降低效率，却能给你的系统带来更高的可扩展性和可维护性。

常见的工厂模式有以下 3 种：

- 简单工厂：利用静态方法定义一个简单的工厂，将创建产品的逻辑抽离出来。
- 工厂方法：定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，把实例化推迟到子类。
- 抽象工厂：提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。

1.4 Servlet 请求转发与重定向

1.4.1 转发

请求转发是服务器内部把对一个 request/response 的处理权，移交给另外一个 Servlet。该过程对于客户端而言是透明的，并且转发后传输的信息不会丢失。

```
request.getRequestDispatcher("/welcome.jsp").forward(request, response);
```

1.4.2 重定向

重定向是服务端在处理完请求后，由 response 向客户端发出响应，其告诉客户端接下来需要发送另一个请求去获取资源。随后客户端便会根据 response 的响应内容向服务端发

出指定请求。在这个过程中，两次 request 是相互独立的，内容不会共享。

```
response.sendRedirect(request.getContextPath() + "/welcome.jsp");
```

1.4.3 二者的区别

- 转发在服务器端完成；重定向在客户端完成；
- 转发速度快；重定向速度慢；
- 转发是同一次请求；重定向是两次不同请求；
- 转发不会执行转发后的代码；重定向会执行重定向之后的代码；
- 转发地址栏没有变化；重定向地址栏有变化；
- 转发必须是在同一台服务器下完成；重定向可以在不同的服务器下完成；

2. 假设

需要注意，实验 E2 是在 E1 基础上进行的，所以在做 E2 前请先完成 E1，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。当然对于一些之前没接触过的内容，如：Java XML 解析、Servlet 请求转发和重定向等，可以通过一些 demo 现学现卖。

- 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 Java 反射机制，能够利用 Java 反射来调用某个类中的方法；
- 了解单例和工厂设计模式，能够使用 Java 语言实现这两种模式；

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。

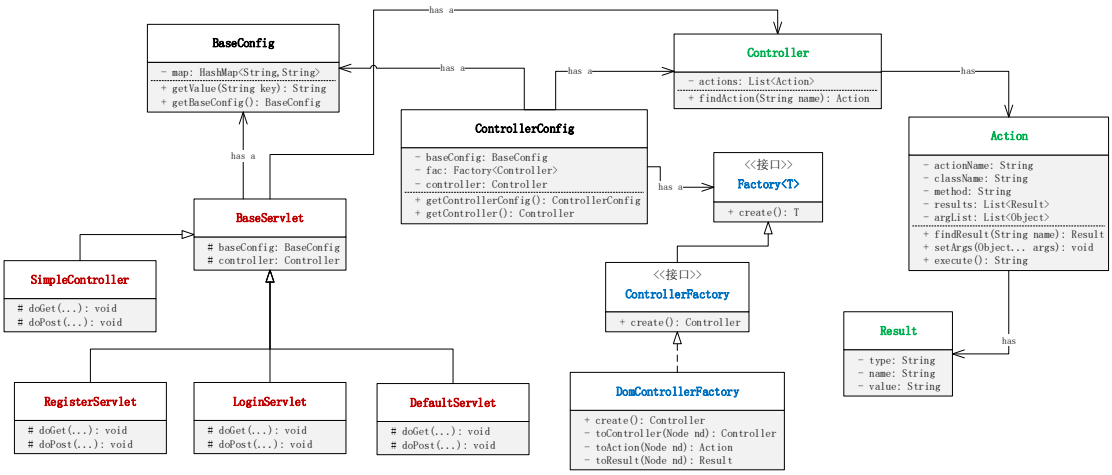
3. 实现或证明

3.1 项目包结构及简要描述

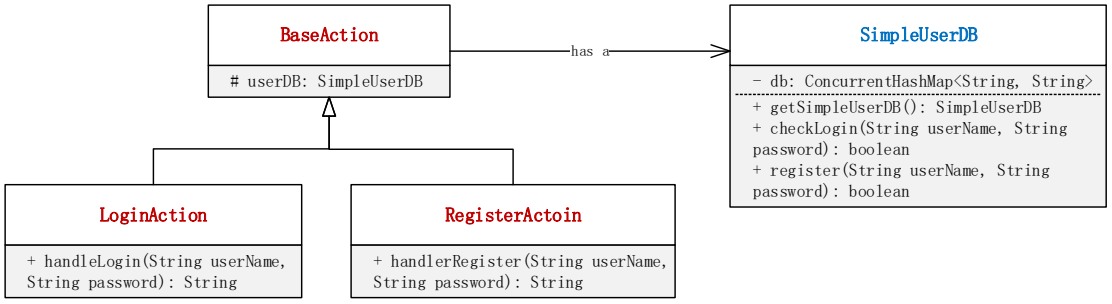
```
SimpleController
  |- src/main/java
    |- southday.j2eework.sc.ustc.controller      # 包根目录
      |- BaseServlet.java                        # 基Servlet，里面包含一些Servlet共用的资源，是其他Servlet的基类
      |- SimpleController.java                   # Servlet，用于处理*.sc的请求
      |- servlet                                 # Servlet包，包含一些处理具体业务的Servlet
        |- DefaultServlet.java                   # 默认Servlet，当请求未匹配时，被转发到默认Servlet进行处理
        |- LoginServlet.java                     # 处理登陆业务 login.sc
        |- RegisterServlet.java                  # 处理注册业务 register.sc
      |- config                                  # 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
        |- BaseConfig.java                       # 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
        |- ControllerConfig.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
      |- bean                                    # bean包，包含与UseSC中controller.xml文件里定义内容相对应的JavaBean对象类
        |- Controller.java                       # 与controller.xml中定义的<controller>标签相对应
        |- Action.java                           # 与controller.xml中定义的<action>标签相对应
        |- Result.java                           # 与controller.xml中定义的<result>标签相对应
      |- factory                                 # factory包，工厂模式，里面包含了用于创建对象的工厂类
        |- Factory.java                          # 抽象工厂，提供接口：T create() throws Exception;
        |- ControllerFactory.java                # 用于生成Controller对象的工厂
        |- dom                                   # dom包，基于DOM 的 XML解析来实现Controller对象创建的工厂
          |- DomControllerFactory.java           # 基于DOM 的 XML解析来创建Controller对象的具体实现类
      |- util                                    # util包，包含项目工具类
        |- FileUtil.java                         # 包含与文件资源处理相关的常用方法，如：关闭资源、获取类加载路径等

UseSC
  |- src/main/java
    |- southday.j2eework.water.ustc              # 包根目录
      |- action                                  # action包，包含了具体业务对应的action类
        |- BaseAction.java                       # 基Action，包含了一些Action共用的资源，是其他Action的基类
        |- LoginAction.java                      # 登陆业务对应的Action
        |- RegisterAction.java                   # 注册业务对应的Action
      |- db                                      # db包，服务端管理数据库资源的包
        |- SimpleUserDB.java                     # 自制的简易User数据库，内部使用ConcurrentHashMap实现，单例模式
  |- src/main/resources                          # Web项目的资源文件包
    |- config                                    # config包，包含了开发者自定义的配置文件
      |- controller.xml                          # 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
      |- files-locations.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                             # Web项目部署包
    |- welcome.html                              # 项目首页 html
    |- welcome.jsp                               # 项目首页 jsp
    |- WEB-INF                                   # Web配置包
      |- web.xml                                 # Web项目配置文件
    |- pages                                     # pages包，存放返回结果页面
      |- failure.jsp                             # “请求失败”页面
      |- no-req-resource.jsp                     # “为找到请求资源”页面
      |- unkonwn-action.jsp                      # “未知Action”页面
      |- welcome.jsp                             # “欢迎”页面（用户已登陆）
```

3.2 UML 类图

限于篇幅和显示，UML 类图中我只描述重要的部分，对一些不必要的细节内容，我将会省略。

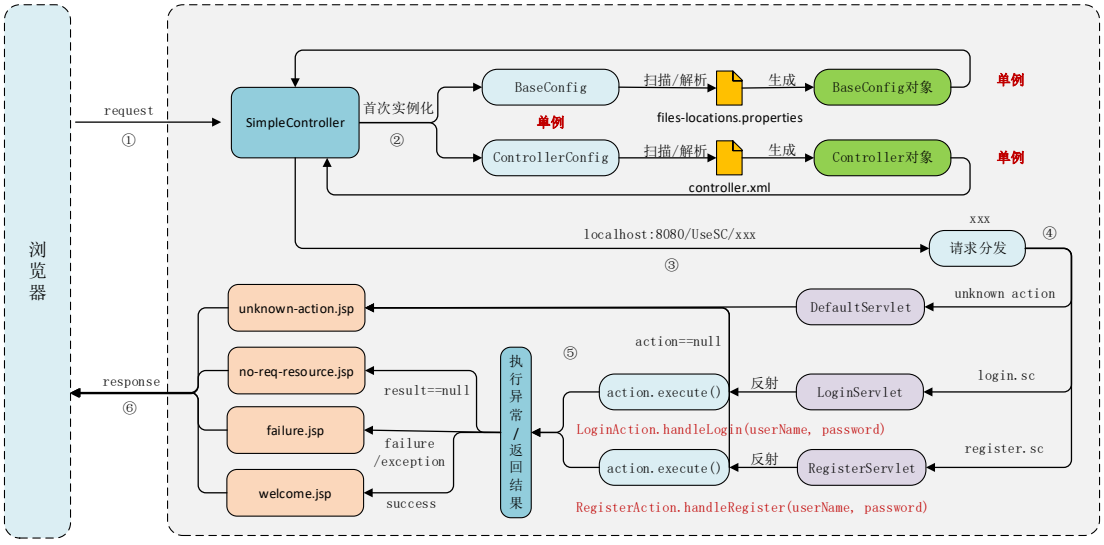


SimpleController 工程 UML 类图



UseSC 工程 UML 类图

3.3 请求处理流程图



3.4 Action 与 SimpleUserDB

1）UseSC 工程中，在 southday.j2eework.water.ustc.action 包下创建 Action 类：BaseAction、LoginAction、RegisterAction。3 个类的主要代码如下：

BaseAction：

```
public abstract class BaseAction {
    protected SimpleUserDB userDB = SimpleUserDB.getSimpleUserDB();
}
```

LoginAction：

```
public class LoginAction extends BaseAction {
    public String handleLogin(String userName, String password) {
        return userDB.checkLogin(userName, password) ? "success" :
"failure";
    }
}
```

RegisterAction：

```
public class RegisterAction extends BaseAction {
    public String handleRegister(String userName, String password) {
        userDB.register(userName, password);
        return "success";
    }
}
```

2）UseSC 工程中，在 southday.j2eework.water.ustc.db 包内创建 SimpleUserDB 类。该类基于单例模式，用于模拟数据库资源，并且提供相关操作，其内部使用 ConcurrentHashMap 来实现。（限于篇幅关系，贴出来的代码中已去掉部分注释）如下：

```
public class SimpleUserDB {
    private final ConcurrentHashMap<String, String> db = new
ConcurrentHashMap<>();
    private SimpleUserDB() {}

    private static class SimpleUserDBHolder {
        private static SimpleUserDB userDB = new SimpleUserDB();
    }

    public static SimpleUserDB getSimpleUserDB() {
        return SimpleUserDBHolder.userDB;
    }

    public boolean checkLogin(String userName, String password) {
        return db.get(userName) != null &&
password.equals(db.get(userName));
    }

    public boolean register(String userName, String password) {
        db.putIfAbsent(userName, password);
```

```
        return true;
    }
}
```

3.5 配置文件与 BaseConfig

1）UseSC 工程中，在 src/main/resources 目录下创建 config 目录，并在该目录下创建配置文件：controller.xml、files-locations.properties。内容如下：

controller.xml：

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

<sc-configuration>
    <controller>
        <action name="login"
class="southday.j2eework.water.ustc.action.LoginAction"
method="handleLogin">
            <result name="success" type="forward"
value="/pages/welcome.jsp"></result>
            <result name="faiulre" type="redirect"
value="/pages/failure.jsp"></result>
        </action>
        <action name="register"
class="southday.j2eework.water.ustc.action.RegisterAction"
method="handleRegister">
            <result name="success" type="forward"
value="/pages/welcome.jsp"></result>
        </action>
    </controller>
</sc-configuration>
```

files-locations.properties：

```
controller.xml=config/controller.xml
unkonwn-action.jsp=/pages/unkonwn-action.jsp
no-req-resources.jsp=/pages/no-req-resources.jsp
failure.jsp=/pages/failure.jsp
welcome.jsp=/pages/welcome.jsp
welcome.html=/welcome.html
```

2）SimpleController 工程中，southday.j2eework.sc.ustc.controller.config 目录下的 BaseConfig 类会在首次实例化时去扫描 UseSC 工程中的 files-locations.properties 文件，并将这些配置保存在本对象中（基于单例模式，并通过 HashMap 实现）。若后期要调整这些文件的存放位置，只需要修改 files-locations.properties 中相应的配置即可。如下：

```
public class BaseConfig {
```

```
    private static final String FILES_LOCATIONS_PROPS_PATH =
getPathOfFilesLocationsProps();
    public static final String ACTION_NAME_LOGIN = "login";
    public static final String ACTION_NAME_REGISTER = "register";
    public static final String RESULT_NAEM_SUCCESS = "success";
    public static final String RESULT_NAME_FAILURE = "failure";
    public static final String RESULT_TYPE_FORWARD = "forward";
    public static final String RESULT_TYPE_REDIRECT = "redirect";
    private HashMap<String, String> map = new HashMap<>();

    private BaseConfig() {
        init();
    }

    private void init() {
        Properties props = new Properties();
        InputStream ins = null;
        try {
            ins = new BufferedInputStream(new
FileInputStream(FILES_LOCATIONS_PROPS_PATH));
            props.load(ins);
            for (Map.Entry<Object, Object> e : props.entrySet())
                map.put((String)e.getKey(), (String)e.getValue());
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            FileUtil.close(ins);
        }
    }

    private static class BaseConfigHolder {
        private static BaseConfig config = new BaseConfig();
    }

    public static BaseConfig getBaseConfig() {
        return BaseConfigHolder.config;
    }

    public String getValue(String key) {
        return map.get(key);
    }

    private static String getPathOfFilesLocationsProps() {
```

```
        return FileUtil.CLASSES_PATH +
"config/files-locations.properties";
    }
}
```

3.6 Controller 的生成

我个人觉得 Controller 对象是通过扫描 controller.xml 文件生成的，而 controller.xml 文件是静态资源，不会在服务器运行过程中被改变，所以 Controller 对象也不应该被改变，并且该对象应该是全局共享一个实例，所以需要单例模式进行控制。

1）在实现过程过程中，我通过将 ControllerConfig 类设置为单例模式，以达到控制 Controller 对象为单例的效果。如下：

ControllerConfig：

```
public class ControllerConfig {
    private static final BaseConfig baseConfig = BaseConfig.getBaseConfig();
    private static final String CONTROLLER_XML_PATH =
getPathOfControllerXML();
    private Factory<Controller> fac = new
DomControllerFactory(CONTROLLER_XML_PATH);
    private Controller controller = null;

    private ControllerConfig() {
        try {
            controller = fac.create();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private static class ControllerConfigHolder {
        private static ControllerConfig config = new ControllerConfig();
    }

    public static ControllerConfig getControllerConfig() {
        return ControllerConfigHolder.config;
    }

    public static Controller getController() {
        return ControllerConfigHolder.config.controller;
    }

    private static String getPathOfControllerXML() {
        return FileUtil.CLASSES_PATH +
baseConfig.getValue("controller.xml");
```

```
    }
}
```

2）对于 controller.xml 的解析是由 DomControllerFactory 类完成的，代码如下：

DomControllerFactory：

```
public class DomControllerFactory implements ControllerFactory {
    private String controllerXMLPath = null;
    private static DocumentBuilder dBuilder = null;

    public DomControllerFactory() {
        this(null);
    }

    public DomControllerFactory(String controllerXMLPath) {
        this.controllerXMLPath = controllerXMLPath;
        init();
    }

    private void init() {
        try {
            dBuilder =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Controller create() throws Exception {
        Document doc = dBuilder.parse(controllerXMLPath);
        Node controllerNode =
doc.getElementsByTagName("controller").item(0);
        return toController(controllerNode);
    }

    private Controller toController(Node controllerNode) throws Exception {
        NodeList nodeList = controllerNode.getChildNodes();
        Controller controller = new Controller();
        List<Action> actionList = new ArrayList<>();
        for (int i = 0, len = nodeList.getLength(); i < len; i++) {
            Node node = nodeList.item(i);
            if (node.getNodeType() != Node.ELEMENT_NODE)
                continue;
            Action action = toAction(node);
```

```
            actionList.add(action);
        }
        controller.setActions(actionList);
        return controller;
    }

    private Action toAction(Node actionNode) throws Exception {
        Element actionElement = (Element)actionNode;
        Action action = new Action();
        action.setActionName(actionElement.getAttribute("name"));
        action.setClassName(actionElement.getAttribute("class"));
        action.setMethodName(actionElement.getAttribute("method"));

        NodeList nodeList = actionElement.getChildNodes();
        List<Result> resultList = new ArrayList<>();
        for (int i = 0, len = nodeList.getLength(); i < len; i++) {
            Node node = nodeList.item(i);
            if (node.getNodeType() != Node.ELEMENT_NODE)
                continue;
            Result result = toResult(node);
            resultList.add(result);
        }
        action.setResults(resultList);
        return action;
    }

    private Result toResult(Node resultNode) throws Exception {
        Element resultElement = (Element)resultNode;
        Result result = new Result();
        result.setName(resultElement.getAttribute("name"));
        result.setType(resultElement.getAttribute("type"));
        result.setValue(resultElement.getAttribute("value"));
        return result;
    }
    public String getControllerXMLPath() {
        return controllerXMLPath;
    }
    public void setControllerXMLPath(String controllerXMLPath) {
        this.controllerXMLPath = controllerXMLPath;
    }
}
```

3）为了测试生成的 Controller 对象是否正确，就需要将该对象进行输出。通过 fastjson 可以将 Bean 对象转为 json，方便检验生成结果。在 SimpleController 项目的 pom.xml 中添加

如下配置：

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.28</version>
</dependency>
```

导入 fastjson 包后，在 Bean 对象（Controller、Action、Result）中重写 toString()方法，如下（只展示 Controller 类中的 toString()，其他两个同理）：

```
@Override
public String toString() {
    return JSON.toJSONString(this);
}
```

特别注意：Bean 对象中要提供公共的 getXXX()方法（XXX 为对象属性），否则 fastjson 无法获取该属性的值。

4）最后通过一些在线工具来将 json 串格式化，方便阅读，比如：https://www.sojson.com/editor.html。

3.7 Java 反射：action.execute()

SimpleController 工程中，southday.j2eework.sc.ustc.controller.bean 目录下的 Action 类提供方法：execute()，其通过反射机制来执行指定类（由 className 属性决定）的指定方法（由 methodName 属性决定）。方法 setArgs(Object... args)则用来设置要执行的指定方法的参数值。具体代码如下（已省略部分代码和注释）：

```
public class Action {
    private String actionName;
    private String className;
    private String methodName;
    private List<Result> results;
    private List<Object> argList = new ArrayList<>();

    public Result findResult(String name) {
        for (Result r : results)
            if (name.equals(r.getName()))
                return r;
        return null;
    }

    private Class<?>[] getArgTypes() {
        List<Class<?>> argsTypeList = new ArrayList<>();
        for (Object arg : argList)
            argsTypeList.add(arg.getClass());
        Class<?>[] res = new Class<?>[argsTypeList.size()];
        return argsTypeList.toArray(res);
```

```
    }

    public void setArgs(Object... args) {
        argList.clear();
        for (Object arg : args)
            if (arg != null)
                argList.add(arg);
    }

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

    @Override
    public String toString() {
        return JSON.toJSONString(this);
    }
}
```

3.8 请求转发与重定向

1）浏览器发送的请求（http://localhost:8080/UseSC/xxx）首先通过 SimpleController 处理，其根据 xxx 来判断要将请求分发给哪个子 Servlet 要后处理，或者找不到匹配的 action 时，就分发给 DefaultServlet。

```
public class SimpleController extends BaseServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        doPost(req, resp);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String servletName = req.getServletPath();
        String actionName =
servletName.substring(servletName.lastIndexOf("/")+1,
servletName.lastIndexOf("."));
        if (BaseConfig.ACTION_NAME_LOGIN.equals(actionName)) {
```

```
            req.getRequestDispatcher("/login.sc").forward(req, resp);
        } else if (BaseConfig.ACTION_NAME_REGISTER.equals(actionName)) {
            req.getRequestDispatcher("/register.sc").forward(req, resp);
        } else {
            req.getRequestDispatcher("/default.sc").forward(req, resp);
        }
    }
}
```

2）LoginServlet 收到 login.sc 请求后，开始进行登陆业务的处理。

- 如果找不到 action，则转发到：“unknown-action.jsp”；
- 如果 action.execute()执行异常，则重定向到：“failure.jsp”；
- 如果执行后所得结果 result==null，则转发到：“no-req-resources.jsp”；
- 否则根据 result 的 type 和 value 属性进行下一步处理：
  - type=”forward”，value=“success”，则转发到：“welcome.jsp”；
  - type=”redirect”，value=”failure”，则重定向到：“failure.jsp”；

```
public class LoginServlet extends BaseServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        doPost(req, resp);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        Action action =
controller.findAction(BaseConfig.ACTION_NAME_LOGIN);
        if (action == null) {

req.getRequestDispatcher(baseConfig.getValue("unkonwn-action.jsp")).forw
ard(req, resp);
            return;
        }
        action.setArgs(req.getParameter("userName"),
req.getParameter("password"));
        String resultName = null;
        try {
            resultName = action.execute();
        } catch (Exception e) {
            resp.sendRedirect(req.getContextPath() +
baseConfig.getValue("failure.jsp"));
            return;
        }
```

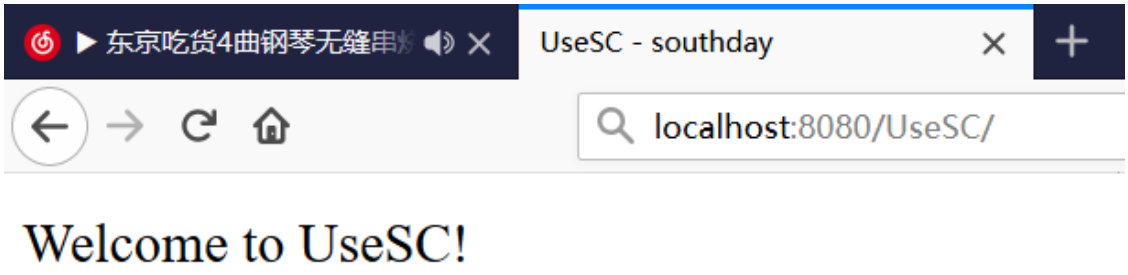
```
        Result result = action.findResult(resultName);
        if (result == null) {

req.getRequestDispatcher(baseConfig.getValue("no-req-resources.jsp")).fo
rward(req, resp);
            return;
        }
        if (BaseConfig.RESULT_TYPE_FORWARD.equals(result.getType())) {
            req.getRequestDispatcher(result.getValue()).forward(req,
resp);
        } else if
(BaseConfig.RESULT_TYPE_REDIRECT.equals(result.getType())) {
            resp.sendRedirect(req.getContextPath() + result.getValue());
        } else {

req.getRequestDispatcher(baseConfig.getValue("no-req-resources.jsp")).fo
rward(req, resp);
        }
    }
}
```

3）RegisterServlet 和 LoginServlet 的处理操作类似，这里不做过多的叙述。DefaultServlet 则是直接将请求转发到：“unknown-action.jsp”。

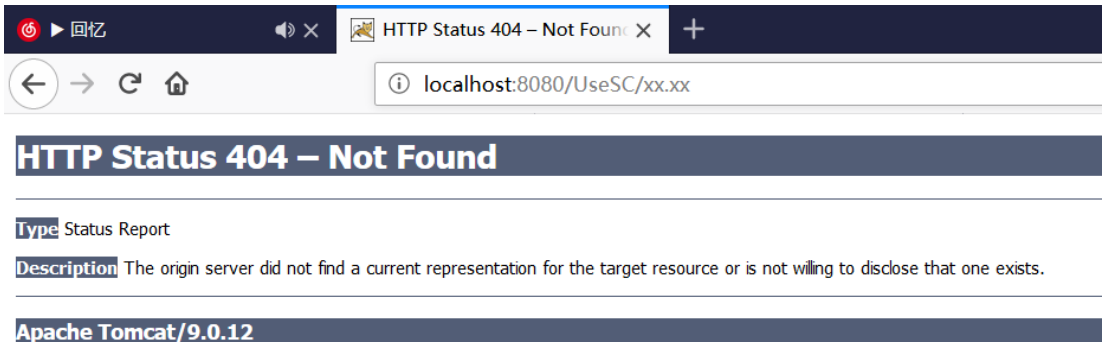
4）我觉得正常的逻辑应该是：网站向外提供 login.sc、register.sc 和其他 Action，当某些用户（未登录）想要越权使用某些功能时，服务端就会拦截该请求，然后转发到登陆页面。很显然，上面的处理逻辑不是这样的，需要改进。在实验 E3 中就会涉及到拦截器。

3.9 测试结果

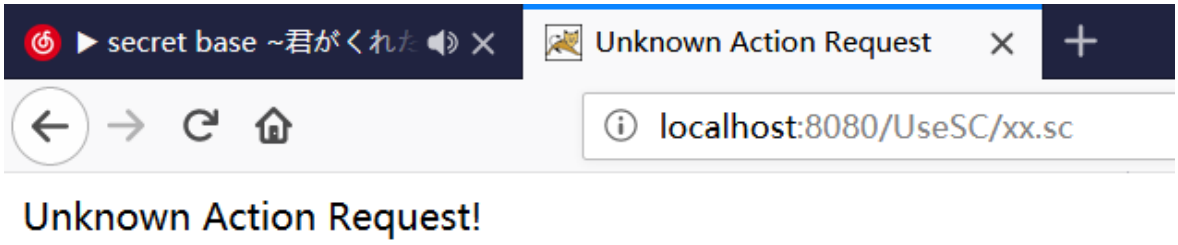
1）测试主页：http://localhost:8080/UseSC/，成功访问主页。



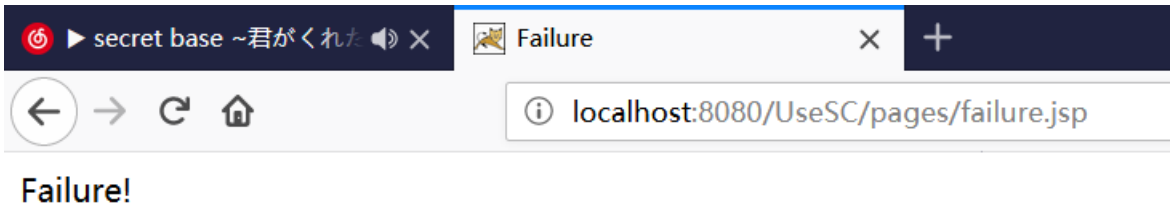
2）测试非法 action：http://localhost:8080/UseSC/xx.xx，404-Not Found。



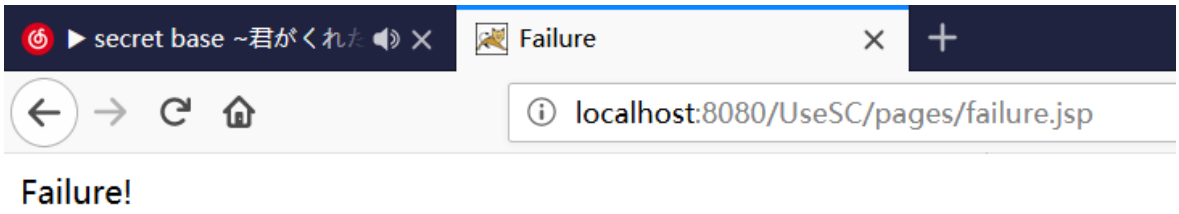
3）测试 xx.sc 请求：http://localhost:8080/UseSC/xx.sc，无法识别的 action。



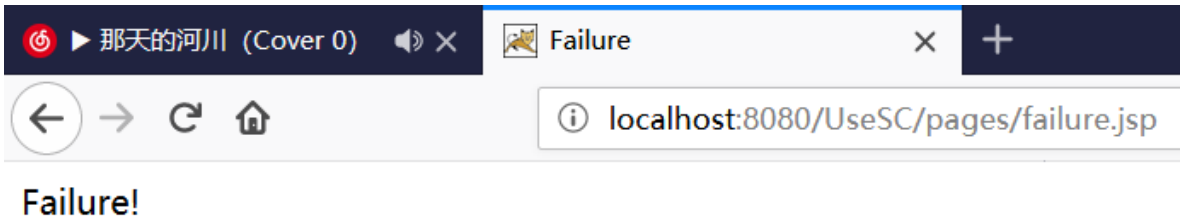
4）测试登陆：http://localhost:8080/UseSC/login.sc，失败，因为缺少参数：userName 和 password。



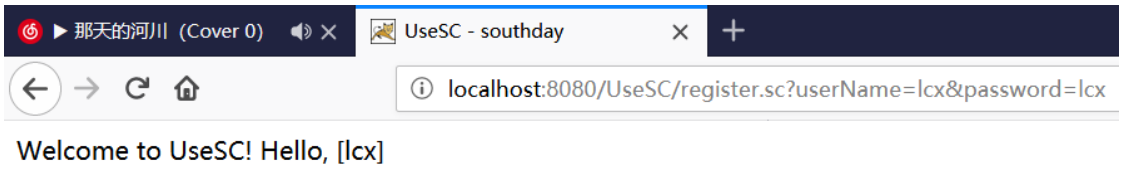
5）测试登陆：http://localhost:8080/UseSC/login.sc?userName=lcx，失败，因为缺少参数：password。



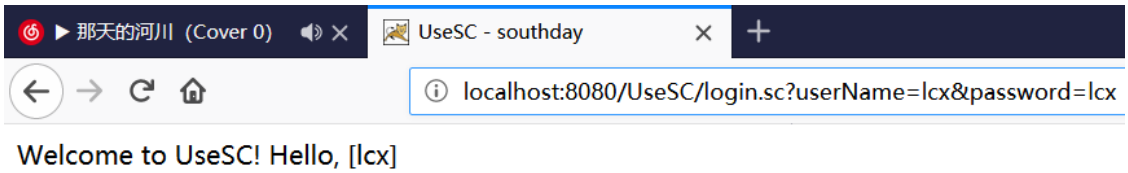
6）测试注册：http://localhost:8080/UseSC/register.sc?password=lcx。失败，因为缺少参数：userName。



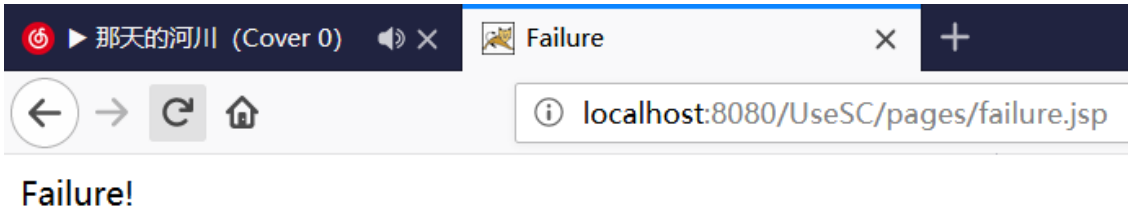
7）测试注册：http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx，成功，跳转到 welcome.jsp 页面（其实应该将用户信息保存在 session 中，然后重定向的）。



8）测试登陆（使用之前注册的账号 lcx）：http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx，成功，跳转到 welcome.jsp 页面。



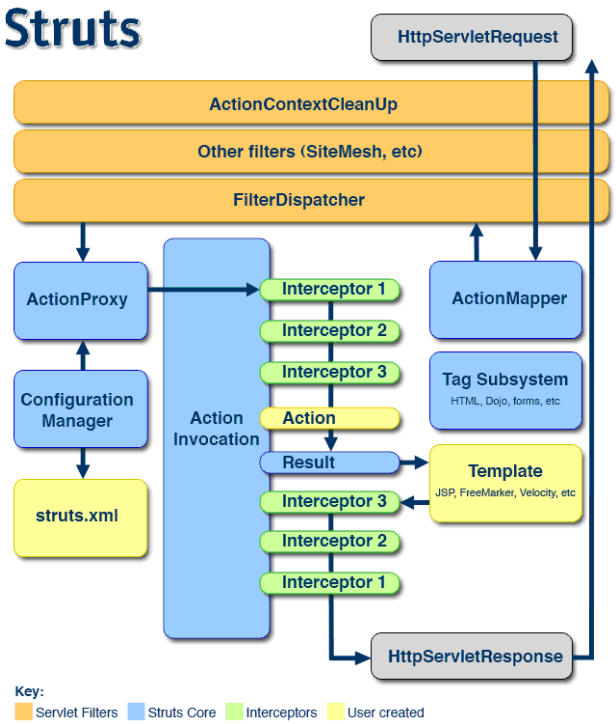
9）测试登陆（使用 lcx 账号，填写错误密码 xcl）：http://localhost:8080/UseSC/login.sc?userName=lcx&password=xcl，失败，因为密码错误。



3.10 对 Struts 2 控制器的理解

Struts2 框架由三部分构成：核心控制器、业务控制器和用户实现的业务逻辑组件。在这三部分中，struts2 框架提供了核心控制器 StrutsPrepareAndExecuteFilter，而用户需要实现业务控制层和业务逻辑层。

Struts2 请求处理流程如下：



1）客户端初始化一个指向 Servlet 容器（例如 Tomcat）的请求。

2）这个请求经过一系列的过滤器（Filter）（这些过滤器中有一个叫做 ActionContextCleanUp 的可选过滤器，这个过滤器对于 Struts2 和其他框架的集成很有帮助，例如：SiteMesh、Plugin）。

3）接着 FilterDispatcher 被调用，FilterDispatcher 询问 ActionMapper 来决定这个请求是否需要调用某个 Action。

4）如果 ActionMapper 决定需要调用某个 Action，FilterDispatcher 把请求的处理交给 ActionProxy。

5）ActionProxy 通过 Configuration Manager 询问框架的配置文件，找到需要调用的 Action 类。

6）ActionProxy 创建一个 ActionInvocation 的实例。

7）ActionInvocation 实例使用命名模式来调用，在调用 Action 的过程前后，涉及到相关拦截器（Intercepter）的调用。

8）一旦 Action 执行完毕，ActionInvocation 负责根据 struts、xml 中的配置找到对应的返回结果。返回结果通常是（但不总是，也可能是另外的一个 Action 链）一个需要被表示的 JSP 或者 FreeMarker 的模版。在表示的过程中可以使用 Struts2 框架中继承的标签。在这个过程中需要涉及到 ActionMapper。

3.11 基于配置、注解的控制器各自的优缺点

3.11.1 实现

1）基于 XML 配置的实现：

web.xml：

```
<web-app>
  <!-- ********************************* servlet ********************************* -->
  <servlet>
     <servlet-name>sc</servlet-name>
     <servlet-class>southday.j2eework.sc.ustc.controller.SimpleController</servlet-class>
  </servlet>

  <servlet>
     <servlet-name>login</servlet-name>
     <servlet-class>southday.j2eework.sc.ustc.controller.servlet.LoginServlet</servlet-class>
  </servlet>

  <servlet>
     <servlet-name>register</servlet-name>
     <servlet-class>southday.j2eework.sc.ustc.controller.servlet.RegisterServlet</servlet-class>
  </servlet>
  <!-- ********************************* servlet ********************************* -->

  <!-- ***************************** servlet-mapping ***************************** -->
  <servlet-mapping>
     <servlet-name>sc</servlet-name>
     <url-pattern>*.sc</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
     <servlet-name>login</servlet-name>
     <url-pattern>/login.sc</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
     <servlet-name>register</servlet-name>
     <url-pattern>/register.sc</url-pattern>
  </servlet-mapping>
  <!-- ***************************** servlet-mapping ***************************** -->
</web-app>
```

2）基于注解的实现：

SimpleController：

```
@WebServlet(name="sc", urlPatterns="*.sc")
public class SimpleController extends BaseServlet {// ...}
```

LoginServlet：

```
@WebServlet(name="login", urlPatterns="/login.sc")
public class LoginServlet extends BaseServlet {// ...}
```

RegisterServlet：

```
@WebServlet(name="register", urlPatterns="/register.sc")
public class RegisterServlet extends BaseServlet {// ...}
```

3.11.2 基于 XML 配置的优缺点

优点：

1）xml 作为可扩展标记语言最大的优势在于开发者能够为软件量身定制适用的标记，使代码更加通俗易懂。

2）利用 xml 配置能使软件更具扩展性。例如 Spring 将 class 间的依赖配置在 xml 中，最大限度地提升应用的可扩展性。

3）具有成熟的验证机制确保程序正确性。利用 Schema 或 DTD 可以对 xml 的正确性进行验证，避免了非法的配置导致应用程序出错。

4）修改配置而无需变动现有程序。

缺点：

1）需要解析工具或类库的支持。

2）解析 xml 势必会影响应用程序性能，占用系统资源。

3）配置文件过多导致管理变得困难。

4）编译期无法对其配置项的正确性进行验证，或要查错只能在运行期。

5）IDE 无法验证配置项的正确性无能为力。

6）查错变得困难。往往配置的一个手误导致莫名其妙的错误。

7）开发人员不得不同时维护代码和配置文件，开发效率变得低下。

8）配置项与代码间存在潜规则。改变了任何一方都有可能影响另外一方。

3.11.3 基于注解的优缺点

优点：

1）保存在 class 文件中，降低维护成本。

2）无需工具支持，无需解析。

3）编译期即可验证正确性，查错变得容易。

4）提升开发效率。

缺点：

1）若要对配置项进行修改，不得不修改 Java 文件，重新编译打包应用。

2）配置项编码在 Java 文件中，可扩展性差。

4. 结论

4.1 总结

通过此次实验，我学习和巩固了以下知识：

1）Java XML 解析。本次实验中我只使用了 DOM 的方式解析，不过我留了接口，下次实验中再通过其他方式解析 XML。

2）Java 反射的使用。在 Action 类的 execute()方法中，通过反射调用指定类的指定方法。

3）单例模式和工厂模式的运用。本次实验中，多次使用单例模式：SimpleUserDB、BaseConfig、ControllerConfig；对于 Bean 对象的创建，使用了工厂模式，定义了接口 Factory，为后期扩展提供支持。

4）了解了 Servlet 请求转发和重定向的区别和实现方式。

5）加强了 Java 编程能力，包括：包、类结构的设计与具体业务的实现。在实现过程中，考虑系统的可扩展性、健壮性、可维护性等，同时也避免过度设计，造成不必要的效能低下。

6）对 Struts2 控制器有了更深的理解。在完成该实验前，我没看过关于 Struts2 控制器的内容，但是写完代码，听老师讲到这部分内容后，发现我的设计和 Struts2 控制器的设计有几分相似，也算是个值得开心的事。

4.2 问题及看法

1）本次实验中，我没有实现 Model 层，而是在 Servlet 中把业务逻辑实现了。对于简单的业务逻辑还行，但对于复杂的业务，如果都放在 Servlet 中处理，会让 Servlet 类变得混乱不堪，无法扩展和维护。Servlet 应该只做任务分发的操作，具体业务的处理应该放到 Model 层来完成，这是在接下来的实验中需要改善的地方。

2）尽管基于扫描配置文件来动态处理请求，但 SmipleController 工程和 UseSC 工程却存在紧耦合，比如：controller.xml、files-locations.properties 文件的存放位置。作为一个框架提供者，需要将这些耦合的部分作为规约，让大家都遵守。

3）是否应该将 Action 类也单例化？答案是否定的，因为 Action 是有状态的（其内部有属性供外部配置）。如果实现单例，那么不同线程对 Action 里面的属性进行修改，就会造成程序错误。

4）单例模式的实现有很多方法，本例中使用的是静态内部类来实现。这种方式存在一个不足点：需要额外的工作（Serializable、transient、readResolve()）来实现序列化，否则每次反序列化一个序列化的对象实例时都会创建一个新的实例。

5）在实现 SimpleUserDB 时，我遇到了“鱼与熊掌不可兼得”的问题。SimpleUserDB 内部使用 ConcurrentHasMap 来实现，而该容器是自动扩容的，所以要想对容量大小进行控制就需要手动加锁，但是加锁后就无法体现 ConcurrentHashMap.put()的性能。本例中，最后我选择放弃对注册用户数量大小的控制，虽然这有可能导致恶意注册程序攻击，使得内存溢出。

4.3 编程心得

这一块单独提出来，是因为这次实验让我对这两个编程技术感触很深。

4.3.1 面向接口编程

面向接口编程，不一定要有实现，可以先把程序整体逻辑框架完成，再进行具体实现和优化。比如下面的代码：

- fac.create()是个空方法，具体内容还没实现；
- action.execute()也是空方法，为了表达程序逻辑而设置的存在；

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    // 先大概写个处理逻辑框架，之后再具体实现，后期再优化代码
    String servletName = req.getServerName();
    String actionName =
servletName.substring(servletName.lastIndexOf("/")+1,
servletName.lastIndexOf("."));
    Factory<Controller> fac = new SimpleControllerFactory();
    Controller controller = fac.create();
    Action action = controller.findAction(actionName);
    if (action == null) {
        // 响应客户端信息为：不可识别的 action 请求
        return;
    }
    // 利用反射 invoke clzz.method
    String resultName = action.execute();
    Result result = action.findResult(resultName);
    if (result == null) {
        // 响应客户端信息为：没有请求的资源
        return;
    }
    // 将 result 中的 value 指向的资源按 type 所定义的方式返回到客户端
}
```

除了提高系统的可扩展性、可维护性，这么做还有个好处：好多时候编写程序需要大块的时间段，而事实却不如意，一旦思路被打断（原因可能是：电话、吃饭、睡觉等），要再次进入状态就需要花一定时间。通过面向接口编程或伪码、注释等形式将程序的整体逻辑框架写出，一方面可以审视逻辑，排查逻辑错误，另一方面可以减少再次进入状态的时间，提高效率。

4.3.2 对象桩

比如我的 Controller 对象是需要通过扫描和解析 controller.xml 文件生成的，但是我想把 XML 解析的逻辑放在后面再来写，现在想先测试程序的整体流程是否正确。而整个流程的测试需要 Controller 对象，这时候就可以通过硬编码手动创建 Controller 对象。如下：

```
public static Controller getController() {
    Controller controller = new Controller();
    List<Action> actions = new ArrayList<>();
    Action a1 = new Action();
    a1.setActionName("login");
    a1.setClassName("southday.j2eework.water.ustc.action.LoginActio");
    a1.setMethodName("handleLogin");
```

```
    List<Result> results = new ArrayList<>();
    Result r1 = new Result();
    r1.setName("success");
    r1.setType("forward");
    r1.setValue("/pages/welcome.jsp");
    Result r2 = new Result();
    r2.setName("failure");
    r2.setType("redirect");
    r2.setValue("/pages/failure.jsp");

    results.add(r1);
    results.add(r2);
    a1.setResults(results);
    actions.add(a1);
    controller.setActions(actions);

    return controller;
}
```

5. 参考文献

[1] XML 解析——Java 中 XML 的四种解析方式：
https://www.cnblogs.com/longqingyang/p/5577937.html
[2] Java 处理 XML 的三种主流技术及介绍：
https://www.ibm.com/developerworks/cn/xml/dm-1208gub/index.html
[3] Java 解析 XML 文件：https://blog.csdn.net/zflovecf/article/details/78908788
[4] [java 开发篇][dom 模块] 遍历解析 xml：
https://www.cnblogs.com/liuzhipenglove/p/7232487.html
[5] 你真的会写单例模式吗-------Java 实现：
https://www.cnblogs.com/andy-zhou/p/5363585.html
[6] 快速理解 Java 中的五种单例模式：https://www.cnblogs.com/hupp/p/4487521.html
[7] 【Servlet】深入浅出 JavaServlet 重定向和请求转发：
https://www.cnblogs.com/HDK2016/p/7056935.html
[8] Java 获取工程路径的几种方法：https://www.iteblog.com/java_get_file_absolute_path/
[9] java 获得项目绝对路径：https://blog.csdn.net/rogerjava/article/details/7568466
[10] Struts1 和 Struts2 核心控制器的执行原理：
https://www.cnblogs.com/beijiguangyong/archive/2013/01/31/2890248.html
[11] Struts 2 Core 2.5.18 API：https://struts.apache.org/maven/struts2-core/apidocs/index.html
[12] Core Developers Guide：https://struts.apache.org/core-developers/
[13] Annotation 和 xml 各自作为配置项的优点与缺点：
https://blog.csdn.net/lzhang616/article/details/46681919
[14] Java 读取 .properties 配置文件的几种方式：
https://www.cnblogs.com/sebastian-tyd/p/7895182.html
[15] Java 读写 Properties 配置文件：https://www.cnblogs.com/xudong-bupt/p/3758136.html