轻量级 J2EE 框架应用

E 3　A Simple Controller with Interceptors

学号：SA18225170

姓名：李朝喜

报告撰写时间：2018/12/13

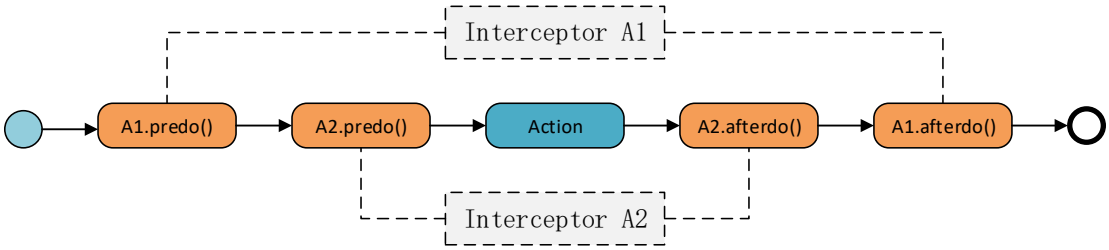
文档目录

1. 主题概述

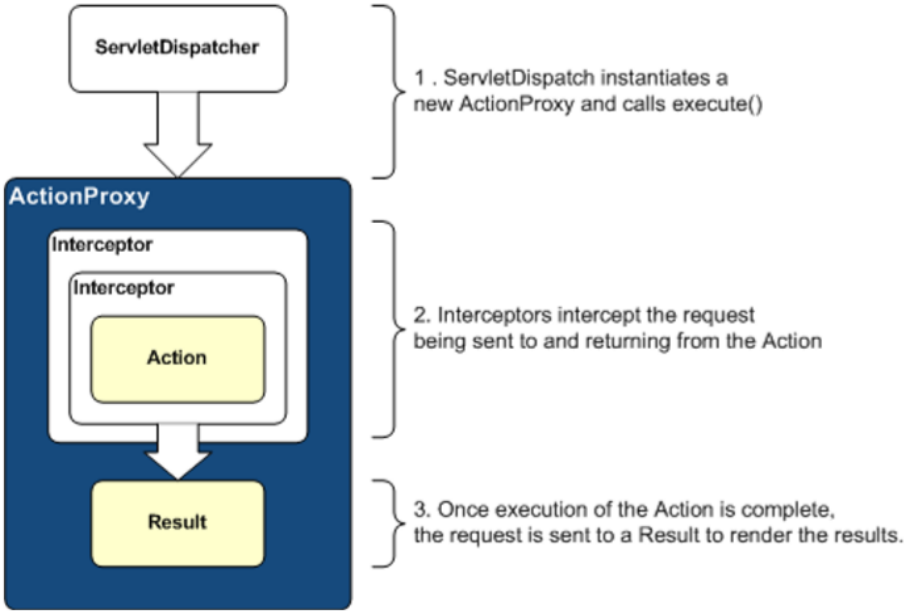
1.1 拦截器 Interceptor

拦截器是 Web 开发中常用的技术（面向切面编程—AOP：Aspect-Oriented Programming），其基于某种规则拦截外界对服务端资源的访问，并且可以实现：在访问资源前执行某个操作（predo），在访问资源后也执行某个操作（afterdo）。

拦截器的调用是一种栈式调用结构，如图：



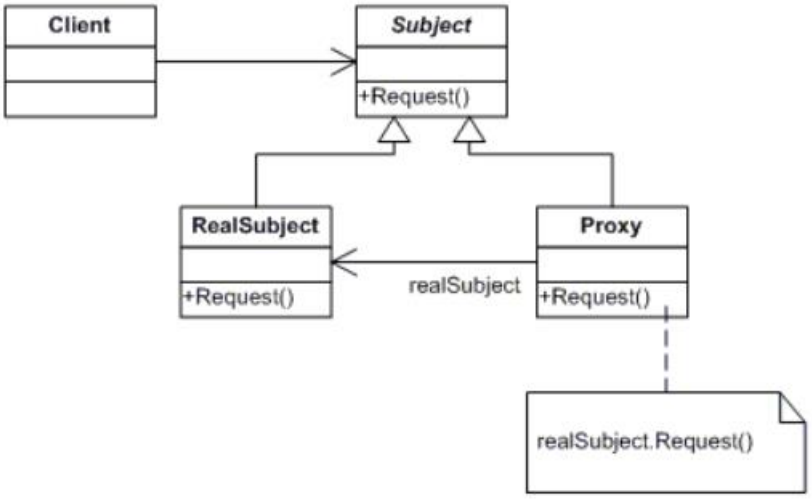
目前主流的 Web 后端开发框架都提供对拦截器的支持，用户可以自定义拦截器，然后通过可插拔的方式配置到 Action 中。拦截器基本上是通过动态代理的方式来实现的，如 struts2 中就会生成一个 ActionProxy 对象来代理实际的 Action 处理请求，而拦截器就作用于代理的过程中。如下图：



1.2 Java 代理模式：静态&动态

1.2.1 代理模式

定义：给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。UML 类图如下：



（图源：http://i.imgur.com/oh3VMNs.gif）

1.2.2 静态代理

若代理类在程序运行前就已经存在，那么这种代理方式被成为 静态代理，这种情况下的代理类通常都是我们在 Java 代码中定义的。通常情况下，静态代理中的代理类和委托类会实现同一接口或是派生自相同的父类。

1.2.3 动态代理

代理类在程序运行时创建的代理方式被成为 动态代理。这种情况下，代理类并不是在 Java 代码中定义的，而是在运行时根据我们在 Java 代码中的“指示”动态生成的。相比于静态代理，动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。

2. 假设

需要注意，实验 E3 是在 E2 基础上进行的，所以在做 E3 前请先完成 E2，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；
- 对线程安全和 Java 并发编程知识有基本了解，能够使用 Java 开发简单的多线程程序，并且通过某种加锁机制来同步各线程对资源的访问；

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
         |- ActionInterface.java                # Action类实现的接口，用于实现Java动态代理
         |- Result.java                         # 与controller.xml中定义的<result>标签相对应
         |- Interceptor.java                    # 与controller.xml中定义的<interceptor>标签相对应
         |- InterceptorRef.java                 # 与controller.xml中定义的<interceptor-ref>标签相对应
         |- SCConfiguration.java                # 与controller.xml中定义的<sc-configuration>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
         |- SCConfigurationFactory.java         # 用于生成SCConfiguration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（已弃用，改为dom4j解析）
         |- dom4j                               # dom4j包，基于DOM4J 的 XML解析来实现SCConfiguration对象创建的工厂
            |- DOM4JSCConfigurationFactory.java # 基于DOM4J的 XML解析来创建SCConfiguration对象的具体实现类
      |- proxy                                  # proxy包，包含与代理机制相关的类
         |- LogInvocationHandler.java           # 针对日志记录拦截器的InvocationHandler，用于实现Java动态代理
      |- service                                # service包，包含模型层（业务逻辑处理）的相关类
         |- UserService.java                    # 接口，定义了与User有关的相关业务逻辑的抽象方法
         |- impl                                # impl包，包含了针对模型层接口的具体实现类
            |- UserServiceImpl.java             # UserService接口的具体实现类
      |- 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
         |- ReflectUtil.java                    # 包含与反射处理相关的常用方法
```

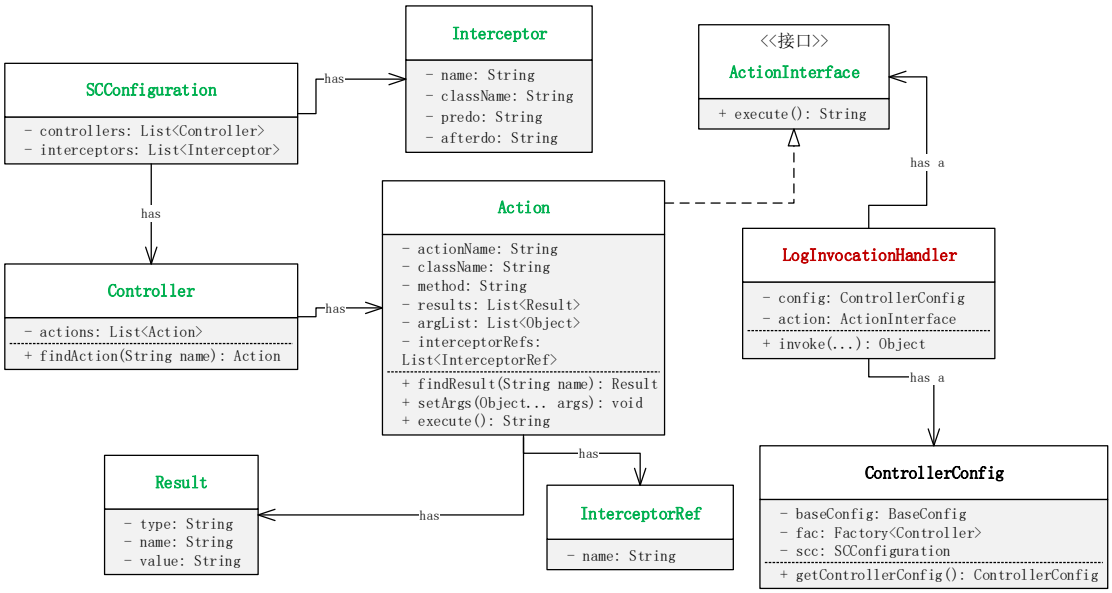
UseSC

```
|- 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
      |- interceptor                           # interceptor包，包含拦截器相关类
         |- LogInterceptor.java                # 日志记录拦截器
      |- log                                   # log包，包含与日志记录操作相关的类
         |- LogProperties.java                 # 用于加载和保存log.properties配置文件中的信息
         |- LogWriter.java                     # 用于写日志
         |- bean                               # bean包，包含日志文件xml中相对应的bean对象类
            |- Log.java                        # 与log.xml中定义的<log>标签相对应
            |- ActionLog.java                  # 与log.xml中定义的<action>标签相对应
   |- 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
         |- log.properties                     # 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
   |- 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）
```

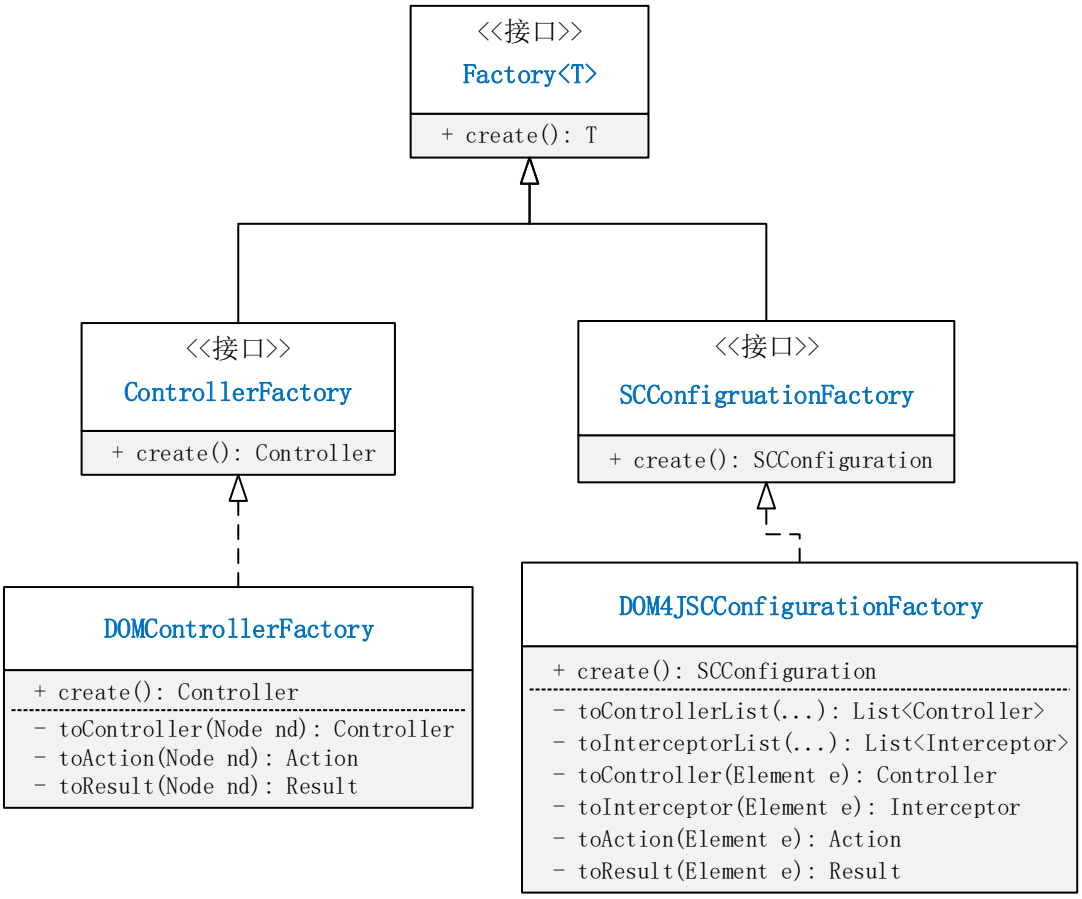
3.2 UML 类图

3.2.1 SimpleController UML 类图

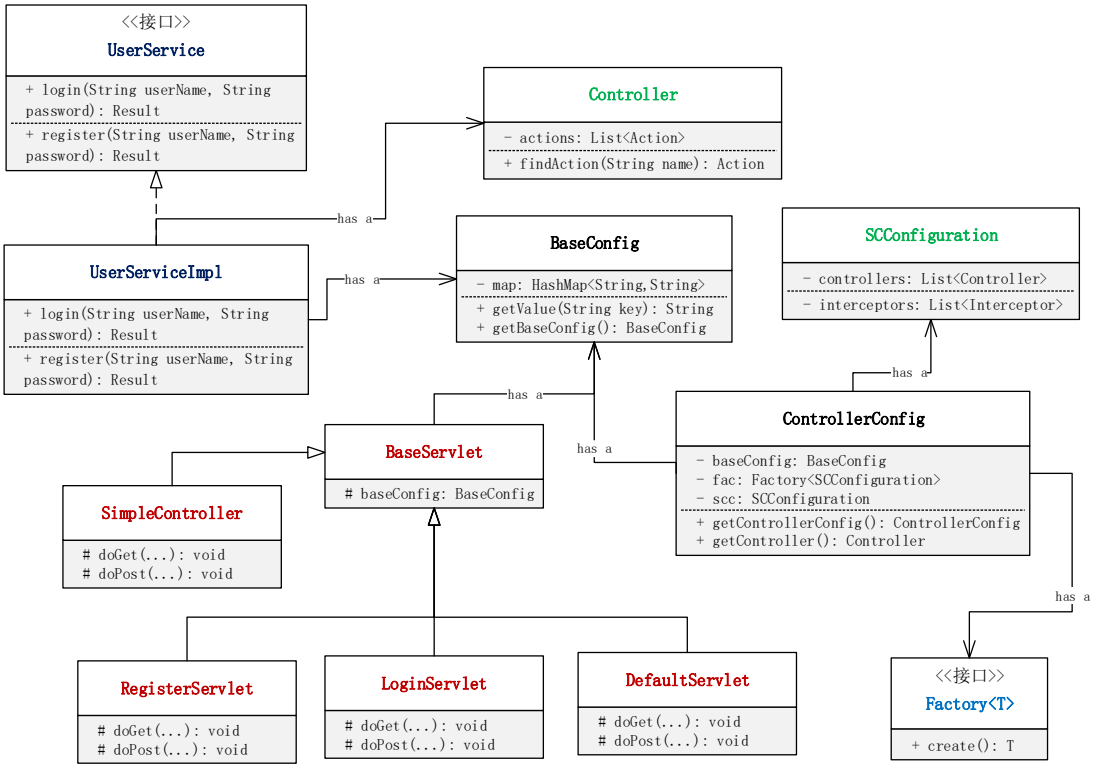
由于类比较多，一张图展示不了全部内容，所以我分片展示。



Bean 及相关类-UML 类图

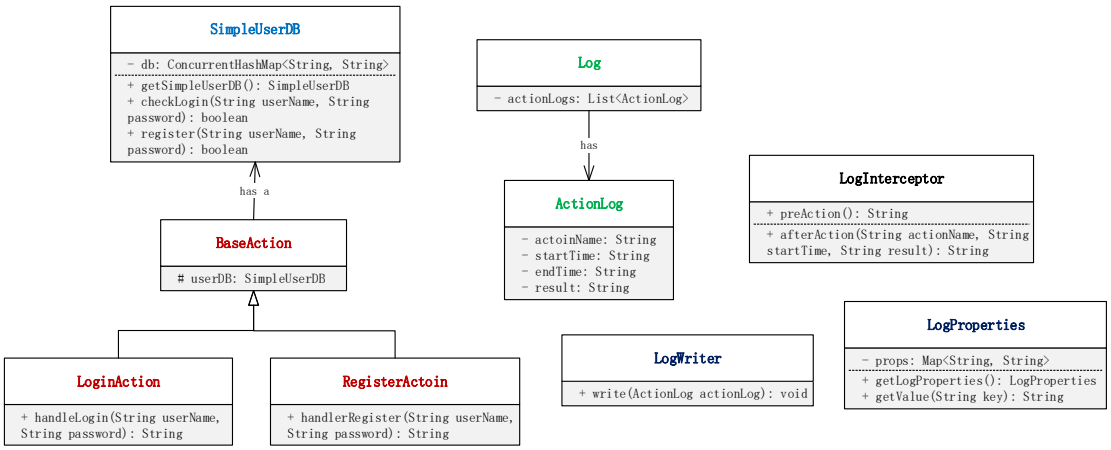


Factory-UML 类图

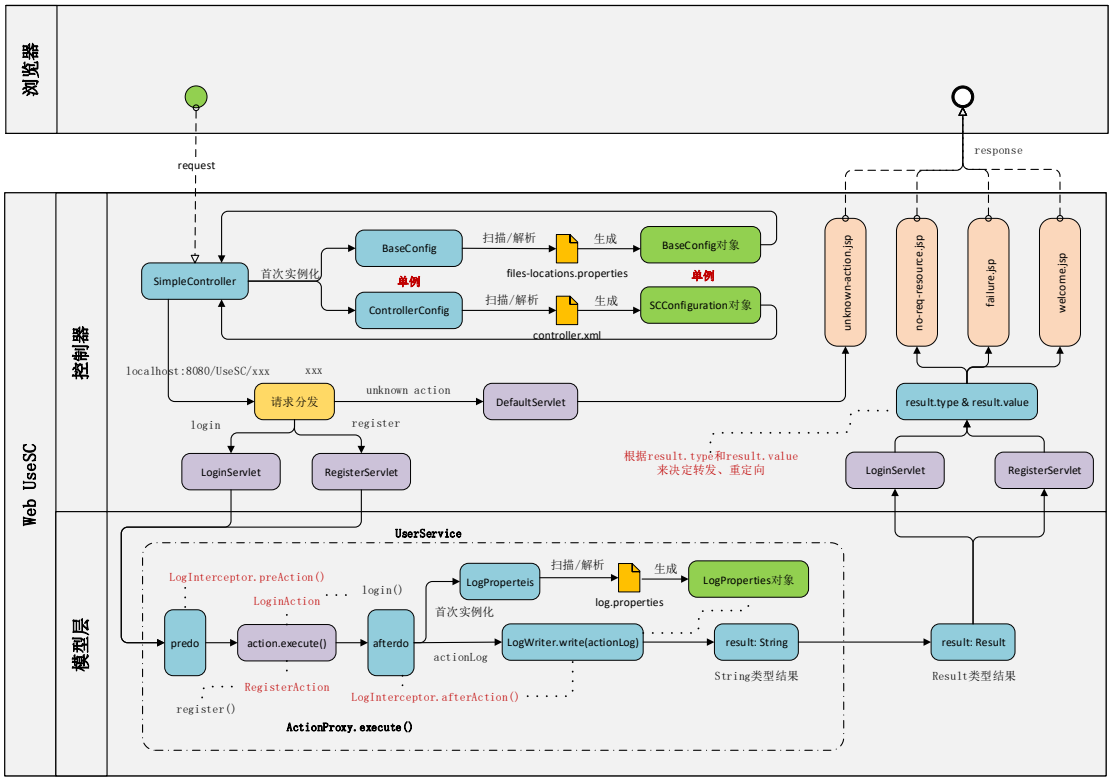


Servlet、Service 层相关类-UML 类图

3.2.2 UseSC UML 类图



3.3 请求处理流程图



3.4 日志记录

1）在 UseSC 工程的 src/main/resource/config 目录下添加 log.properties 文件，用于记录与日志操作相关的配置信息。目前只配置了日志文件 log.xml 的保存位置，如下：

```
log.xml=F:/J2EE-SC/log.xml
```

2）创建与 log.xml 中定义标签<log>、<action>相对应的 Bean 类：Log、ActioinLog；Log 中包含 List<ActionLog>，ActionLog 中有属性：name、startTime、endTime、result；

3）用单例模式实现 LogProperties 类，首次实例化该类时，就会去扫描 log.propreties 文件，然后将<key, value>配置信息保存在内置的 map 中。通过提供接口 getValue(String key)来返回指定配置信息；

4）编写 LogWriter 类，专注于写日志；内部使用 DOM4J 来实现与 xml 文件有关的操作，并且使用了多线程技术，每次外部调用 LogWriter.write(actionLog)方法时，都会新开一个线程去执行写日志的任务（需要对写日志操作进行同步处理，内存回收交给 GC 来处理），LogWriter.java 代码如下：

```
public class LogWriter {
//    private static final String LOG_XML_PATH = "F:/J2EE-SC/log.xml";
    private static final String LOG_XML_PATH =
LogProperties.getLogProperties().getValue("log.xml");
    private static final Document doc;
    private static final OutputFormat ofmt;

    static {
        File xmlFile = new File(LOG_XML_PATH);
        try {
```

```
            if (xmlFile.exists())
                doc = new SAXReader().read(xmlFile);
            else {
                doc = DocumentHelper.createDocument();
                doc.addElement("log");
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        // OutputFormat.createCompactFormat(); 是压缩版本的
        // OutputFormat.createPrettyPrint(); 是格式化版本的（可视）
        ofmt = OutputFormat.createPrettyPrint();
        ofmt.setEncoding("UTF-8");
    }

    public static void write(ActionLog actionLog) {
        new Thread(new Woker(actionLog)).start();
    }

    private static class Woker implements Runnable {
        private ActionLog actionLog;

        public Woker(ActionLog actionLog) {
            this.actionLog = actionLog;
        }

        @Override
        public void run() {
            addLog();
        }

        private void addLog() {
            Element root = doc.getRootElement();
            synchronized (LogWriter.class) {
                String threadName = "[" + actionLog.getActionName() + "-" +
Thread.currentThread().getName() + "]";
                System.out.println(threadName + " start log...");
                addActionLogNode(root);
                writeLog();
                System.out.println(threadName + " end log.");
            }
        }
```

```
        private void addActionLogNode(Element root) {
            Element actionElement = root.addElement("action");
            Element name = actionElement.addElement("name");
            name.setText(actionLog.getActionName());
            Element startTime = actionElement.addElement("s-time");
            startTime.setText(actionLog.getStartTime());
            Element endTime = actionElement.addElement("e-time");
            endTime.setText(actionLog.getEndTime());
            Element result = actionElement.addElement("result");
            result.setText(actionLog.getResult());
        }

        private void writeLog() {
            XMLWriter writer = null;
            try {
                writer = new XMLWriter(new FileWriter(new
File(LOG_XML_PATH)), ofmt);
                writer.write(doc);
            } catch (Exception e) {
                throw new RuntimeException(e);
            } finally {
                try {
                    if (writer != null)
                        writer.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

3.5 实现拦截器

3.5.1 配置文件与 Bean

1）在 UseSC 工程的 controller.xml 文件中添加与拦截器相关的配置，如下：

```
<sc-configuration>
    <interceptor name="log" class="southday.j2eework.water.ustc.interceptor.LogInterceptor"
        predo="preAction" afterdo="afterAction">
    </interceptor>
    <controller>
        <action name="login" class="southday.j2eework.water.ustc.action.LoginAction" method="handleLogin">
            <interceptor-ref name="log"></interceptor-ref>
            <result name="success" type="forward" value="/pages/welcome.jsp"></result>
            <result name="failure" type="redirect" value="/pages/failure.jsp"></result>
        </action>
        <action name="register" class="southday.j2eework.water.ustc.action.RegisterAction" method="handleRegister">
            <interceptor-ref name="log"></interceptor-ref>
            <result name="success" type="forward" value="/pages/welcome.jsp"></result>
        </action>
    </controller>
</sc-configuration>
```

2）在 SimpleController 工程的 bean 包中添加与 controller.xml 文件中定义的<sc-configuration>、<interceptor>、<interceptor-ref>标签相对应的 bean 类：SCConfiguration、Interceptor、InterceptorRef；

3）在 bean 包中定义 ActionInterface 接口，让 Action 类实现该接口，为后面使用动态代理实现拦截器做铺垫；ActionInterface.java 代码如下：

```
public interface ActionInterface {
    String execute() throws Exception;
    String getActionName();
    List<InterceptorRef> getInterceptorRefs();
}
```

3.5.2 XML 解析

1）在 E2 实验中使用了原生的 DOM 方式来解析 controller.xml 文件，本次实验中则使用 DOM4J 来解析 XML 文件，使用起来比原生的更方便；

2）在 factory 包中定义接口 SCConfigurationFactory，继承 Factory<SCConfiguration>，在 dom4j 包中定义实现类 DOM4JSCConfigurationFactory 来解析 controller.xml 文件，并最终生成 SCConfiguration 对象；由于代码篇幅较长，并且解析方式和 E2 的 DOMControllerFactory 中的类似，所以这里就不再贴出 DOM4JSCConfigurationFactory 的代码；

3.5.3 Java 动态代理

1）本次实验使用 Java 原生的动态代理方式来实现拦截器；创建包：southday.j2eework.sc.ustc.controller.proxy，在 proxy 包中定义 LogInvocationHandler 类，重写 invoke 方法，以实现日志记录的拦截功能；LogInvocationHandler.java 代码如下：

```
public class LogInvocationHandler implements InvocationHandler {
    // SimpleController 工程和 UseSC 工程紧耦合
    private static final String LOG_INTERCEPTOR_NAME = "log";
    ControllerConfig config = ControllerConfig.getControllerConfig();
    ActionInterface action;

    public LogInvocationHandler(ActionInterface action) {
        this.action = action;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        // 只对 execute()进行日志记录
        if (!"execute".equals(method.getName()))
            return method.invoke(action, args);
        List<InterceptorRef> refs = action.getInterceptorRefs();
        Interceptor logInterceptor = config.findInterceptor(refs,
LOG_INTERCEPTOR_NAME);
        if (logInterceptor == null)
```

```
            return method.invoke(action, args);
        // predo() -> startTime
        Object obj = config.getObjResource(logInterceptor.getClassName());
        String startTime =
ReflectUtil.execute1(logInterceptor.getClassName(),
logInterceptor.getPredo(), obj);
        String result = (String)method.invoke(action, args);
        // afterdo(actionName, startTime, result)
        Object afterdoArgs[] = {action.getActionName(), startTime, result};
        ReflectUtil.execute1(logInterceptor.getClassName(),
logInterceptor.getAfterdo(), obj, afterdoArgs);
        return result;
    }
}
```

2）LogInvocationHandler 类中调用的 ReflectUtil 类中的方法，如下：

```
public class ReflectUtil {
    public static Object newInstance(String className) throws Exception {
        Class<?> clzz = Class.forName(className);
        return clzz.getDeclaredConstructor().newInstance();
    }

    public static String execute1(String className, String methodName,
Object obj, Object... args) throws Exception {
        Class<?> clzz = Class.forName(className);
        Method method = clzz.getMethod(methodName, getArgTypes(args));
        String res = (String)method.invoke(obj, args);
        return res;
    }

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

3）特别注意，使用反射来 newInstance()是很耗时的，并且本次实验的 LogInterceptor 是无状态的，所以应该设置成单例的形式，一方面可以节省资源，另一方面可以加快响应速度；针对该问题，我在 ControllerConfig 类中设置了 Map<String, Object> objResources 属性，用于保存通过反射生成的单例对象，并且通过双重检测的方式来同步对 objResources 资源的写操作；每次外部都通过 getObjResources(String className)方法来获取资源对象，如果发现不存在，则通过反射机制来生成该对象，并 put 到 objResources 中；ControllerConfig 类中相

关的代码片段如下：

```
private Map<String, Object> objResources = new HashMap<>();

public Object getObjResource(String className) throws Exception {
    Object obj = objResources.get(className);
    if (obj == null) {
        synchronized (this) {
            obj = objResources.get(className);
            if (obj != null)
                return obj;
            obj = ReflectUtil.newInstance(className);
            objResources.putIfAbsent(className, obj);
        }
    }
    return obj;
}
```

3.6 模型层 Service

1）实验 E2 中没有模型层，业务逻辑处理都放在了控制器中（LoginServlet、RegisterServlet）；本次实验中，我划分出了模型层（也称业务层），将业务逻辑的处理从 Servlet 中抽取出来，放到 Service 中执行；让 Servlet 只对 Service 处理的结果进行操作（判断与视图分发等）；

2）在 SimpleController 工程中建包：southday.j2eework.sc.ustc.controller.service，用于存放模型层相关类、接口等；在 service 包下创建接口 UserService，并定义方法：login、register；

3）在 service 包下创建子包 impl，存放 service 中接口的具体实现类；在 impl 包下创建 UserServiceImpl 类，实现 UserService 接口；具体代码如下：

UserService.java：

```
public interface UserService {
    Result login(String userName, String password) throws Exception;
    Result register(String userName, String password) throws Exception;
}
```

UserServiceImpl.java：（register 和 login 逻辑类似，所以下面的代码中只贴出与 login 相关的部分）

```
public class UserServiceImpl implements UserService {
    private static final BaseConfig baseConfig = BaseConfig.getBaseConfig();
    private static final Controller controller =
ControllerConfig.getControllerConfig().getController();

    @Override
    public Result login(String userName, String password) throws Exception
{
```

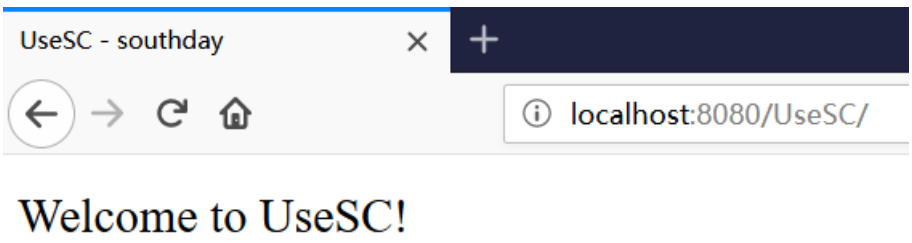
```
        Action action =
controller.getAction(BaseConfig.ACTION_NAME_LOGIN);
        Result result = new Result();
        if (action == null) {
            result.setType(BaseConfig.RESULT_TYPE_FORWARD);

result.setValue(baseConfig.getValue(BaseConfig.RESULT_VALUE_UNKNOWN_ACTI
ON));
            return result;
        }
        action.setArgs(userName, password);
        ActionInterface actionProxy = newProxyInstance(action);
        String resultName = null;
        try {
            resultName = actionProxy.execute();
        } catch (Exception e) {
            result.setType(BaseConfig.RESULT_TYPE_REDIRECT);

result.setValue(baseConfig.getValue(BaseConfig.RESULT_VALUE_FAILURE));
            return result;
        }
        Result realResult = action.getResult(resultName);
        if (realResult == null) {
            result.setType(BaseConfig.RESULT_TYPE_FORWARD);

result.setValue(baseConfig.getValue(BaseConfig.RESULT_VALUE_NO_REQ_RES))
;
            return result;
        }
        return realResult;
    }

    private ActionInterface newProxyInstance(Action action) {
        return (ActionInterface)Proxy.newProxyInstance(
                        ActionInterface.class.getClassLoader(),
                        new Class<?>[] {ActionInterface.class},
                        new LogInvocationHandler(action));
    }

}
```

4）从 UserServiceImpl 的代码中可以看到，处理结果被包装为 Result 对象返回；这样方便控制器 Servlet 统一处理结果，比如 LoginServlet 中的处理如下：

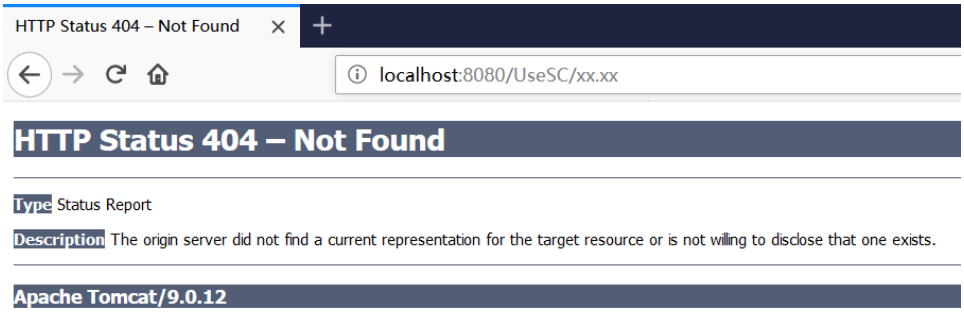
```
public class LoginServlet extends BaseServlet {
    private static final long serialVersionUID = 1L;

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

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        UserService userService = new UserServiceImpl();
        Result result = null;
        try {
            result = userService.login(req.getParameter("userName"),
req.getParameter("password"));
        } catch (Exception e) {
            resp.sendRedirect(req.getContextPath() +
baseConfig.getValue(BaseConfig.RESULT_VALUE_FAILURE));
            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(BaseConfig.RESULT_VALUE_NO_
REQ_RES)).forward(req, resp);
        }
    }
}
```

3.7 测试结果

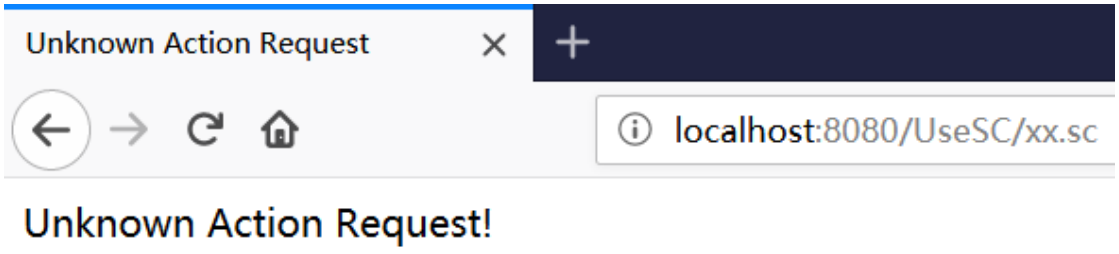
1）访问主页：http://localhost:8080/UseSC/，成功访问主页；



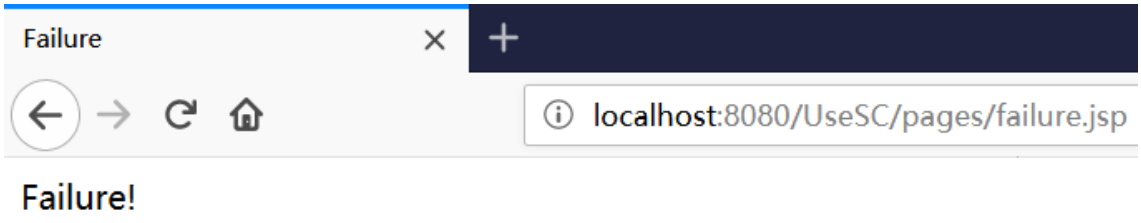
2）非法 URL：http://localhost:8080/UseSC/xx.xx，404-Not found；



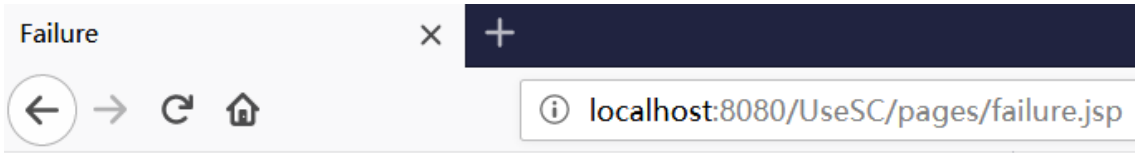
3）非法请求：http://localhost:8080/UseSC/xx.sc，无法识别的 action；



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

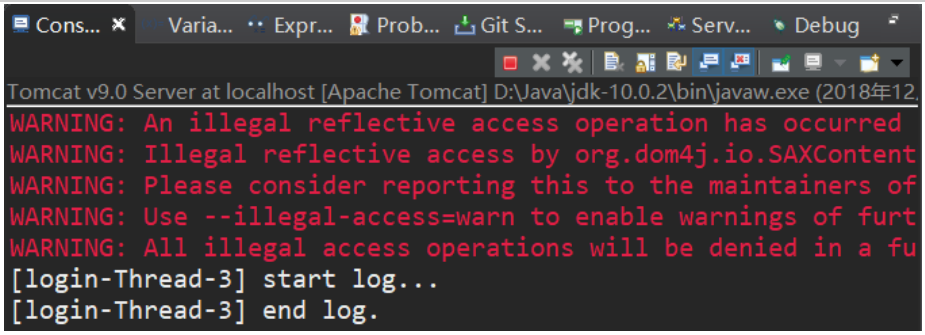


5）测试登陆：http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx，失败，用户账号 lcx 不存在；



控制台输出：

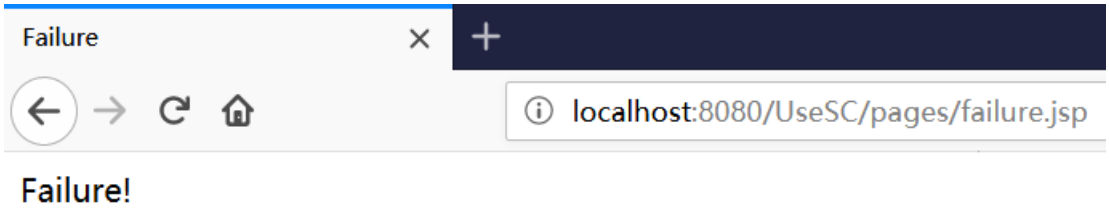
```
[login-Thread-3] start log...
[login-Thread-3] end log.
```



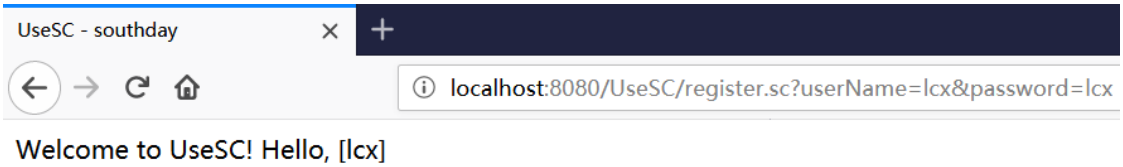
查看日志 F:\J2EE\log.xml：

```
<log>
  <action>
    <name>login</name>
    <s-time>2018-12-10 23:56:31</s-time>
    <e-time>2018-12-10 23:56:31</e-time>
    <result>failure</result>
  </action>
</log>
```

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；



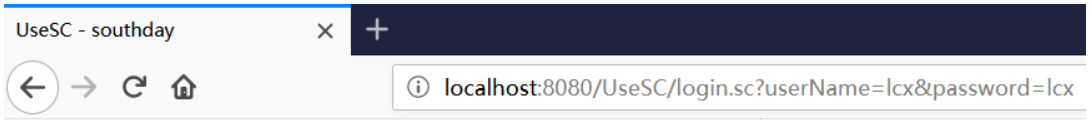
7）测试注册：http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx，注册成功，跳转到 welcome 页面；



查看日志 F:\J2EE\log.xml：

```
<log>
  <action>
    <name>login</name>
    <s-time>2018-12-10 23:56:31</s-time>
    <e-time>2018-12-10 23:56:31</e-time>
    <result>failure</result>
  </action>
  <action>
    <name>register</name>
    <s-time>2018-12-10 23:59:35</s-time>
    <e-time>2018-12-10 23:59:35</e-time>
    <result>success</result>
  </action>
</log>
```

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；

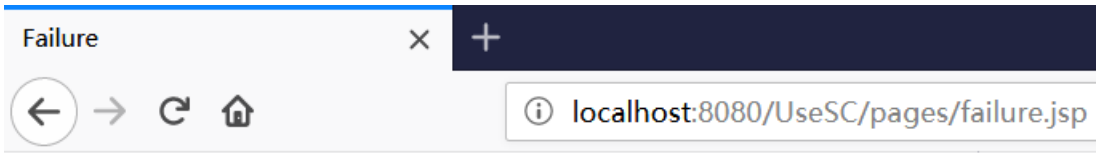


Welcome to UseSC! Hello, [lcx]

查看日志 F:\J2EE\log.xml：

```
<log>
  <action>
    <name>login</name>
    <s-time>2018-12-10 23:56:31</s-time>
    <e-time>2018-12-10 23:56:31</e-time>
    <result>failure</result>
  </action>
  <action>
    <name>register</name>
    <s-time>2018-12-10 23:59:35</s-time>
    <e-time>2018-12-10 23:59:35</e-time>
    <result>success</result>
  </action>
  <action>
    <name>login</name>
    <s-time>2018-12-11 00:01:56</s-time>
    <e-time>2018-12-11 00:01:56</e-time>
    <result>success</result>
  </action>
</log>
```

9）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!

查看日志 F:\J2EE\log.xml：

```
<log>
  <action>
    <name>login</name>
    <s-time>2018-12-10 23:56:31</s-time>
    <e-time>2018-12-10 23:56:31</e-time>
    <result>failure</result>
  </action>
  <action>
    <name>register</name>
    <s-time>2018-12-10 23:59:35</s-time>
```

```
    <e-time>2018-12-10 23:59:35</e-time>
    <result>success</result>
  </action>
  <action>
    <name>login</name>
    <s-time>2018-12-11 00:01:56</s-time>
    <e-time>2018-12-11 00:01:56</e-time>
    <result>success</result>
  </action>
  <action>
    <name>login</name>
    <s-time>2018-12-11 00:04:01</s-time>
    <e-time>2018-12-11 00:04:01</e-time>
    <result>failure</result>
  </action>
</log>
```

3.8 MVC 中 Controller 的功能及其合理性

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

结合 E1、E2、E3 及其他相关经验，我给出如下表格：

Controller 可具备的功能	是否合理	理由
1.接收请求	Y	
2.过滤器	Y	
3.拦截器	Y	
4.请求分发	Y	
5.业务逻辑处理	N	业务逻辑处理应该放在 Model 层来实现
6.视图渲染	N	视图方面的处理应该放在 View 层来实现
7.调用模型层来处理业务	Y	
8.调用视图层来响应客户	Y	

4. 结论

4.1 总结

通过本次实验，我学会了如何使用 Java 动态代理来实现拦截器，同时也对 Web 后端框架中拦截器的实现有了更深的认识。

相较与 E2，我重构了部分代码，一方面是由于新的需求，另一方面是由于之前的设计存在不合理的地方。重构主要体现在：

1）ControllerConfig 类，将根元素对象由 Controller 转为 SCConfiguration；

2）添加了 ReflectUtil 工具类，将涉及到反射的操作都封装在该工具类中；

3）增加了 Model 层（UserService、UserServiceImpl），将业务逻辑的处理从 Servlet 中抽离出来，放到 Model 层实现；

4）Model 层对于 login()和 register()操作返回的不再是 String 对象，而是封装为 Result 对象返回，这样便于 Servlet 统一处理结果；

5）解析 controller.xml 使用新的工厂 DOM4JSCConfigurationFactory，方便且高效；

6）通过动态代理 actionProxy.execute()来执行相关操作，满足了日志记录的需求；

在做该实验的过程中，并非一帆风顺，我也遇到了好多需要抉择、权衡的问题。而程序设计、编程能力的提升，就是在思考、博弈、猜测、验证的过程中产生的。

4.2 问题及看法

4.2.1 InvocationHandler 无法复用问题

本例中之所以将 InvocationHandler（原命名为 ActonInvocationHandler）改名为 LogInvocationHandler，正是因为其实现无法复用。为了避免混淆，特意标注为 Log。

上面所说的复用为：为多个拦截器 List<Interceptor>提供支持，因为 action 中会注册不只一个拦截器<interceptor-ref>，所以需要实现一个栈式调用的拦截器链。

造成无法复用的原因：反射调用拦截器的 preAction()、afterAction()时，传入的参数不统一。如：LogInterceptor 的 preAction()参数为空，而 afterAction()需要参数 actionName、startTime、result。若此时又添加一个拦截器，其 preAction()和 afterAction()又是另一种定义，那么目前的 LogInvocationHandler 是无法提供支持的。

解决这个问题的关键就是：让所有拦截器的 preAction()、afterAction()都接收统一的参数（规定），这样在反射调用时就可以泛化；此外，还要注意拦截器中参数与结果的传递，以确保外部程序、拦截器能获取到其所需要的数据。参考 SpringMVC 中的拦截器实现，它就是统一了参数，如下：

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response,
                         Object handler) throws Exception {
    return true;
}
```

4.2.2 ActionInterface 接口方法混乱问题

因为 Java 动态代理需要使用接口来完成，所以我创建了 ActionInterface 接口。本来只在该接口中定义了 execute()方法，但是后来因为 LogInvocationHandler 中实现的需要，又往里面加了 getActionName()、getInterceptorRefs()方法。这就让我感觉接口 ActionInterface 的设计有点不伦不类：其存在的意义只是因为动态代理的需要，而非设计上的存在合理性。

当然，在动态生成代理对象时，也可以直接用 Action.class 来生成，即：不需要通过 ActionInterface 接口也可以达到效果。但是查阅了一些动态代理的实现原理后，知道生成的代理类对象，会对被代理对象的所有方法都进行代理（进一步封装）。所以，考虑到效率问题，才使用 ActionInterface 接口的，毕竟 Action 类中的方法比较多。

为了避免在调用 actionProxy 的其他方法（getActionName()、getInterceptorRefs()）时也记录日志，就需要在 invoke()方法中添加方法名的判断：只对 execute()方法进行日志记录。

4.2.3 日志记录的线程相关问题

1）本次实验中所有的日志都写入同一个文件 log.xml，这就需要考虑对该文件并发操作的同步问题。

2）既然需要同步，就说明会存在阻塞的情况。然而太慢的响应对用户显然是不友好的，所以写日志的操作不能是单线程的。

3）考虑到要实现多线程写日志，如果我们对每一个请求都创建一个线程去执行任务，那么高并发情况下，我们的服务器将不堪重负。因为创建过多的线程占用了大量的内存资源，而对这些资源的回收（GC）是不确定的。

4）既然如此，那么就只能实现一个线程池，自己来管理线程资源。设置任务阻塞队列，当有线程空闲时，就从任务队列中取一个任务去执行，这样就能保证服务器内存资源不被无节制的占用。但是，结合我目前的状况（各种作业、编程能力不足等）以及 E3 实验的权重，我决定不花时间去写线程池，直接用简单粗暴的 1 个请求 1 个线程的方式来实现。

4.2.4 日志记录的效率相关问题

本次实验中，针对每次请求都要写日志，这样好吗？撇开创建线程和线程资源占用问题，光是 I/O 操作也是不小的开销，虽然是异步执行，但是同样需要 CPU 时间。

比较普遍的做法是，每隔一段时间（1 天、1 小时等）写一次日志，在这期间先将数据保存在内存中或内置数据库中。或者我们也可以实现每个 n 次请求就写一次日志，这些都要比每次请求写一次日志要好的多。

另一个要注意的问题，每次我们写日志都是在之前的日志中进行追加的，就说明在这之前我们要把原文件（log.xml）中的所有节点都加载到内存中。这样的实现，当日志文件过大时，读取和解析 log.xml 就有一笔不小的时间开销，此外还有较大的内存开销，显然不太现实。解决的方法，其实上面已经提到了，就是以时间来切片，将日志分为不同的部分，这样每次只需读取指定的小片段，就不会造成过高的时间和空间的开销。

5. 参考文献

[1] java 中采用 dom4j 解析 xml 文件：https://www.cnblogs.com/hongwz/p/5514786.html
[2] dom4j 解析 xml 字符串实例：http://www.cnblogs.com/superjt/p/3310307.html
[3] DOM4J 创建 XML 文件和追加元素节点：
https://blog.csdn.net/chianz632/article/details/80358073
[4] 使用 XStream 实现 Java 对象与 XML 互相转换：
https://segmentfault.com/a/1190000012435867
[5] java 动态代理实现与原理详细分析：https://www.cnblogs.com/gonjan-blog/p/6685611.html