

Mybatis 第二天

框架课程

# 1. 课程计划

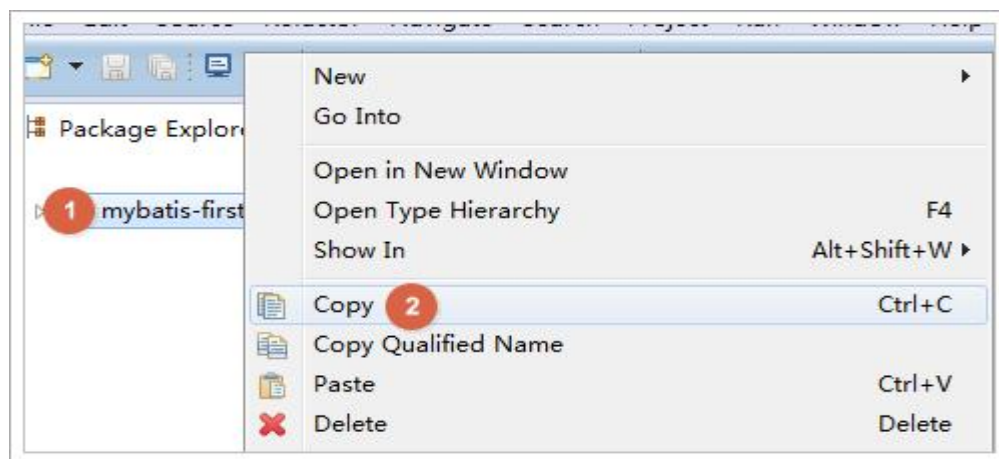
- 1、输入映射和输出映射
  - a) 输入参数映射
  - b) 返回值映射
- 2、动态 sql
  - a) If 标签
  - b) Where 标签
  - c) Sql 片段
  - d) Foreach 标签
- 3、关联查询
  - a) 一对一关联
  - b) 一对多关联
- 4、Mybatis 整合 spring
  - a) 如何整合 spring
  - b) 使用原始的方式开发 dao
  - c) 使用 Mapper 接口动态代理
- 5、Mybatis 逆向工程（了解）

## 2. 输入映射和输出映射

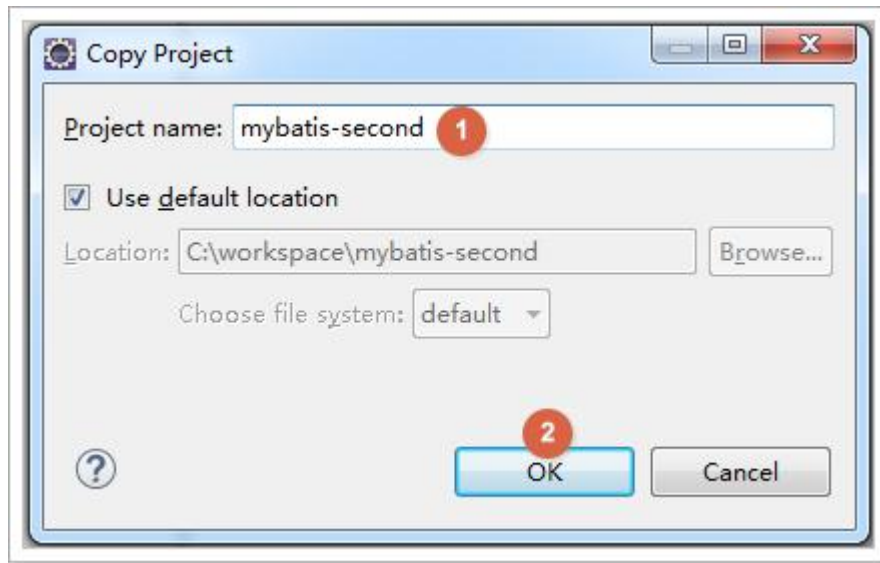
Mapper.xml 映射文件中定义了操作数据库的 sql，每个 sql 是一个 statement，映射文件是 mybatis 的核心。

### 2.1. 环境准备

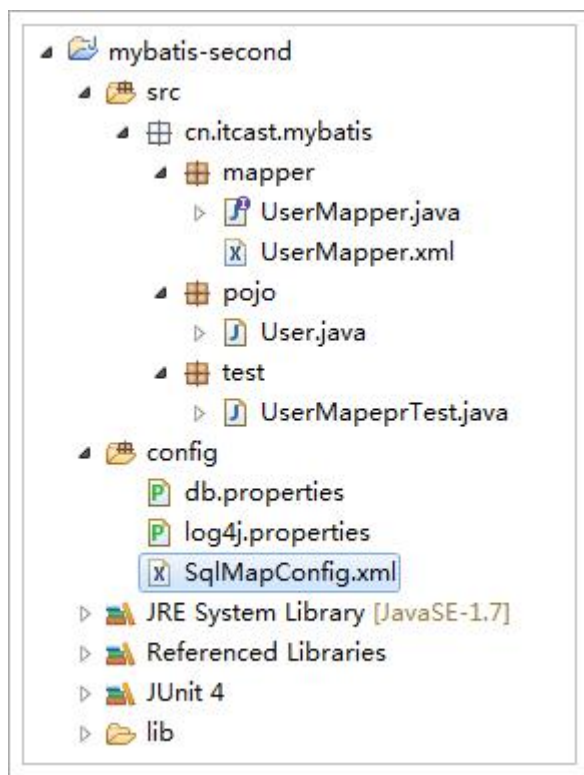
1. 复制昨天的工程，按照下图进行



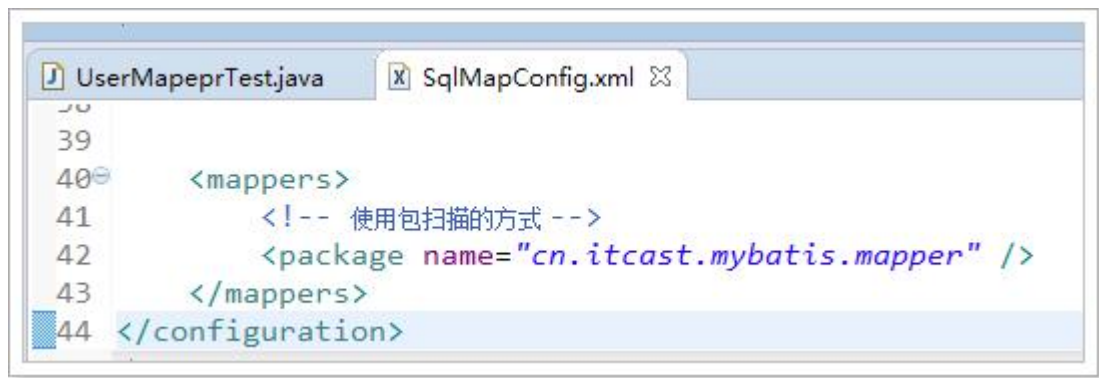
2. 如下图粘贴，并更名



3. 只保留 Mapper 接口开发相关的文件，其他的删除  
最终效果如下图：



4. 如下图修改 SqlMapConfig.xml 配置文件。Mapper 映射器只保留包扫描的方式



## 2.2. parameterType(输入类型)

### 2.2.1. 传递简单类型

参考第一天内容。

使用#{ }占位符，或者\${ }进行 sql 拼接。

### 2.2.2. 传递 pojo 对象

参考第一天的内容。

Mybatis 使用 ognl 表达式解析对象字段的值，#{ }或者\${ }括号中的值为 pojo 属性名称。

### 2.2.3. 传递 pojo 包装对象

开发中通过可以使用 pojo 传递查询条件。

查询条件可能是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如查询用户信息的时候，将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

包装对象：Pojo 类中的一个属性是另外一个 pojo。

需求：根据用户名模糊查询用户信息，查询条件放到 QueryVo 的 user 属性中。

### 2.2.3.1. 编写 QueryVo

```
public class QueryVo {  
    // 包含其他的 pojo  
    private User user;  
  
    public User getUser() {  
        return user;  
    }  
    public void setUser(User user) {  
        this.user = user;  
    }  
}
```

### 2.2.3.2. Sql 语句

SELECT \* FROM user WHERE username LIKE '%张%'

### 2.2.3.3. Mapper.xml 文件

在 UserMapper.xml 中配置 sql，如下图。



### 2.2.3.4. Mapper 接口

在 UserMapper 接口中添加方法，如下图：



### 2.2.3.5. 测试方法

在 UserMapperTest 增加测试方法，如下：

```
@Test
public void testQueryUserByQueryVo() {
    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象，整合之后，交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 使用 userMapper 执行查询，使用包装对象
    QueryVo queryVo = new QueryVo();
    // 设置 user 条件
    User user = new User();
    user.setUsername("张");
    // 设置到包装对象中
    queryVo.setUser(user);

    // 执行查询
    List<User> list = userMapper.queryUserByQueryVo(queryVo);
    for (User u : list) {
        System.out.println(u);
    }

    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    sqlSession.close();
}
```

### 2.2.3.6. 效果

测试结果如下图：

```
DEBUG [main] - ==> Preparing: SELECT * FROM `user` WHERE username LIKE '%张%'
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==          Total: 3
User [id=10, username=张三, sex=1, birthday=Thu Jul 10 00:00:00 CST 2014, address=北京市]
User [id=16, username=张小明, sex=1, birthday=null, address=河南郑州]
User [id=24, username=张三丰, sex=1, birthday=null, address=河南郑州]
```

## 2.3. resultType(输出类型)

### 2.3.1. 输出简单类型

需求:查询用户表数据条数

sql: SELECT count(\*) FROM `user`

#### 2.3.1.1. Mapper.xml 文件

在 UserMapper.xml 中配置 sql, 如下图:



#### 2.3.1.2. Mapper 接口

在 UserMapper 添加方法, 如下图:



#### 2.3.1.3. 测试方法

在 UserMapeprTest 增加测试方法, 如下:

```
@Test
public void testQueryUserCount() {
    // mybatis 和 spring 整合, 整合之后, 交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象, 整合之后, 交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
```

```

// 使用 userMapper 执行查询用户数据条数
int count = userMapper.queryUserCount();
System.out.println(count);

// mybatis 和 spring 整合，整合之后，交给 spring 管理
sqlSession.close();
}

```

#### 2.3.1.4. 效果

测试结果如下图：

```

DEBUG [main] - ==> Preparing: SELECT count(*) FROM `user`
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==      Total: 1
14

```

注意：输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。

#### 2.3.2. 输出 pojo 对象

参考第一天内容

#### 2.3.3. 输出 pojo 列表

参考第一天内容。

### 2.4. resultMap

`resultType` 可以指定将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 `resultMap` 将字段名和属性名作一个对应关系，`resultMap` 实质上还需要将查询结果映射到 pojo 对象中。

`resultMap` 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。

需求：查询订单表 `order` 的所有数据

sql: `SELECT id, user_id, number, createtime, note FROM `order``



### 2.4.1. 声明 pojo 对象

数据库表如下图：

```
CREATE TABLE `order` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `user_id` int(11) NOT NULL COMMENT '下单用户id',  
  `number` varchar(32) NOT NULL COMMENT '订单号',  
  `createtime` datetime NOT NULL COMMENT '创建订单时间',  
  `note` varchar(100) DEFAULT NULL COMMENT '备注',  
  PRIMARY KEY (`id`),  
  KEY `FK_orders_1` (`user_id`),  
  CONSTRAINT `FK_order_id` FOREIGN KEY (`user_id`) REFEREN  
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

Order 对象：

```
public class Order {  
    // 订单 id  
    private int id;  
    // 用户 id  
    private Integer userId;  
    // 订单号  
    private String number;  
    // 订单创建时间  
    private Date createtime;  
    // 备注  
    private String note;  
    get/set...  
}
```

### 2.4.2. Mapper.xml 文件

创建 OrderMapper.xml 配置文件，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<!-- namespace: 命名空间，用于隔离 sql，还有一个很重要的作用，Mapper 动态代理开发的时候使用，需要指定 Mapper 的类路径 -->  
<mapper namespace="com.cloudtcc.mybatis.mapper.OrderMapper">  
    <!-- 查询所有的订单数据 -->  
    <select id="queryOrderAll" resultType="order">  
        SELECT id, user_id,  
        number,  
        createtime, note FROM `order`
```

```
</select>
</mapper>
```

### 2.4.3. Mapper 接口

编写接口如下：

```
public interface OrderMapper {
    /**
     * 查询所有订单
     *
     * @return
     */
    List<Order> queryOrderAll();
}
```

### 2.4.4. 测试方法

编写测试方法 OrderMapperTest 如下：

```
public class OrderMapperTest {
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void init() throws Exception {
        InputStream inputStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
        this.sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    }

    @Test
    public void testQueryAll() {
        // 获取 sqlSession
        SqlSession sqlSession = this.sqlSessionFactory.openSession();
        // 获取 OrderMapper
        OrderMapper orderMapper =
sqlSession.getMapper(OrderMapper.class);

        // 执行查询
        List<Order> list = orderMapper.queryOrderAll();
        for (Order order : list) {
            System.out.println(order);
        }
    }
}
```

```
}
```

## 2.4.5. 效果

测试效果如下图：

```
DEBUG [main] - ==> Preparing: SELECT id, user_id, number, createtime, note FROM `order`
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==          Total: 3
Order [id=3, userId=null, number=1000010, createtime=Wed Feb 04 13:22:35 CST 2015, note=null]
Order [id=4, userId=null, number=1000011, createtime=Tue Feb 03 13:22:41 CST 2015, note=null]
Order [id=5, userId=null, number=1000012, createtime=Thu Feb 12 16:13:23 CST 2015, note=null]
```

发现 userId 为 null

解决方案：使用 resultMap

## 2.4.6. 使用 resultMap

由于上边的 mapper.xml 中 sql 查询列(user\_id)和 Order 类属性(userId)不一致，所以查询结果不能映射到 pojo 中。

需要定义 resultMap，把 orderResultMap 将 sql 查询列(user\_id)和 Order 类属性(userId)对应起来

改造 OrderMapper.xml，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- namespace: 命名空间，用于隔离 sql，还有一个很重要的作用，Mapper 动态代理开发的时候使用，需要指定 Mapper 的类路径 -->
<mapper namespace="com.cloudtcc.mybatis.mapper.OrderMapper">

    <!-- resultMap 最终还是要将结果映射到 pojo 上，type 就是指定映射到哪一个 pojo -->
    <!-- id: 设置 ResultMap 的 id -->
    <resultMap type="order" id="orderResultMap">
        <!-- 定义主键，非常重要。如果是多个字段，则定义多个 id -->
        <!-- property: 主键在 pojo 中的属性名 -->
        <!-- column: 主键在数据库中的列名 -->
        <id property="id" column="id" />

        <!-- 定义普通属性 -->
        <result property="userId" column="user_id" />
    </resultMap>

```

```

        <result property="number" column="number" />
        <result property="createtime" column="createtime" />
        <result property="note" column="note" />
    </resultMap>

    <!-- 查询所有的订单数据 -->
    <select id="queryOrderAll" resultMap="orderResultMap">
        SELECT id, user_id,
            number,
            createtime, note FROM `order`
    </select>

</mapper>

```

### 2.4.7. 效果

只需要修改 Mapper.xml 就可以了，再次测试结果如下：

```

DEBUG [main] - ==> Preparing: SELECT id, user_id, number, createtime, note FROM `order`
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==          Total: 3
Order [id=3, userId=1, number=1000010, createtime=Wed Feb 04 13:22:35 CST 2015, note=null]
Order [id=4, userId=1, number=1000011, createtime=Tue Feb 03 13:22:41 CST 2015, note=null]
Order [id=5, userId=10, number=1000012, createtime=Thu Feb 12 16:13:23 CST 2015, note=null]

```

## 3. 动态 sql

通过 mybatis 提供的各种标签方法实现动态拼接 sql。

需求：根据性别和名字查询用户

查询 sql:

```
SELECT id, username, birthday, sex, address FROM `user` WHERE sex = 1 AND
username LIKE '%张%'
```

### 3.1. If 标签

#### 3.1.1. Mapper.xml 文件

UserMapper.xml 配置 sql，如下：

```

<!-- 根据条件查询用户 -->
<select id="queryUserByWhere" parameterType="user" resultType="user">
    SELECT id, username, birthday, sex, address FROM `user`

```

```
WHERE sex = #{sex} AND username LIKE
    '%${username}%'
</select>
```

### 3.1.2. Mapper 接口

编写 Mapper 接口，如下图：



### 3.1.3. 测试方法

在 UserMapperTest 添加测试方法，如下：

```
@Test
public void testQueryUserByWhere() {
    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象，整合之后，交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 使用 userMapper 执行根据条件查询用户
    User user = new User();
    user.setSex("1");
    user.setUsername("张");

    List<User> list = userMapper.queryUserByWhere(user);

    for (User u : list) {
        System.out.println(u);
    }

    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    sqlSession.close();
}
```

### 3.1.4. 效果

测试效果如下图：

```
DEBUG [main] - ==> Preparing: SELECT id, username, birthday, sex, address FROM `user`
DEBUG [main] - ==> Parameters: 1(String)
DEBUG [main] - <==      Total: 3
User [id=10, username=张三, sex=1, birthday=Thu Jul 10 00:00:00 CST 2014, address=北京市]
User [id=16, username=张小明, sex=1, birthday=null, address=河南郑州]
User [id=24, username=张三丰, sex=1, birthday=null, address=河南郑州]
```

如果注释掉 `user.setSex("1")`，测试结果如下图：

```
DEBUG [main] - ==> Preparing: SELECT id, username, birthday, sex, address FROM `user`
DEBUG [main] - ==> Parameters: null
DEBUG [main] - <==      Total: 0
DEBUG [main] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4C
```

测试结果二很显然不合理。

按照之前所学的，要解决这个问题，需要编写多个 sql，查询条件越多，需要编写的 sql 就更多了，显然这样是不靠谱的。

解决方案，使用动态 sql 的 if 标签

### 3.1.5. 使用 if 标签

改造 UserMapper.xml，如下：

```
<!-- 根据条件查询用户 -->
<select id="queryUserByWhere" parameterType="user" resultType="user">
    SELECT id, username, birthday, sex, address FROM `user`
    WHERE 1=1
    <if test="sex != null and sex != ''">
        AND sex = #{sex}
    </if>
    <if test="username != null and username != ''">
        AND username LIKE
        '%${username}%'
    </if>
</select>
```

注意字符串类型的数据需要做不等于空字符串校验。

### 3.1.6. 效果

```
DEBUG [main] - ==> Preparing: SELECT id, username, birthday, sex, address FROM `user` WHERE 1=1 AND us
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==      Total: 3
User [id=10, username=张三, sex=1, birthday=Thu Jul 10 00:00:00 CST 2014, address=北京市]
User [id=16, username=张小明, sex=1, birthday=null, address=河南郑州]
User [id=24, username=张三丰, sex=1, birthday=null, address=河南郑州]
```

如上图所示，测试 OK

## 3.2. Where 标签

上面的 sql 还有 where 1=1 这样的语句，很麻烦

可以使用 where 标签进行改造

改造 UserMapper.xml，如下

```
<!-- 根据条件查询用户 -->
<select id="queryUserByWhere" parameterType="user" resultType="user">
    SELECT id, username, birthday, sex, address FROM `user`
    <!-- where 标签可以自动添加 where，同时处理 sql 语句中第一个 and 关键字 -->
    <where>
        <if test="sex != null">
            AND sex = #{sex}
        </if>
        <if test="username != null and username != ''">
            AND username LIKE
            '%${username}%'
        </if>
    </where>
</select>
```

### 3.2.1. 效果

测试效果如下图：

```
DEBUG [main] - ==> Preparing: SELECT id, username, birthday, sex, address FROM `user` WHERE sex = ? AND
DEBUG [main] - ==> Parameters: 1(String)
DEBUG [main] - <==      Total: 3
User [id=10, username=张三, sex=1, birthday=Thu Jul 10 00:00:00 CST 2014, address=北京市]
User [id=16, username=张小明, sex=1, birthday=null, address=河南郑州]
User [id=24, username=张三丰, sex=1, birthday=null, address=河南郑州]
```

### 3.3. Sql 片段

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的。

把上面例子中的 `id, username, birthday, sex, address` 提取出来，作为 sql 片段，如下：

```
<!-- 根据条件查询用户 -->
<select id="queryUserByWhere" parameterType="user" resultType="user">
  <!-- SELECT id, username, birthday, sex, address FROM `user` -->
  <!-- 使用 include 标签加载 sql 片段; refid 是 sql 片段 id -->
  SELECT <include refid="userFields" /> FROM `user`
  <!-- where 标签可以自动添加 where 关键字，同时处理 sql 语句中第一个 and 关键字 -->
  <where>
    <if test="sex != null">
      AND sex = #{sex}
    </if>
    <if test="username != null and username != ''">
      AND username LIKE
        '%${username}%'
    </if>
  </where>
</select>

<!-- 声明 sql 片段 -->
<sql id="userFields">
  id, username, birthday, sex, address
</sql>
```

如果要使用别的 Mapper.xml 配置的 sql 片段，可以在 refid 前面加上对应的 Mapper.xml 的 namespace

例如下图

```
<!-- 根据性别和用户名查询 -->
<select id="queryUserByWhere1" parameterType="user" resultType="user">
  SELECT <include refid="cn.itcast.mybatis.mapper.OrderMapper.userFields" /> FROM `user`
  <!-- where 标签有两个作用：1，添加 where 关键字；2，处理第一个 and 关键字 -->
  <where>
    <if test="sex != null and sex != ''">
      AND sex = #{sex}
    </if>
    <if test="username != null and username != ''">
      AND username LIKE '%${username}%'
    </if>
  </where>
</select>
```



## 3.4. foreach 标签

向 sql 传递数组或 List，mybatis 使用 foreach 解析，如下：

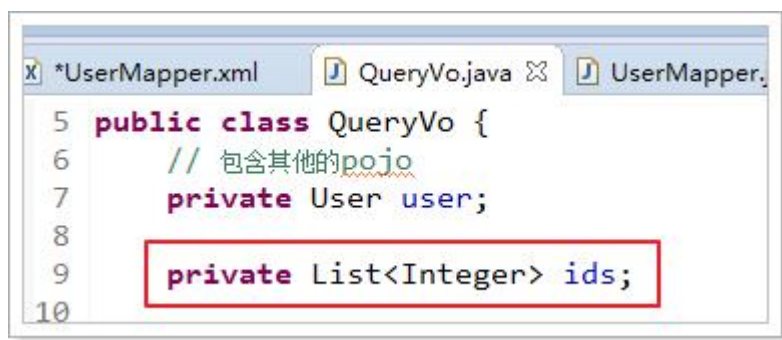
根据多个 id 查询用户信息

查询 sql:

```
SELECT * FROM user WHERE id IN (1,10,24)
```

### 3.4.1. 改造 QueryVo

如下图在 pojo 中定义 list 属性 ids 存储多个用户 id，并添加 getter/setter 方法



### 3.4.2. Mapper.xml 文件

UserMapper.xml 添加 sql，如下：

```
<!-- 根据 ids 查询用户 -->
<select id="queryUserByIds" parameterType="queryVo" resultType="user">
    SELECT * FROM `user`
    <where>
        <!-- foreach 标签，进行遍历 -->
        <!-- collection: 遍历的集合，这里是 QueryVo 的 ids 属性 -->
        <!-- item: 遍历的项目，可以随便写，但是和后面的#{ }里面要一致 -->
        <!-- open: 在前面添加的 sql 片段 -->
        <!-- close: 在结尾处添加的 sql 片段 -->
        <!-- separator: 指定遍历的元素之间使用的分隔符 -->
        <foreach collection="ids" item="item" open="id IN (" close=")"
            separator=", ">
            #{item}
        </foreach>
    </where>
</select>
```

测试方法如下图：

```
@Test
public void testQueryUserByIds() {
    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象，整合之后，交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 使用 userMapper 执行根据条件查询用户
    QueryVo queryVo = new QueryVo();
    List<Integer> ids = new ArrayList<>();
    ids.add(1);
    ids.add(10);
    ids.add(24);
    queryVo.setIds(ids);

    List<User> list = userMapper.queryUserByIds(queryVo);

    for (User u : list) {
        System.out.println(u);
    }

    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    sqlSession.close();
}
```

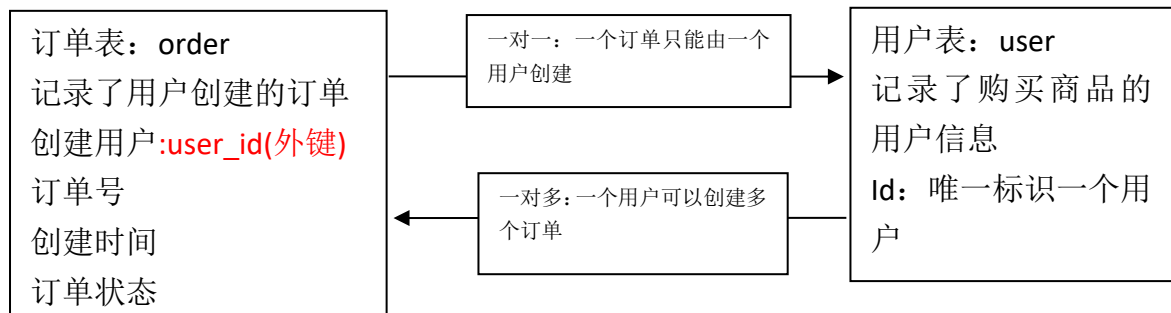
### 3.4.3. 效果

测试效果如下图：

```
DEBUG [main] - ==> Preparing: SELECT * FROM `user` WHERE id IN ( ? , ? , ? )
DEBUG [main] - ==> Parameters: 1(Integer), 10(Integer), 24(Integer)
DEBUG [main] - <==          Total: 3
User [id=1, username=李四, sex=2, birthday=null, address=null]
User [id=10, username=张三, sex=1, birthday=Thu Jul 10 00:00:00 CST 2014, address=北京市]
User [id=24, username=张三丰, sex=1, birthday=null, address=河南郑州]
```

## 4. 关联查询

### 4.1. 商品订单数据模型



### 4.2. 一对一查询

需求：查询所有订单信息，关联查询下单用户信息。

注意：因为一个订单信息只会是一个人下的订单，所以从查询订单信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

sql 语句：

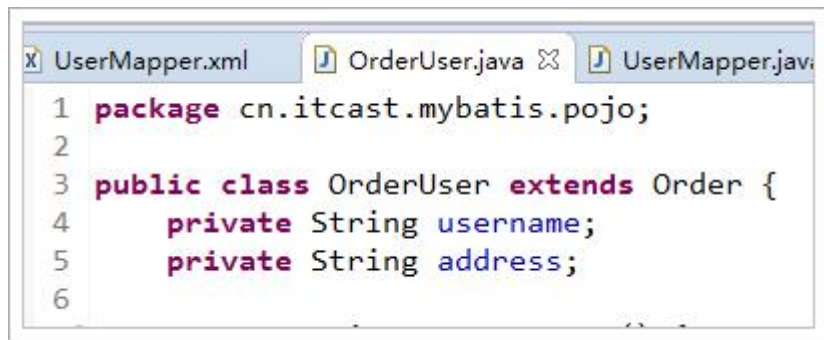
```
SELECT
    o.id,
    o.user_id userId,
    o.number,
    o.createtime,
    o.note,
    u.username,
    u.address
FROM
    `order` o
LEFT JOIN `user` u ON o.user_id = u.id
```

#### 4.2.1. 方法一：使用 resultType

使用 resultType，改造订单 pojo 类，此 pojo 类中包括了订单信息和用户信息这样返回对象的时候，mybatis 自动把用户信息也注入进来了

#### 4.2.1.1. 改造 pojo 类

OrderUser 类继承 Order 类后 OrderUser 类包括了 Order 类的所有字段，只需要定义用户的信息字段即可，如下图：



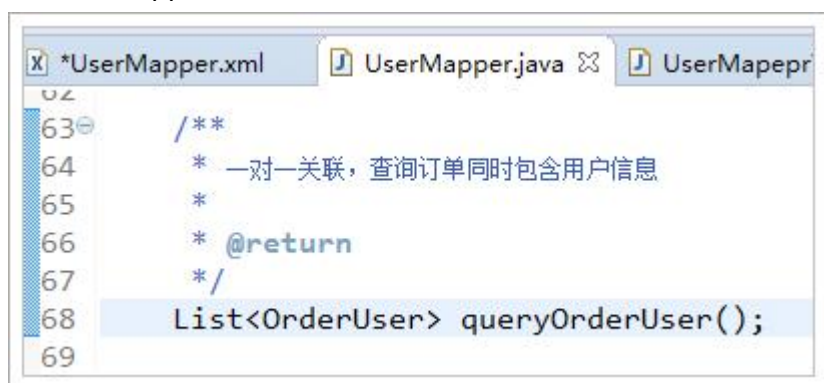
#### 4.2.1.2. Mapper.xml

在 UserMapper.xml 添加 sql，如下

```
<!-- 查询订单，同时包含用户数据 -->
<select id="queryOrderUser" resultType="orderUser">
    SELECT
    o.id,
    o.user_id
    userId,
    o.number,
    o.createtime,
    o.note,
    u.username,
    u.address
    FROM
    `order` o
    LEFT JOIN `user` u ON o.user_id = u.id
</select>
```

#### 4.2.1.3. Mapper 接口

在 UserMapper 接口添加方法，如下图：



#### 4.2.1.4. 测试方法:

在 UserMapperTest 添加测试方法, 如下:

```
@Test
public void testQueryOrderUser() {
    // mybatis 和 spring 整合, 整合之后, 交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象, 整合之后, 交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

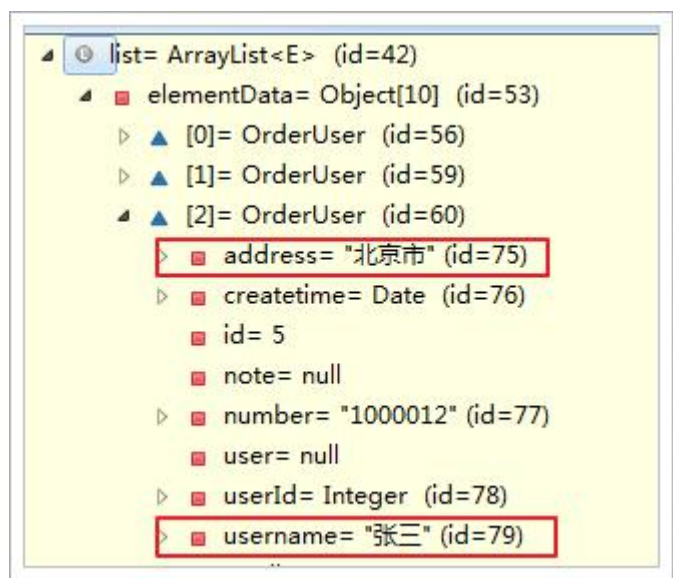
    // 使用 userMapper 执行根据条件查询用户
    List<OrderUser> list = userMapper.queryOrderUser();

    for (OrderUser ou : list) {
        System.out.println(ou);
    }

    // mybatis 和 spring 整合, 整合之后, 交给 spring 管理
    sqlSession.close();
}
```

#### 4.2.1.5. 效果

测试结果如下图:



#### 4.2.1.6. 小结

定义专门的 pojo 类作为输出类型, 其中定义了 sql 查询结果集所有的字段。此方法较为简单, 企业中使用普遍。

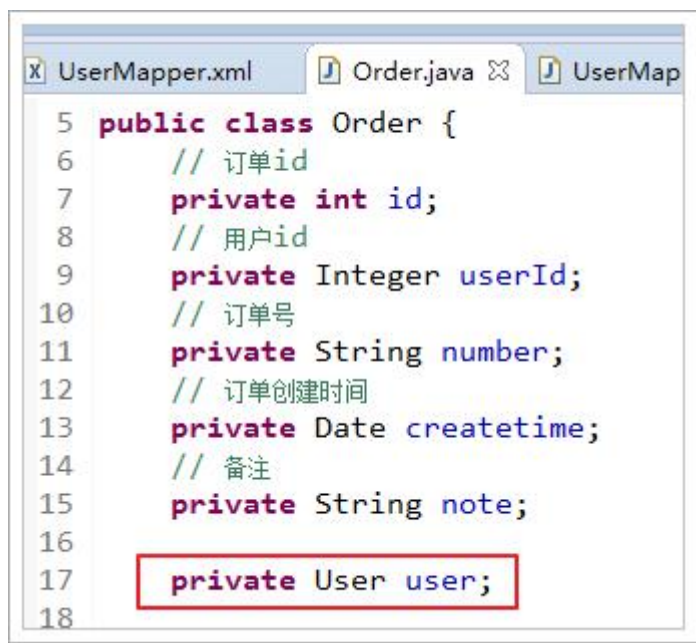
## 4.2.2. 方法二：使用 resultMap

使用 resultMap，定义专门的 resultMap 用于映射一对一查询结果。

### 4.2.2.1. 改造 pojo 类

在 Order 类中加入 User 属性，user 属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个 User 对象存储关联查询的用户信息。

改造 Order 如下图：



### 4.2.2.2. Mapper.xml

这里 resultMap 指定 orderUserResultMap，如下：

```
<resultMap type="order" id="orderUserResultMap">
    <id property="id" column="id" />
    <result property="userId" column="user_id" />
    <result property="number" column="number" />
    <result property="createtime" column="createtime" />
    <result property="note" column="note" />

    <!-- association : 配置一对一属性 -->
    <!-- property:order 里面的 User 属性名 -->
    <!-- javaType:属性类型 -->
    <association property="user" javaType="user">
        <!-- id:声明主键，表示 user_id 是关联查询对象的唯一标识-->
        <id property="id" column="user_id" />
    </association>
</resultMap>
```

```

        <result property="username" column="username" />
        <result property="address" column="address" />
    </association>

</resultMap>

<!-- 一对一关联，查询订单，订单内部包含用户属性 -->
<select id="queryOrderUserResultMap" resultMap="orderUserResultMap">
    SELECT
    o.id,
    o.user_id,
    o.number,
    o.createtime,
    o.note,
    u.username,
    u.address
    FROM
    `order` o
    LEFT JOIN `user` u ON o.user_id = u.id
</select>

```

#### 4.2.2.3. Mapper 接口

编写 UserMapper 如下图：



#### 4.2.2.4. 测试方法

在 UserMapperTest 增加测试方法，如下：

```

@Test
public void testQueryOrderUserResultMap() {
    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象，整合之后，交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
}

```

```

// 使用 userMapper 执行根据条件查询用户
List<Order> list = userMapper.queryOrderUserResultMap();

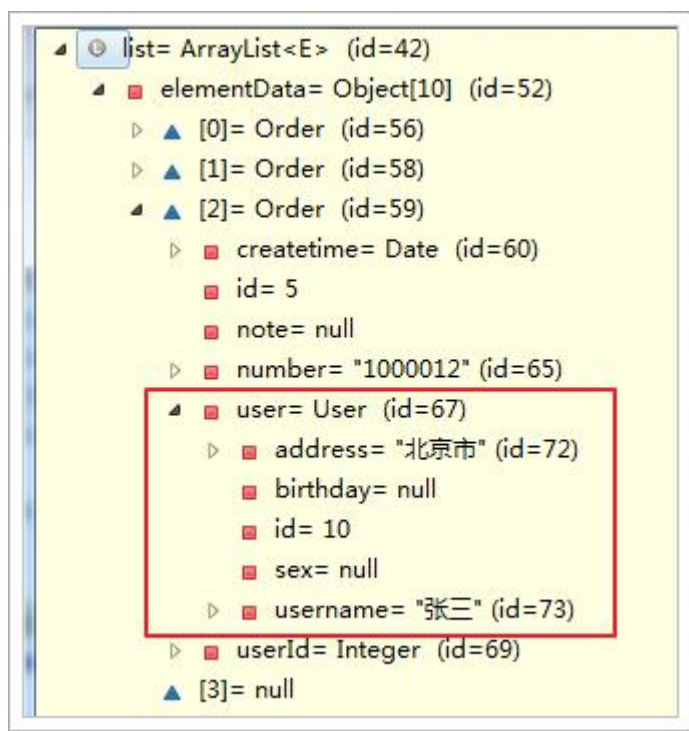
for (Order o : list) {
    System.out.println(o);
}

// mybatis 和 spring 整合，整合之后，交给 spring 管理
sqlSession.close();
}

```

#### 4.2.2.5. 效果

测试效果如下图：



### 4.3. 一对多查询

案例：查询所有用户信息及用户关联的订单信息。  
用户信息和订单信息为一对多关系。

sql 语句：

```

SELECT
    u.id,
    u.username,
    u.birthday,

```



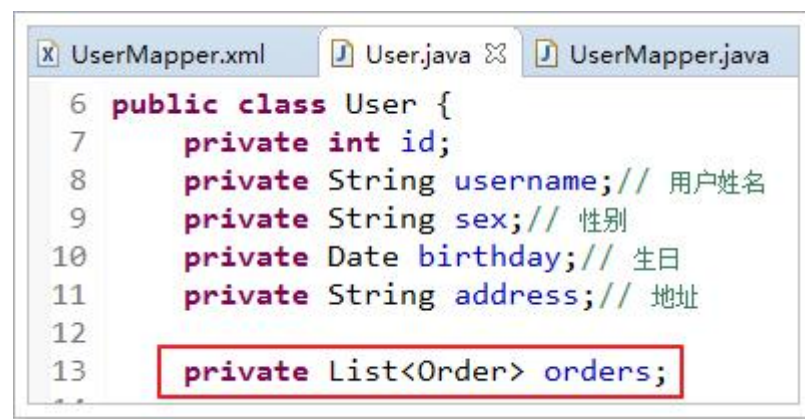
```

    u.sex,
    u.address,
    o.id oid,
    o.number,
    o.createtime,
    o.note
FROM
    `user` u
LEFT JOIN `order` o ON u.id = o.user_id

```

### 4.3.1. 修改 pojo 类

在 User 类中加入 List<Order> orders 属性,如下图:



### 4.3.2. Mapper.xml

在 UserMapper.xml 添加 sql, 如下:

```

<resultMap type="user" id="userOrderResultMap">
    <id property="id" column="id" />
    <result property="username" column="username" />
    <result property="birthday" column="birthday" />
    <result property="sex" column="sex" />
    <result property="address" column="address" />

    <!-- 配置一对多的关系 -->
    <collection property="orders" javaType="list" ofType="order">
        <!-- 配置主键, 是关联 Order 的唯一标识 -->
        <id property="id" column="oid" />
        <result property="number" column="number" />
        <result property="createtime" column="createtime" />
        <result property="note" column="note" />
    
```

```

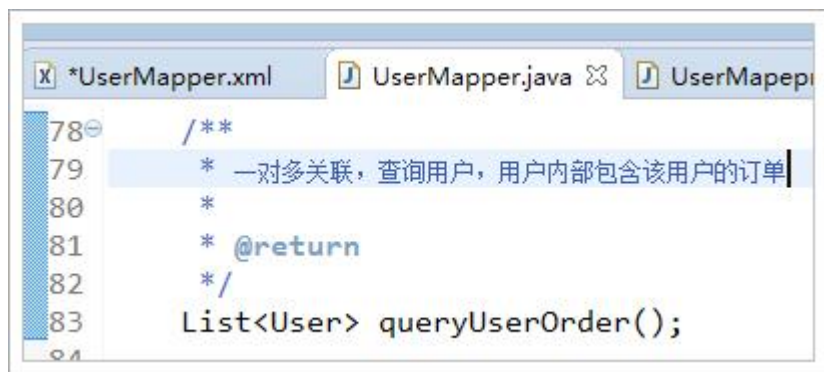
    </collection>
</resultMap>

<!-- 一对多关联，查询订单同时查询该用户下的订单 -->
<select id="queryUserOrder" resultMap="userOrderResultMap">
    SELECT
    u.id,
    u.username,
    u.birthday,
    u.sex,
    u.address,
    o.id oid,
    o.number,
    o.createtime,
    o.note
    FROM
    `user` u
    LEFT JOIN `order` o ON u.id = o.user_id
</select>

```

### 4.3.3. Mapper 接口

编写 UserMapper 接口，如下图：



### 4.3.4. 测试方法

在 UserMapperTest 增加测试方法，如下

```

@Test
public void testQueryUserOrder() {
    // mybatis 和 spring 整合，整合之后，交给 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 创建 Mapper 接口的动态代理对象，整合之后，交给 spring 管理
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
}

```

```

// 使用 userMapper 执行根据条件查询用户
List<User> list = userMapper.queryUserOrder();

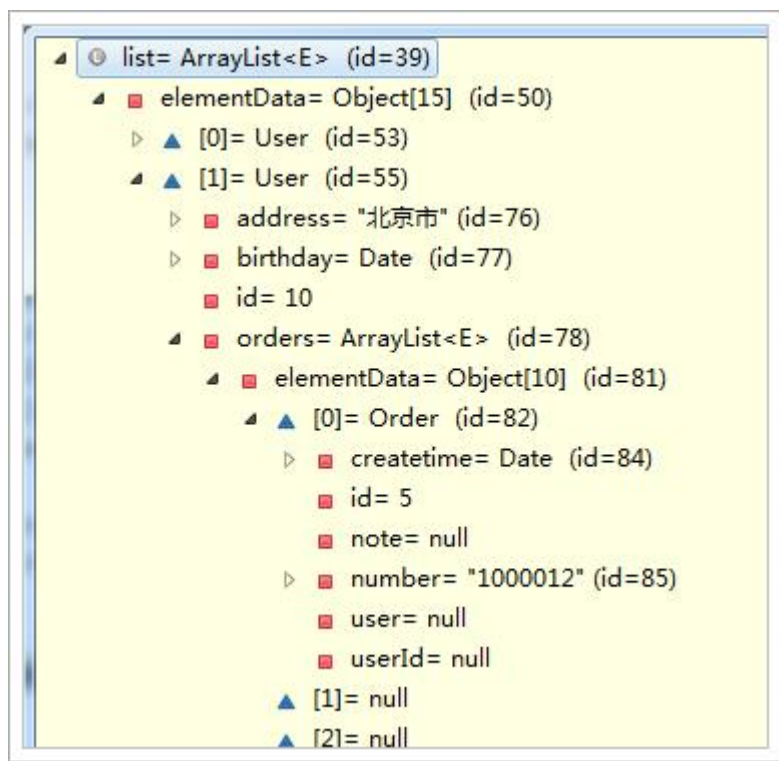
for (User u : list) {
    System.out.println(u);
}

// mybatis 和 spring 整合，整合之后，交给 spring 管理
sqlSession.close();
}

```

### 4.3.5. 效果

测试效果如下图：



## 5. Mybatis 整合 spring



















### 5.1. 整合思路

- 1、SqlSessionFactory 对象应该放到 spring 容器中作为单例存在。
- 2、传统 dao 的开发方式中，应该从 spring 容器中获得 sqlSession 对象。
- 3、Mapper 代理形式中，应该从 spring 容器中直接获得 mapper 的代理对象。
- 4、数据库的连接以及数据库连接池事务管理都交给 spring 容器来完成。

### 5.2. 整合需要的 jar 包

- 1、spring 的 jar 包
- 2、Mybatis 的 jar 包
- 3、Spring+mybatis 的整合包。
- 4、Mysql 的数据库驱动 jar 包。
- 5、数据库连接池的 jar 包。

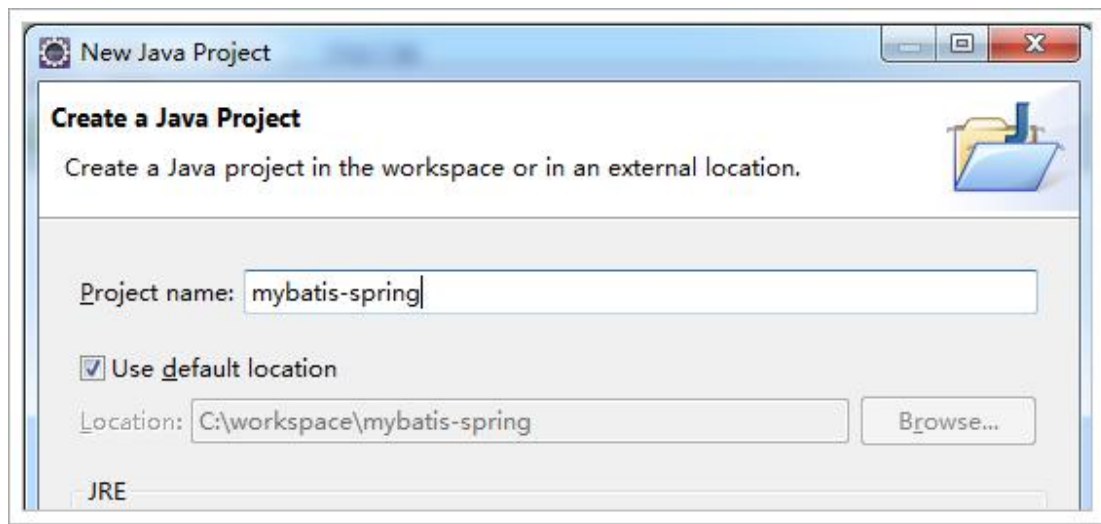
jar 包位置如下所示：

01Mybatis ▶ 01课前资料 ▶ jar包 ▶ mybatis与spring整合全部jar包(包括springmvc)		
新建文件夹		
名称	修改日期	类型
 mybatis-3.2.7.jar	2016/5/13 23:34	JAR 文件
 mybatis-spring-1.2.2.jar	2016/5/13 23:34	JAR 文件
 mysql-connector-java-5.1.7-bin.jar	2016/5/13 23:34	JAR 文件
 slf4j-api-1.7.5.jar	2016/5/13 23:34	JAR 文件
 slf4j-log4j12-1.7.5.jar	2016/5/13 23:34	JAR 文件
 spring-aop-4.1.3.RELEASE.jar	2016/5/13 23:34	JAR 文件
 spring-aspects-4.1.3.RELEASE.jar	2016/5/13 23:34	JAR 文件
 spring-beans-4.1.3.RELEASE.jar	2016/5/13 23:34	JAR 文件
 spring-context-4.1.3.RELEASE.jar	2016/5/13 23:34	JAR 文件
 spring-context-support-4.1.3.RELEAS...	2016/5/13 23:33	JAR 文件
 spring-core-4.1.3.RELEASE.jar	2016/5/13 23:34	JAR 文件
 spring-expression-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-jdbc-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-jms-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-messaging-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-tx-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-web-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件
 spring-webmvc-4.1.3.RELEASE.jar	2016/5/13 23:33	JAR 文件

## 5.3. 整合的步骤

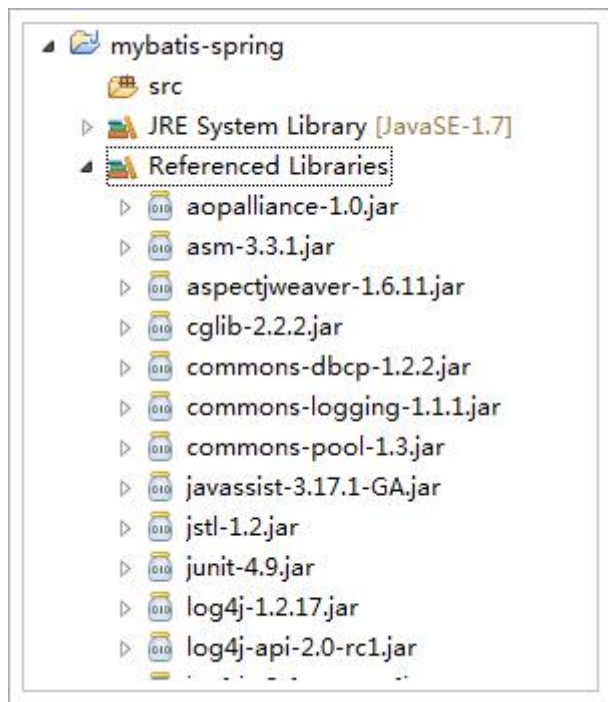
### 5.3.1. 创建工程

如下图创建一个 java 工程：



### 5.3.2. 导入 jar 包

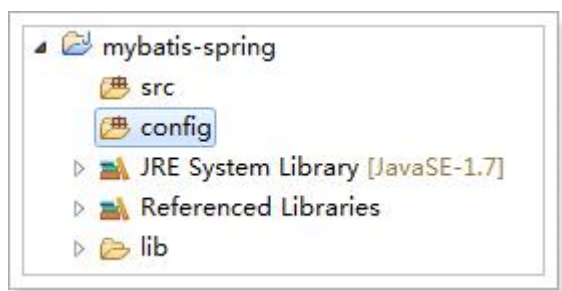
前面提到的 jar 包需要导入，如下图：



### 5.3.3. 加入配置文件

1. mybatisSpring 的配置文件
2. 的配置文件 sqlmapConfig.xml
  - a) 数据库连接及连接池
  - b) 事务管理（暂时可以不配置）
  - c) sqlSessionSessionFactory 对象，配置到 spring 容器中
  - d) mapeer 代理对象或者是 dao 实现类配置到 spring 容器中。

创建资源文件夹 config 拷贝加入配置文件，如下图



#### 5.3.3.1. SqlMapConfig.xml

配置文件是 SqlMapConfig.xml，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <!-- 设置别名 -->
  <typeAliases>
    <!-- 2. 指定扫描包，会把包内所有的类都设置别名，别名的名称就是类名，大小写不敏感 -->
    <package name="com.cloudtcc.mybatis.pojo" />
  </typeAliases>
</configuration>
```

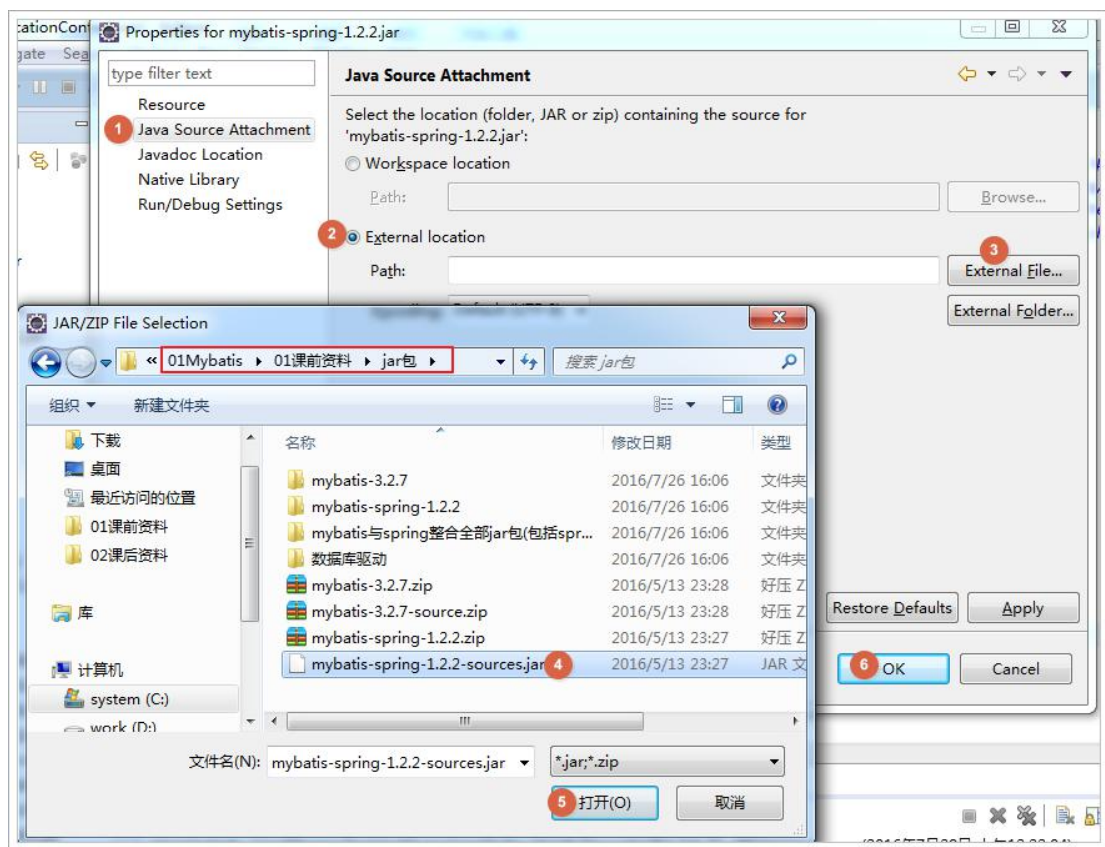
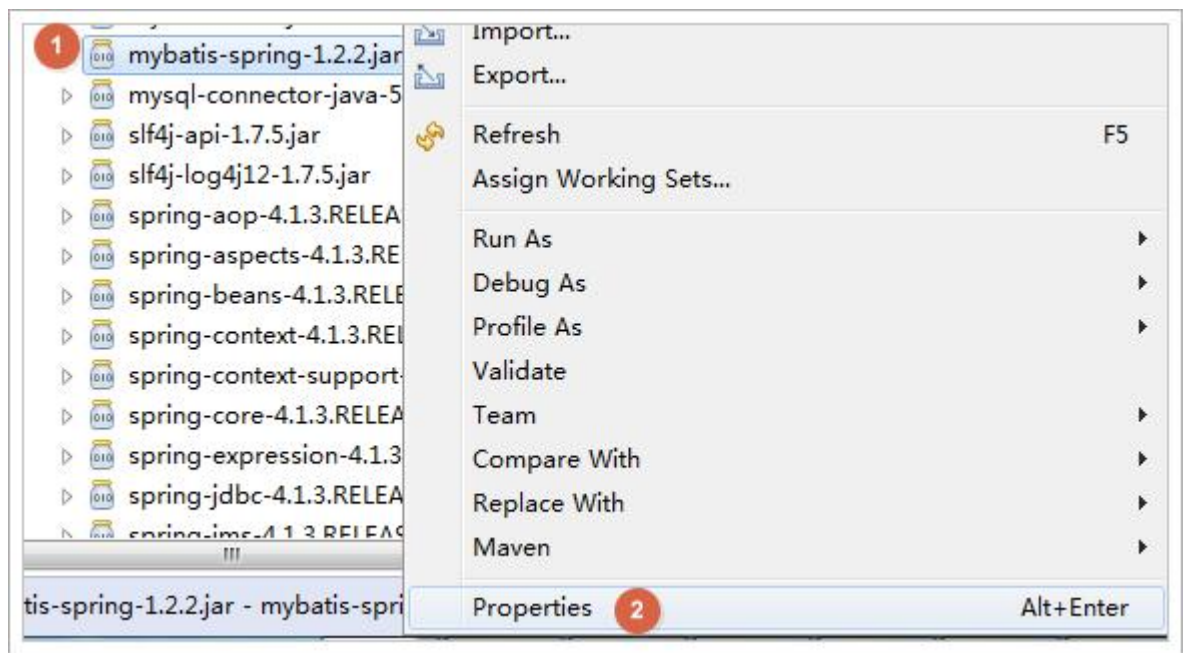
#### 5.3.3.2. applicationContext.xml

SqlSessionFactoryBean 属于 mybatis-spring 这个 jar 包

对于 spring 来说，mybatis 是另外一个架构，需要整合 jar 包。



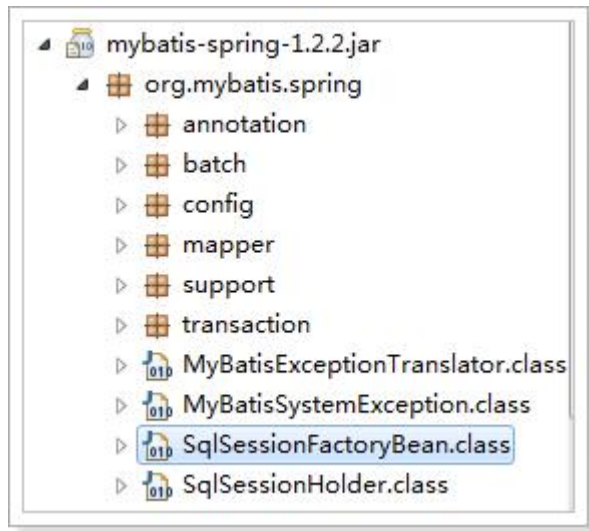
在项目中加入 mybatis-spring-1.2.2.jar 的源码，如下图



效果，如下图所示，图标变化，表示源码加载成功：



整合 Mybatis 需要的是 SqlSessionFactoryBean，位置如下图：



applicationContext.xml，配置内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <!-- 加载配置文件 -->
    <context:property-placeholder location="classpath:db.properties" />

    <!-- 数据库连接池 -->
    <bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="${jdbc.driver}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
```



```

        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="10" />
        <property name="maxIdle" value="5" />
    </bean>

    <!-- 配置 SqlSessionFactory -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 配置 mybatis 核心配置文件 -->
        <property name="configLocation"
value="classpath:SqlMapConfig.xml" />
        <!-- 配置数据源 -->
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

### 5.3.3.3. db.properties

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8
jdbc.username=root
jdbc.password=root

```

### 5.3.3.4. log4j.properties

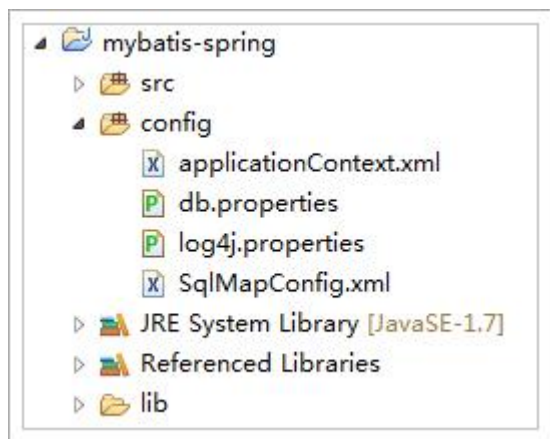
```

# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

```

### 5.3.3.5. 效果:

加入的配置文件最终效果如下:



## 5.4. Dao 的开发

两种 dao 的实现方式:

- 1、原始 dao 的开发方式
- 2、使用 Mapper 代理形式开发方式
  - a) 直接配置 Mapper 代理
  - b) 使用扫描包配置 Mapper 代理

需求:

1. 实现根据用户 id 查询
2. 实现根据用户名模糊查询
3. 添加用户

### 5.4.1. 创建 pojo

```
public class User {  
    private int id;  
    private String username; // 用户姓名  
    private String sex; // 性别  
    private Date birthday; // 生日  
    private String address; // 地址  
  
    get/set...  
}
```

### 5.4.2. 传统 dao 的开发方式

原始的 DAO 开发接口+实现类来完成。

需要 dao 实现类需要继承 SqlSessionDaoSupport 类

#### 5.4.2.1. 实现 Mapper.xml

编写 User.xml 配置文件, 如下:

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="test">  
    <!-- 根据用户 id 查询 -->
```

```

<select id="queryUserById" parameterType="int" resultType="user">
    select * from user where id = #{id}
</select>

<!-- 根据用户名模糊查询用户 -->
<select id="queryUserByUsername" parameterType="string"
    resultType="user">
    select * from user where username like '%${value}%'
</select>

<!-- 添加用户 -->
<insert id="saveUser" parameterType="user">
    <selectKey keyProperty="id" keyColumn="id" order="AFTER"
        resultType="int">
        select last_insert_id()
    </selectKey>
    insert into user
    (username,birthday,sex,address)
    values
    (#{username},#{birthday},#{sex},#{address})
</insert>

</mapper>

```

#### 5.4.2.2. 加载 Mapper.xml

在 SqlMapConfig 如下图进行配置:



#### 5.4.2.3. 实现 UserDao 接口

```

public interface UserDao {
    /**
     * 根据 id 查询用户
     *
     * @param id
     * @return

```

```

    */
    User queryUserById(int id);

    /**
     * 根据用户名模糊查询用户列表
     *
     * @param username
     * @return
     */
    List<User> queryUserByUsername(String username);

    /**
     * 保存
     *
     * @param user
     */
    void saveUser(User user);
}

```

#### 5.4.2.4. 实现 UserDaoImpl 实现类

编写 DAO 实现类，实现类必须集成 SqlSessionDaoSupport

SqlSessionDaoSupport 提供 getSession() 方法来获取 SqlSession

```

public class UserDaoImpl extends SqlSessionDaoSupport implements
UserDao {
    @Override
    public User queryUserById(int id) {
        // 获取 SqlSession
        SqlSession sqlSession = super.getSession();

        // 使用 SqlSession 执行操作
        User user = sqlSession.selectOne("queryUserById", id);

        // 不要关闭 sqlSession

        return user;
    }
}

```

```

@Override
public List<User> queryUserByUsername(String username) {
    // 获取 SqlSession
    SqlSession sqlSession = super.getSession();
}

```

```

        // 使用 SqlSession 执行操作
        List<User> list = sqlSession.selectList("queryUserByUsername",
username);

        // 不要关闭 sqlSession

        return list;
    }

    @Override
    public void saveUser(User user) {
        // 获取 SqlSession
        SqlSession sqlSession = super.getSqlSession();

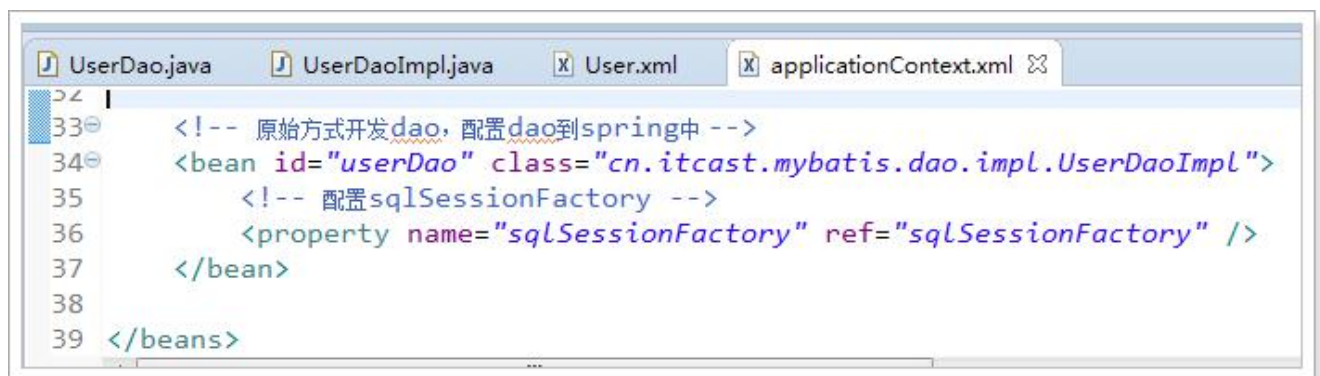
        // 使用 SqlSession 执行操作
        sqlSession.insert("saveUser", user);

        // 不用提交,事务由 spring 进行管理
        // 不要关闭 sqlSession
    }
}

```

#### 5.4.2.5. 配置 dao

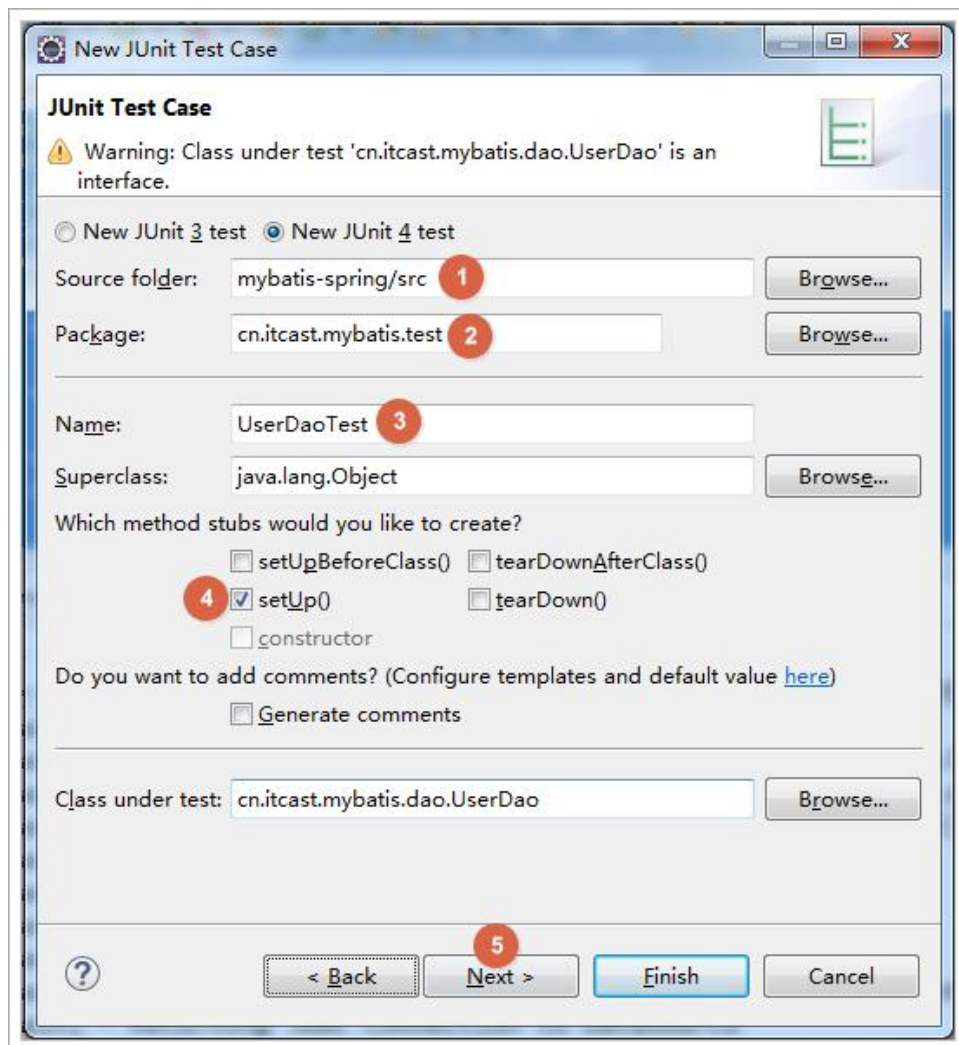
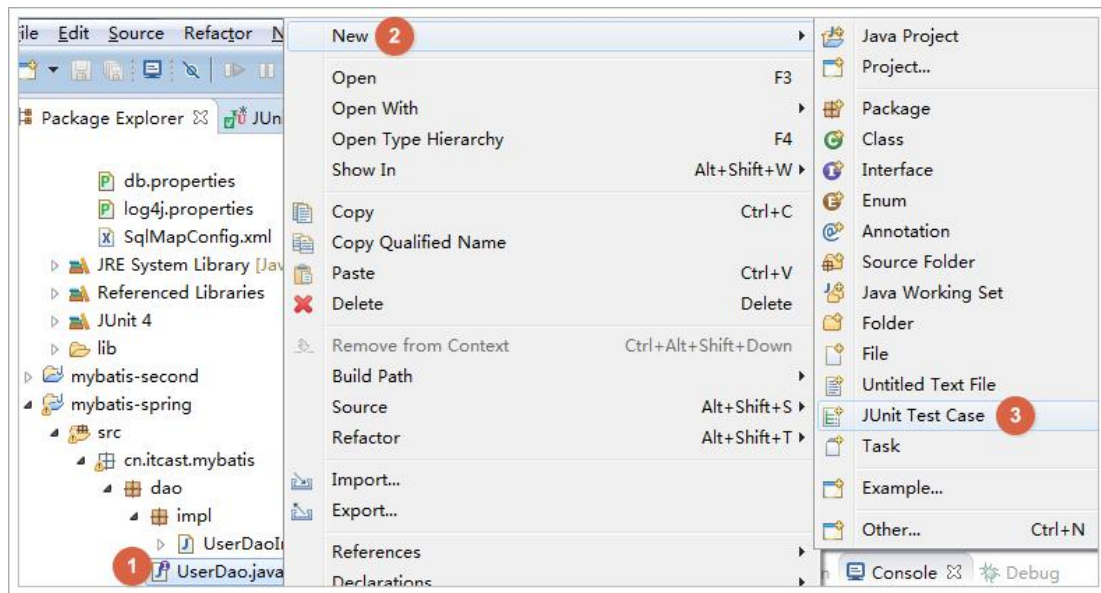
把 dao 实现类配置到 spring 容器中，如下图

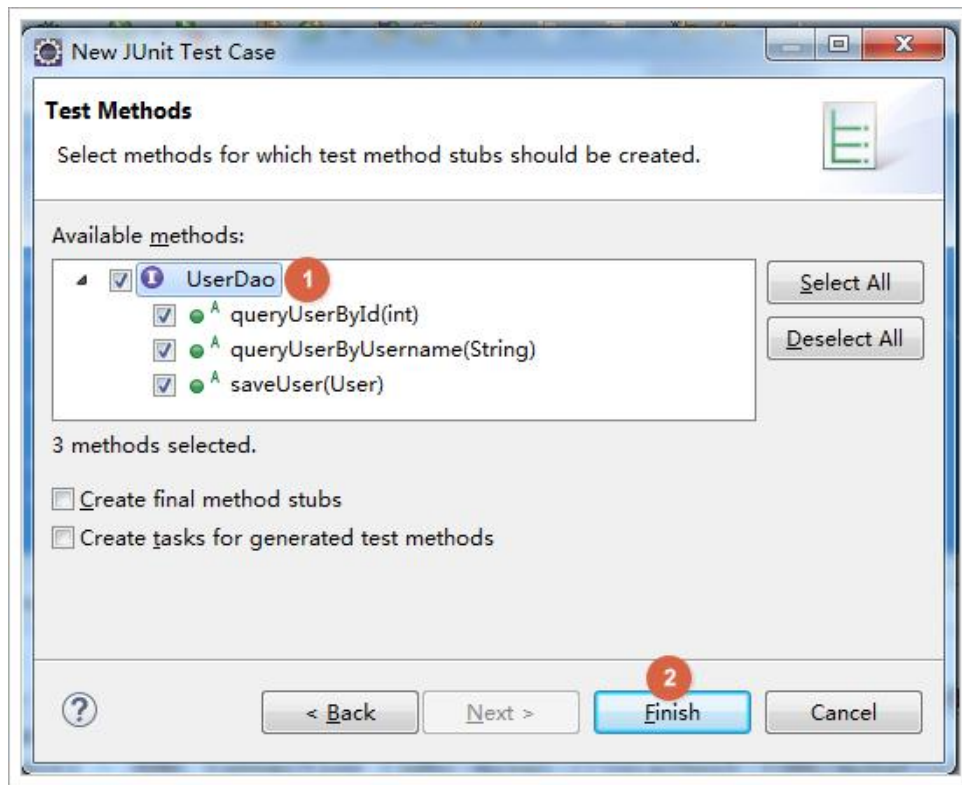


#### 5.4.2.6. 测试方法

创建测试方法，可以直接创建测试 Junit 用例。

如下图所示进行创建。





编写测试方法如下：

```
public class UserDaoTest {
    private ApplicationContext context;

    @Before
    public void setUp() throws Exception {
        this.context = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
    }

    @Test
    public void testQueryUserById() {
        // 获取 userDao
        UserDao userDao = this.context.getBean(UserDao.class);

        User user = userDao.queryUserById(1);
        System.out.println(user);
    }

    @Test
    public void testQueryUserByUsername() {
        // 获取 userDao
        UserDao userDao = this.context.getBean(UserDao.class);
```

```

        List<User> list = userDao.queryUserByUsername("张");
        for (User user : list) {
            System.out.println(user);
        }
    }

    @Test
    public void testSaveUser() {
        // 获取 userDao
        UserDao userDao = this.context.getBean(UserDao.class);

        User user = new User();
        user.setUsername("曹操");
        user.setSex("1");
        user.setBirthday(new Date());
        user.setAddress("三国");
        userDao.saveUser(user);
        System.out.println(user);
    }
}

```

### 5.4.3. Mapper 代理形式开发 dao

#### 5.4.3.1. 实现 Mapper.xml

编写 UserMapper.xml 配置文件，如下：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.cloudtcc.mybatis.mapper.UserMapper">
    <!-- 根据用户 id 查询 -->
    <select id="queryUserById" parameterType="int" resultType="user">
        select * from user where id = #{id}
    </select>

    <!-- 根据用户名模糊查询用户 -->
    <select id="queryUserByUsername" parameterType="string"
        resultType="user">
        select * from user where username like '%${value}%'
    </select>

```



```

<!-- 添加用户 -->
<insert id="saveUser" parameterType="user">
    <selectKey keyProperty="id" keyColumn="id" order="AFTER"
        resultType="int">
        select last_insert_id()
    </selectKey>
    insert into user
    (username,birthday,sex,address) values
    (#{username},#{birthday},#{sex},#{address})
</insert>
</mapper>

```

### 5.4.3.2. 实现 UserMapper 接口

```

public interface UserMapper {
    /**
     * 根据用户 id 查询
     *
     * @param id
     * @return
     */
    User queryUserById(int id);

    /**
     * 根据用户名模糊查询用户
     *
     * @param username
     * @return
     */
    List<User> queryUserByUsername(String username);

    /**
     * 添加用户
     *
     * @param user
     */
    void saveUser(User user);
}

```

### 5.4.3.3. 方式一：配置 mapper 代理

在 applicationContext.xml 添加配置

MapperFactoryBean 也是属于 mybatis-spring 整合包

```
<!-- Mapper 代理的方式开发方式一，配置 Mapper 代理对象 -->
<bean id="userMapper"
class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 配置 Mapper 接口 -->
    <property name="mapperInterface"
value="com.cloudtcc.mybatis.mapper.UserMapper" />
    <!-- 配置 sqlSessionSessionFactory -->
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

### 5.4.3.4. 测试方法

```
public class UserMapperTest {
    private ApplicationContext context;

    @Before
    public void setUp() throws Exception {
        this.context = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
    }

    @Test
    public void testQueryUserById() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        User user = userMapper.queryUserById(1);
        System.out.println(user);
    }

    @Test
    public void testQueryUserByUsername() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        List<User> list = userMapper.queryUserByUsername("张");

        for (User user : list) {
            System.out.println(user);
        }
    }
}
```

```

    }
    @Test
    public void testSaveUser() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        User user = new User();
        user.setUsername("曹操");
        user.setSex("1");
        user.setBirthday(new Date());
        user.setAddress("三国");

        userMapper.saveUser(user);
        System.out.println(user);
    }
}

```

#### 5.4.3.5. 方式二：扫描包形式配置 mapper

```

<!-- Mapper 代理的方式开发方式二，扫描包方式配置代理 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 配置 Mapper 接口 -->
    <property name="basePackage" value="com.cloudtcc.mybatis.mapper"
/>
</bean>

```

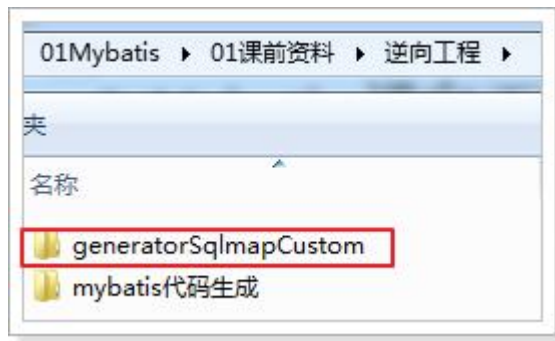
每个 mapper 代理对象的 id 就是类名，首字母小写

## 6. Mybatis 逆向工程

使用官方网站的 Mapper 自动生成工具 mybatis-generator-core-1.3.2 来生成 po 类和 Mapper 映射文件

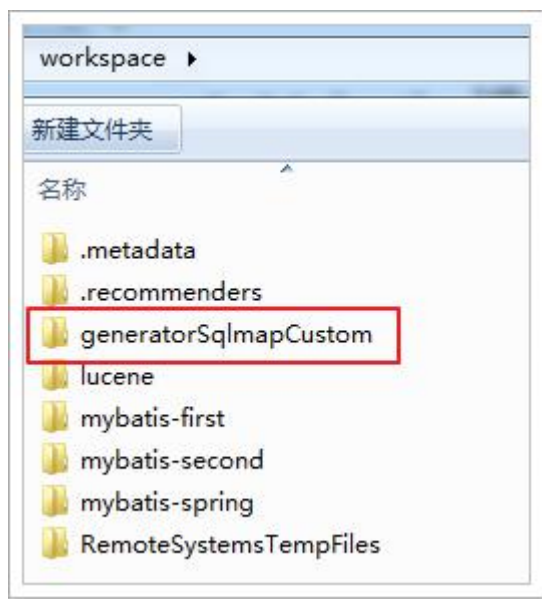
### 6.1. 导入逆向工程

使用课前资料已有逆向工程，如下图：



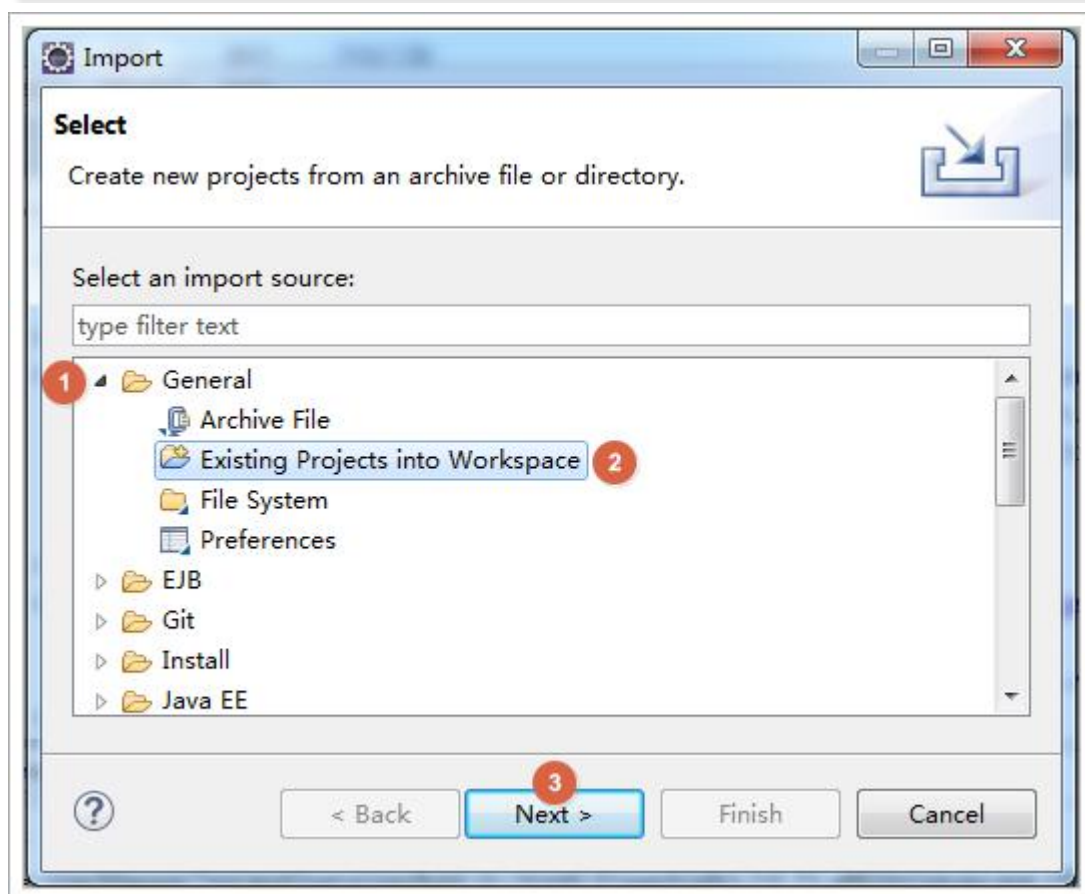
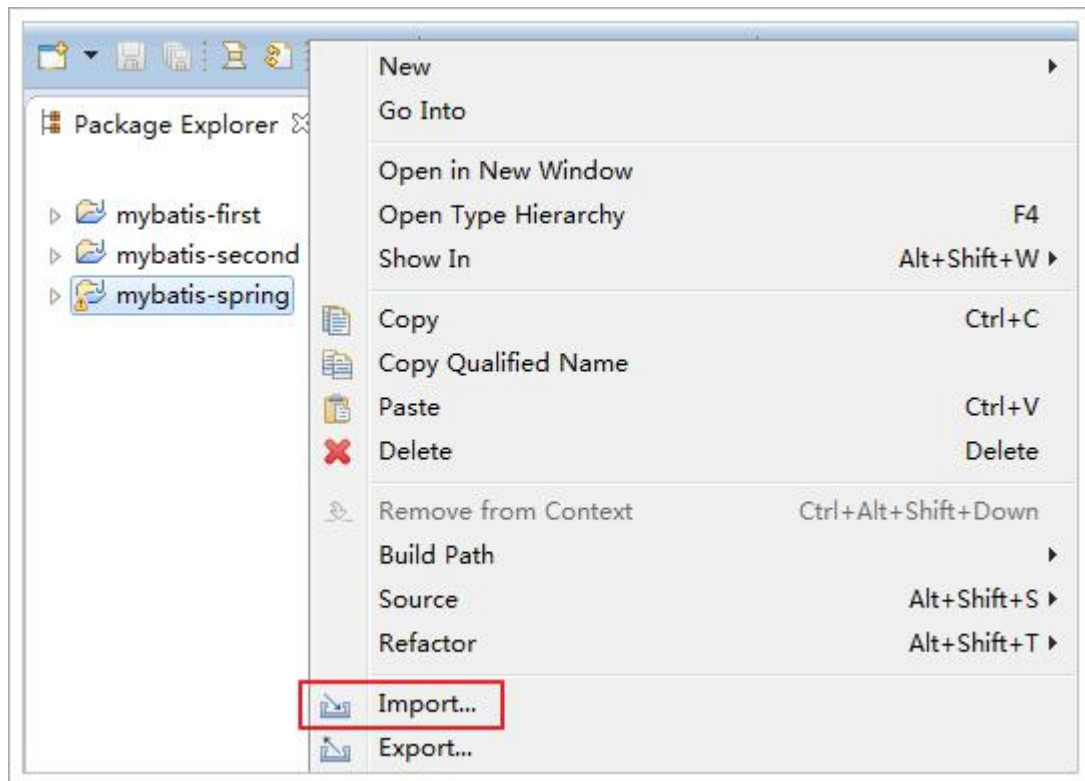
### 6.1.1. 复制逆向工程到工作空间中

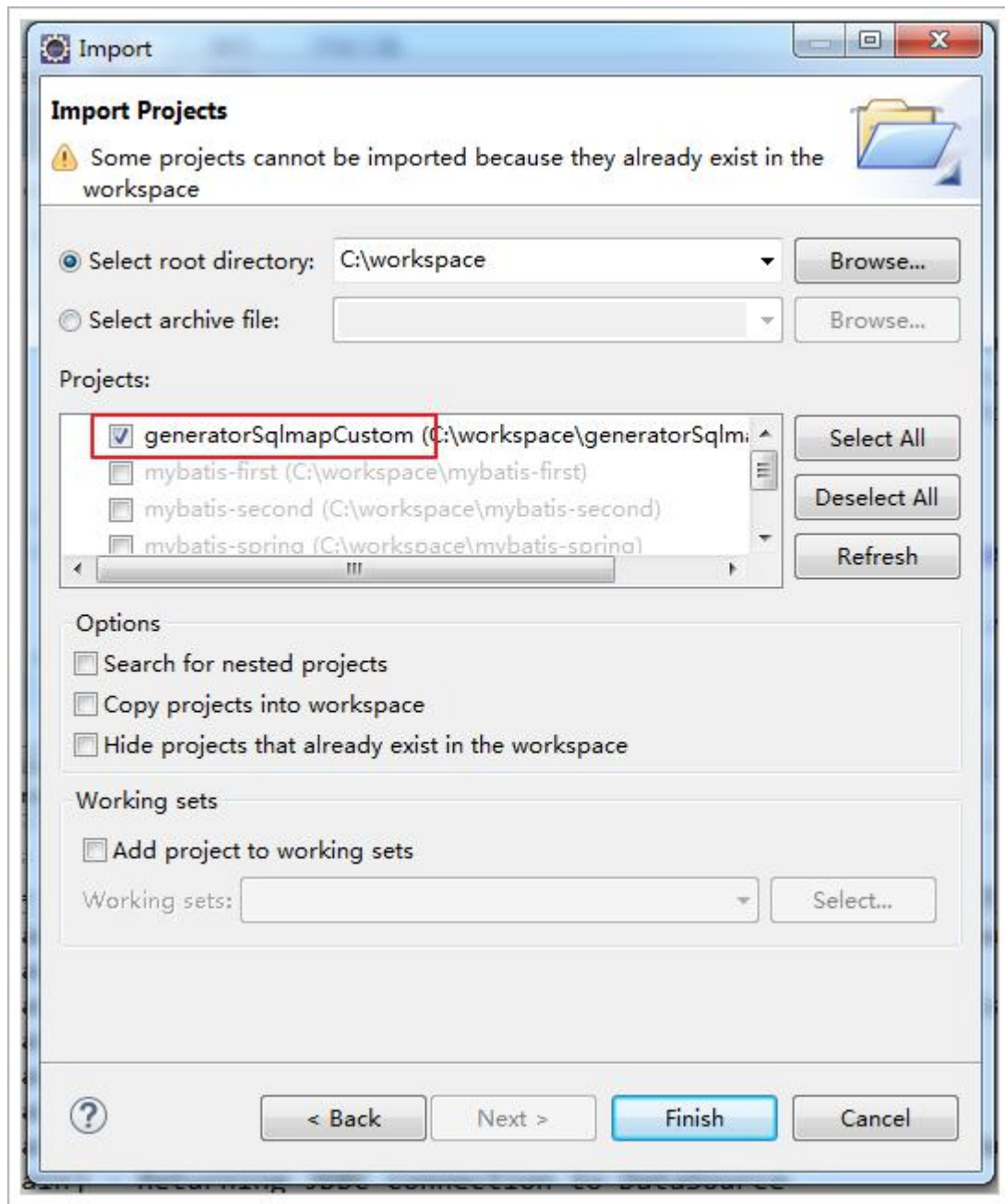
复制的效果如下图：



### 6.1.2. 导入逆向工程到 eclipse 中

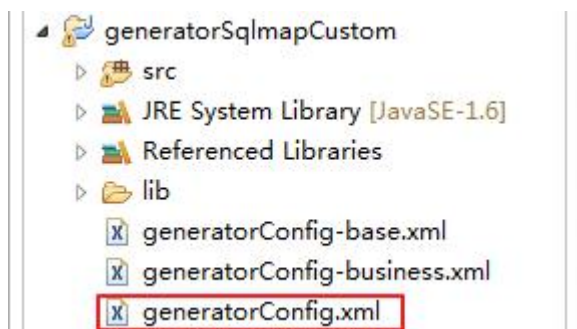
如下图方式进行导入：





## 6.2. 修改配置文件

在 generatorConfig.xml 中配置 Mapper 生成的详细信息，如下图：



注意修改以下几点:

1. 修改要生成的数据库表
2. pojo 文件所在包路径
3. Mapper 所在的包路径

配置文件如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
  <context id="testTables" targetRuntime="MyBatis3">
    <commentGenerator>
      <!-- 是否去除自动生成的注释 true: 是 : false:否 -->
      <property name="suppressAllComments" value="true" />
    </commentGenerator>
    <!--数据库连接的信息: 驱动类、连接地址、用户名、密码 -->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"
      connectionURL="jdbc:mysql://localhost:3306/mybatis"
      userId="root" password="root">
    </jdbcConnection>
    <!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"
      connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"
      userId="yycg" password="yycg"> </jdbcConnection> -->

    <!-- 默认false, 把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer, 为
    true时把JDBC DECIMAL
      和 NUMERIC 类型解析为java.math.BigDecimal -->
    <javaTypeResolver>
      <property name="forceBigDecimals" value="false" />
    </javaTypeResolver>

    <!-- targetProject:生成PO类的位置 -->
    <javaModelGenerator targetPackage="com.cloudtcc.ssm.po"
      targetProject=".\\src">
      <!-- enableSubPackages:是否让schema作为包的后缀 -->
      <property name="enableSubPackages" value="false" />
      <!-- 从数据库返回的值被清理前后的空格 -->
      <property name="trimStrings" value="true" />
    </javaModelGenerator>
    <!-- targetProject:mapper映射文件生成的位置 -->
    <sqlMapGenerator targetPackage="com.cloudtcc.ssm.mapper"
```

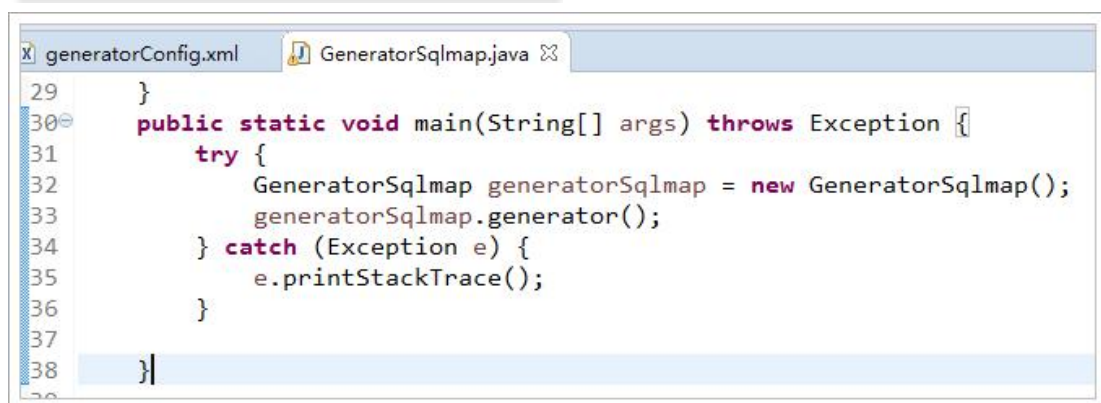
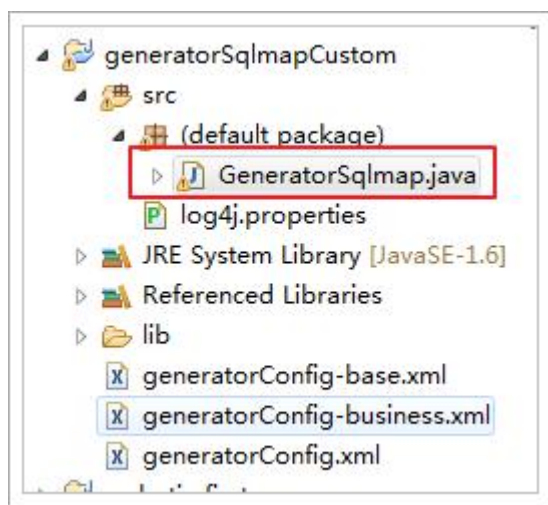
```

        targetProject=".\\src">
        <!-- enableSubPackages:是否让schema作为包的后缀 -->
        <property name="enableSubPackages" value="false" />
    </sqlMapGenerator>
    <!-- targetPackage: mapper接口生成的位置 -->
    <javaClientGenerator type="XMLMAPPER"
        targetPackage="com.cloudtcc.ssm.mapper"
targetProject=".\\src">
        <!-- enableSubPackages:是否让schema作为包的后缀 -->
        <property name="enableSubPackages" value="false" />
    </javaClientGenerator>
    <!-- 指定数据库表 -->
    <table schema="" tableName="user"></table>
    <table schema="" tableName="order"></table>
    </context>
</generatorConfiguration>

```

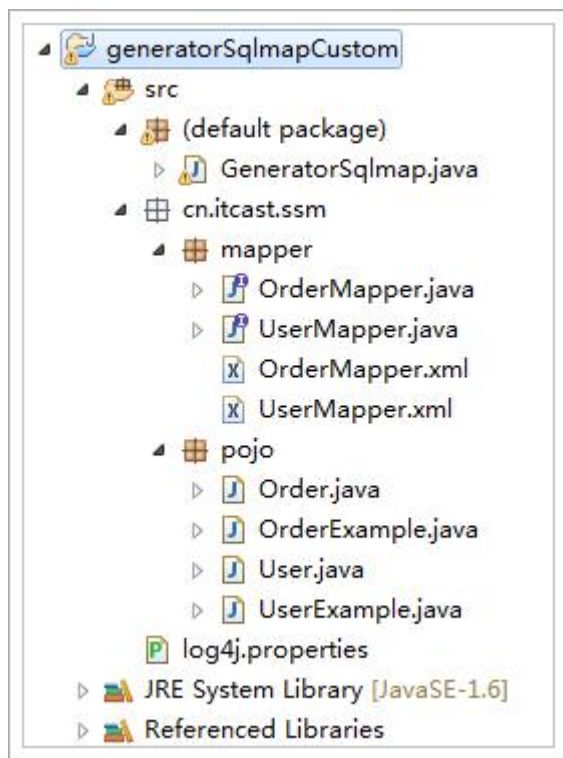
## 6.3. 生成逆向工程代码

找到下图所示的 java 文件，执行工程 main 主函数，



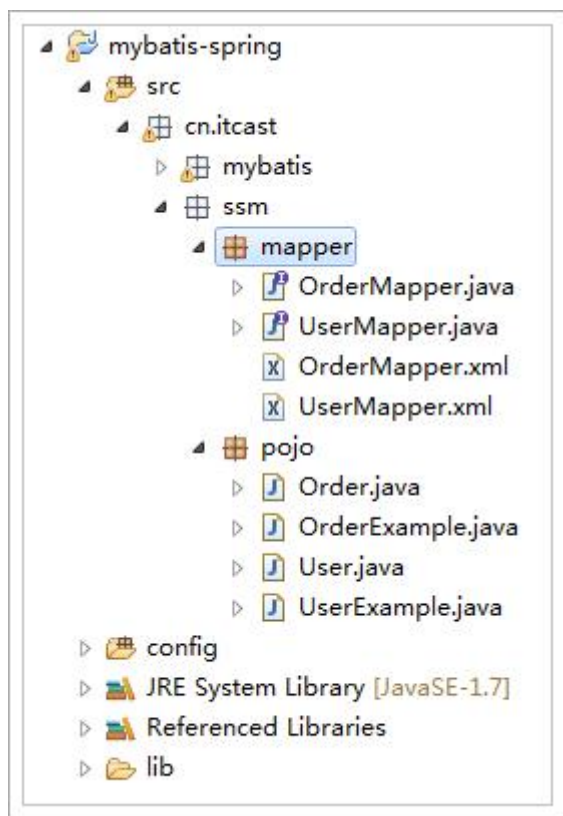


刷新工程，发现代码生成，如下图：



## 6.4. 测试逆向工程代码

1. 复制生成的代码到 mybatis-spring 工程，如下图



## 2. 修改 spring 配置文件

在 applicationContext.xml 修改

```
<!-- Mapper 代理的方式开发，扫描包方式配置代理 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 配置 Mapper 接口，如果需要加载多个包，直接写进来，中间用，分隔 -->
    <!-- <property name="basePackage"
value="com.cloudtcc.mybatis.mapper" /> -->
    <property name="basePackage" value="com.cloudtcc.ssm.mapper" />
</bean>
```

## 3. 编写测试方法：

```
public class UserMapperTest {
    private ApplicationContext context;

    @Before
    public void setUp() throws Exception {
        this.context = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
    }

    @Test
    public void testInsert() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        User user = new User();
        user.setUsername("曹操");
        user.setSex("1");
        user.setBirthday(new Date());
        user.setAddress("三国");

        userMapper.insert(user);
    }

    @Test
    public void testSelectByExample() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        // 创建 User 对象扩展类，用户设置查询条件
        UserExample example = new UserExample();
        example.createCriteria().andUsernameLike("%张%");
    }
}
```

```

        // 查询数据
        List<User> list = userMapper.selectByExample(example);

        System.out.println(list.size());
    }

    @Test
    public void testSelectByPrimaryKey() {
        // 获取 Mapper
        UserMapper userMapper = this.context.getBean(UserMapper.class);

        User user = userMapper.selectByPrimaryKey(1);
        System.out.println(user);
    }
}

```

注意：

1. 逆向工程生成的代码只能做单表查询
2. 不能在生成的代码上进行扩展，因为如果数据库变更，需要重新使用逆向工程生成代码，原来编写的代码就被覆盖了。
3. 一张表会生成 4 个文件