

轻量级 J2EE 框架应用

E 6 A Simple Controller with DAO pattern & O/R mapping

学号：SA18225170

姓名：李朝喜

报告撰写时间：2018/12/29

文档目录

1. 主题概述

- 1.1 ORM
- 1.2 延迟加载

2. 假设

- 2.1 知识背景
- 2.2 环境背景

3. 实现或证明

- 3.1 项目包结构及简要描述
- 3.2 UML 类图
- 3.3 or_mapping.xml 与解析
- 3.4 SQLFactory
- 3.5 DBUtil
- 3.6 LazyLoadProxy
- 3.7 Conversation
- 3.8 修改 BaseDAO、UserDAO、UserServiceImpl 逻辑
- 3.9 测试结果

4. 结论

- 4.1 总结
- 4.2 问题及看法

5. 参考文献

1. 主题概述

1.1 ORM

对象关系映射（Object Relational Mapping，简称 ORM）模式是为了解决面向对象与关系数据库存在的互不匹配现象的技术。简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。

ORM 技术特点：

1) 提高了开发效率。由于 ORM 可以自动对 Entity 对象与数据库中的 Table 进行字段与属性的映射，所以我们实际可能已经不需要一个专用的、庞大的数据访问层。

2) ORM 提供了对数据库的映射，不用 sql 直接编码，能够像操作对象一样从数据库获取数据。

ORM 技术的缺点：ORM 会牺牲程序的执行效率和造成开发思维模式固定。从系统结构上来看,采用 ORM 的系统一般都是多层系统，系统的层次多了，效率就会降低。ORM 是一种完全的面向对象的做法，而面向对象的做法也会对性能产生一定的影响。

1.2 延迟加载

延迟加载(lazy load)是(也称为懒加载)Hibernate3 关联关系对象默认的加载方式，延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。

延迟加载，可以简单理解为，只有在使用的时候，才会发出 sql 语句进行查询。延迟加载的有效期是在 session 打开的情况下，当 session 关闭后，会报异常。当调用 load 方法加载对象时，返回代理对象，等到真正用到对象的内容时才发出 sql 语句。

2. 假设

需要注意，实验 E6 是在 E5 基础上进行的，所以在做 E6 前请先完成 E5，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。

- 具备基本的 SQL 编程能力；
- 理解 Java 基础知识，并且具备基本的 Java 编程能力，如：理解 Java 中对象封装、继承等概念，以及具备 JavaSE 基本开发能力；
- 了解代理模式，能够使用 Java 实现静态和动态代理方式；
- 了解 Java 反射机制，能够利用 Java 反射来调用某个类中的方法；
- 了解 JDBC 相关概念，能够使用 JDBC 与数据库进行交互（CURD）；
- 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>
- 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	
- SimpleController.java	
- action	
- Actionface.java	
- ActionSupport.java	
- DefaultAction.java	
- config	
- BaseConfig.java	
- ControllerConfig.java	
- bean	
- Controller.java	
- Action.java	
- Result.java	
- Interceptor.java	
- InterceptorRef.java	
- SCConfiguration.java	
- factory	
- ActionFactory.java	
- ResultFactory.java	
- ControllerFactory.java	
- SCConfigurationFactory.java	
- DOM4JSCConfigurationFactory.java	
- dao	
- BaseDAO.java	
- DBConfiguration.java	
- factory	
- Factory.java	
- interceptor	
- ParameterInterceptor.java	
- orm	
- Configuration.java	
- Conversation.java	
- ORMMappingFactory.java	
- SQLFactory.java	
- bean	
- ID.java	
- ORMMapping.java	
- ORMClass.java	
- Property.java	
- proxy	
- ProxyFactory.java	
- cglib	
- LogActionProxy.java	
- LazyLoadProxy.java	
- transformer	
- XML2HTMLTransformer.java	
- util	
- CommonUtil.java	
- FileUtil.java	
- ReflectUtil.java	
- DBUtil.java	

包根目录
Servlet, 用于处理*.sc的请求
action包, 里面包含了Action接口定义及默认实现类等
Action接口
Action的默认实现类, 框架使用者可以继承该类
默认Action, 当未找到匹配的action时, 就返回DefaultAction
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
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
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包, 包含与UseSC中controller.xml文件里定义内容相对应的JavaBean对象类
与controller.xml中定义的<controller>标签相对应
与controller.xml中定义的<action>标签相对应
与controller.xml中定义的<result>标签相对应
与controller.xml中定义的<interceptor>标签相对应
与controller.xml中定义的<interceptor-ref>标签相对应
与controller.xml中定义的<sc-configuration>标签相对应
factory包, 工厂模式, 里面包含了用于创建对象的工厂类
用于生成Action对象的工厂
用于生成Result对象的工厂
用于生成Controller对象的工厂
用于生成SCConfiguration对象的工厂
基于DOM4J的XML解析来创建SCConfiguration对象的具体实现类
dao包, 包含各类DAO
基本DAO, 作为其他DAO的父类存在
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
factory包, 目前包含工厂的抽象定义(接口Factory)
抽象工厂, 提供接口: T create() throws Exception;
interceptor包, 用于存放拦截器的相关类、接口等
参数拦截器, 目前仅用于给Action代理对象(动态)填充参数
orm包, 包含了处理O/R Mapping的相关类
or_mapping.xml在内存中的映射总类(单例)
实现与数据库的交互, 将交互操作封装为对对象的操作
解析or_mapping.xml的工厂类
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包, 包含与UseSC中or_mapping.xml文件里定义内容相对应的JavaBean对象类
与or_mapping.xml中定义的<id>标签相对应
与or_mapping.xml中定义的<OR-Mapping>标签相对应
与or_mapping.xml中定义的<class>标签相对应
与or_mapping.xml中定义的<property>标签相对应
proxy包, 包含与代理机制相关的类
代理工厂, 用于生成代理类对象
cglib包, 关于Action的动态代理, 使用cglib技术来实现
针对日志记录的Action代理拦截器
针对PO对象属性实现延迟加载的代理类
transformer包, 包含各类转换器
将xml转为html的转换器, 目前使用xslt技术实现
util包, 包含项目工具类
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
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
包含与反射处理相关的常用方法
包含与数据库操作相关的常用方法, 目前提供insert()、fillParams()方法

UseSC

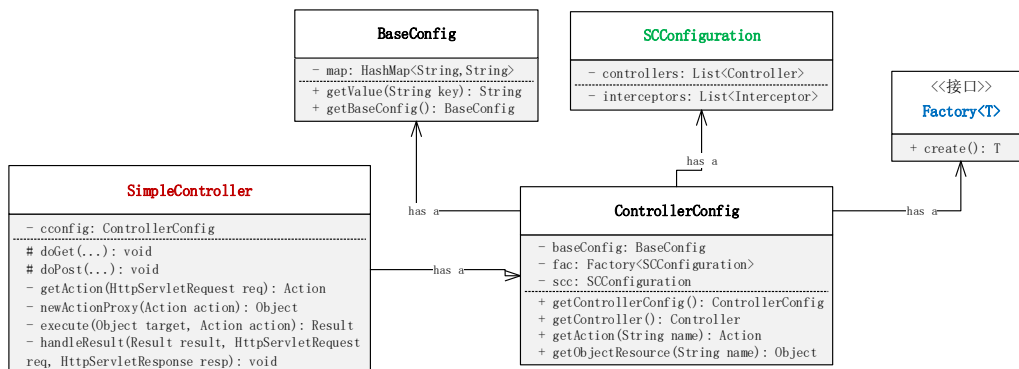
```

|- src/main/java
  |- southday.j2eework.water.ustc
    |- action
      |- LoginAction.java
      |- RegisterAction.java
      |- LogoutAction.java
    |- bean
      |- User.java
    |- db
      |- SimpleUserDB.java
    |- interceptor
      |- LogInterceptor.java
    |- log
      |- LogProperties.java
      |- LogWriter.java
      |- bean
        |- Log.java
        |- ActionLog.java
    |- service
      |- UserService.java
      |- impl
        |- UserServiceImpl.java
    |- dao
      |- UserDao.java
  |- src/main/resources
    |- config
      |- controller.xml
      |- files-locations.properties
      |- log.properties
      |- jdbc-mysql.properties
      |- jdbc-postgresql.properties
      |- or_mapping.xml
    |- src/main/webapp
      |- welcome.jsp
      |- WEB-INF
        |- web.xml
      |- pages
        |- failure.jsp
        |- no-req-resource.jsp
        |- unknown-action.jsp
        |- welcome.jsp
        |- success_view.xml
        |- success_view.xsl
  
```

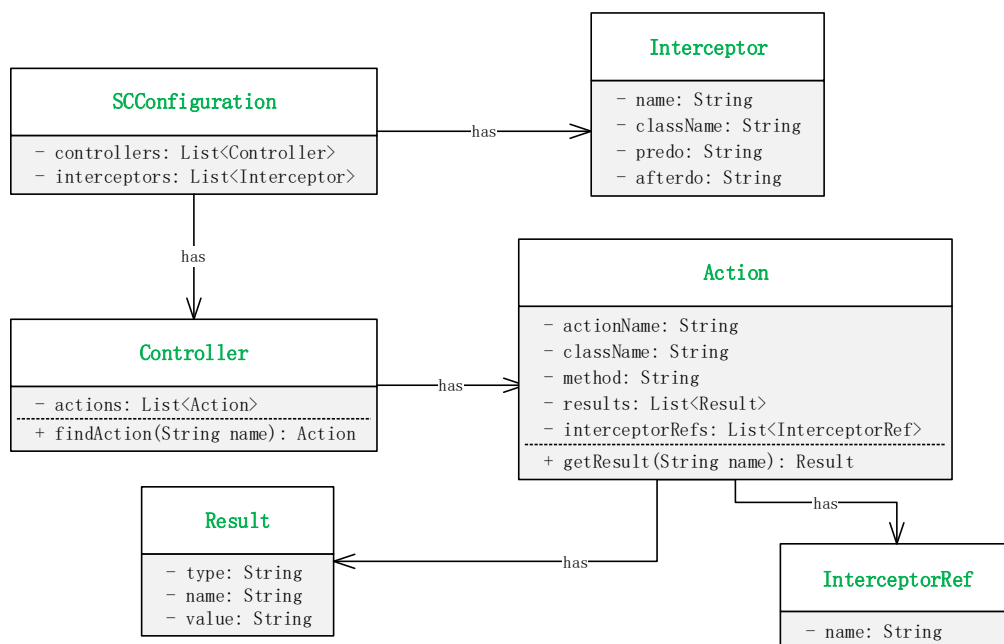
包根目录
 # action包, 包含了具体业务对应的action类
 # 登陆业务对应的Action
 # 注册业务对应的Action
 # 退出登陆对应的Action
 # bean包
 # UserBean
 # db包, 服务端管理数据库资源的包
 # 自制的简易User数据库, 内部使用ConcurrentHashMap实现, 单例模式
 # interceptor包, 包含拦截器相关类
 # 日志记录拦截器
 # log包, 包含与日志记录操作相关的类
 # 用于加载和保存log.properties配置文件中的信息
 # 用于写日志
 # bean包, 包含日志文件xml中相对应的bean对象类
 # 与log.xml中定义的<log>标签相对应
 # 与log.xml中定义的<action>标签相对应
 # service包, 包含模型层(业务逻辑处理)的相关类
 # 接口, 定义了与User有关的相关业务逻辑的抽象方法
 # impl包, 包含了针对模型层接口的具体实现类
 # UserService接口的具体实现类
 # dao包, 包含各类DAO
 # 实现BaseDAO的具体类, 包含query, insert, update, delete等方法的实现
 # Web项目的资源文件包
 # config包, 包含了开发者自定义的配置文件
 # 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, 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, 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, 里面包含了DBC用于创建、管理数据库连接的配置属性(针对MySQL)
 # 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, O/R Mapping配置文件
 # Web项目部署包
 # 项目首页.jsp
 # Web配置包
 # Web项目配置文件
 # pages包, 存放返回结果页面
 # “请求失败”页面
 # “为找到请求资源”页面
 # “未知Action”页面
 # “欢迎”页面(用户已登陆)
 # “登陆成功”页面的XML配置
 # “登陆成功”页面的XSL配置

3.2 UML 类图

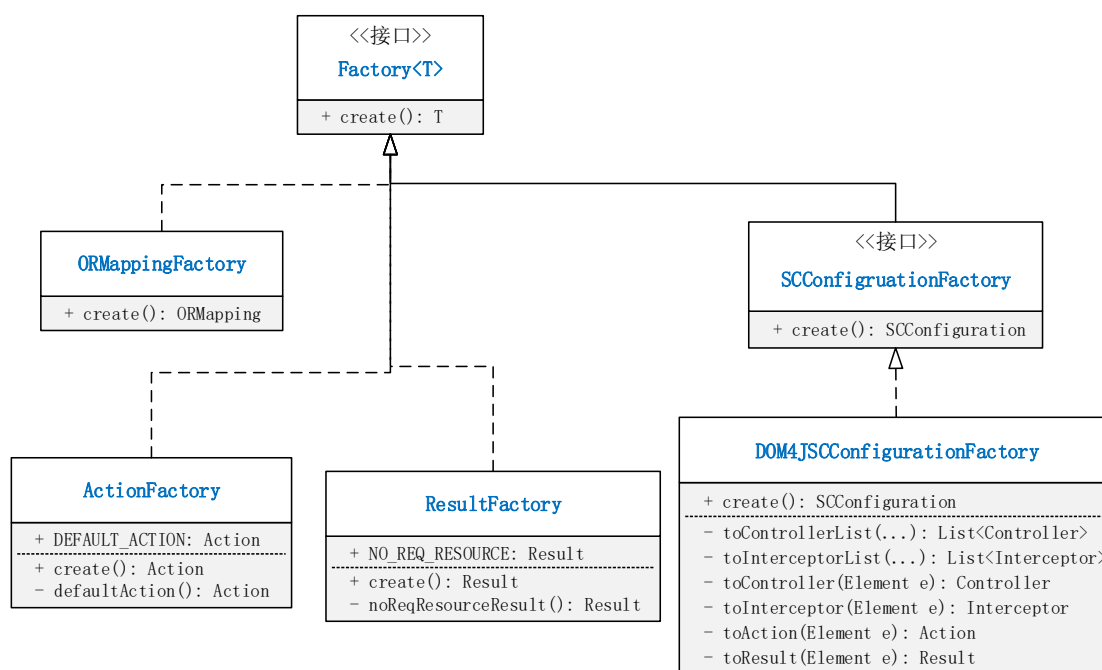
3.2.1 SimpleController UML 类图



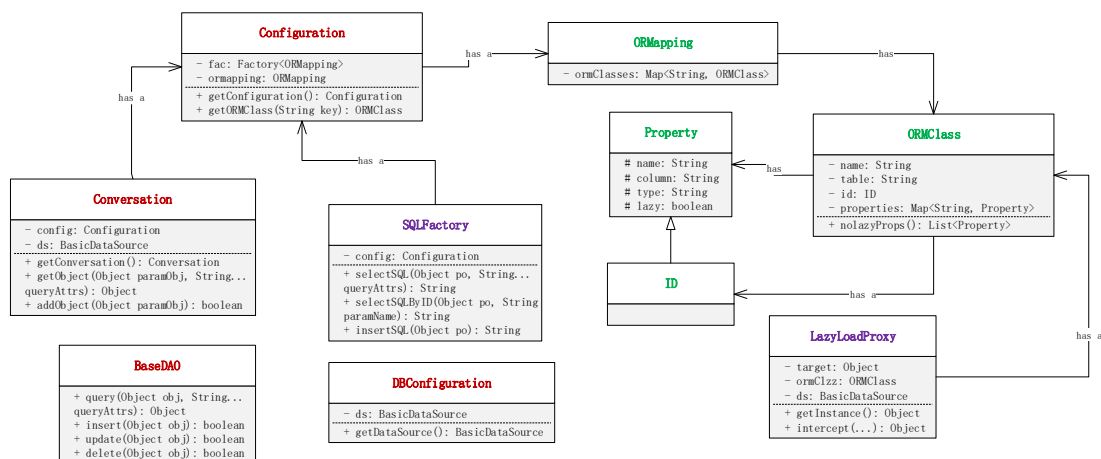
Servlet 相关类-UML 类图



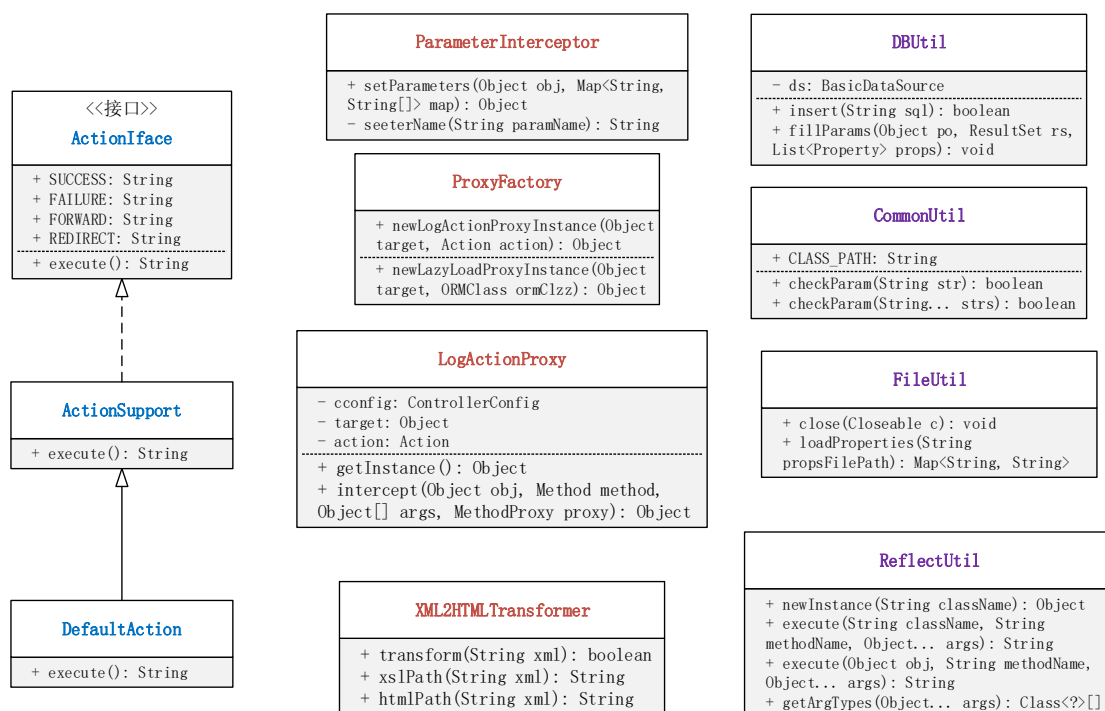
contoller.xml 对应的相关 Bean 类-UML 类图



Factory-UML 类图

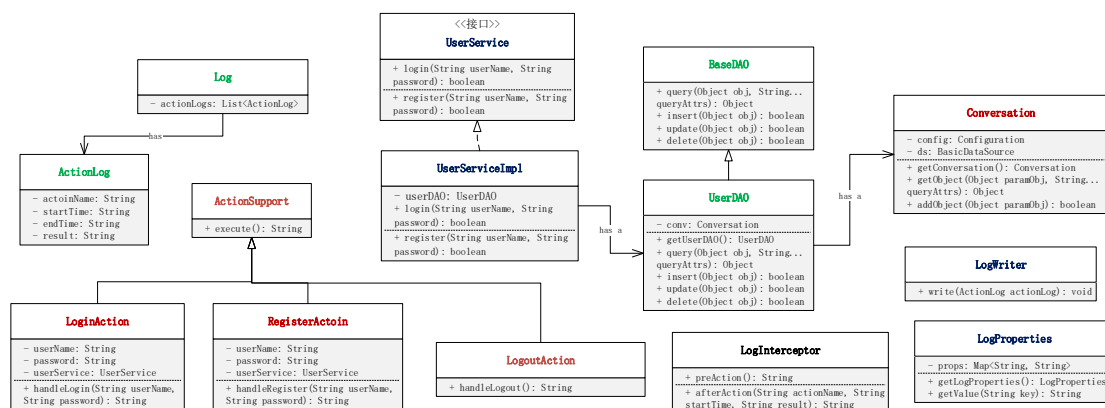


ORM 相关类-UML 类图



其他相关类-UML 类图

3.2.2 UseSC UML 类图



3.3 or_mapping.xml 与解析

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

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

<OR-Mapping>
  <class>
    <name>southday.j2eework.water.ustc.bean.User</name>
    <table>user_t</table>
    <id>
      <name>userId</name>
      <column>user_id</column>
      <type>java.lang.Integer</type>
    </id>
    <property>
      <name>userName</name>
      <column>user_name</column>
      <type>java.lang.String</type>
      <lazy>false</lazy>
    </property>
    <property>
      <name>password</name>
      <column>password</column>
      <type>java.lang.String</type>
      <lazy>true</lazy>
    </property>
  </class>
</OR-Mapping>
```

2) 在 SimpleController 工程中创建包：southday.j2eework.sc.ustc.controller.orm.bean，在该包中创建对应于 or_mapping.xml 中标签<OR-Mapping>、<class>、<id>、<property>的 Bean 类：ORMapping、ORMClass、ID、Property，其中 ID 继承 Property。

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

```
public class ORMMappingFactory implements Factory<ORMapping> {
    String ormappingXMLPath = null;
    private static final SAXReader saxReader = new SAXReader();

    public ORMMappingFactory(String ormappingXMLPath) {
        this.ormappingXMLPath = ormappingXMLPath;
    }

    @SuppressWarnings("unchecked")
    @Override
    public ORMMapping create() throws Exception {
        Document doc = saxReader.read(ormappingXMLPath);
        Element root = doc.getRootElement();
        ORMMapping ormapping = new ORMMapping();
        ormapping.setOrmClasses(toORMClasses(root.elements("class")));
        return ormapping;
    }

    private Map<String, ORMClass> toORMClasses(List<Element> elements) throws Exception {
        Map<String, ORMClass> ormClasses = new HashMap<>();
        for (Element e : elements) {
            ORMClass clzz = toORMClass(e);
            ormClasses.put(clzz.getName(), clzz);
        }
        return ormClasses;
    }

    @SuppressWarnings("unchecked")
    private ORMClass toORMClass(Element element) throws Exception {
        ORMClass clzz = new ORMClass();
        clzz.setId(toID(element.element("id")));
        clzz.setName(element.elementTextTrim("name"));
        clzz.setProperties(toProperties(element.elements("property")));
        clzz.setTable(element.elementTextTrim("table"));
        return clzz;
    }
}
```

```

private ID toID(Element element) throws Exception {
    ID id = new ID();
    id.setName(element.elementTextTrim("name"));
    id.setColumn(element.elementTextTrim("column"));
    id.setType(element.elementTextTrim("type"));
    return id;
}

private Map<String, Property> toProperties(List<Element> elements) throws Exception {
    Map<String, Property> properties = new HashMap<>();
    for (Element e : elements) {
        Property prop = toProperty(e);
        properties.put(prop.getName(), prop);
    }
    return properties;
}

private Property toProperty(Element element) throws Exception {
    Property prop = new Property();
    prop.setName(element.elementTextTrim("name"));
    prop.setColumn(element.elementTextTrim("column"));
    prop.setLazy("true".equals(element.elementTextTrim("lazy")));
    prop.setType(element.elementTextTrim("type"));
    return prop;
}
}

```

4) 在包 southday.j2eework.sc.ustc.controller.orm 中创建类 Configuration，作为 or_mapping.xml 在内存中的映射，使用单例实现。

```

public class Configuration {
    private static final String ORMAPPING_XML_PATH = getPathOfORMappingXML();
    private Factory<ORMapping> fac = new ORMMappingFactory(ORMAPPING_XML_PATH);
    private ORMMapping ormapping;
    private Map<String, ORMClass> classes;

    private Configuration() {
        try {
            ormapping = fac.create();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        classes = ormapping.getOrmClasses();
    }

    private static class ConfigurationHolder {
        private static Configuration config = new Configuration();
    }

    public static Configuration getConfiguration() {
        return ConfigurationHolder.config;
    }

    public ORMClass getORMClass(String key) {
        /* 当对象为动态代理生成的子类时，
        * key就可能为这类形式：southday.j2eework.water.ustc.bean.User$$EnhancerByCGLIB$$4110e76b
        * 为了能正确获取到ORMClass，这里做个转换，把从$到后面的内容全部切掉，只保留父类类路径
        */
        int index = key.indexOf('$');
        return index < 0 ? classes.get(key) : classes.get(key.substring(0, index));
    }

    private static String getPathOfORMappingXML() {
        return BaseConfig.getBaseConfig().getFilePathInClassesDIR("or_mapping.xml");
    }
}

```

3.4 SQLFactory

在包 southday.j2eework.sc.ustc.controller.orm 中创建类 SQLFactory，用于动态生成相关 SQL 语句。目前支持 SELECT 和 INSERT 语句的生成。

- 1) 对于 selectSQL(Object po, String... queryAttrs)，构造思想如下：
 - a) 获取非懒加载属性集：nolazyCol，用其作为要 SELECT 的属性；
 - b) 使用 queryAttrs 中的属性来构造 WHERE 语句的查询条件；

c) 通过反射来取到 queryAttrs 中属性的值，构造 WHERE 条件；
代码如下：

```
public static String selectSQL(Object po, String... queryAttrs) throws Exception {
    ORMClass ormClzz = config.getORMClass(po.getClass().getName());
    List<String> nolaazyCol = nolaazyColumns(ormClzz);
    StringBuilder sql = new StringBuilder();
    sql.append("SELECT ");
    for (String col : nolaazyCol)
        sql.append(col + ", ");
    sql.deleteCharAt(sql.length()-2); // 去掉最后多余的 ','
    // SELECT attr1, attr2
    sql.append("FROM " + ormClzz.getTable() + " WHERE ");
    // SELECT attr1, attr2 FROM table WHERE
    Map<String, Property> properties = ormClzz.getProperties();
    for (String attr : queryAttrs) {
        String col = properties.get(attr).getColumn();
        String getter = CommonUtil.getterName(attr);
        Object value = ReflectUtil.execute(po, getter);
        if (value.getClass() == String.class)
            sql.append(col + " = '" + value.toString() + "' AND ");
        else
            sql.append(col + " = " + value.toString() + " AND ");
    }
    // 如果查询属性为空，则默认使用id查询
    if (queryAttrs.length <= 0) {
        String col = ormClzz.getId().getColumn(); // id基本都为Integer，下面不用判断是否为String类型
        String getter = CommonUtil.getterName(ormClzz.getId().getName());
        Object value = ReflectUtil.execute(po, getter);
        sql.append(col + " = " + value.toString() + " AND ");
    }
    // SELECT attr1, attr2 FROM table WHERE a = 1 AND b = 'x' AND
    sql.delete(sql.length()-5, sql.length()); // 去掉最后多余的' AND '
    // SELECT attr1, attr2 FROM table WHERE a = 1 AND b = 'x'
    return sql.toString();
}

private static List<String> nolaazyColumns(ORMClass ormClzz) {
    List<String> nolaazy = new ArrayList<>();
    for (Property prop : ormClzz.nolaazyProps())
        nolaazy.add(prop.getColumn());
    return nolaazy;
}
```

- 2) 对于 selectSQLByID(Object po, String propName)，构造思想如下：
 - a) 获取 propName 对应的 column，作为 SELECT 的属性；
 - b) 通过 po 对应 ORMClass 类获取 ID，作为 WHERE 语句的查询条件；
 - c) 通过反射来获取到 id 的值，构造 WHERE 条件；

代码如下：

```
public static String selectSQLByID(Object po, String propName) throws Exception {
    ORMClass ormClzz = config.getORMClass(po.getClass().getName());
    String col = ormClzz.getProperties().get(propName).getColumn();
    StringBuilder sql = new StringBuilder();
    sql.append("SELECT ").append(col).append(" FROM ").append(ormClzz.getTable());
    sql.append(" WHERE ").append(ormClzz.getId().getColumn()).append(" = ");
    String methodName = CommonUtil.getterName(ormClzz.getId().getName());
    Object value = ReflectUtil.execute(po, methodName);
    sql.append(value.toString());
    return sql.toString();
}
```

- 3) 对于 insertSQL(Object po)，构造思想如下：
 - a) 通过 po 对应的 ORMClass 类获取其所有属性（不包括 ID），作为 INSERT 的属性；
 - b) 通过反射从 po 中获取这些属性的值，构造 VALUES()子句；

代码如下：

```

public static String insertSQL(Object po) throws Exception {
    ORMClass ormClzz = config.getORMClass(po.getClass().getName());
    StringBuilder sql = new StringBuilder();
    sql.append("INSERT INTO ").append(ormClzz.getTable()).append("(");
    // INSERT INTO table(
    List<String> valueProps = new ArrayList<>();
    for (Map.Entry<String, Property> map : ormClzz.getProperties().entrySet()) {
        Property prop = map.getValue();
        sql.append(prop.getColumn() + ", ");
        valueProps.add(prop.getName());
    }
    // INSERT INTO table(attr1, attr2,
    // 将最后的 ', ' 换为 ')'
    sql.replace(sql.length()-2, sql.length()-1, ")");
    sql.append("VALUES(");
    // INSERT INTO table(attr1, attr2) VALUES(
    for (String vprop : valueProps) {
        String getter = CommonUtil.getterName(vprop);
        Object value = ReflectUtil.execute(po, getter);
        if (value.getClass() == String.class)
            sql.append("'" + value.toString() + "', ");
        else
            sql.append(value.toString() + ", ");
    }
    // INSERT INTO table(attr1, attr2) VALUES('lcx', 2,
    // 将最后的 ', ' 换为 ')'
    sql.replace(sql.length()-2, sql.length(), ")");
    // INSERT INTO table(attr1, attr2) VALUES('lcx', 2)
    return sql.toString();
}

```

3.5 DBUtil

DBUtil 中目前提供了两个公用的方法：insert(String sql)，用于执行插入语句；fillParams(Object po, ResultSet rs, List<Property> props)，使用查询结果 rs 来构造 JavaBean 对象（PO 对象），主要使用反射机制。代码如下：

```

public class DBUtil {
    private static BasicDataSource ds = DBConfiguration.getDataSource();

    public static boolean insert(String sql) throws Exception {
        Connection conn = ds.getConnection();
        conn.setAutoCommit(false);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(sql);
        conn.commit();
        conn.close();
        return true;
    }

    public static void fillParams(Object po, ResultSet rs, Property... props) throws Exception {
        fillParams(po, rs, Arrays.asList(props));
    }

    public static void fillParams(Object po, ResultSet rs, List<Property> props) throws Exception {
        if (rs.next()) {
            for (Property prop : props) {
                String setter = CommonUtil.setterName(prop.getName());
                if (Integer.class.getName().equals(prop.getType())) {
                    int value = rs.getInt(prop.getColumn());
                    ReflectUtil.execute(po, setter, value);
                } else { // else 默认为String
                    String value = rs.getString(prop.getColumn());
                    ReflectUtil.execute(po, setter, value);
                }
            }
        }
    }
}

```

3.6 LazyLoadProxy

属性的懒加载，就是在第一次查询时不获取值，等到真正用到的时候再去查取值。我的实现是：当外部调用对象懒加载属性的 `get()` 方法时，就去数据库查询相应的值，并将该值 `set` 到相应属性中，以此实现懒加载。

根据上面的描述，在进行动态代理时，就需要排除不需要代理的方法：

- a) 非“get”开头的方法；
- b) “getXXX()”，但 XXX 不是懒加载属性；

此外，如果调用原 `getXXX()` 方法后获取的值非空，则不用再进行数据库查询取值操作；代码如下：

```
public class LazyLoadProxy implements MethodInterceptor {
    private Object target;
    private ORMClass ormClzz;
    private BasicDataSource ds = DBConfiguration.getDataSource();

    public LazyLoadProxy(Object target, ORMClass ormClzz) {
        this.target = target;
        this.ormClzz = ormClzz;
    }

    public Object getInstance() {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        return enhancer.create();
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
        String methodName = method.getName();
        Object originalValue = proxy.invokeSuper(obj, args);
        if (!isNeedProxy(methodName) || originalValue != null)
            return originalValue;
        Object idvalue = ReflectUtil.execute(obj, CommonUtil.getterName(ormClzz.getId().getName()));
        if (idvalue == null)
            return originalValue;
        String propName = CommonUtil.getPropNameFromSetterOrGetter(methodName);
        String sql = SQLFactory.selectSQLByID(obj, propName);
        System.out.println("[LazyLoadProxy]-[intercept] SQL: " + sql);
        Connection conn = ds.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sql);
        Property prop = ormClzz.getProperties().get(propName);
        // 目前只支持Integer, String属性getXXX()方法的代理，代理其他类型的getter方法时会出错
        DBUtil.fillParams(obj, rs, prop);
        conn.close(); // 在rs使用完后关闭conn
        return proxy.invokeSuper(obj, args);
    }

    private boolean isNeedProxy(String methodName) {
        // 只对getXXX()做代理
        if (methodName.indexOf("get") != 0)
            return false;
        // 只对XXX 设置为LazyLoad做代理
        Map<String, Property> properties = ormClzz.getProperties();
        String propName = CommonUtil.getPropNameFromSetterOrGetter(methodName);
        if (propName.equals(ormClzz.getId().getName()))
            return ormClzz.getId().isLazy();
        else
            return properties.get(propName).isLazy();
    }
}
```

3.7 Conversation

Conversation 类中定义了数据库操作 CRUD 方法，每个方法将对象操作解释为目标数据库的 DML 或 DDL，通过 JDBC 完成数据库持久化。我的 Conversation 类（单例）中定义了 `getObject()` 和 `addObject()` 方法，以支持 login 和 register 业务。

- 1) 对于 `getObject()`，过程如下：
 - a) 获取 paramObject 对应的 ORMClass 对象 `ormClzz`；
 - b) 通过反射生成目标实例 `target`；

c) 通过 ProxyFactory 来生成目标实例 target 的懒加载代理对象 proxy;

d) 通过 SQLFactory 来生成 SELECT SQL 语句;

e) 通过 JDBC 来执行查询, 得到结果集: ResultSet rs;

d) 通过 DBUtil 的 fillParams() 和 rs 来为 proxy 填充参数值;

2) 对于 addObject(), 过程很简单: 通过 SQLFactory 生成 INSERT SQL 语句, 然后调用 DBUtil 的 insert(sql) 方法, 完毕。

代码如下:

```
public class Conversation {
    private Configuration config = Configuration.getConfiguration();
    private BasicDataSource ds = DBConfiguration.getDataSource();

    private Conversation() {}

    private static class ConversationHolder {
        private static Conversation conv = new Conversation();
    }

    public static Conversation getConversation() {
        return ConversationHolder.conv;
    }

    public Object getObject(Object paramObj, String... queryAttrs) throws Exception {
        Class<?> clzz = paramObj.getClass();
        ORMClass ormClzz = config.getORMClass(clzz.getName());
        Object target = ReflectUtil.newInstance(clzz.getName());
        Object proxy = ProxyFactory.newLazyLoadProxyInstance(target, ormClzz);
        String sql = SQLFactory.selectSQL(paramObj, queryAttrs);
        System.out.println("[Conversation]-[getObject] SQL: " + sql);
        Connection conn = ds.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sql);
        DBUtil.fillParams(proxy, rs, ormClzz.nolazyProps());
        conn.close(); // 在rs使用完毕后关闭conn
        return proxy;
    }

    public boolean addObject(Object paramObj) throws Exception {
        String sql = SQLFactory.insertSQL(paramObj);
        System.out.println("[Conversation]-[addObject] SQL: " + sql);
        return DBUtil.insert(sql);
    }
}
```

3.8 修改 BaseDAO、UserDAO、UserServiceImpl 逻辑

1) 修改 E5 的 BaseDAO 和 UserDAO 类, 改为对对象进行操作(即: 参数为 Object)。UserDAO 中通过 Conversation 类来执行数据库相关操作, 如下:

```
public abstract class BaseDAO {
    public abstract Object query(Object obj, String... queryAttrs) throws Exception;
    public abstract boolean insert(Object obj) throws Exception;
    public abstract boolean update(Object obj) throws Exception;
    public abstract boolean delete(Object obj) throws Exception;
}
```



```

public class UserDao extends BaseDAO {
    private Conversation conv = Conversation.getConversation();

    private UserDao() {}

    private static class UserDaoHolder {
        private static UserDao usrDAO = new UserDao();
    }

    public static UserDao getUserDAO() {
        return UserDaoHolder.usrDAO;
    }

    @Override
    public Object query(Object obj, String... queryAttrs) throws Exception {
        return conv.getObject(obj, queryAttrs);
    }

    @Override
    public boolean insert(Object obj) throws Exception {
        return conv.addObject(obj);
    }

    @Override
    public boolean update(Object obj) throws Exception {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean delete(Object obj) throws Exception {
        // TODO Auto-generated method stub
        return false;
    }
}

```

2) UserServiceImpl 类中的 login 和 register 方法，改为先生成参数对象 paramObj，然后调用 UserDao 的方法。如下：

```

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

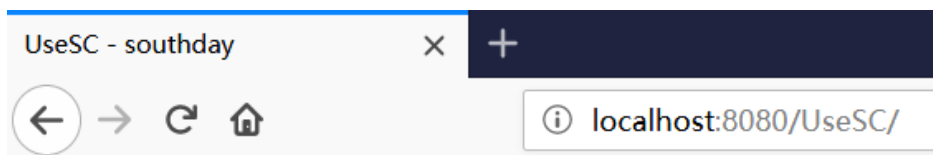
@Override
public boolean register(String userName, String password) throws Exception {
    if (!CommonUtil.checkParam(userName, password))
        return false;
    User paramUser = new User();
    paramUser.setUserName(userName);
    paramUser.setPassword(password);
    return userDAO.insert(paramUser);
}

```

3.9 测试结果

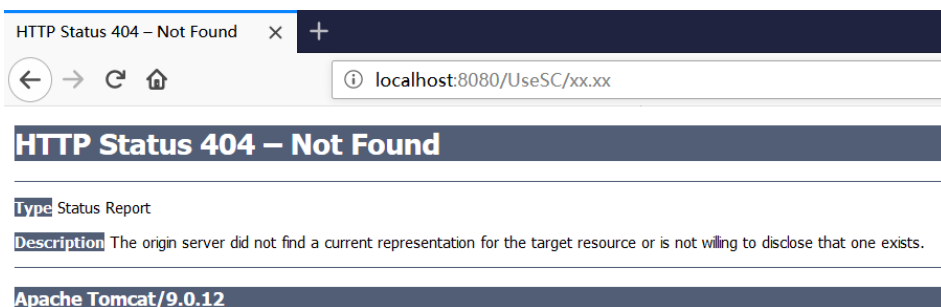
3.9.1 MySQL 部分

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

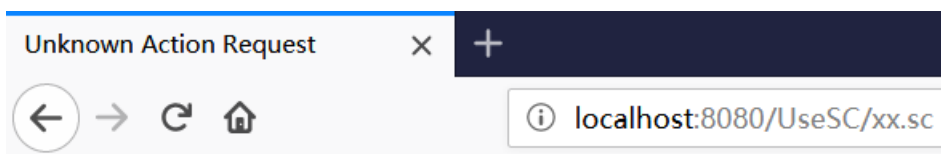


Welcome to UseSC!

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

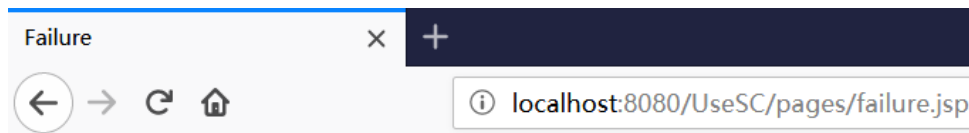


- 3) 非法请求: <http://localhost:8080/UseSC/xx.sc>, 无法识别的 action;



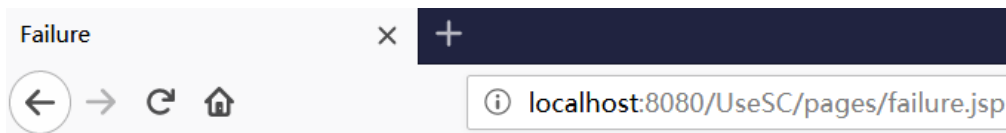
Unknown Action Request!

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



Failure!

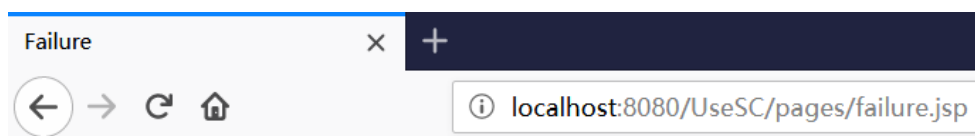
- 5) 测试登陆: <http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx>, 失败, 用户账号 lcx 不存在;



Failure!

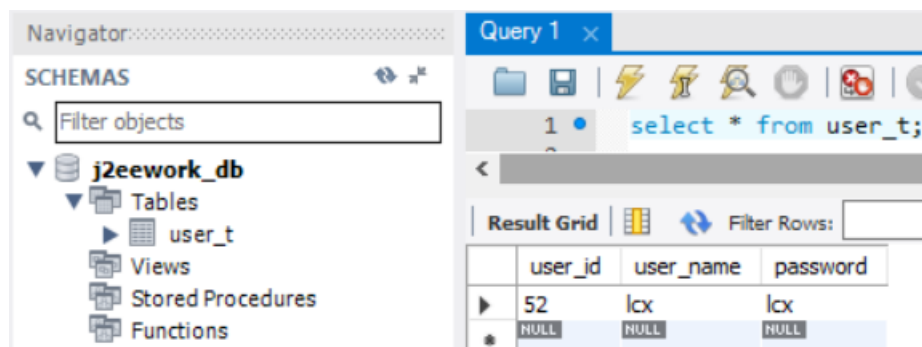
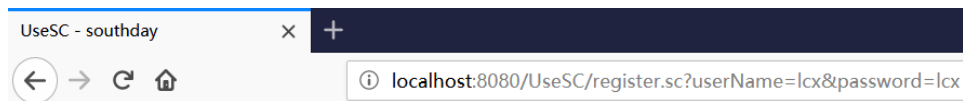
```
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE user_name = 'lcx'
```

- 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;



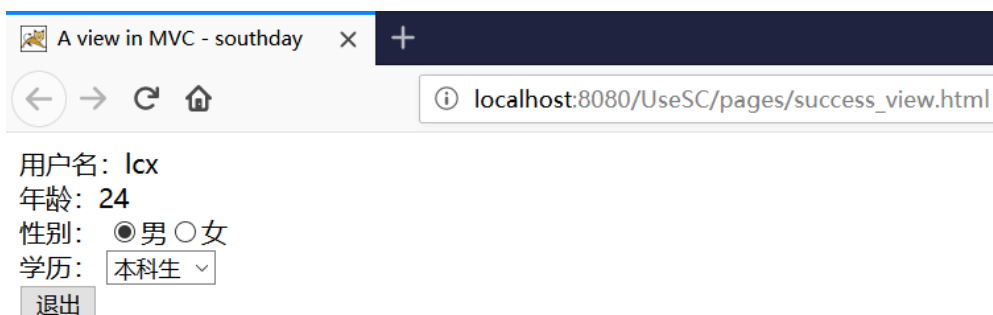
Failure!

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;



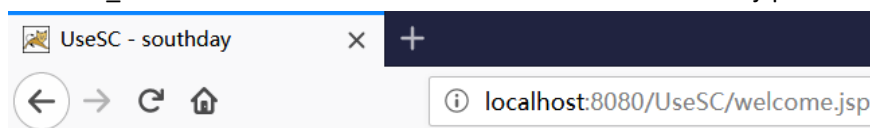
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name) VALUES('lcx', 'lcx')

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;



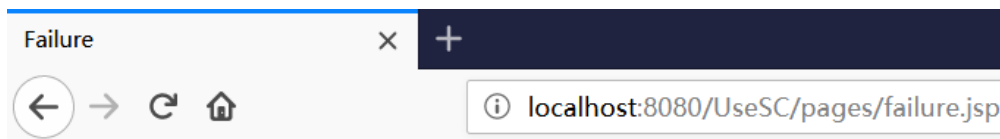
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE user_name = 'lcx'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id = 52

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



Welcome to UseSC!

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



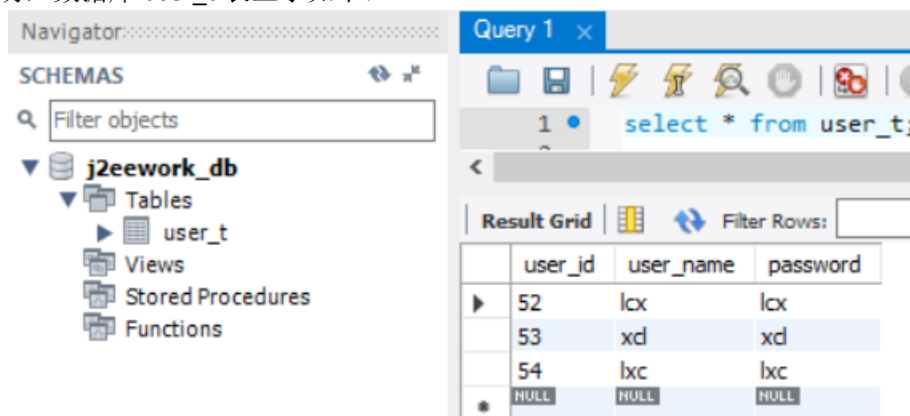
Failure!

```
[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
= 52
```

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

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

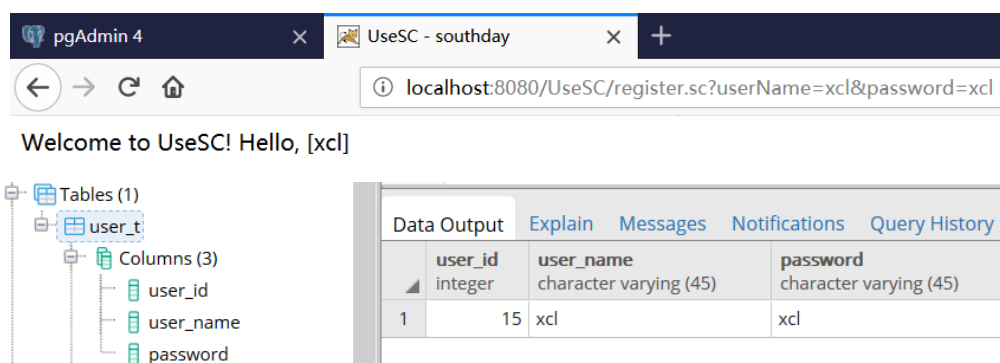
注册成功，数据库 user_t 表显示如下：



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

3.9.2 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；

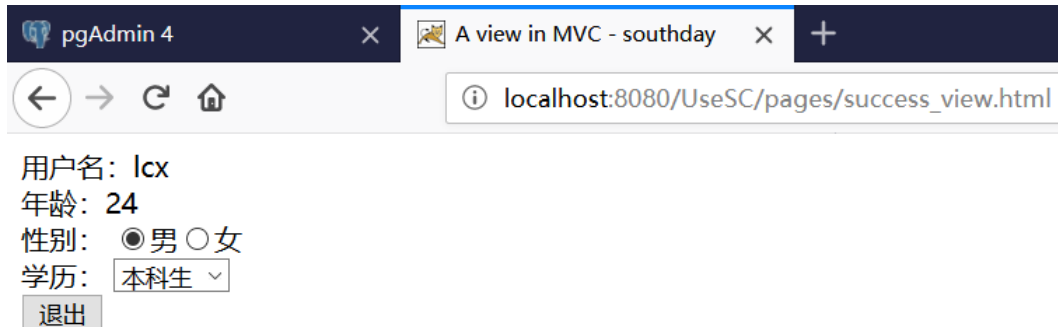


Data Output	Explain	Messages	Notifications	Query History
<div> <div>Tables (1)</div> <div> <div>user_t</div> <div>Columns (3)</div> <div> <div>user_id</div> <div>user_name</div> <div>password</div> </div> </div> </div>	<div> <div>user_id</div> <div>integer</div> </div>	<div> <div>user_name</div> <div>character varying (45)</div> </div>	<div> <div>password</div> <div>character varying (45)</div> </div>	
1	15	xcl	xcl	

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

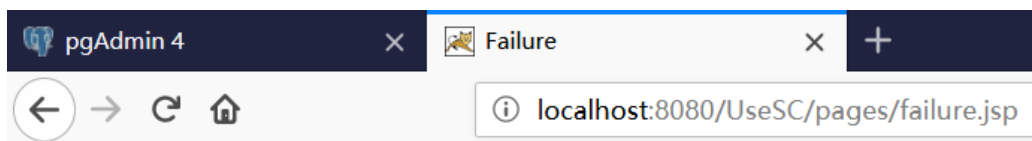
2) 注册成功后, 用 xcl 账号登陆:

<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
= 15
```

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
= 15
```

4) 使用<userName, password>=<lcX, lcX>, <cXl, cXl>注册账号:

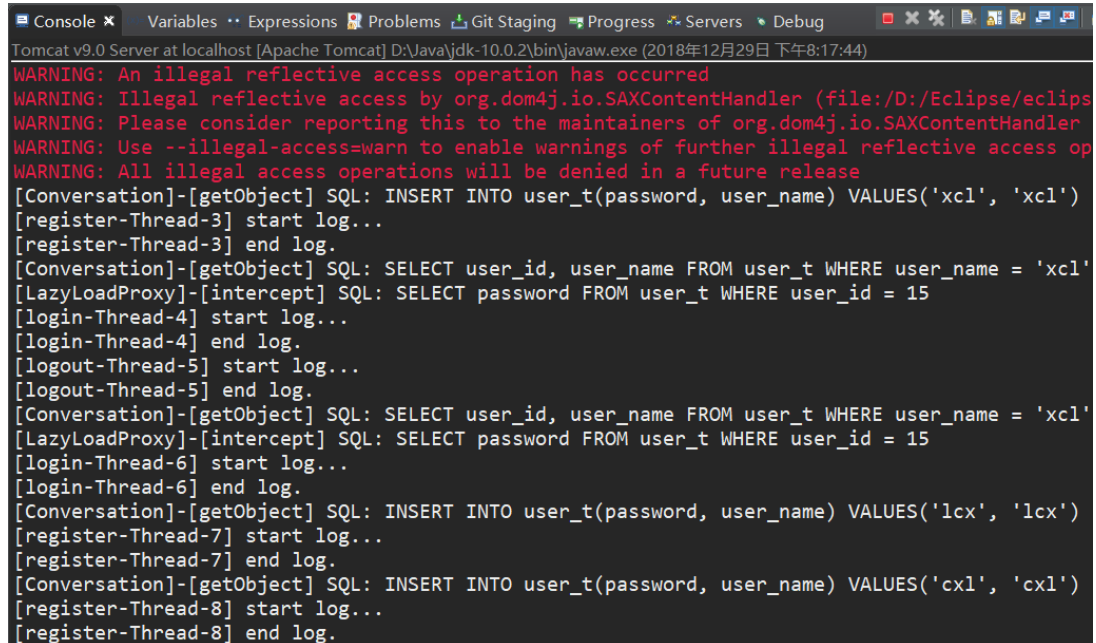
- <http://localhost:8080/UseSC/register.sc?userName=lcX&password=lcX>
- <http://localhost:8080/UseSC/register.sc?userName=cXl&password=cXl>

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	15	xcl	xcl	
2	16	lcX	lcX	
3	17	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('cx1', 'cx1')
```



The screenshot shows the Eclipse IDE's console window. The title bar indicates it's running Tomcat v9.0 Server at localhost. The console output includes several warning messages about illegal reflective access operations, followed by a series of log entries and SQL queries. The queries involve inserting and selecting data from a table named 'user_t'.

```
Tomcat v9.0 Server at localhost [Apache Tomcat] D:\Java\jdk-10.0.2\bin\javaw.exe (2018年12月29日 下午8:17:44)
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.dom4j.io.SAXContentHandler (file:/D:/Eclipse/eclipse
WARNING: Please consider reporting this to the maintainers of org.dom4j.io.SAXContentHandler
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access op
WARNING: All illegal access operations will be denied in a future release
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name) VALUES('cx1', 'cx1')
[register-Thread-3] start log...
[register-Thread-3] end log.
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE user_name = 'cx1'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id = 15
[login-Thread-4] start log...
[login-Thread-4] end log.
[logout-Thread-5] start log...
[logout-Thread-5] end log.
[Conversation]-[getObject] SQL: SELECT user_id, user_name FROM user_t WHERE user_name = 'cx1'
[LazyLoadProxy]-[intercept] SQL: SELECT password FROM user_t WHERE user_id = 15
[login-Thread-6] start log...
[login-Thread-6] end log.
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name) VALUES('lcx', 'lcx')
[register-Thread-7] start log...
[register-Thread-7] end log.
[Conversation]-[getObject] SQL: INSERT INTO user_t(password, user_name) VALUES('cx1', 'cx1')
[register-Thread-8] start log...
[register-Thread-8] end log.
```

4. 结论

4.1 总结

本次实验加深了我对 ORM 的理解，同时也了解了懒加载存在的意义，并且在实际项目中通过代理模式实现了对对象属性的懒加载。

临近期末，实验做得比较匆忙，感受就不是很多。在动态生成 SQL 部分，也是怎么简单怎么实现，有好多复杂的内容没有考虑到，比如：关联查询、子查询等；将结果集映射为 JavaBean 对象的实现也是简单粗暴，目前只支持 Integer 和 String 两种类型，并且也只能映射为一个 JavaBean 对象，而实际开发中经常是映射为一组对象（List）的。

4.2 问题及看法

4.2.1 java.sql.SQLException: Operation not allowed after ResultSet closed

本来在 DBUtil 中我还定义了 query(sql)方法，返回的是 ResultSet，然后外部获得 ResultSet 对象后，就可以去调用 DBUtil 的 fillParams()方法来填充参数值。但是运行时就报了上面的错，很明显：我在 conn.close()后还使用 ResultSet 对象的相关操作。

解决方法：

1) 将 query(sql)中的 conn.close()去掉，即不主动关闭连接，通过 jdbc-xxxxsql.properties 中配置自动回收属性来使 Connection 自动回收到连接池；

2) 不提供 query(sql)方法，让外部自己写 query，等使用完 ResultSet 对象后再 conn.close()；

我最终采取了第 2 种方案，因为觉得第 1 种具有不确定性，自动回收也可能使性能低下。最后的结果就类似下面的代码段：

```
Connection conn = ds.getConnection();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
Property prop = ormClzz.getProperties().get(propName);
DBUtil.fillParams(obj, rs, prop);
conn.close(); // 在 rs 使用完毕后关闭 conn
```

4.2.2 代理类类名与<class>中定义的<name>不匹配

```
public static String selectSQLByID(Object po, String propName) throws Exception {
    ORMClass ormClzz = config.getORMClass(po.getClass().getName());
    String col = ormClzz.getProperties().get(propName).getColumn();
    StringBuilder sql = new StringBuilder();
    sql.append("SELECT ").append(col).append(" FROM ").append(ormClzz.getTable());
    sql.append(" WHERE ").append(ormClzz.getId().getColumn()).append(" = ");
    String methodName = CommonUtil.getterName(ormClzz.getId().getName());
    Object value = ReflectUtil.execute(po, methodName);
    sql.append(value.toString());
    return sql.toString();
}
```

框中的代码报了空指针异常，我把 ormClzz 和 getProperties()等都输出后，发现 ormClzz 为 null，很惊讶。然后我继续输出了 po.getClass().getName()，得到的是这一串字符：southday.j2ework.water.ustc.bean.User\$\$EnhancerByCGLIB\$\$4110e76b，很明显和我的

or_mapping.xml 中定义的<name> southday.j2eework.water.ustc.bean.User</name>不匹配。

出现那串类路径的原因是参数 po 为代理对象。CGLIB 使用继承的方式来创建代理对象的，所以这里的 po 其实是 JVM 动态生成的 User 的子类对象（代理对象）。

而我的目的是，即使 po 是 User 的子类对象，我也要拿到 User 的 ORMClass。为了一劳永逸，我直接在 Configuration 的 getORMClass()中增加了对于动态代理子类对象的处理逻辑，如下：

```
public ORMClass getORMClass(String key) {  
    /* 当对象为动态代理生成的子类时，  
     * key就可能为这类形式: southday.j2eework.water.ustc.bean.User$$EnhancerByCGLIB$$4110e76b  
     * 为了能正确获取到ORMClass，这里做个转换，把从$到后面的内容全部切掉，只保留父类类路径  
     */  
    int index = key.indexOf('$');  
    return index < 0 ? classes.get(key) : classes.get(key.substring(0, index));  
}
```

5. 参考文献

- [1] Hibernate 学习笔记-懒加载 Lazy=true: <https://www.2cto.com/kf/201605/506464.html>
- [2] Hibernate ORM: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/
- [3] Introduction to the Java Persistence API:
<https://docs.oracle.com/javaee/5/tutorial/doc/bnbpz.html>
- [4] ORM 是什么？如何理解 ORM: <https://www.cnblogs.com/huanhang/p/6054908.html>