轻量级 J2EE 框架应用

E 7　A Simple Controller with DI

学号：SA18225170

姓名：李朝喜

报告撰写时间：2018/12/30

文档目录

1. 主题概述

1.1、1.2、1.3 的内容均摘自文章：“架构师之路】依赖注入原理---IoC 框架”。

1.1 IOC

IOC 是 Inversion of Control 的缩写，多数书籍翻译成“控制反转”，还有些书籍翻译成为“控制反向”或者“控制倒置”。

1996 年，Michael Mattson 在一篇有关探讨面向对象框架的文章中，首先提出了 IOC 这个概念。简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC 理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦，如下图：

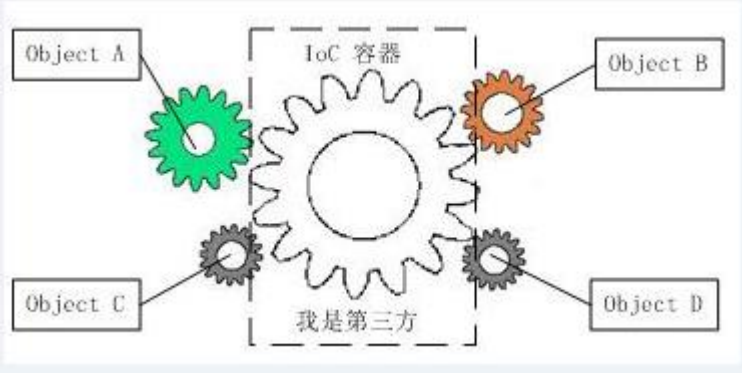


图 1（图源：https://images2018.cnblogs.com/blog/711792/201808/711792-20180802091842184-469210910.png）

1.2 “控制反转”的由来

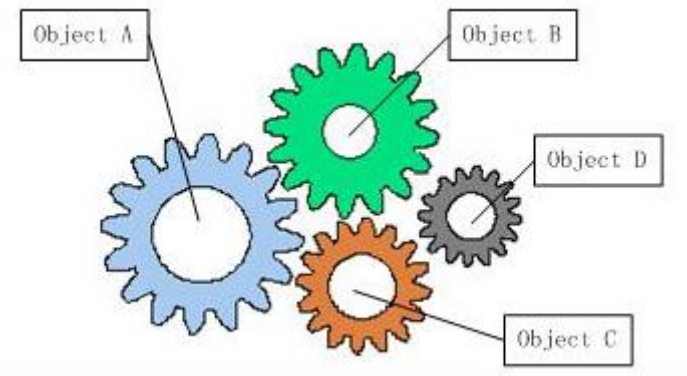


图 2（图源：https://images2018.cnblogs.com/blog/711792/201808/711792-20180802091728701-343878555.png）

软件系统在没有引入 IOC 容器之前，如图 2 所示，对象 A 依赖于对象 B，那么对象 A 在初始化或者运行到某一点的时候，自己必须主动去创建对象 B 或者使用已经创建的对象 B。无论是创建还是使用对象 B，控制权都在自己手上。

软件系统在引入 IOC 容器之后，这种情形就完全改变了，如图 1 所示，由于 IOC 容器的加入，对象 A 与对象 B 之间失去了直接联系，所以，当对象 A 运行到需要对象 B 的时候，IOC 容器会主动创建一个对象 B 注入到对象 A 需要的地方。

通过前后的对比，我们不难看出来：对象 A 获得依赖对象 B 的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

1.3 DI

依赖注入 DI，其实是 IOC 的别名，历史渊源如下：

2004 年，Martin Fowler 探讨了同一个问题，既然 IOC 是控制反转，那么到底是“哪些方面的控制被反转了呢？”，经过详细地分析和论证后，他得出了答案：“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由 IOC 容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖注入（DI：Dependency Injection）”。他的这个答案，实际上给出了实现 IOC 的方法：注入。所谓依赖注入，就是由 IOC 容器在运行期间，动态地将某种依赖关系注入到对象之中。

2. 假设

需要注意，实验 E7 是在 E6 基础上进行的，所以在做 E7 前请先完成 E6，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；
- 了解单例模式，能够使用 Java 实现单例模式；
- 了解 Java 反射机制，能够利用 Java 反射来调用某个类中的方法；
- 能够使用 DOM4J 解析 XML 文件；

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
- 已安装好 MySQL：https://dev.mysql.com/downloads/mysql/
- 已安装好 PostgreSQL：
  https://www.enterprisedb.com/downloads/postgres-postgresql-downloads

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

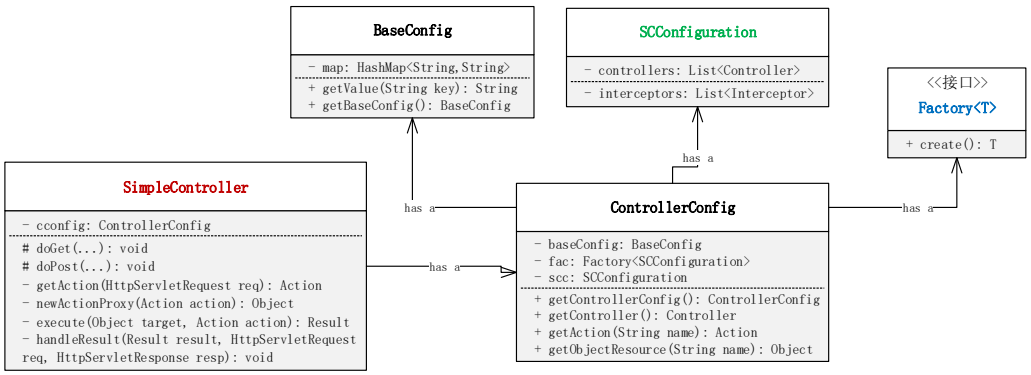
3.1 项目包结构及简要描述

```
SimpleController
  |- src/main/java
    |- southday.j2eework.sc.ustc.controller        # 包根目录
      |- SimpleController.java                     # Servlet，用于处理*.sc的请求
      |- action                                    # action包，里面包含了Action接口定义及默认实现类等
        |- ActionIface.java                        # Action接口
        |- ActionSupport.java                      # Action的默认实现类，框架使用者可以继承该类
        |- DefaultAction.java                      # 默认Action，当未找到匹配的action时，就返回DefaultAction
      |- 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
          |- 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
          |- ActionFactory.java                    # 用于生成Action对象的工厂
          |- ResultFactory.java                    # 用于生成Result对象的工厂
          |- ControllerFactory.java                # 用于生成Controller对象的工厂
          |- SCConfigurationFactory.java           # 用于生成SCConfiguration对象的工厂
          |- DOM4JSCConfigurationFactory.java      # 基于DOM4J的 XML解析来创建SCConfiguration对象的具体实现类
      |- dao                                       # dao包，包含各类DAO
        |- BaseDAO.java                            # 基本DAO，作为其他DAO的父类存在
        |- DBConfiguration.java                    # 用于管理数据源DataSource，因为配有MySQL和PostgreSQL，并且多个类中都用到了DataSource
      |- di                                        # di包，包含依赖注入的相关处理类
        |- DICongiuration.java                     # di.xml在内存中的映射总类（单例）
        |- DIer.java                               # 依赖注入器，用于完成依赖注入
        |- SCDIFactory.java                        # 解析di.xml的工厂类
        |- bean                                    # bean包，包含与UseSC中di.xml文件里定义内容相对应的JavaBean对象类
          |- DIBean.java                           # 与di.xml中定义的<bean>标签相对应
          |- DIField.java                          # 与di.xml中定义的<field>标签相对应
          |- SCDI.java                             # 与di.xml中定义的<sc-di>标签相对应
      |- factory                                   # factory包，目前包含工厂的抽象定义（接口Factory）
        |- Factory.java                            # 抽象工厂，提供接口：T create() throws Exception;
      |- interceptor                               # interceptor包，用于存放拦截器的相关类、接口等
        |- ParameterInterceptor.java               # 参数拦截器，目前仅用于给Action代理对象（动态）填充参数
      |- orm                                       # orm包，包含了处理O/R Mapping的相关类
        |- ORMConfiguration.java                   # or_mapping.xml在内存中的映射总类（单例）
        |- ORMConversation.java                    # 实现与数据库的交互，将交互操作封装为对对象的操作
        |- ORMappingFactory.java                   # 解析or_mapping.xml的工厂类
        |- SQLFactory.java                         # 用于构造SQL语句的工厂，目前提供查询（SELECT）和添加（INSERT）语句的构造
        |- bean                                    # bean包，包含与UseSC中or_mapping.xml文件里定义内容相对应的JavaBean对象类
          |- ID.java                               # 与or_mapping.xml中定义的<id>标签相对应
          |- ORMapping.java                        # 与or_mapping.xml中定义的<OR-Mapping>标签相对应
          |- ORMClass.java                         # 与or_mapping.xml中定义的<class>标签相对应
          |- Property.java                         # 与or_mapping.xml中定义的<property>标签相对应
      |- proxy                                     # proxy包，包含与代理机制相关的类
        |- ProxyFactory.java                       # 代理工厂，用于生成代理类对象
        |- cglib                                   # cglib包，关于Action的动态代理，使用cglib技术来实现
          |- LogActionProxy.java                   # 针对日志记录的Action代理拦截器
          |- LazyLoadProxy.java                    # 针对PO对象属性实现延迟加载的代理类
      |- transformer                               # transformer包，包含各类转换器
        |- XML2HTMLTransformer.java                # 将xml转为html的转换器，目前使用xslt技术实现
      |- util                                      # util包，包含项目工具类
        |- CommonUtil.java                         # 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
        |- FileUtil.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
        |- ReflectUtil.java                        # 包含与反射处理相关的常用方法
        |- DBUtil.java                             # 包含与数据库操作相关的常用方法，目前提供insert()、fillParams()方法
```

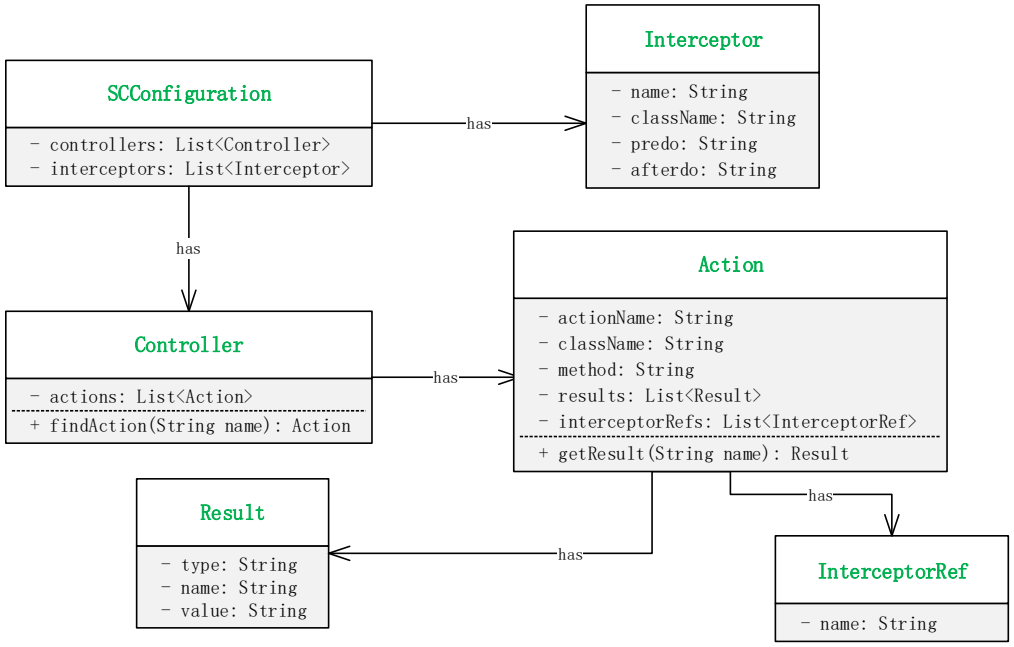
```
UseSC
  |- src/main/java
    |- southday.j2eework.water.ustc          # 包根目录
      |- action                              # action包，包含了具体业务对应的action类
        |- LoginAction.java                  # 登陆业务对应的Action
        |- RegisterAction.java               # 注册业务对应的Action
        |- LogoutAction.java                 # 退出登陆对应的Action
      |- bean                                # bean包
        |- User.java                         # UserBean
      |- 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
      |- 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
      |- dao                                 # dao包，包含各类DAO
        |- UserDAO.java                      # 实现BaseDAO的具体类，包含query, insert, update, delete等方法的实现
  |- 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
      |- jdbc-mysql.properties               # jdbc-mysql.properties，里面包含了DBCP用于创建、管理数据库连接的配置属性（针对MySQL）
      |- jdbc-postgresql.properties          # jdbc-postgresql.properties，里面包含了 DBCP用于创建、管理数据库连接的配置属性（针对PostgreSQL）
      |- or_mapping.xml                      # or_mapping.xml，O/R Mapping配置文件
      |- di.xml                              # di.xml，里面包含了有关依赖注入的相关配置信息
  |- src/main/webapp                         # Web项目部署包
    |- 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）
      |- success_view.xml                    # “登陆成功”页面的XML配置
      |- success_view.xsl                    # “登陆成功”页面的XSL配置
```

3.2 UML 类图

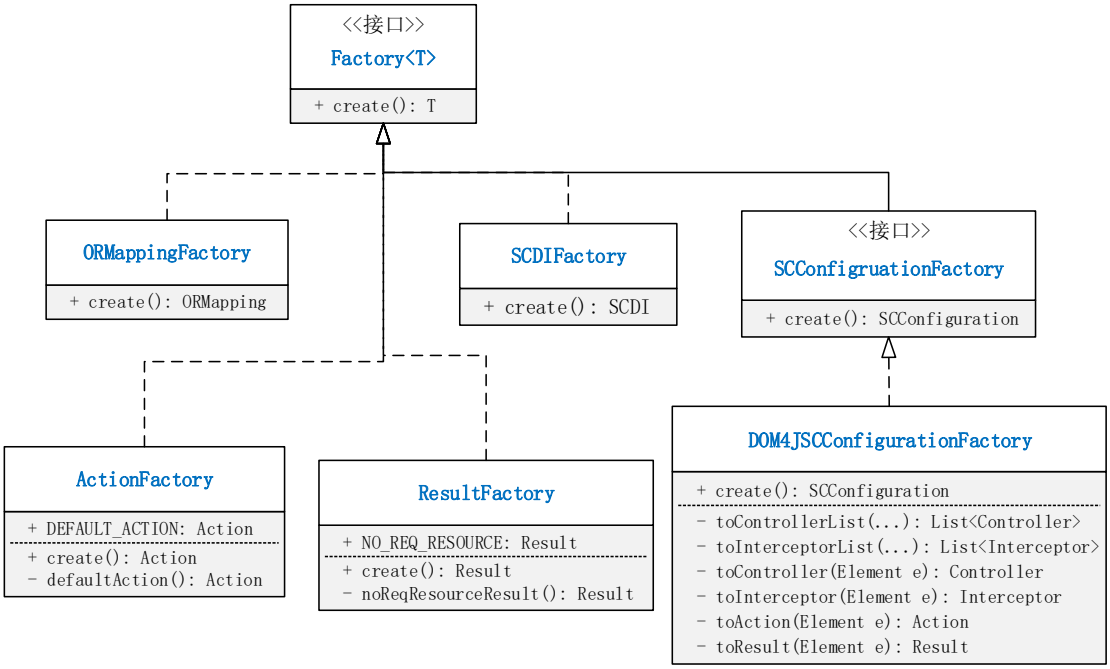
3.2.1 SimpleController UML 类图



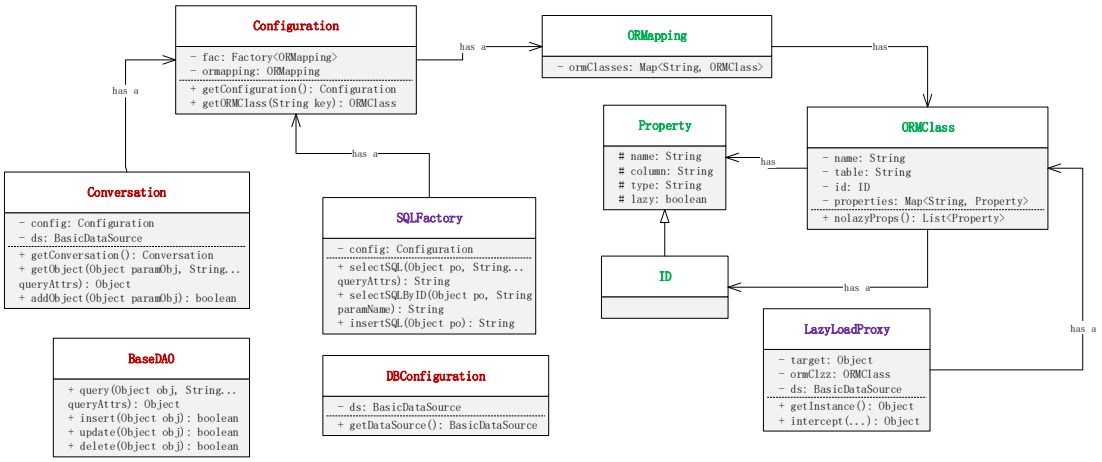
Servlet 相关类-UML 类图



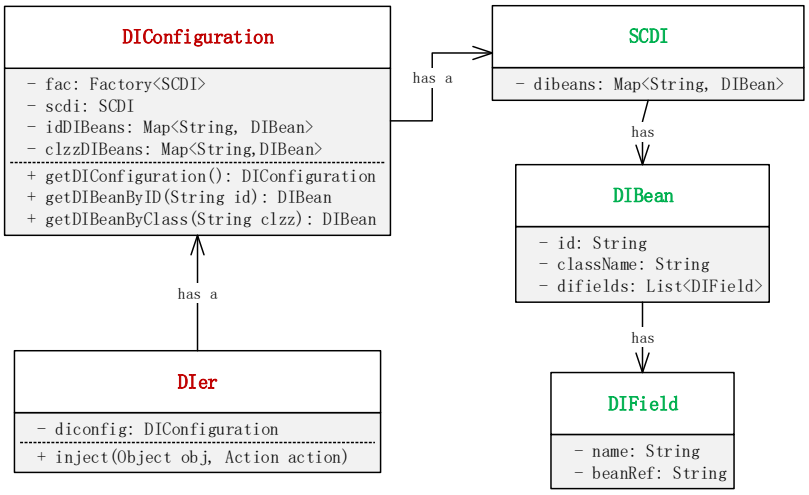
contoller.xml 对应的相关 Bean 类-UML 类图



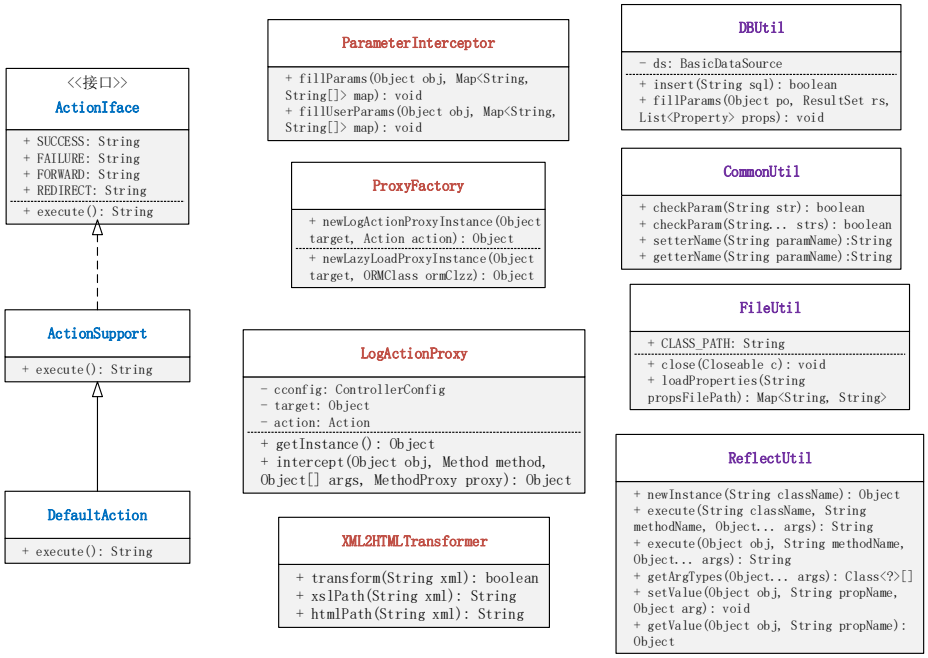
Factory-UML 类图



ORM 相关类-UML 类图

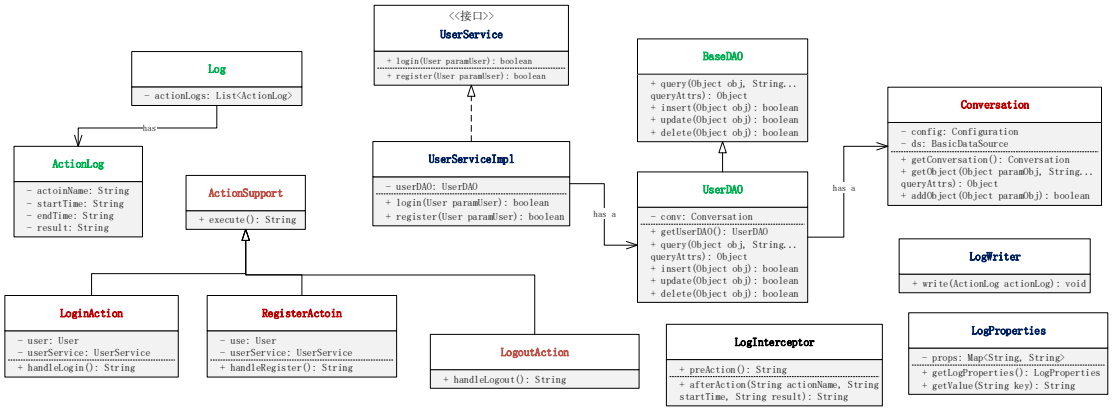


DI 相关类-UML 类图



其他相关类-UML 类图

3.2.2 UseSC UML 类图



3.3 doPost 处理流程

1）根据请求 req 获取 aciont；

2）根据 action 构造代理对象 proxy；

3）Dier 对 proxy 进行依赖注入；

4）ParameterInterceptor 对 proxy 中的 user（如果有）进行参数填充（userName、password）；

5）执行 proxy 中的 action 指定 method，返回结果 result；

6）hadleResult()方法对结果 result 进行分发；

代码片段如下：

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

3.4 di.xml 与解析

1）在 UseSC 工程的 config 目录下创建 di.xml 文件，写入如下内容：

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

<sc-di>
    <bean id="user" class="southday.j2eework.water.ustc.bean.User"></bean>
    <bean id="loginAction" class="southday.j2eework.water.ustc.action.LoginAction">
        <field name="user" bean-ref="user"></field>
    </bean>
    <bean id="registerAction" class="southday.j2eework.water.ustc.action.RegisterAction">
        <field name="user" bean-ref="user"></field>
    </bean>
    <bean id="defaultAction" class="southday.j2eework.sc.ustc.controller.action.DefaultAction"></bean>
    <bean id="logoutAction" class="southday.j2eework.water.ustc.action.LogoutAction"></bean>
</sc-di>
```

2）在 SimpleController 工程中创建包：southday.j2eework.sc.ustc.controller.di.bean，在该包中创建对应于 di.xml 中标签<sc-di>、<bean>、<field>的 Bean 类：SCDI、DIBean、DIField。

3）在包 southday.j2eework.sc.ustc.controller.di 中创建类 SCDIFactory（实现 Factory<SCDI>），使用 DOM4J 进行 XML 文件解析，解析代码如下：

```java
public class SCDIFactory implements Factory<SCDI> {
    String diXMLPath = null;
    private static final SAXReader saxReader = new SAXReader();

    public SCDIFactory(String diXMLPath) {
        this.diXMLPath = diXMLPath;
    }

    @SuppressWarnings("unchecked")
    @Override
    public SCDI create() throws Exception {
        Document doc = saxReader.read(diXMLPath);
        Element root = doc.getRootElement();
        SCDI scdi = new SCDI();
        scdi.setDibeans(toDIBeans(root.elements("bean")));
        return scdi;
    }

    @SuppressWarnings("unchecked")
    private Map<String, DIBean> toDIBeans(List<Element> elements) {
        Map<String, DIBean> dibeans = new HashMap<>();
        for (Element e : elements) {
            DIBean dibean = new DIBean();
            dibean.setId(e.attributeValue("id"));
            dibean.setClassName(e.attributeValue("class"));
            dibean.setDifields(toDIFields(e.elements("field")));
            dibeans.put(dibean.getId(), dibean);
        }
        return dibeans;
    }

    private List<DIField> toDIFields(List<Element> elements) {
        List<DIField> difields = new ArrayList<>();
        for (Element e : elements) {
            DIField difield = new DIField();
            difield.setName(e.attributeValue("name"));
            difield.setBeanRef(e.attributeValue("bean-ref"));
            difields.add(difield);
        }
        return difields;
    }
}
```

3.5 DIConfiguration

在包 southday.j2eework.sc.ustc.controller.di 中创建类 DIConfiguration，作为 di.xml 在内存中的映射，使用单例实现。此外，为了加快检索，该类中除了提供基于<bean>的<id>进行检索的 map，还提供了基于<bean>的<class>进行查询的 map，在创建实例时对 map 进行初始化。代码如下：

```java
public class DIConfiguration {
    private static final String DI_XML_PATH = getPathOfDIXML();
    private Factory<SCDI> fac = new SCDIFactory(DI_XML_PATH);
    private SCDI scdi;
    private Map<String, DIBean> idDIBeans;
    private Map<String, DIBean> clzzDIBeans;

    private DIConfiguration() {
        try {
            scdi = fac.create();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        idDIBeans = scdi.getDibeans();
        clzzDIBeans = new HashMap<>();
        for (Map.Entry<String, DIBean> e : idDIBeans.entrySet())
            clzzDIBeans.put(e.getValue().getClassName(), e.getValue());
    }

    private static class DIConfigurationHolder {
        private static DIConfiguration diconfig = new DIConfiguration();
    }
```

```
    public static DIConfiguration getDIConfiguration() {
        return DIConfigurationHolder.diconfig;
    }

    public DIBean getDIBeanByID(String id) {
        return idDIBeans.get(id);
    }

    public DIBean getDIBeanByClass(String clzz) {
        /* 当对象为动态代理生成的子类时，
         * clzz就可能为这类形式：southday.j2eework.water.ustc.action.LoginAction$$EnhancerByCGLIB$$2b939c0d
         * 为了能正确获取到DIBean，这里做个转换，把从$到后面的内容全部切掉，只保留父类类路径
         */
        int index = clzz.indexOf('$');
        return index < 0 ? clzzDIBeans.get(clzz) : clzzDIBeans.get(clzz.substring(0, index));
    }

    private static String getPathOfDIXML() {
        return BaseConfig.getBaseConfig().getFilePathInClassesDIR("di.xml");
    }
}
```

3.6 DIer 注入器

DIer 注入的步骤：

a）根据 action 的 className 获取对应的 dibean；如果 dibean 为空或者 dibean 中没有依赖的对象（List<DIField>.size()<=0），直接返回，否则进入下一步；

b）通过 dibean.getDifields()获取到依赖对象的集合，遍历该集合。diconfig 获取<bean-ref>对应的 DIBean ref，如果为空，则 continue（检查下一个<bean-ref>）；当 ref 不为空时，根据其属性 className 动态生成对象 obj，通过 ReflectUtil.setValue()方法将对象 obj 注入到目标对象 target 中；

代码如下：

```
public class DIer {
    private static final DIConfiguration diconfig = DIConfiguration.getDIConfiguration();

    public static void inject(Object target, Action action) {
        DIBean dibean = diconfig.getDIBeanByClass(action.getClassName());
        if (dibean == null || dibean.getDifields().size() <= 0)
            return;
        try {
            for (DIField difield : dibean.getDifields()) {
                DIBean ref = diconfig.getDIBeanByID(difield.getBeanRef());
                if (ref == null)
                    continue;
                Object obj = ReflectUtil.newInstance(ref.getClassName());
                ReflectUtil.setValue(target, difield.getName(), obj);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

其中所用到的 ReflectUtil 中的方法代码片段如下：

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

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

    public static Object setValue(Object obj, String propName, Object arg) throws Exception {
        return execute(obj, CommonUtil.setterName(propName), arg);
    }

    public static Object getValue(Object obj, String propName) throws Exception {
        return execute(obj, CommonUtil.getterName(propName));
    }
}
```

3.7 ParameterInterceptor 参数填充

在 DIer 对 proxy 进行注入后，就要对 proxy 中的 user 对象进行参数填充，而该功能是由 ParameterInterceptor 的 fillUserParams()方法来完成的。

参数填充步骤：

a）先判断 proxy 是否需要填充参数，即 proxy 是否依赖 User 对象；如果不需要填充，则直接 return；

b）若需要填充，则先通过 ReflectUtil 的 getValue()方法获取 proxy 中的 user 对象；

c）调用 fillParams(user, map)进一步给 user 对象填充参数；填充过程就是根据 req 请求参数名称（如：userName、password）来构造 setter 方法，然后通过反射注入参数值。这个过程都被封装在了 ReflectUtil 的 setValue()方法中了。

ParameterInterceptor.java 代码如下：

```
public class ParameterInterceptor {
    private static final DIConfiguration diconfig = DIConfiguration.getDIConfiguration();

    public static void fillParams(Object obj, Map<String, String[]> map) {
        try {
            for (Map.Entry<String, String[]> e : map.entrySet()) {
                String paramName = e.getKey();
                String value = e.getValue()[0];
                ReflectUtil.setValue(obj, paramName, value);
            }
        } catch (NoSuchMethodException e) {
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void fillUserParams(Object obj, Map<String, String[]> map) {
        if (!isNeedFillUserParams(obj))
            return;
        Object user = null;
        try {
            user = ReflectUtil.getValue(obj, "user");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        fillParams(user, map);
    }

    private static boolean isNeedFillUserParams(Object obj) {
        DIBean dibean = diconfig.getDIBeanByClass(obj.getClass().getName());
        if (dibean == null || dibean.getDifields().size() <= 0)
            return false;
        for (DIField f : dibean.getDifields())
            if ("user".equals(f.getName()))
                return true;
        return false;
    }
}
```

3.8 修改 LoginAction、RegisterAction

之前写的 LoginAction 和 RegisterAction 都是 ActionBean，里面自带了 userName 和 password 属性，现在要将其改为依赖 User 对象。修改如下：

注意：一定要提供 getUser()和 setUser()方法，否则反射调用时会出异常！

LoginAction.java：

```
public class LoginAction extends ActionSupport {
    private User user;
    private UserService userService = UserServiceImpl.getUserService();

    public String handleLogin() throws Exception {
        return userService.login(user) ? SUCCESS : FAILURE;
    }

    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

RegisterAction.java：

```
public class RegisterAction extends ActionSupport {
    private User user;
    private UserService userService = UserServiceImpl.getUserService();

    public String handleRegister() throws Exception {
        return userService.register(user) ? SUCCESS : FAILURE;
    }

    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

3.9 修改 UserService、UserServiceImpl

之前写的 UserService 和 UserServiceImpl 的 login 和 register 方法接收的参数是 String 的 userName 和 password，现在改为接收参数 User 对象。修改如下：

UserService.java：

```
public interface UserService {
    boolean login(User paramUser) throws Exception;
    boolean register(User paramUser) throws Exception;
}
```

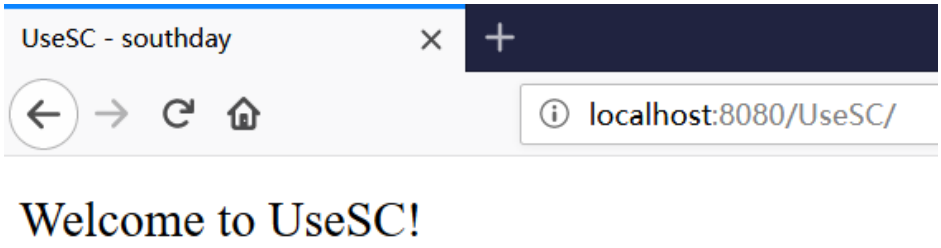
UserServiceImpl.java：

```java
public class UserServiceImpl implements UserService {
    private UserDAO userDAO = UserDAO.getUserDAO();

    private UserServiceImpl() {}

    private static class UserServiceImplHolder {
        private static UserService userService = new UserServiceImpl();
    }

    public static UserService getUserService() {
        return UserServiceImplHolder.userService;
    }

    @Override
    public boolean login(User paramUser) throws Exception {
        if (!checkUserParam(paramUser))
            return false;
        User retUser = (User)userDAO.query(paramUser, "userName");
        return paramUser.getPassword().equals(retUser.getPassword());
    }

    @Override
    public boolean register(User paramUser) throws Exception {
        if (!checkUserParam(paramUser))
            return false;
        return userDAO.insert(paramUser);
    }

    private boolean checkUserParam(User user) {
        return user != null && CommonUtil.checkParam(user.getUserName(), user.getPassword());
    }
}
```

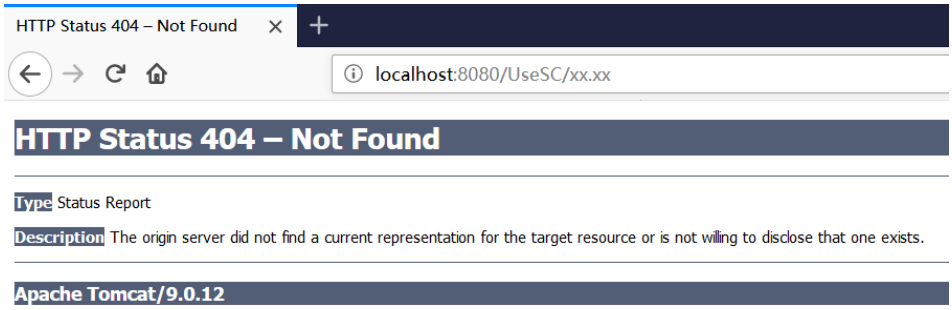
3.10 测试结果

3.10.1 MySQL 部分

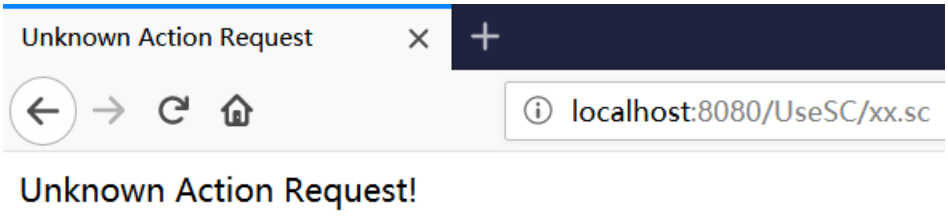
1）访问主页：http://localhost:8080/UseSC/，成功访问主页；



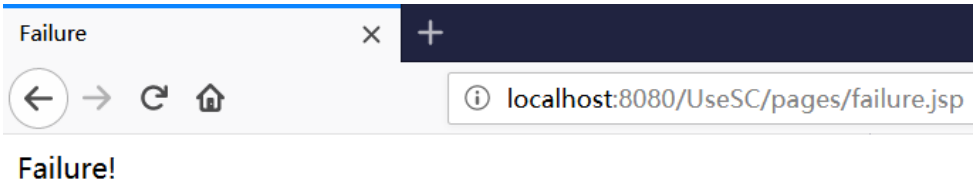
2）非法 URL：http://localhost:8080/UseSC/xx.xx，404-Not found；



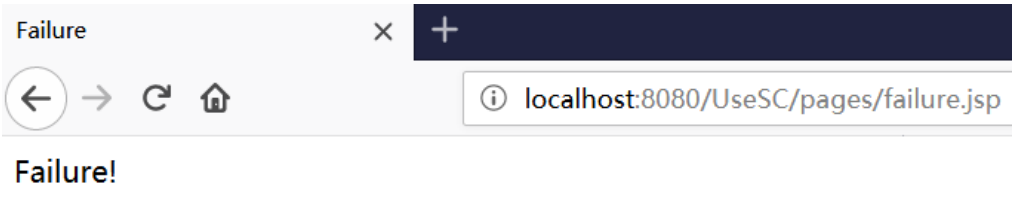
3）非法请求：http://localhost:8080/UseSC/xx.sc，无法识别的 action；



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

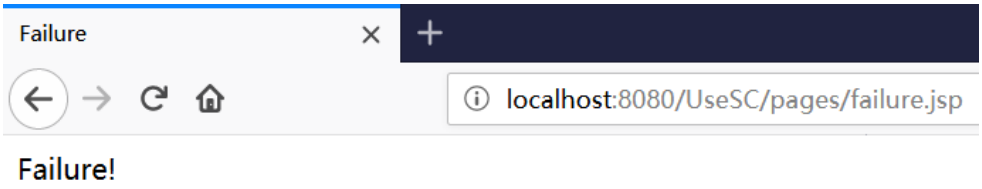


5）测试登陆：http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx，失败，用户账号 lcx 不存在；

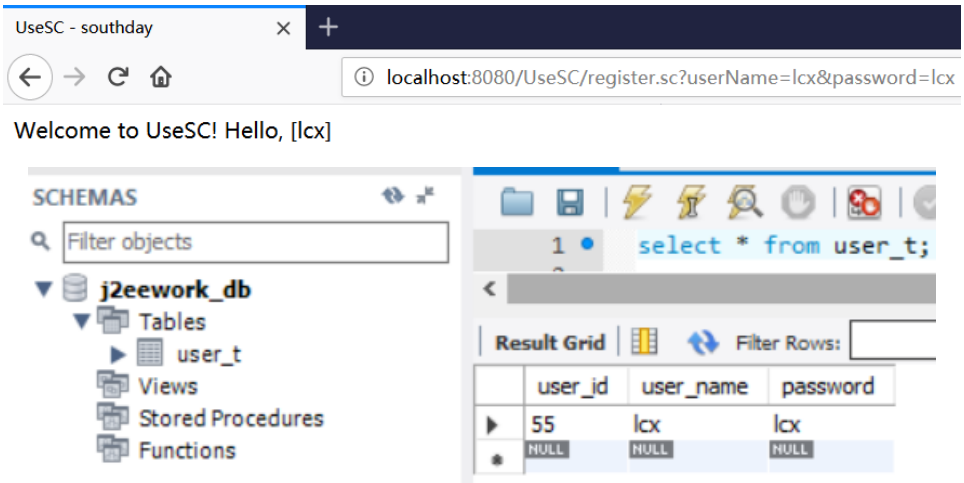



```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE user_name = 'lcx'
```

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；

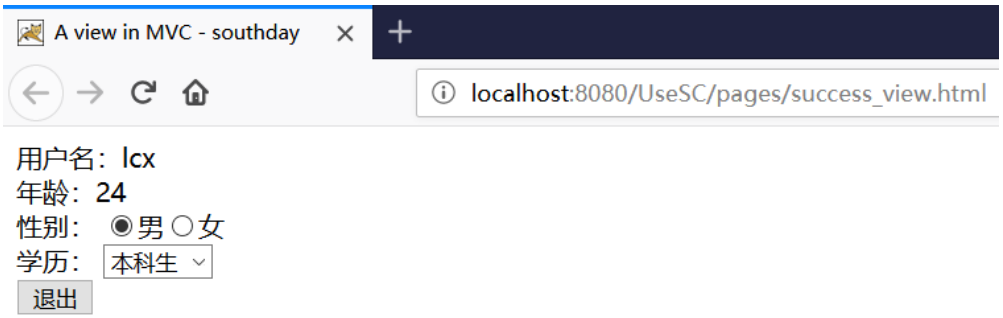


7）测试注册：http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx，注册成功，跳转到 welcome 页面；



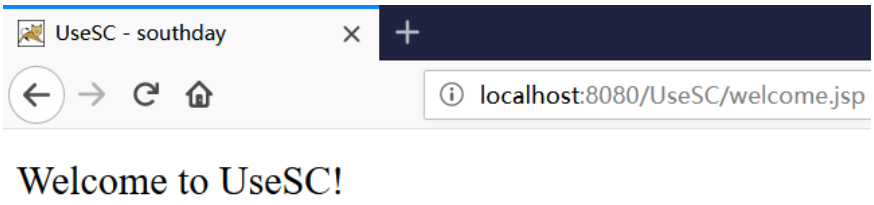
```
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('lcx', 'lcx')
```

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；

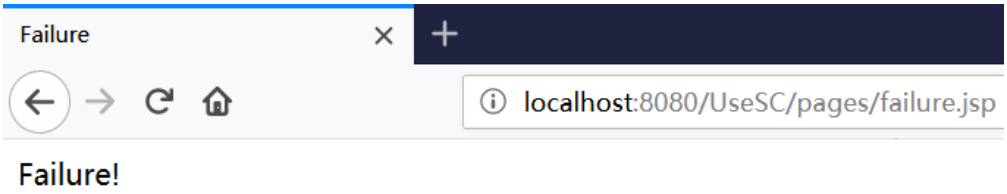



```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE
user_name = 'lcx'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id
= 55
```

9）在 success_view.html 页面点击“退出”，返回到首页（welcome.jsp）；



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；

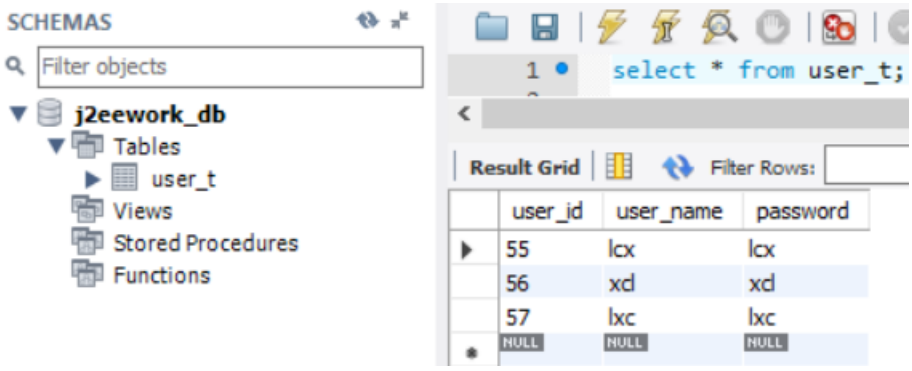



```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE
user_name = 'lcx'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id
= 55
```

11）使用<userName, password>=<xcl, xcl>, <lxc, lxc>注册账号：

- http://localhost:8080/UseSC/register.sc?userName=xcl&password=xcl
- http://localhost:8080/UseSC/register.sc?userName=lxc&password=lxc

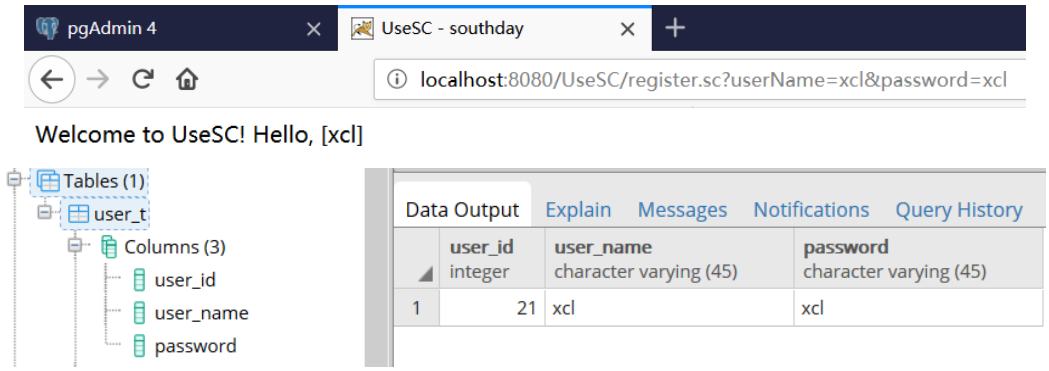
注册成功，数据库 user_t 表显示如下：




```
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('xcl', 'xcl')
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('lxc', 'lxc')
```

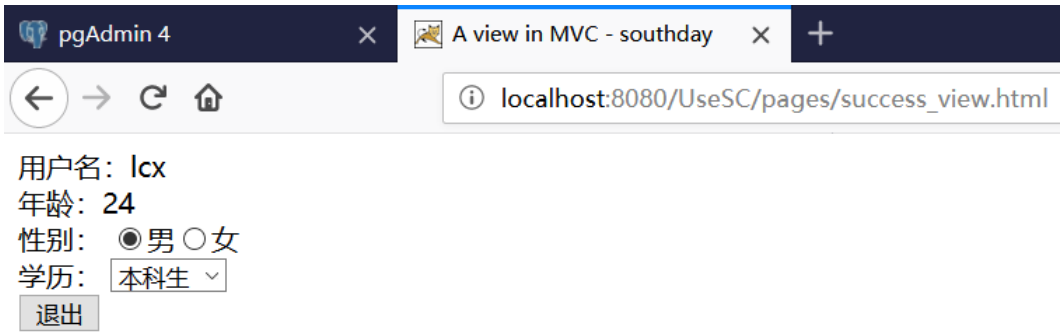
3.10.2 PostgreSQL 部分

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




```
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('xcl', 'xcl')
```

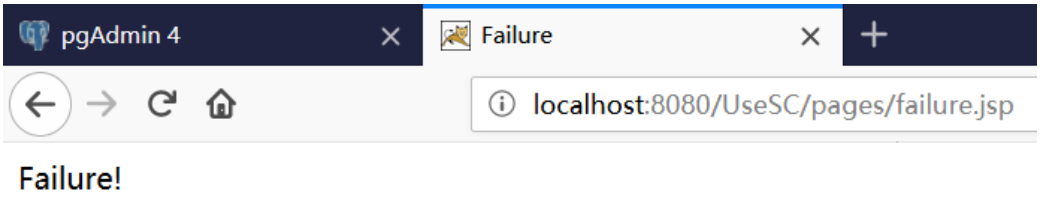
2）注册成功后，用 xcl 账号登陆：http://localhost:8080/UseSC/login.sc?userName=xcl&password=xcl，登陆成功，跳转到 success_view.html 页面；




```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE
user_name = 'xcl'
```

```
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id
= 21
```

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

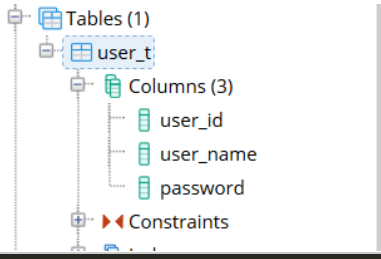



```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE
user_name = 'xcl'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id
= 21
```

4）使用<userName, password>=<lcx, lcx>, <cxl, cxl>注册账号：

- http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx
- http://localhost:8080/UseSC/register.sc?userName=cxl&password=cxl

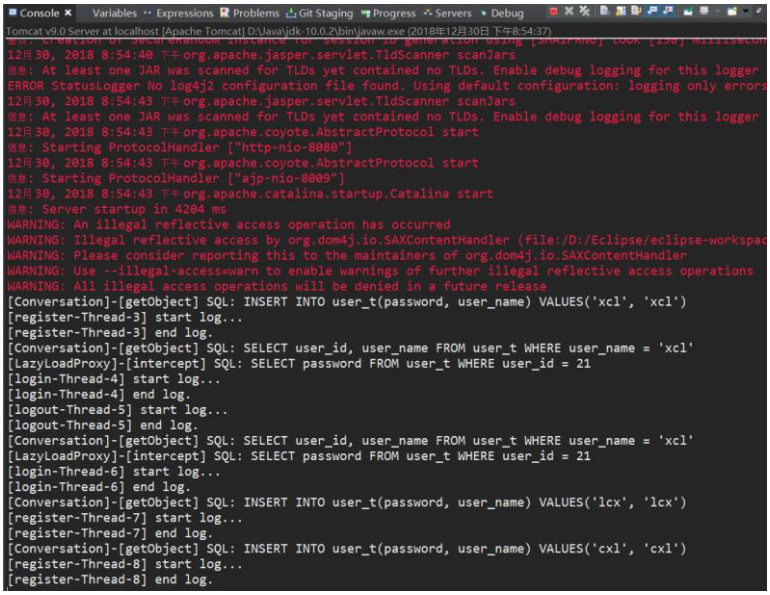
注册成功，数据库 user_t 表显示如下：



Data Output | Explain | Messages | Notifications | Query History

	user_id integer	user_name character varying (45)	password character varying (45)
1	21	xcl	xcl
2	22	lcx	lcx
3	23	cxl	cxl

```
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('lcx', 'lcx')
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name)
VALUES('cxl', 'cxl')
```



4. 结论

4.1 总结

通过本次实验，我认识了何为依赖注入，并且亲自实现了简单的依赖注入功能。说到底还是对反射的利用。纵观 E2~E7，Java 反射无处不在，难以言表，只能双击 666。

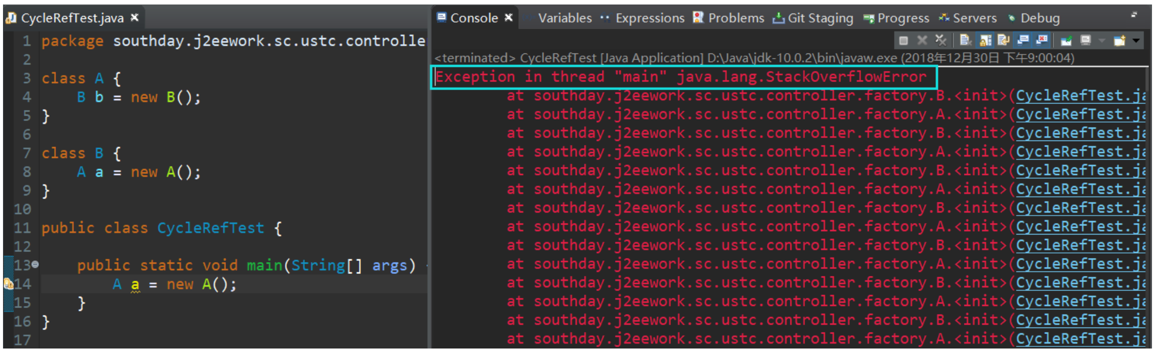
轻量级 J2EE 这门课的作业对我帮助很大，一方面让我了解了框架的底层原理；另一方面，在实现简易框架的过程中，对程序的设计、决策、利弊权衡、重构等，都是对我程序设计能力的锻炼和考验，我也因此而得到成长。

我一直觉得学某个东西（特别是我们 IT 行业，新技术层出不穷），应该要以一个明确的目标来作为驱动，这样学起来效率更高。J2EE 的作业就是典型的目标驱动学习案例。学而不用，过一段时间就忘了，大脑会选择性淘汰那些暂时不必要的东西，而把重要的东西保留下来。所以，不单要学习，还要致用，更要温故，这样才能变得更加强大。

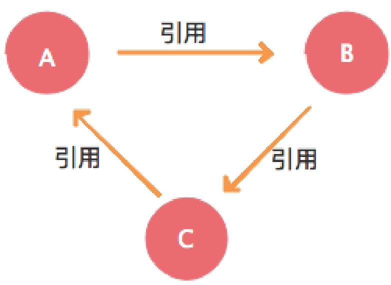
4.2 问题及看法

4.2.1 依赖注入循环引用问题

循环引用是很容易想到的问题，我的设计中没有处理该类问题。下面是我实现循环引用的 demo，运行后瞬间：StackOverflow。



正常情况下程序都不这么设计的，但是写在 di.xml 配置文件中，难免出现配置错误的情况，这时就很容易出现循环引用的问题。Spring 中对循环引用做了相应的处理。



（图源：https://img-blog.csdn.net/20180817100220505?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2NoZW4yNTI2MjY0/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70）

当 Spring 容器在创建 A 时，会发现其引用了 B，从而会先去创建 B。同样的，创建 B 时，会先去创建 C，而创建 C 时，又先去创建 A。最后 A、B、C 之间互相等待，谁都没法创建成功。

要想打破这个环，那么这个环中至少需要有一个 bean 可以在自身的依赖还没有得到满足前，就能够被创建出来（最起码要被实例化出来，可以先不注入其需要的依赖）。这种 bean 只能是通过属性注入依赖的类，因为它们可以先使用默认构造器创建出实例，然后再通过 setter 方法注入依赖。而通过构造器注入依赖的类，在它的依赖没有被满足前，无法被实例化。更详细请看文章：“理解 Spring 循环引用（循环依赖）”

5. 参考文献

[1] 架构师之路】依赖注入原理---IoC 框架：https://www.cnblogs.com/jhli/p/6019895.html
[2] 理解 Spring 循环引用（循环依赖）：
https://blog.csdn.net/chen2526264/article/details/80673598
[3] 从循环引用谈依赖倒置原则：https://www.cnblogs.com/yangecnu/p/3689402.html
[4] spring 循环依赖注入：https://www.jianshu.com/p/49e88dae7107
[5] github 上一篇比较贴切的依赖注入举例：
https://github.com/android-cn/blog/tree/master/java/dependency-injection