

轻量级 J2EE 框架应用

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

学号: **SA18225433**

姓名: 杨帆

报告撰写时间: **2018/1/6**

1.主题概述

1、ORM。

ORM 为 Object Relational Mapping，即对象关系映射，是为了解决面向对象与关系数据库之间不匹配的现象的技术，即通过描述对象和数据库之间映射的元数据，将对象自动持久化到关系数据库中，使得只需要像平时一样操作对象即可实现数据库的 crud 操作，因此我们采用编码的方式，为每一种数据库访问操作提供封装的方法，这样的做法提高了开发效率，不用 sql 编码，但是增加了系统层次，降低了运行效率。

2、Cglib 动态代理。

在 orm 映射关系中涉及到 lazyloading 的实现，被代理对象为 UserBean 类等实体类，考虑使用 cglib 动态代理来实现懒加载，代理类需要实现 MethodInterceptor 接口，重写 interceptor 方法，在其中添加需要在被代理方法执行前后的操作，这样当被代理对象执行方法时，就会调用 interceptor 方法。

3、懒加载

需要实现实体对象的延迟加载，即需要加载的对象不是一次加载完全，只是加载了必须的部分，没有加载的部分，只有当真正访问到的时候才去加载，所以设置了 lazyloading 的属性在创建代理对象时，并不会被加载，这样做的优点是，占用较少的系统资源，本次作业使用 cglib 模拟懒加载的实现，即对于设置了懒加载的对象属性，在进行查询操作时忽略，在代理类的 interceptor 中监听对象的访问情况，当调用 getxxx 方法时，表示该对象属性进行了访问，若为懒加载属性则去数据库中查询数据并返回给代理对象。

2. 假设


本次作业能够使用 IDEA 实现 Controller 具有 dao 层，能实现访问数据库并操作数据库的数据，根据映射关系文件解析出对象与数据库字段的对应关系，实现对象与数据库字段的对应关系，能够简单地操作对象向数据库存入数据，同时使用 cglib 动态代理模拟查询数据懒加载，当对象需要使用的時候才进行真正的查询。

3. 实现或证明

1. 实现成果

e6: <https://github.com/saaaaaaail/J2eee6>

2. 基于 E5。在 UseSC 工程中新建一个 XML 文件名为 `or_mapping.xml`，格式可参考如下：

在 resource 目录下新建文件  `or_mapping.xml`，

```
<?xml version="1.0" encoding="UTF-8" ?>
<OR-Mapping>
  <jdbc>
    <property name="driver" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/sc?useSSL=false"></property>
    <property name="username" value="root"></property><property name="password" value="123456"></property>
  </jdbc>
  <class name="sail.ustc.model.UserBean" table="user">
    <property name="userId" column="id" type="String" lazy="false"></property>
    <property name="userName" column="name" type="String" lazy="true"></property>
    <property name="userPass" column="password" type="String" lazy="true"></property>
  </class>
</OR-Mapping>
```

3. `or_mapping.xml` 中定义 JDBC 节点和 Class 节点。JDBC 节点为 `java jdbc` 属性配置；class 节点为 O/R 映射的实现。如，示例中对 `UserBean` 与 `user` 表作了映射。如图定义了 jdbc 节点和 Class 节点，jdbc 定义了 3 条属性 `driver`、`url`、`username`，Class 节点定义了 3 条属性 `userId`、`userName`、`userPass`。

4. 在 `SimpleController` 工程包 `sc.ustc.dao` 中新建 `Configuration` 类与 `Conversation` 类。`Configuration` 负责解析 UseSC 工程的配置 `or_mapping.xml`；`Conversation` 负责完成将对象操作映射为数据表操作，即在 `Conversation` 中定义数据操作 CRUD 方法，每个方法将对象操作解释成目标数据库的 DML 或 DDL，通过 JDBC 完成数据持久化。

在 `dao` 包中新建 `Configuration` 类，用来解析对象映射关系 xml 文件，首先使用类加载器获得文件的路径，然后创建 `DocumentBuilderFactory` 类获得文件构建类 `DocumentBuilder`，进而得到 xml 文件的 `document` 对象，

```

public Document getDocument() {
    try {
        String fileString = this.getClass().getClassLoader().getResource("or_mapping.xml").getPath();
        System.out.println("getdoc: "+fileString);
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document document = db.parse(new File(fileString));
        return document;
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

在解析 orm 文件之前，先创建三个 bean 类，用来存放解析文件的数据，JDBCEntity 用以存放 jdbc 节点解析出来的数据，

```

public class JDBCEntity {
    private String driver;
    private String url;
    private String username;
    private String userpass;
}

```

ORMUserEntity 类用以存放 userBean 类的 class 节点的属性数据，

```

public class ORMUserEntity {
    private String name;
    private String table;
    private List<PropertyEntity> pclist;
}

```

PropertyEntity 类用以存放 bean 类 Property 属性，包括其与数据库字段的映射字段的数据，

```

public class PropertyEntity {
    private String name;
    private String column;
    private String type;
    private String lazy;
}

```

然后使用 configJDBC()方法去解析对象映射关系文件获得连接 jdbc 的属性，存储到

```

//类静态对象，存储jdbc信息
private static JDBCEntity jdbcEntity;

```

JDBCEntity 类对象中

```

public JDBCEntity configJDBC() {
    if(jdbcEntity==null) {
        jdbcEntity = new JDBCEntity();
        Document document = getDocument();
        NodeList list = document.getElementsByTagName("jdbc");
        NodeList clist = list.item(index: 0).getChildNodes();
        for (int i = 0; i < clist.getLength(); i++) {
            Node node = clist.item(i);
            NamedNodeMap nodeMap = node.getAttributes();
            String name = nodeMap.getNamedItem("name").getNodeValue();
            String value = nodeMap.getNamedItem("value").getNodeValue();
            System.out.println("jdbcConfig: " + name + " " + value);
            switch (name) {
                case "driver":
                    jdbcEntity.setDriver(value);
                    break;
                case "url":
                    jdbcEntity.setUrl(value);
                    break;
                case "username":
                    jdbcEntity.setUsername(value);
                    break;
                case "password":
                    jdbcEntity.setUserpass(value);
                    break;
            }
        }
    }
    return jdbcEntity;
}

```

然后使用 configUserORM(String bean)方法去解析 bean 类与数据库的映射关系节点，存

储到类静态变量中

```

//类静态对象，存储映射关系
private static ORMUserEntity ormUserEntity;

```

```

public ORMUserEntity configUserORM(String bean) {
    if(ormUserEntity == null) {
        ormUserEntity = new ORMUserEntity();
        List<PropertyEntity> properties = new ArrayList<>();
        Document document = getDocument();
        NodeList list = document.getElementsByTagName("class");
        System.out.println("configUserORM: listlength: " + list.getLength());
        String beanClass = bean;
        System.out.println("configUserORM: beanClass " + bean);
        for (int i = 0; i < list.getLength(); i++) {
            Node node = list.item(i);
            NamedNodeMap nodeMap = node.getAttributes();
            String name = nodeMap.getNamedItem("name").getNodeValue();
            String table = nodeMap.getNamedItem("table").getNodeValue();
            if (name.equals(beanClass)) {
                ormUserEntity.setName(name);
                ormUserEntity.setTable(table);
                NodeList clist = node.getChildNodes();
                System.out.println("configUserORM: node: " + node.getNodeName());
                System.out.println("configUserORM: clist: " + clist);
                for (int j = 0; j < clist.getLength(); j++) {
                    NamedNodeMap cNodeMap = clist.item(j).getAttributes();
                    String pname = cNodeMap.getNamedItem("name").getNodeValue();
                    String pcolumn = cNodeMap.getNamedItem("column").getNodeValue();
                    String ptype = cNodeMap.getNamedItem("type").getNodeValue();
                    String plazy = cNodeMap.getNamedItem("lazy").getNodeValue();
                    PropertyEntity pe = new PropertyEntity(pname, pcolumn, ptype, plazy);
                    properties.add(pe);
                }
                ormUserEntity.setPclist(properties);
            }
        }
    }
    return ormUserEntity;
}

```

以上将映射关系保存到实体类中，然后创建 Conversation 类中 getConnection()方法，用来获得数据库连接，

```

private static Connection getConnection() {
    try {
        Configuration configuration = Configuration.getInstance();
        JDBCEntity jdbcEntity = configuration.getConfigJDBC();
        Class.forName(jdbcEntity.getDriver());
        Connection connection = DriverManager.getConnection(
            jdbcEntity.getUrl(),
            jdbcEntity.getUsername(),
            jdbcEntity.getUserpass());
        return connection;
    } catch (ClassNotFoundException e) {
    }
}

```

创建 getObject(Object o)方法进行数据库的查询操作，首先通过反射获得要查询类的全限定名，使用 Configuration 类获得映射关系，获得表名，获得属性对应关系 list，

```

Configuration configuration = Configuration.getInstance();
//获得类名
Class cls = o.getClass();
System.out.println("getObject: "+cls.getName());
//获得映射关系
ORMUserEntity ormUserEntity = configuration.configUserORM(cls.getName());
//表名
String table = ormUserEntity.getTable();
String className = ormUserEntity.getName();
System.out.println("table: "+ table);

//使代理对象与原始对象类属性一致

ResultSet resultSet=null;
//属性对应关系
List<PropertyEntity> list = ormUserEntity.getPclist();

```

匹配到数据库的 id 字段，读取到对象的 id 属性的属性名，使用反射获得该对象的 id 属性的值，然后拼接 sql 语句，查询所有属性，获得结果集，

```

for(int i=0;i<list.size();i++){
    if(list.get(i).getColumn().equals("id")){
        //获得对象属性
        Field f = cls.getDeclaredField(list.get(i).getName());
        f.setAccessible(true);
        System.out.println("f: "+list.get(i).getName());
        //获得属性值
        String value = f.get(o).toString();

        String SQL = "select * from " + table +
            " where " + list.get(i).getColumn() + " = " + value;
        //String SQL = "select * from " + table + " where name = 'sail'";
        System.out.println(SQL);
        connection = getConnection();
        Statement statement = connection.createStatement();
        resultSet = statement.executeQuery(SQL);
    }
}

```

使用类对象的 newInstance 方法获得对象实例，遍历结果集，获得属性名，使用反射获得属性的 field 对象，保存对应属性的值到对象实例中，


```
//构造返回对象
Object newObject = cls.newInstance();

while (resultSet.next()) {
    //获得对象属性
    for (int i = 0; i < list.size(); i++) {
        Field f = cls.getDeclaredField(list.get(i).getName());
        f.setAccessible(true);
        //设置返回对象的属性值

        String v = resultSet.getString(list.get(i).getColumn());
        System.out.println("returnValue: " + v);
        f.set(newObject, v);
    }
}

return newObject;
```

5.修改 E 5 UseSC 中的 UserDao 代码，使用 Conversation，将 UserDao 中的数据的 CRUD 操作全部修改为对对象的 CRUD 操作：

如，查询 userId 为 100 的 UserBean 对象的代码为 Conversation.getObject(new UserBean(100))；删除 userId 为 100 的 UserBean 对象的代码为 Conversation.deleteObject(new UserBean(100));(如果考虑对不同类型属性都可查询，可使用泛型作为查询方法参数类型，或者使用方法重载)

将 SimpleController 工程打包为 jar，放到 UseSC 工程 lib 目录下，修改 UserDao 类中的 CRUD 操作，

```
@Override
public Object query(String s) {

    UserBean newUserBean = null;
    try {
        newUserBean = (UserBean) Conversation.getObject(new UserBean(s));
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return newUserBean;
}
```

```

@Override
public boolean delete(String s) {
    try {
        return Conversation.deleteObject(new UserBean(s));
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}

```

6.将修改后的 SimpleController 打包为 jar，UseSC 打包为 war，部署到 Tomcat 中测试。

控制台打印输出 Configuration 类的解析结果，

```

configUserORM:listlength: 1
configUserORM:beanClass sail.ustc.model.UserBean
configUserORM:name, table: sail.ustc.model.UserBean, user
configUserORM:node: class
configUserORM:clist: [class: null]
configUserORM:pname: userId
configUserORM:pname: userName
configUserORM:pname: userPass
table: user
FIELD: id
select id from user where id = 2
getdoc: /G:/IntelliJIDEA/workspaces_web/J2eee7/UseSC/out/artifacts/UseSC_war_exploded/WEB-INF/classes/or_mapping.xml
jdbcConfig: driver com.mysql.jdbc.Driver
jdbcConfig: url jdbc:mysql://localhost:3306/sc?useSSL=false
jdbcConfig: username root
jdbcConfig: password 123456
sql value: 2

```

在输入框输入 id，密码框输入密码，

首先会使用 loadObject 方法通过 id 进行懒加载的查询获得 userBean 对象，然后进行密码比较会调用 getPassword 方法，则会判断 password 属性是否为空，为空则到数据库去查询返回具有该属性的代理对象，控制台打印结果如图，

```

methodString: getUserPass
true
调用了get方法!!
查询userpass
getObject: sail.ustc.model.UserBean
UserBean
idValue: 2
select password from user where id = 2
returnValue: 1
methodString: setUserPass
false
查询userpass结束返回
methodString: getUserPass
true
调用了get方法!!
signIn中第二次getpassword: true
success

```

7.将 5 中测试的数据库修改为另一个 DBMS，仅修改 **Conversation** 代码，重新进行打包和部署，并测试结果。如将 **mysql** 修改为 **sqlite**。

将 **mysql** 修改为 **sqlite**，修改 **UserDAO** 代码，初始化数据库驱动与数据库的 **url** 地址，并且获得 **sqlite** 数据库的连接对象，

```

private UserDAO() {
    /*
    userPassword = "123456";
    userName = "root";
    driver = "com.mysql.jdbc.Driver";
    url = "jdbc:mysql://localhost:3306/sc?useSSL=false&serverTimezone=UTC";
    */
    driver="org.sqlite.JDBC";
    url="jdbc:sqlite:G:\\sqliteDB\\identifier.sqlite";

    //connection = openDBConnectionMySQL();
    connection = openDBConnectionSQLite();
}

```

8.实现对象属性 **lazy-loading**（可通过代理模式 **Proxy Pattern** 实现）。

实现对象属性的懒加载，即查询对象时不查询具体的值，仅返回一个空对象，当需要对对象的某条属性时，再去查询数据库获得属性值再去执行操作，使用 **cglib** 动态代理实现对象属性的懒加载，创建一个代理类 **UserProxy** 实现 **MethodInterceptor** 接口，创建代理

类的普通对象 `//业务类对象`
`private Object target;`，重写 **intercept** 方法，

```

@Override
public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

```

获得代理类执行属性方法时的方法名，使用正则表达式匹配方法是否为属性的 **get** 方法，如果是 **get** 表示需要使用属性值，

```
Object obj=null;
UserBean userBean = (UserBean) o;
String methodString = method.getName();
//String methodString = "gets";
System.out.println("methodString: "+methodString);
String regex = "get[a-zA-Z]*";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(methodString);
System.out.println(matcher.matches());
if(matcher.matches()){
    System.out.println("调用了get方法!!");
}
```

紧接着判断是 `get` 哪一个属性来查询对应的属性值，获得普通的未代理对象 `target`，判断对应的属性是否为 `null`，如果为空则进行数据库查询操作，获得返回对象，同时更新普通的未代理对象 `target`，同时使用 `setter` 方法更新代理对象 `userBean` 的对应属性，

```
switch (methodString){
    case "getId":
        break;
    case "getName":{
        UserBean ub = (UserBean) target;
        if(ub.getUserName()==null){
            System.out.println("查询username");
            String username = sail.ustc.dao.Conversation.getFieldById(target, field: "name");
            ub.setUserName(username);
            target = ub;
            userBean.setUserName(username);
            System.out.println("查询username结束返回");
        }
        break;}
    case "getPass":{
        UserBean ub = (UserBean) target;
        if(ub.getUserPass()==null){
            System.out.println("查询userpass");
            String password = sail.ustc.dao.Conversation.getFieldById(target, field: "password");
            ub.setUserPass(password);
            target = ub;
            userBean.setUserPass(password);
            System.out.println("查询userpass结束返回");
        }
        break;}
}
```

最后执行代理对象本来应该需要执行的方法，

```
obj = methodProxy.invokeSuper(userBean, objects);
```

还需要编写一个获得代理对象的方法 `getInstance(Object target)`，传入目标对象 `target`，返回动态代理对象，

```

public Object getInstance(Object target) {
    this.target=target;//业务对象赋值
    Enhancer enhancer = new Enhancer();//创建加强器，
    enhancer.setSuperclass(this.target.getClass());//
    //设置回调：对于代理类上所有方法的调用，都会调用Call
    enhancer.setCallback(this);
    //创建动态代理类对象并返回
    return enhancer.create();
}

```

同时还需要编写懒加载的对应 crud 方法 loadObject(Object o)，通过类名查询对象映射关系，获得属性 list 然后遍历属性判断是否为懒加载，拼接 sql 语句需要查询的字段，如果为懒加载字段，则不添加，

```

//选择没有懒加载的数据
for(PropertyEntity pe:list){
    if(pe.getLazy().equals("false")){
        String column = pe.getColumn();
        FIELD.append(" "+column+",");
    }
}
char lastChar = FIELD.charAt(FIELD.length()-1);
if(lastChar==' ') {
    FIELD.deleteCharAt(FIELD.length() - 1);
}
System.out.println("FIELD: "+FIELD);

```

然后获得 id 属性值，使用 id 属性值查询对应的数据，

```

//获得id属性值
for(PropertyEntity pe:list){
    if(pe.getColumn().equals("id")){
        Field f = cls.getDeclaredField(pe.getName());
        f.setAccessible(true);
        //获得id属性值
        String value = f.get(o).toString();

        //sql
        String SQL = "select "+FIELD+" from "+table+" where "+pe.getColumn()+" = "+value;
        System.out.println(SQL);

        Connection connection = getConnection();
        PreparedStatement pstmt = connection.prepareStatement(SQL);
        resultSet = pstmt.executeQuery();
    }
}

```

最后构建返回对象，遍历结果集，通过反射将属性值填充到对象中，

```
//构建返回对象
Object newObj = cls.newInstance();
while (resultSet.next()) {
    for (PropertyEntity pe: list) {
        if (pe.getLazy().equals("false")) {
            Field f = cls.getDeclaredField(pe.getName());
            f.setAccessible(true);
            String value = resultSet.getString(pe.getColumn());
            System.out.println("sql value: " + value);
            f.set(newObj, value);
        }
    }
}
```

4. 结论

对主题的总结，结果评论，发现的问题，或你的建议和看法。

本次作业学习了数据操作的层次 `dao`，了解了 `dao` 层存在的作用以及为什么要添加 `dao` 层，同时学习了对象映射关系的思想，通过反射实现操作对象属性与数据库字段的一一对应，同时编写代码解析了映射关系文件，重新改写了 `dao` 层操作数据库的方式，直接操作 `Bean` 对象，完成数据库的查询操作，同时学习了懒加载的思想，编写懒加载方式的查询，通过 `cglib` 动态代理实现判断当对象调用 `get` 方法时去查询对应属性，最后改变数据库驱动，使用 `sqlite` 替换 `mysql` 成功连接到 `sqlite`。

第一个问题，关于 `java.lang.NoClassDefFoundError: org/objectweb/asm/Type`;

使用 `cglib` 动态代理时，添加依赖包以后报错，查询资料得知为 `cglib` 版本问题，更换 `cglib` 版本为更高的版本后即可。

第二个问题，使用反射无法获得动态代理类的属性与方法并修改；

在添加 `UserBean` 类的动态代理对象时，使用通用的代理类后，使用反射无法判断代理对象所代理的目标对象是哪个，即无法修改代理对象的属性值，解决方法为创建专用于 `UserBean` 的代理类对象 `UserProxy`，使用类型强转获得原始类对象是哪个，并使用 `setter` 方法设置代理对象的属性值。

5.参考文献

- 1.参阅 [github](#) 学长代码编写风格
- 2.自己实现一个简单的 [Mybatis](#) 框架
- 3.[ORM 是什么？ 如何理解 ORM](#)
- 4.[什么是 ORM?为啥要是用 ORM?](#)
- 5.[JAVA 反射中的 `getFields\(\)`方法和 `getDeclaredFields \(\)`方法的区别](#)
- 6.[java 使用反射给对象属性赋值的两种方法](#)
- 7.[Java 使用反射通过对象属性获取属性的值](#)
- 8.[Java DOM 处理 XML 时调用 `getChildNodes` 函数，子节点个数问题](#)
- 9.[获取 DOM 节点的几种方式](#)
- 10.[Java 动态代理之 JDK 实现和 Cglib 实现（简单易懂）](#)
- 11.[StringBuilder &&StringBuffer 删除最后一个字符的方法](#)
- 12.[使用 Method 类获取方法的全名称](#)
- 13.[关于 java.lang.NoClassDefFoundError: org/objectweb/asm/Type](#)
- 14.[java 正则表达式——规则表](#)
- 15.[ORM 的概念, ORM 到底是什么](#)
- 16.[Cglib 动态代理实现原理](#)