Mybatis开发dao

- Mybatis在项目中主要使用的地方就是开发dao（数据访问层），所以下面讲解一下mybatis开发dao的方法。有两种方式：原始dao开发方式、mapper代理开发方式（推荐）。

1、需求

- 1、 根据用户ID来查询用户信息；
- 2、 根据用户名称来模糊查询用户信息列表；
- 3、添加用户；

2、开发dao的实现类

SqlSession使用范围

- 通过入门程序，大家可以看出，在测试代码中，有大量的重复代码。所以我们第一反应就是想给它抽取出共性的部分，但是SqlSession、SqlSessionFactory、SqlSessionFactoryBuilder有着各自的生命周期，因为这些生命周期的不同，抽取时要有针对性的处理。所以在抽取之前，我们先来了解并总结下它们三个的生命周期。
- 1, SqlSessionFactoryBuilder
  - 它的作用只是通过配置文件创建SqlSessionFactory，所以只要创建出SqlSessionFactory，它就可以销毁了。所以说，它的生命周期是在方法之内。
- 2, SqlSessionFactory
  - 它的作用是创建SqlSession的工厂，工厂一旦创建，除非应用停掉，不要销毁。所以说它的生命周期是在应用范围内。这里可以通过单例模式来管理它。
  - 在mybatis整合spring之后，最好的处理方式是把SqlSessionFactory交由spring来做单例管理。
- 3, SqlSession
  - SqlSession是一个面向用户（程序员）的接口，它的默认实现是DefaultSqlSession。
  - Mybatis是通过SqlSession来操作数据库的。

- SqlSession中不仅包含要处理的SQL信息，还包括一些数据信息，所以说它是线程不安全的，因此它最佳的生命周期范围是在方法体之内。

3、原始dao层写法

- 需要向dao实现类中注入SqlSessionFactory，在方法体内通过SqlSessionFactory创建SqlSession
- 要注意SqlSession和SqlSessionFactory的生命周期。

代码:

UserDao.java

```
package com.lanou.dao;

import com.lanou.domain.User;

import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public interface UserDao {
    User findUserById(int id);

    List<User> findUsersByName(String username);

    void insertUser(User user);
}
```

UserDaoImpl.java

```
package com.lanou.dao;

import com.lanou.domain.User;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;

import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public class UserDaoImpl implements UserDao {
    //注入SqlSessionFactory
```

```
    private SqlSessionFactory sqlSessionFactory;

    //使用构造方法来初始化SqlSessionFactory
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {
        this.sqlSessionFactory = sqlSessionFactory;
    }

    public User findUserById(int id) {
        //通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //返回结果集
        return sqlSession.selectOne("test.findUserById", id);

    }

    public List<User> findUsersByName(String username) {
        //通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全
        SqlSession sqlSession = sqlSessionFactory.openSession();
        return sqlSession.selectList("test.findUsersByName", username);

    }

    public void insertUser(User user) {
        //通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全
        SqlSession sqlSession = sqlSessionFactory.openSession();
        sqlSession.insert("test.insertUser", user);

    }
}
```

UserDaoTest.java

```
package com.lanou.test;

import com.lanou.dao.UserDao;
import com.lanou.dao.UserDaoImpl;
import com.lanou.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Before;
import org.junit.Test;

import java.io.InputStream;
import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public class UserDaoTes {
```

```
    //声明全局的SqlSessionFactory
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        // 1、读取配置文件
        String resource = "SqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resourc
e);
        // 2、根据配置文件创建SqlSessionFactory
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputS
tream);
    }

    @Test
    public void testFindUserById() {
        //构造UserDao对象
        UserDao userDao = new UserDaoImpl(sqlSessionFactory);
        //调用UserDao对象的方法
        User user = userDao.findUserById(1);

        System.out.println(user);
    }

    @Test
    public void testFindUsersByName() {
        //构造UserDao对象
        UserDao userDao = new UserDaoImpl(sqlSessionFactory);
        //调用UserDao对象的方法
        List<User> list = userDao.findUsersByName("张三");

        System.out.println(list);
    }

    @Test
    public void testInsertUser() {
        //构造UserDao对象
        UserDao userDao = new UserDaoImpl(sqlSessionFactory);
        //构造User对象
        User user = new User();
        user.setUsername("王五");
        user.setAddress("庄河");

        //调用UserDao对象的方法
        userDao.insertUser(user);

        System.out.println(user.getId());
    }
}
```

问题总结

原始dao开发存在一些问题：

- 1.存在一定量的模板代码。比如：通过SqlSessionFactory创建SqlSession；调用SqlSession的方法操作数据库；关闭Sqlsession。
- 2.存在一些硬编码。调用SqlSession的方法操作数据库时，需要指定statement的id，这里存在了硬编码。

4、Mapper代理开发方式（推荐）

- Mapper代理的开发方式，程序员只需要编写mapper接口（相当于dao接口）即可。Mybatis会自动的为mapper接口生成动态代理实现类。 不过要实现mapper代理的开发方式，需要遵循一些开发规范。

4.1 开发规范

- 1、 mapper接口的全限定名要和mapper映射文件的namespace的值相同。
- 2、 mapper接口的方法名称要和mapper映射文件中的statement的id相同；
- 3、 mapper接口的方法参数只能有一个，且类型要和mapper映射文件中statement的parameterType的值保持一致。
- 4、 mapper接口的返回值类型要和mapper映射文件中statement的resultType值或resultMap中的type值保持一致；
- 通过规范式的开发mapper接口，可以解决原始dao开发当中存在的问题：
  - 1、 模板代码已经去掉；
  - 2、 剩下去不掉的操作数据库的代码，其实就是一行代码。这行代码中硬编码的部分，通过第一和第二个规范就可以解决。

4.2 编程步骤

- 1、 根据需求创建po类
- 2、 编写全局配置文件
- 3、 根据需求编写映射文件
- 4、 加载映射文件
- 5、 编写mapper接口
- 6、 编写测试代码

4.2.1 编写mapper映射文件

- 重新定义mapper映射文件UserMapper.xml（内容同Users.xml，除了namespace的值），放到新创建的目录mapper下

```xml
<?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：此时用mapper代理方式，它的值必须等于对应mapper接口的全限定名  -->
<mapper namespace="com.lanou.mapper.UserMapper">
    <!-- 根据用户ID，查询用户信息 -->
    <!--
        [id]：statement的id，要求在命名空间内唯一
        [parameterType]：入参的java类型，可是是简单类型、POJO、HashMap
        [resultType]：查询出的单条结果集对应的java类型
        [#{}]：  表示一个占位符?
        [#{id}]：表示该占位符待接收参数的名称为id。
        注意：如果参数为简单类型时，#{}里面的参数名称可以是任意定义
     -->
    <select id="findUserById" parameterType="int"
            resultType="com.lanou.domain.User">
        SELECT * FROM USER WHERE id = #{id}
    </select>


    <!-- 根据用户名称模糊查询用户信息列表 -->
    <!--
        [${}]：表示拼接SQL字符串，即不加解释的原样输出
        [${value}]：表示要拼接的是简单类型参数。
         注意：
        1、如果参数为简单类型时，${}里面的参数名称必须为value
        2、${}会引起SQL注入，一般情况下不推荐使用。
        但是有些场景必须使用${}，比如order by ${colname}
    -->
    <select id="findUsersByName"
            parameterType="java.lang.String"
            resultType="com.lanou.domain.User">
        SELECT * FROM USER WHERE username LIKE '%${value}%'
    </select>

    <!-- 添加用户之自增主键返回（selectKey方式） -->
    <!--
        [selectKey标签]：通过select查询来生成主键
        [keyProperty]：指定存放生成主键的属性
        [resultType]：生成主键所对应的Java类型
        [order]：指定该查询主键SQL语句的执行顺序，相对于insert语句，此时选用AFTER
        [last_insert_id]：MySQL的函数，要配合insert语句一起使用
     -->
    <insert id="insertUser" parameterType="com.lanou.domain.User">
        <selectKey keyProperty="id" resultType="int" order="AFTER">
            SELECT LAST_INSERT_ID()
        </selectKey>
```

```
        INSERT INTO USER(username,sex,birthday,address)
        VALUES (#{username},#{sex},#{birthday},#{address})
    </insert>

</mapper>
```

4.2.2 加载mapper映射文件

```
<!--加载mapper文件-->
    <mappers>
        <mapper resource="User.xml"/>
        <mapper resource="mapper/UserMapper.xml"/>
    </mappers>
```

4.2.3 编写mapper接口

- 内容同UserDao接口一样

```
package com.lanou.mapper;

import com.lanou.domain.User;

import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public interface UserMapper {
    //根据用户ID来查询用户信息
    public User findUserById(int id);

    //根据用户名称来模糊查询用户信息列表
    public List<User> findUsersByName(String username);

    //添加用户
    public void insertUser(User user);

}
```

4.2.4 编写测试代码

```
package com.lanou.test;

import com.lanou.domain.User;
```

```
import com.lanou.mapper.UserMapper;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Before;
import org.junit.Test;

import java.io.InputStream;
import java.util.Date;
import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public class UserMapperTest {
    // 声明全局的SqlSessionFactory
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        // 1、读取配置文件
        String resource = "SqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resourc
e);
        // 2、根据配置文件创建SqlSessionFactory
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputS
tream);
    }

    @Test
    public void testFindUserById() {
        // 创建SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        // 调用mapper对象的方法
        User user = userMapper.findUserById(1);

        System.out.println(user);
        // 关闭SqlSession
        sqlSession.close();

    }

    @Test
    public void testFindUsersByName() {
        // 创建SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        // 调用mapper对象的方法
```

```
        List<User> list = userMapper.findUsersByName("张三");

        System.out.println(list);
        // 关闭SqlSession
        sqlSession.close();
    }

    @Test
    public void testInsertUser() {
        // 创建SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        //构造User对象
        User user = new User();
        user.setUsername("王二麻子");
        user.setAddress("辽阳");
        user.setBirthday(new Date());
        user.setSex("女");

        // 调用mapper对象的方法
        userMapper.insertUser(user);

        System.out.println(user.getId());

        //执行SqlSession的commit操作
        sqlSession.commit();
        // 关闭SqlSession
        sqlSession.close();
    }
}
```

Mybatis全局配置文件

- SqlMapConfig.xml是mybatis的全局配置文件，它的名称可以是任意命名的。

1、全部配置内容

- SqlMapConfig.xml的配置内容和顺序如下（顺序不能乱）：
  - Properties（属性）
  - Settings（全局参数设置）
  - typeAliases（类型别名）

- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）
- environments（环境信息集合）
  - environment（单个环境信息）
    - transactionManager（事物）
    - dataSource（数据源）
- mappers（映射器）

2、常用配置详解

- 2.1 Properties
  - SqlMapConfig.xml文件中可以引用java属性文件中的配置信息

db.properties配置信息如下：

```
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEnc
oding=utf8
db.username=root
db.password=1
```

SqlMapConfig.xml使用properties标签后，如下所示：

```
<?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>
    <!-- 通过properties标签，读取java配置文件的内容 -->
    <properties resource="db.properties"/>

    <!-- 配置mybatis的环境信息 -->
    <environments default="development">

        <environment id="development">
            <!-- 配置JDBC事务控制，由mybatis进行管理 -->
            <transactionManager type="JDBC"></transactionManager>
            <!-- 配置数据源，采用dbcp连接池 -->
            <dataSource type="POOLED">
                <property name="driver" value="${db.driver}"/>
                <property name="url" value="${db.url}"/>
```

```
                <property name="username" value="${db.username}"/>
                <property name="password" value="${db.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--加载mapper文件-->
    <mappers>
        <mapper resource="User.xml"/>
        <mapper resource="mapper/UserMapper.xml"/>
    </mappers>

</configuration>
```

使用${}，可以引用已经加载的java配置文件中的信息。

注意：mybatis将按照下面的顺序加载属性：
- Properties标签体内定义的属性首先被读取
- Properties引用的属性会被读取，如果发现上面已经有同名的属性了，那后面会覆盖前面的值
- parameterType接收的值会最后被读取，如果发现上面已经有同名的属性了，那后面会覆盖前面的值

所以说，mybatis读取属性的顺序由高到低分别是：parameterType接收的属性值、properties引用的属性、properties标签内定义的属性。

- 2.2 Setttings
    - mybatis全局配置参数，全局参数将会影响mybatis的运行行为
- 2.3 typeAliases
    - 别名是使用是为了在映射文件中，更方便的去指定入参和结果集的类型，不再用写很长的一段全限定名。
- 2.4 typeAliases

SqlMapConfig.xml配置信息如下：

```
<!-- 定义别名 -->
    <typeAliases>
        <!-- 单个定义别名 -->
        <typeAlias type="com.lanou.domain.User" alias="user"/>

        <!-- 批量定义别名（推荐） -->
        <!-- [name]：指定批量定义别名的类包，别名为类名（首字母大小写都可） -->
        <package name="com.lanou.domain.po"/>
```

```
    </typeAliases>
```

在UserMapper.xml中使用如下：

```
<select id="findUserById" parameterType="int"
            resultType="user">
        SELECT * FROM USER WHERE id = #{id}
    </select>
```

- 2.5 mappers
  - 2.5.1 `<mapper resource=''/>` 使用相对于类路径的资源,如：

    ```
    <mapper resource="sqlmap/User.xml"/>
    ```

  - 2.5.2 `<mapper url=''/>` 使用完全限定路径,如：

    ```
     <mapper url="file:////Users/lilyxiao/workspaceweb-ssm/Mybatis01/src/resources/User.xml"/>
    ```

  - 2.5.3 `<mapper class=''/>` 使用mapper接口的全限定名,如：

    ```
    <mapper class="com.lanou.mapper.UserMapper"/>
    ```

    - 注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下；
  - 2.5.4 `<package name=''/>`（推荐）,注册指定包下的所有映射文件,如：

    ```
    <package name="com.lanou.mapper"/>
    ```

    - 注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下；

Mybatis映射文件（重点）

1、输入映射

1.1 ParameterType

- 指定输入参数的java类型，可以使用别名或者类的全限定名。它可以接收简单类型、POJO、HashMap。
  - a) 传递简单类型，参考入门需求：根据用户ID查询用户信息。

```
<select id="findUserById" parameterType='int"
        resultType="user">
    SELECT * FROM USER WHERE id = #{id}
</select>
```

  - b) 传递POJO对象，参考入门需求：添加用户。

```
<insert id="insertUser" parameterType="com.lanou.domain.User">
    <selectKey keyProperty="id" resultType="int" order="AFTER">
        SELECT LAST_INSERT_ID()
    </selectKey>
    INSERT INTO USER(username,sex,birthday,address)
    VALUES (#{username},#{sex},#{birthday},#{address})
</insert>
```

  - c) 传递POJO包装对象
    - 开发中通过pojo传递查询条件，查询条件是综合的查询条件，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。

1.2 resultMap

- resultMap可以进行高级结果映射（一对一、一对多映射）。
- 使用方法
  - 如果查询出来的列名和属性名不一致，通过定义一个resultMap将列名和pojo属性

名之间作一个映射关系。

- 1、 定义resultMap
- 2、 使用resultMap作为statement的输出映射类型

- 定义resultMap

```
 <!-- 定义resultMap -->
    <!--
        [id]：定义resultMap的唯一标识
        [type]：定义该resultMap最终映射的pojo对象
        [id标签]：映射结果集的唯一标识列，如果是多个字段联合唯一，则定义多个id标
签
        [result标签]：映射结果集的普通列
        [column]：SQL查询的列名，如果列有别名，则该处填写别名
        [property]：pojo对象的属性名
    -->
    <resultMap type="user" id="userResultMap">
        <id column="id_" property="id"/>
        <result column="username_" property="username"/>
        <result column="sex_" property="sex"/>
    </resultMap>
```

- 定义statement

```
<!-- 根据ID查询用户信息（学习resultMap） -->
<select id="findUserByIdResultMap" parameterType="int"
        resultMap="userResultMap">
    SELECT id id_,username username_,sex sex_ FROM USER
    WHERE id = #{id}
</select>
```

- 测试

```
@Test
public void findUserByIdResultMapTest() {
    // 创建SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 通过SqlSession，获取mapper接口的动态代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class)
;

    // 调用mapper对象的方法
    User user = userMapper.findUserByIdResultMap(1);
```

```
        System.out.println(user);
        // 关闭SqlSession
        sqlSession.close();
    }
```

1.3 动态SQL（重点）

- 通过Mybatis提供的各种动态标签实现动态拼接sql，使得mapper映射文件在编写SQL时更加灵活，方便。常用动态SQL标签有：if、where、foreach；

1.3.1 if和where

- If标签：作为判断入参来使用的，如果符合条件，则把if标签体内的SQL拼接上。
  - 注意：用if进行判断是否为空时，不仅要判断null，也要判断空字符串‘’；
- Where标签：会去掉条件中的第一个and符号。
- 映射文件

```
<?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：此时用mapper代理方式，它的值必须等于对应mapper接口的全限定名  -->
<mapper namespace="com.lanou.mapper.UserDynamicMapper">
    <!-- 综合查询用户信息，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息）  -->
    <select id="findUsersByQueryVO"
            parameterType="userQueryVO"
            resultType="user">
        SELECT * FROM USER
        <where>
            <if test="userExt != null">
                <if test="userExt.sex != null and userExt.sex != ''">
                    AND sex = #{userExt.sex}
                </if>
                <if test="userExt.username != null and userExt.username != ''">
                    AND username LIKE '%${userExt.username}%'
                </if>
            </if>
        </where>
    </select>
```

```
    <!-- 综合查询用户信息总数，需要传入查询条件复杂，
    比如（用户信息、订单信息、商品信息） -->
    <select id="findUsersCount"
            parameterType="userQueryVO"
            resultType="int">
        SELECT count(1) FROM USER
        <where>
            <if test="userExt != null">
                <if test="userExt.sex != null and userExt.sex != ''">
                    AND sex = #{userExt.sex}
                </if>
                <if test="userExt.username != null and userExt.username
 != ''">
                    AND username LIKE '%${userExt.username}%'
                </if>
            </if>
        </where>
    </select>

</mapper>
```

- Mapper接口

```
package com.lanou.mapper;

import com.lanou.domain.User;
import com.lanou.domain.UserQueryVO;

import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public interface UserDynamicMapper {
    //通过包装类来进行复杂的用户信息综合查询
    List<User> findUsersByQueryVO(UserQueryVO userQueryVO);

    //综合查询用户总数
    int findUsersCount(UserQueryVO userQueryVO);

}
```

- 测试代码

```
    @Test
    public void findUsersByQueryVO() {
        // 创建SqlSession
```

```
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserDynamicMapper userMapper = sqlSession.getMapper(UserDynamic
Mapper.class);

        //构造userQueryVO对象
        UserQueryVO userQueryVO = new UserQueryVO();

        // 此处使用动态SQL，不传username参数
        UserExt userExt = new UserExt();
        userExt.setSex("男");
//        userExt.setUsername("李四");

        userQueryVO.setUserExt(userExt);

        // 调用mapper对象的方法
        List<User> userExts = userMapper.findUsersByQueryVO(userQueryVO
);

        System.out.println(userExts);
        // 关闭SqlSession
        sqlSession.close();
    }

    @Test
    public void findUsersCount() {
        // 创建SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserDynamicMapper userMapper = sqlSession.getMapper(UserDynamic
Mapper.class);

        //构造userQueryVO对象
        UserQueryVO userQueryVO = new UserQueryVO();

        // 此处使用动态SQL，不传username参数
        UserExt userExt = new UserExt();
        userExt.setSex("男");
//        userExt.setUsername("李四");

        userQueryVO.setUserExt(userExt);

        // 调用mapper对象的方法
        int count = userMapper.findUsersCount(userQueryVO);

        System.out.println(count);
        // 关闭SqlSession
        sqlSession.close();
    }
```

1.3.2 SQL片段

- Mybatis提供了SQL片段的功能，可以提高SQL的可重用性。
- 使用sql标签来定义一个SQL片段

```
<!-- 定义SQL片段 -->
    <!--
        [sql标签]：定义一个SQL片段
        [id]：SQL片段的唯一标识
        建议：
            1、SQL片段中的内容最好是以单表来定义
            2、如果是查询字段，则不要写上SELECT
            3、如果是条件语句，则不要写上WHERE
     -->
    <sql id="select_user_where">
        <if test="userExt != null">
            <if test="userExt.sex != null and userExt.sex != ''">
                AND sex = #{userExt.sex}
            </if>
            <if test="userExt.username != null and userExt.username != ''">
                AND username LIKE '%${userExt.username}%'
            </if>
        </if>
    </sql>
```

- 引用SQL片段，使用`<include refid=""/>`来引用SQL片段

```
<!-- 根据用户id来查询用户信息（使用SQL片段） -->
    <!--
        [include标签]：引用已经定义好的SQL片段
        [refid]：引用的SQL片段id
    -->
    <select id="findUserList" parameterType="userQueryVO"
            resultType="user">

        SELECT * FROM USER
        <where>
            <include refid="select_user_where"/>
        </where>
    </select>
    <!-- 综合查询用户信息总数，需要传入查询条件复杂，
    比如（用户信息、订单信息、商品信息） -->
    <select id="findUsersCount2" parameterType="userQueryVO"
            resultType="int">
        SELECT count(1) FROM USER
        <where>
            <include refid="select_user_where"/>
        </where>
    </select>
```

1.3.3 Foreach

- 向sql传递数组或List时，mybatis使用foreach解析数组里的参数并拼接到SQL中。
- 需求：在用户查询列表盒查询总数的statement中增加多个id输入查询。
- SQL：SELECT * FROM user WHERE id IN (1,2,3)
- 定义pojo中的List属性

```
package com.lanou.domain;

import java.util.List;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public class UserQueryVO {
    private UserExt userExt;

    //id集合
    private List<Integer> idList;

    public UserExt getUserExt() {
        return userExt;
    }

    public void setUserExt(UserExt userExt) {
        this.userExt = userExt;
    }

    public List<Integer> getIdList() {
        return idList;
    }

    public void setIdList(List<Integer> idList) {
        this.idList = idList;
    }
}
```

- 映射文件

```
<!-- 定义SQL片段 -->
    <!--
        [sql标签]：定义一个SQL片段
        [id]：SQL片段的唯一标识
```

```
        建议:
            1、SQL片段中的内容最好是以单表来定义
            2、如果是查询字段，则不要写上SELECT
            3、如果是条件语句，则不要写上WHERE
     -->
    <sql id="select_user_where">
        <if test="userExt != null">
            <if test="userExt.sex != null and userExt.sex != ''">
                AND sex = #{userExt.sex}
            </if>
            <if test="userExt.username != null and userExt.username != ''">
                AND username LIKE '%${userExt.username}%'
            </if>
        </if>

        <!-- [foreach标签]：表示一个foreach循环 -->
        <!-- [collection]：集合参数的名称，如果是直接传入集合参数，
                            则该处的参数名称只能填写[list]。 -->
        <!-- [item]：每次遍历出来的对象 -->
        <!-- [open]：开始遍历时拼接的串 -->
        <!-- [close]：结束遍历时拼接的串 -->
        <!-- [separator]：遍历出的每个对象之间需要拼接的字符 -->
        <if test="idList != null and idList.size > 0">
            <foreach collection="idList" item="id"
                     open="AND id IN (" close=")" separator=",">
                #{id}
            </foreach>
        </if>

    </sql>

    <!-- 根据用户id来查询用户信息（使用SQL片段） -->
    <!--
        [include标签]：引用已经定义好的SQL片段
        [refid]：引用的SQL片段id
    -->
    <select id="findUserList" parameterType="userQueryVO"
            resultType="user">

        SELECT * FROM USER
        <where>
            <include refid="select_user_where"/>
        </where>
    </select>
```

- 测试代码

```
@Test
    public void findUsersByQueryVO2() {
```

```
        // 创建SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 通过SqlSession，获取mapper接口的动态代理对象
        UserDynamicMapper userMapper = sqlSession.getMapper(UserDynamic
Mapper.class);

        //构造userQueryVO对象
        UserQueryVO userQueryVO = new UserQueryVO();

        // 此处使用动态SQL，不传username参数
//        UserExt userExt = new UserExt();
//        userExt.setSex("男");
//        userExt.setUsername("李四");
//        userQueryVO.setUserExt(userExt);

        // 创建用户ID集合，然后设置到QueryUserVO对象中
        List<Integer> idList = new ArrayList<Integer>();
        idList.add(1);
        idList.add(2);
        idList.add(3);
        userQueryVO.setIdList(idList);

        // 调用mapper对象的方法
        List<User> users = userMapper.findUserList(userQueryVO);

        System.out.println(users);
        // 关闭SqlSession
        sqlSession.close();
    }
```

- 直接传递List集合
  - 映射文件

```
<!-- 根据用户ID的集合查询用户列表（学习foreach标签之直接传ID集合） -->
<!--
    [foreach标签]：表示一个foreach循环
    [collection]：集合参数的名称，如果是直接传入集合参数，则该处的参数名称只能填
写[list]。
    [item]：定义遍历集合之后的参数名称
    [open]：开始遍历之前需要拼接的SQL串
    [close]：结束遍历之后需要拼接的SQL串
    [separator]：遍历出的每个对象之间需要拼接的字符
 -->
<select id="findUsersByIdList" parameterType="java.util.List"
        resultType="user">
    SELECT * FROM USER
    <where>
        <if test="list != null and list.size > 0">
            <foreach collection="list" item="id"
```

```
                    open="AND id IN (" close=")" separator=",">
                #{id}
            </foreach>
        </if>
    </where>
</select>
```

- Mapper接口

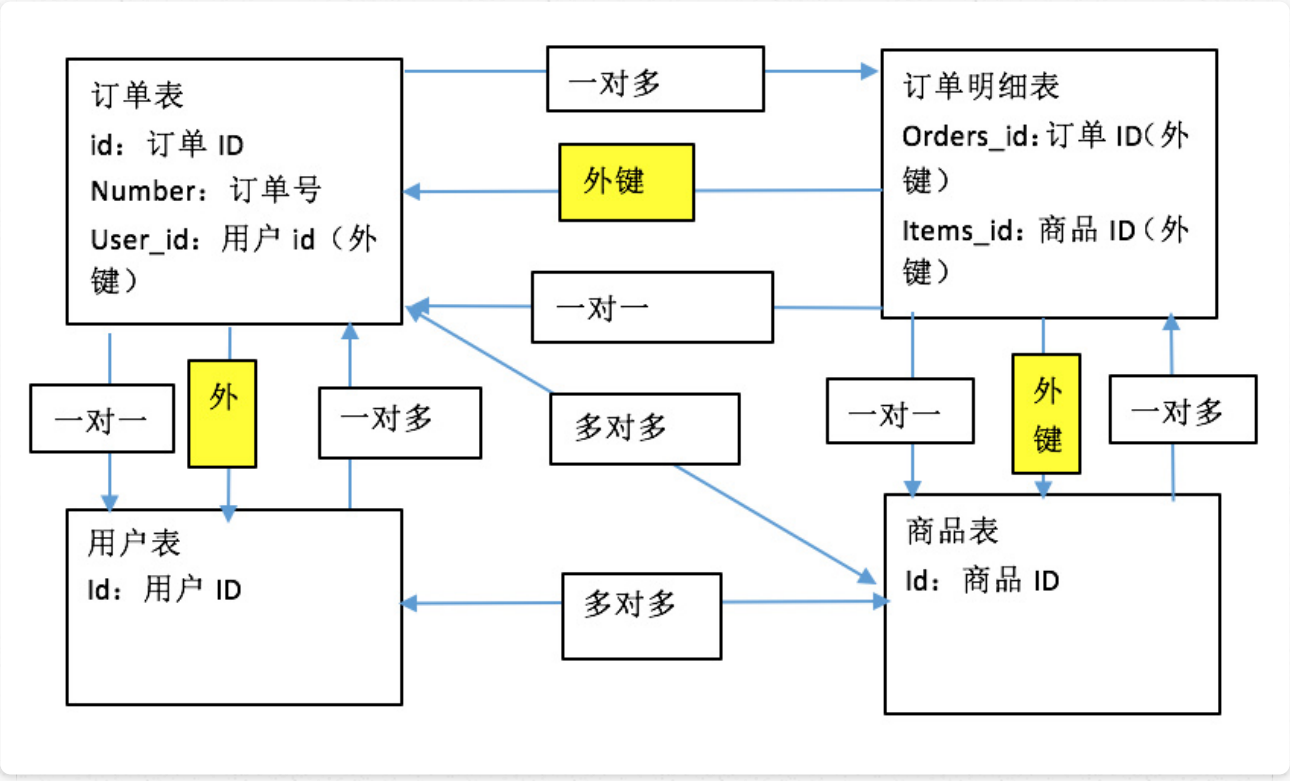
```
List<User> findUsersByIdList(List<Integer> idList);
```

- 测试代码

```
 @Test
public void findUsersByIdListTest() {
    // 创建SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 通过SqlSession，获取mapper接口的动态代理对象
    UserDynamicMapper userMapper = sqlSession.getMapper(UserDynamicMa
pper.class);

    // 构造List<Integer>集合
    List<Integer> idList = new ArrayList<Integer>();
    idList.add(1);
    idList.add(3);
    idList.add(4);

    // 调用mapper对象的方法
    List<User> list = userMapper.findUsersByIdList(idList);
    System.out.println(list);
    // 关闭SqlSession
    sqlSession.close();
}
```

映射关系

- 1、图形分析



- 2、数据表之间有外键关系的业务关系
  - User和Orders：
    - User---->Orders：一个用户可以创建多个订单，一对多
    - Orders--->User：一个订单只由一个用户创建，一对一
  - Orders和OrderDetail：
    - Orders-OrderDetail：一个订单可以包括 多个订单明细，因为一个订单可以购买多个商品，每个商品的购买信息在OrderDetail记录，一对多关系
    - OrderDetail--> Orders：一个订单明细只能包括在一个订单中，一对一
  - OrderDetail和Itesm：
    - OrderDetail---》Itesm：一个订单明细只对应一个商品信息，一对一
    - Itesm--> OrderDetail:一个商品可以包括在多个订单明细 ， 一对多
- 3、数据库表之间没有外键关系的业务关系
  - **Orders和Items：**
    - 这两张表没有直接的外键关系，通过业务及数据库的间接关系分析出它们是多对多的关系。
    - Orders -> OrderDetail ->Items：一个订单可以有多个订单明细，一个订单明细对应一个商品，所以一个订单对应多个商品

    - Items -> OrderDetail -> Orders：一个商品可以对应多个订单明细，一个订单明细对应一个订单，所以一个商品对应多个订单
  - **User和Items：**
    - 这两张表没有直接的外键关系，通过业务及数据库的间接关系分析出它们是多对多的关系。
    - User -> Orders -> OrderDetail -> Items：一个用户有多个订单，一个订单有多个订单明细、一个订单明细对应一个商品，所以一个用户对应多个商品
    - Items -> OrderDetail -> Orders -> User：一个商品对应多个订单明细，一个订单明细对应一个订单，一个订单对应一个用户，所以一个商品对应多个用户
- sql

```
CREATE TABLE Items
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(255)
);
CREATE UNIQUE INDEX Item_id_uindex ON Items (id);
CREATE TABLE OrderDetail
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    orders_id INT(11),
    items_id INT(11),
    CONSTRAINT OrderDetail_Orders_id_fk FOREIGN KEY (orders_id) REFEREN
CES Orders (id),
    CONSTRAINT OrderDetail_Item_id_fk FOREIGN KEY (items_id) REFERENCES
 items
);
CREATE UNIQUE INDEX OrderDetail_id_uindex ON OrderDetail (id);
CREATE INDEX OrderDetail_Item_id_fk ON OrderDetail (items_id);
CREATE INDEX OrderDetail_Orders_id_fk ON OrderDetail (orders_id);
CREATE TABLE Orders
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    number VARCHAR(20),
    user_id INT(11),
    CONSTRAINT Orders_User_id_fk FOREIGN KEY (user_id) REFERENCES User
(id)
);
CREATE UNIQUE INDEX Orders_id_uindex ON Orders (id);
CREATE INDEX Orders_User_id_fk ON Orders (user_id);
CREATE TABLE Student
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    sname VARCHAR(20),
    gender VARCHAR(10),
```

```
    age INT(11)
);
CREATE TABLE User
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    username VARCHAR(20),
    sex VARCHAR(10),
    birthday DATE,
    address VARCHAR(255)
);
CREATE UNIQUE INDEX User_id_uindex ON User (id);
```

1、一对一查询

- 需求：查询订单信息，关联查询创建订单的用户信息
- SQL语句：

```
Select
  Orders.id,
  Orders.user_id,
  Orders.number,
  Orders.createtime,
  Orders.note,
  User.username,
  User.address
from Orders,User
where Orders.user_id = User.id
```

- resultType：复杂查询时，单表对应的po类已不能满足输出结果集的映射，所以要根据需求建立一个扩展类来作为resultType的类型。
- resultType：复杂查询时，单表对应的po类已不能满足输出结果集的映射，所以要根据需求建立一个扩展类来作为resultType的类型。

OrdersExt.java

```
package com.lanou.domain;

/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
//通过此类映射订单和用户查询的结果，让此类继承包括 字段较多的pojo类
public class OrdersExt extends Orders {

    //添加用户属性
```

```
    /*USER.username,
      USER.address */

    private String username;
    private String address;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

}
```

mapper接口

```
/**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public interface OrdersMapper {
    //  进行订单信息查询，包括用户的名称和地址信息
    List<OrdersExt> findOrdersUser();

}
```

映射文件

```
<?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：此时用mapper代理方式，它的值必须等于对应mapper接口的全限定名  --
>
<mapper namespace="com.lanou.mapper.OrdersMapper">

    <!-- 定义查询订单表列名的SQL片段 -->
```

```
    <sql id="select_orders">
        Orders.id,
        Orders.user_id,
        Orders.number,
        Orders.createtime,
        Orders.note
    </sql>
    <!-- 定义查询用户表列名的SQL片段 -->
    <sql id="select_user">
        USER.username,
        USER.address
    </sql>
    <!-- 进行订单信息查询，包括用户的名称和地址信息 -->
    <select id="findOrdersUser"
            resultType="com.lanou.domain.OrdersExt">
        Select
        <include refid="select_orders"/>,
        <include refid="select_user"/>
        from Orders,USER
        where Orders.user_id = USER.id
    </select>
</mapper>
```

测试代码

```
**
 * Created by 蓝鸥科技有限公司  www.lanou3g.com.
 */
public class RelativeMapperTest {
    // 声明全局的SqlSessionFactory
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        // 1、读取配置文件
        String resource = "SqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resourc
e);
        // 2、根据配置文件创建SqlSessionFactory
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputS
tream);
    }

    @Test
    public void testFindOrdersUser() {
        // 创建sqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过SqlSession构造usermapper的代理对象
        OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.c
```

```
lass);
        // 调用usermapper的方法
        List<OrdersExt> list = ordersMapper.findOrdersUser();

        System.out.println(list);

        // 释放SqlSession
        sqlSession.close();
    }

}
```

改进resultMap

在Orders类中，添加User对象

```
public class Orders {
    private int id;
    private String number;//订单编号
    private int user_id;//用户id
    private Date createtime;//订单创建时间
    private String note;//备注

    //用户信息
    private User user;
    //..省略get...
}
```

修改映射文件，在映射文件中添加关系

```
<!-- 进行订单信息查询，包括用户的名称和地址信息（ResultMap） -->
    <select id="findOrdersUserRstMap" resultMap="OrdersUserRstMap">
        Select
        <include refid="select_orders"/>
        ,
        <include refid="select_user"/>
        from orders,user
        where orders.user_id = user.id
    </select>

    <!-- 定义orderUserResultMap -->
    <resultMap type="com.lanou.domain.Orders" id="OrdersUserRstMap">
        <id column="id" property="id"/>
        <result column="user_id" property="user_id"/>
        <result column="number" property="number"/>
        <result column="createtime" property="createtime"/>
        <result column="note" property="note"/>
```

```
        <!-- 映射一对一关联关系的用户对象-->
        <!--
            property: 指定关联对象要映射到Orders的哪个属性上
            javaType: 指定关联对象所要映射的java类型
          -->
        <!-- id标签：指定关联对象结果集的唯一标识，很重要，不写不会报错，但是会影响
性能 -->
        <association property="user" javaType="com.lanou.domain.User">
            <id column="id" property="id"/>
            <result column="username" property="username"/>
            <result column="address" property="address"/>
        </association>
    </resultMap>
```

测试代码

```
 @Test
    public void testFindOrdersUserRstMap() {
        // 创建sqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过SqlSession构造usermapper的代理对象
        OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.c
lass);
        // 调用usermapper的方法
        List<Orders> list = ordersMapper.findOrdersUserRstMap();

        //此处我们采用debug模式来跟踪代码，然后验证结果集是否正确
        System.out.println(list);
        // 释放SqlSession
        sqlSession.close();
    }
```

一对一小结

- **实现一对一查询：**
  - resultType：使用resultType实现较为简单，如果pojo中没有包括查询出来的列名，需要增加列名对应的属性，即可完成映射。如果没有查询结果的特殊要求建议使用resultType。
  - resultMap：需要单独定义resultMap，实现有点麻烦，如果对查询结果有特殊的要求，使用resultMap可以完成将关联查询映射pojo的对象属性中。
  - resultMap可以实现延迟加载，resultType无法实现延迟加载。

2、一对多查询

- 一对多查询和一对一查询的配置基本类似。只是如果使用resultMap的话，映射一对多关联关系要使用collection标签。
- 查询订单信息及订单明细信息
- SQL：

```
SELECT
  Orders.id,
  Orders.user_id,
  Orders.number,
  Orders.createtime,
  Orders.note,
  User.username,
  User.address,
  OrderDetail.id detail_id,
  OrderDetail.items_id,
  OrderDetail.items_num
FROM Orders, User, OrderDetail
WHERE Orders.user_id = User.id
      AND Orders.id = OrderDetail.orders_id
```

- 在Order类中添加订单详情的集合，并提供get／set方法

```
public class Orders {
    private int id;
    private String number;//订单编号
    private int user_id;//用户id
    private Date createtime;//订单创建时间
    private String note;//备注

    //用户信息
    private User user;

    //订单明细
    private List<OrderDetail> detailList;
}
```

- 编写映射文件

```
    <!-- 定义OrdersAndOrderdetailRstMap -->
    <!-- extends：继承已有的ResultMap，值为继承的ResultMap的唯一标示 -->
    <resultMap type="com.lanou.domain.Orders" id="OrdersAndOrderdetailR
stMap" extends="OrdersUserRstMap">
        <!-- 映射关联关系（一对多） -->
        <!-- collection标签：定义一个一对多关系
```

```
            ofType: 指定该集合参数所映射的类型
         -->
        <collection property="detailList"
                    ofType="com.lanou.domain.OrderDetail">
            <id column="id" property="id"/>
            <result column="items_id" property="items_id"/>
            <result column="items_num" property="items_num"/>
        </collection>
    </resultMap>

    <!-- 一对多关联 -->
    <!-- 查询订单信息，包括用户名称、用户地址，订单商品信息（嵌套结果） -->
    <select id="findOrdersAndOrderdetailRstMap" resultMap="OrdersAndOrd
erdetailRstMap">

        Select
        <include refid="select_orders"/>
        ,
        <include refid="select_user"/>
        ,
        OrderDetail.id detail_id,
        OrderDetail.items_id,
        OrderDetail.items_num
        FROM Orders, User, OrderDetail
        WHERE Orders.user_id = User.id
        AND Orders.id = OrderDetail.orders_id
    </select>
```

- 测试代码

```
@Test
    public void testFindOrdersAndOrderdetailRstMap() {
        // 创建sqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过SqlSession构造usermapper的代理对象
        OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.c
lass);
        // 调用usermapper的方法
        List<Orders> list = ordersMapper.findOrdersAndOrderdetailRstMap
();

        //此处我们采用debug模式来跟踪代码，然后验证结果集是否正确
        System.out.println(list);
        // 释放SqlSession
        sqlSession.close();
    }
```

一对多小结

- mybatis使用resultMap的collection对关联查询的多条记录映射到一个list集合属性中。（推荐）
- 使用resultType实现：
  - 需要对结果集进行二次处理。
  - 将订单明细映射到orders中的orderdetails中，需要自己处理，使用双重循环遍历，去掉重复记录，将订单明细放在orderdetails中。

3、多对多查询

- 需求：查询用户信息及用户购买的商品信息，要求将关联信息映射到主pojo的pojo属性中
- SQL语句：

```
SELECT
  Orders.id,
  Orders.user_id,
  Orders.number,
  Orders.createtime,
  Orders.note,
  user.username,
  user.address,
  orderdetail.id detail_id,
  orderdetail.items_id,
  orderdetail.items_num,
  Items.name     items_name
FROM
  Orders,
  User,
  OrderDetail,
  Items
WHERE User.`id` = Orders.`user_id`
      AND Orders.`id` = OrderDetail.`orders_id`
      AND OrderDetail.`items_id` = Items.`id`
```

- 映射思路
  - 将用户信息映射到user中。
  - 在user类中添加订单列表属性List orderslist，将用户创建的订单映射到orderslist
  - 在Orders中添加订单明细列表属性List detailList，将订单的明细映射到detailList

    - 在Orderdetail中添加Items属性，将订单明细所对应的商品映射到Items

- 在User类中添加List ordersList属性

```
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

    //订单信息
    private List<Orders> ordersList;
}
```

- 在Orders类中添加List detailList属性

```
public class Orders {
    private int id;
    private String number;//订单编号
    private int user_id;//用户id
    private Date createtime;//订单创建时间
    private String note;//备注

    //用户信息
    private User user;

    //订单明细
    private List<OrderDetail> detailList;
}
```

- 在OrderDetail类中添加items属性

```
public class OrderDetail {
    private int id;
    private int orders_id;//订单编号
    private int items_id;//商品编号
    private int items_num;//商品数量

    //商品对象
    private Items items;
}
```

- 编写mapper接口

```
    //查询用户及用户购买商品信息（多对多映射之使用resultMap）
    List<User> findUserAndItemsRstMap();
```

- 编写映射文件

```
    <!-- 定义查询订单详情的SQL片段 -->
    <sql id="select_orderdetail">
        OrderDetail.id detail_id,
        OrderDetail.items_id,
        OrderDetail.items_num
    </sql>

    <!-- 定义UserAndItemsRstMap -->
    <resultMap type="com.lanou.domain.User" id="UserAndItemsRstMap">
        <!-- 用户信息 -->
        <!-- id：关联查询用户的唯一标示 -->
        <id column="id" property="id"/>
        <result column="username" property="username"/>
        <result column="address" property="address"/>
        <!-- 订单信息 （一个用户有多个订单） -->
        <collection property="ordersList" ofType="com.lanou.domain.Orde
rs">
            <id column="id" property="id"/>
            <result column="user_id" property="user_id"/>
            <result column="number" property="number"/>
            <result column="createtime" property="createtime"/>
            <result column="note" property="note"/>
            <!-- 订单明细信息（一个订单有多个订单明细） -->
            <collection property="detailList" ofType="com.lanou.domain.
OrderDetail">
                <id column="id" property="id"/>
                <result column="items_id" property="items_id"/>
                <result column="items_num" property="items_num"/>
                <!-- 商品信息 （一个订单明细对应一个商品） -->
                <association property="items" javaType="com.lanou.domai
n.Items">
                    <id column="id" property="id"/>
                    <result column="name" property="name"/>
                </association>
            </collection>
        </collection>
    </resultMap>

    <!-- 查询用户及用户购买商品信息（多对多映射之使用resultMap） -->
    <select id="findUserAndItemsRstMap" resultMap="UserAndItemsRstMap">
        Select
        <include refid="select_orders"/>
        ,
        <include refid="select_user"/>
```

```
        ,
        <include refid="select_orderdetail"/>
        ,
        Items.name items_name

        from Orders,User,OrderDetail,Items

        WHERE User.`id` = Orders.`user_id`
        AND Orders.`id` = OrderDetail.`orders_id`
        AND OrderDetail.`items_id` = Items.`id`
    </select>
```

- 测试代码

```
@Test
    public void testFindUserAndItemsRstMap() {
        // 创建sqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过SqlSession构造usermapper的代理对象
        OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.c
lass);
        // 调用usermapper的方法
        List<User> list = ordersMapper.findUserAndItemsRstMap();

        // 此处我们采用debug模式来跟踪代码，然后验证结果集是否正确
        System.out.println(list);
        // 释放SqlSession
        sqlSession.close();
    }
```

多对多小结

- 将查询用户购买的商品信息明细清单，（用户名、用户地址、购买商品名称、购买商品时间、购买商品数量）
- 针对上边的需求就使用resultType将查询到的记录映射到一个扩展的pojo中，很简单实现明细清单的功能。
- 一对多是多对多的特例，如下需求：
  - 查询用户购买的商品信息，用户和商品的关系是多对多关系。
  - 需求1：
    - 查询字段：用户账号、用户名称、用户性别、商品名称、商品价格(最常见)

        - 企业开发中常见明细列表，用户购买商品明细列表，使用resultType将上边查询列映射到pojo输出。
    - 需求2：
        - 查询字段：用户账号、用户名称、购买商品数量、商品明细（鼠标移上显示明细）
        - 使用resultMap将用户购买的商品明细列表映射到user对象中。
- 总结：
    - 使用resultMap是针对那些对查询结果映射有特殊要求的功能，，比如特殊要求映射成list中包括多个list。

高级映射总结

- resultType：
    - 作用：
        - 将查询结果按照sql列名pojo属性名一致性映射到pojo中。
    - 场合：
        - 常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用resultType将每一条记录映射到pojo中，在前端页面遍历list（list中是pojo）即可。
- resultMap：
    - 使用association和collection完成一对一和一对多高级映射（对结果有特殊的映射要求）。
- association：
    - 作用：
        - 将关联查询信息映射到一个pojo对象中。
    - 场合：
        - 为了方便查询关联信息可以使用association将关联订单信息映射为用户对象的pojo属性中，比如：查询订单及关联用户信息。
        - 使用resultType无法将查询结果映射到pojo对象的pojo属性中，根据对结果集查询遍历的需要选择使用resultType还是resultMap。

- collection：
  - 作用：
    - 将关联查询信息映射到一个list集合中。
  - 场合：
    - 为了方便查询遍历关联信息可以使用collection将关联信息映射到list集合中，比如：查询用户权限范围模块及模块下的菜单，可使用collection将模块映射到模块list中，将菜单列表映射到模块对象的菜单list属性中，这样的作的目的也是方便对查询结果集进行遍历查询。
    - 如果使用resultType无法将查询结果映射到list集合中。

延迟加载

1、什么是延迟加载

- resultMap中的association和collection标签具有延迟加载的功能。
- 延迟加载的意思是说，在关联查询时，利用延迟加载，先加载主信息。需要关联信息时再去按需加载关联信息。这样会大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。

2、设置延迟加载

- Mybatis默认是不开启延迟加载功能的，我们需要手动开启。
- 需要在SqlMapConfig.xml文件中，在标签中开启延迟加载功能。
- lazyLoadingEnabled、aggressiveLazyLoading

设置项	描述	允许值	默认值
lazyLoadingEnabled	全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。	true | false	true
aggressiveLazyLoading	当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。	true | false	true

```
<!-- 开启延迟加载功能 -->
<settings>
    <!-- lazyLoadingEnabled : 延迟加载启用（懒加载）默认是false -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- aggressiveLazyLoading : 积极的懒加载 , false的话是按需加载 默认是true -->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

3、延迟加载思考

- 不使用mybatis提供的association及collection中的延迟加载功能，如何实现延迟加载？？

实现方法如下：

1、定义两个mapper方法：

- 1、查询订单列表
- 2、根据用户id查询用户信息

2、实现思路：

- 先去查询第一个mapper方法，获取订单信息列表
- 在程序中（service），按需去调用第二个mapper方法去查询用户信息。

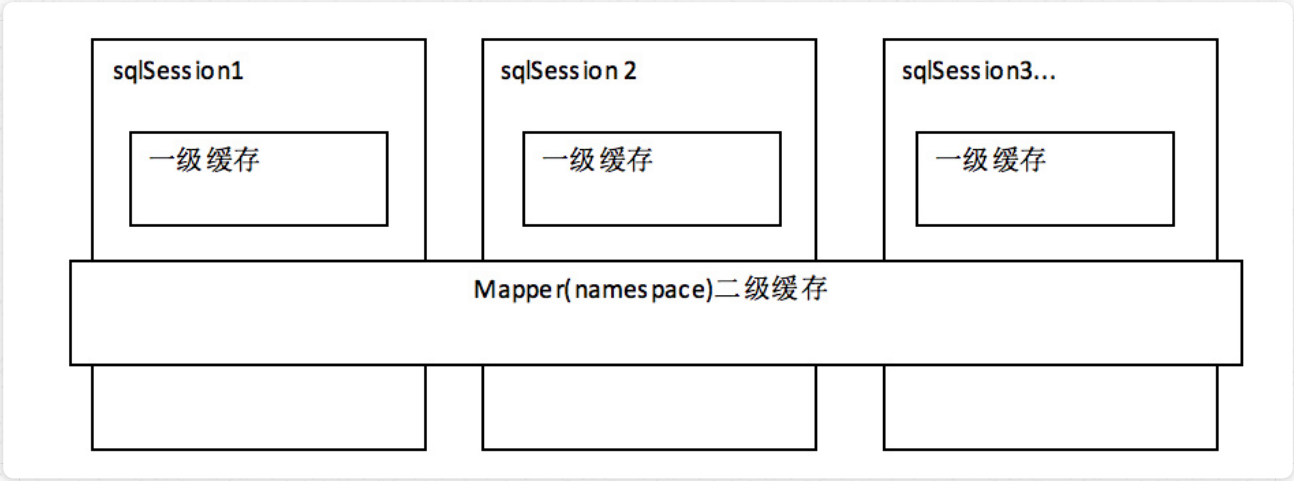
3、总之：

- 使用延迟加载方法，先去查询简单的sql（最好单表，也可以关联查询），再去按需要加载关联查询的其它信息。

查询缓存

1、缓存分析

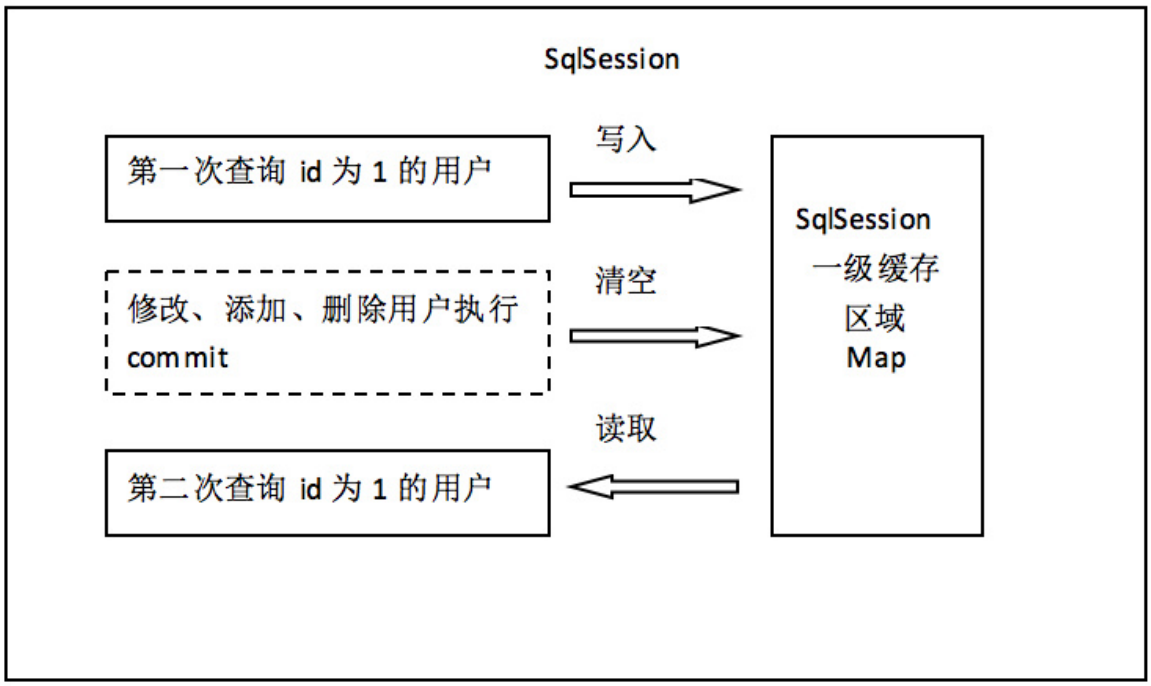
- mybatis提供查询缓存，如果缓存中有数据就不用从数据库中获取，用于减轻数据压力，提高系统性能。



- 一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。
- 二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

2、一级缓存

2.1 原理



- 第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。
- 如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。
- 第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。
- Mybatis默认支持一级缓存。

2.2 应用

- 正式开发，是将mybatis和spring进行整合开发，事务控制在service中。
- 一个service方法中包括，很多mapper方法调用。

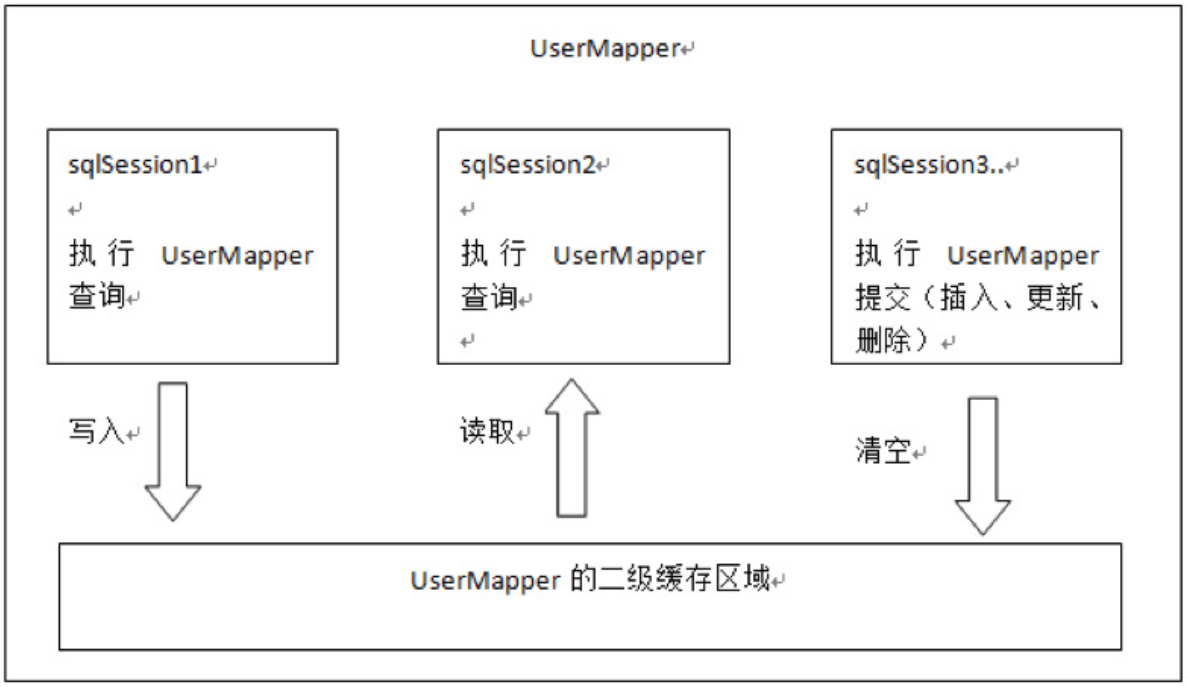
```
service{
    //开始执行时，开启事务，创建SqlSession对象
    //第一次调用mapper的方法findUserById(1)

    //第二次调用mapper的方法findUserById(1)，从一级缓存中取数据
    //方法结束，sqlSession关闭
}
```

- 如果是执行两次service调用查询相同的用户信息，不走一级缓存，因为session方法

结束，sqlSession就关闭，一级缓存就清空。

3、二级缓存

3.1 原理



- 二级缓存是mapper级别的。
- 第一次调用mapper下的SQL去查询用户信息。查询到的信息会存到该mapper对应的二级缓存区域内。
- 第二次调用相同namespace下的mapper映射文件中相同的SQL去查询用户信息。会去对应的二级缓存内取结果。
- 如果调用相同namespace下的mapper映射文件中的增删改SQL，并执行了commit操作。此时会清空该namespace下的二级缓存。

3.2 开启二级缓存

- Mybatis默认是没有开启二级缓存
- 1、 在核心配置文件SqlMapConfig.xml中加入以下内容（开启二级缓存总开关）：
  - 在settings标签中添加以下内容：

```
<!-- 开启二级缓存总开关 -->
<setting name="cacheEnabled" value="true"/>
```

- 2、 在UserMapper映射文件中，加入以下内容，开启二级缓存：

```
<!-- 开启本mapper下的namespace的二级缓存，默认使用的是mybatis提供的PerpetualCa
che -->
<cache></cache>
```

3.3 实现序列化

- 由于二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，所以需要给缓存的对象执行序列化。
- 如果该类存在父类，那么父类也要实现序列化。

3.4 禁用二级缓存

- 该statement中设置userCache=false，可以禁用当前select语句的二级缓存，即每次查询都是去数据库中查询，默认情况下是true，即该statement使用二级缓存。

```
<select id="findUserById" parameterType="int"
        resultType="com.lanou.domain.User" useCache="true">
    SELECT * FROM user WHERE id = #{id}
</select>
```

3.5 刷新二级缓存

- 该statement中设置flushCache=true可以刷新当前的二级缓存，默认情况下如果是select语句，那么flushCache是false。如果是insert、update、delete语句，那么flushCache是true。
- 如果查询语句设置成true，那么每次查询都是去数据库查询，即意味着该查询的二级缓存失效。
- 如果查询语句设置成false，即使用二级缓存，那么如果在数据库中修改了数据，而缓存数据还是原来的，这个时候就会出现脏读。
- flushCache设置如下：

```
<select id="findUserById" parameterType="int"
        resultType="com.lanou.domain.User" useCache="true" flushCache="
true">
        SELECT * FROM user WHERE id = #{id}
</select>
```

Mybatis与Spring整合

1、集成思路

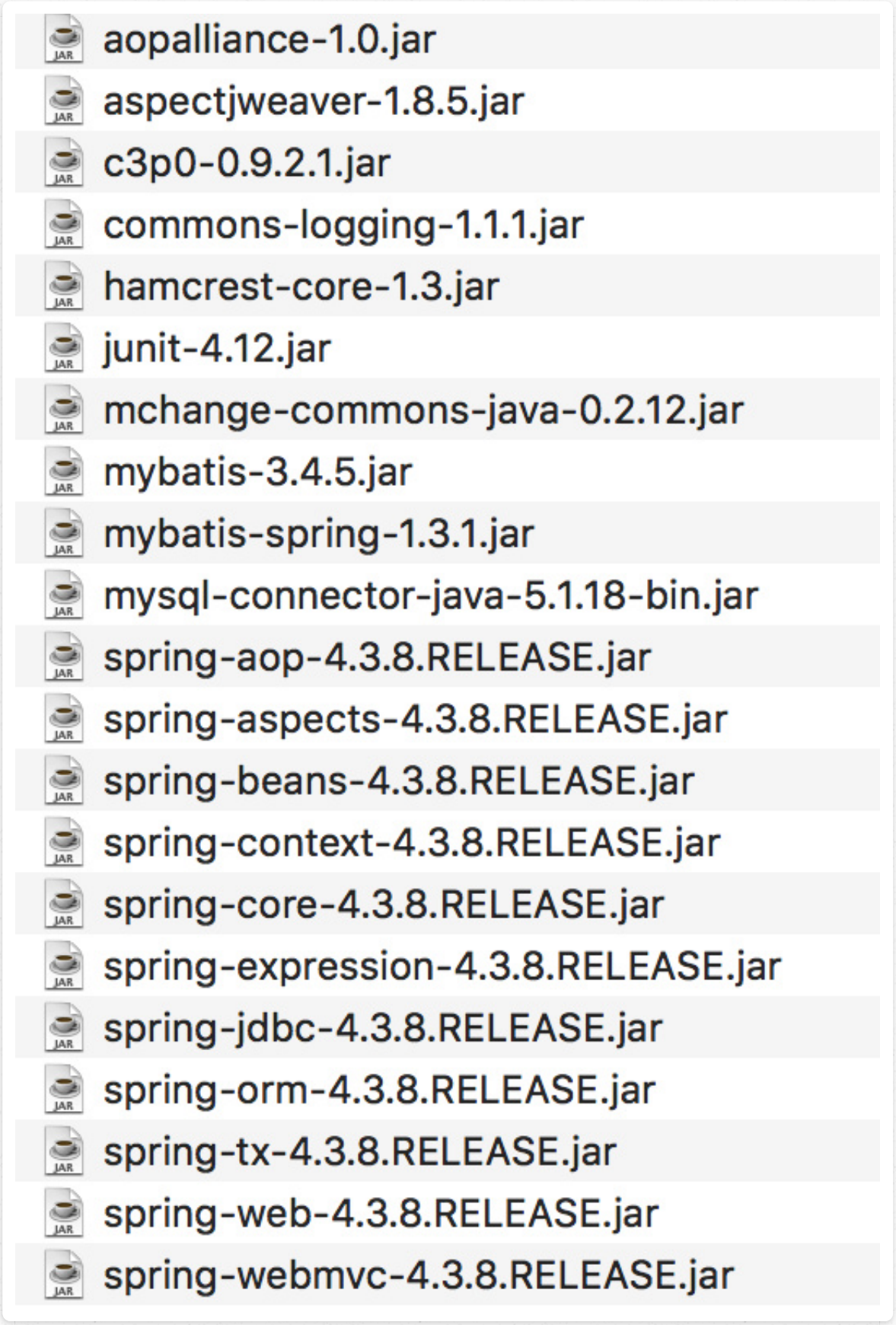
- 需要spring来管理数据源信息。
- 需要spring通过单例方式管理SqlSessionFactory。
- 使用SqlSessionFactory创建SqlSession。（spring和mybatis整合自动完成）
- 持久层的mapper都需要由spring进行管理，spring和mybatis整合生成mapper代理对象。

2、集成步骤

- 2.1、 jar包集成；
- 2.2、 配置文件集成（数据源）；
- 2.3、 SqlSessionFactory集成；
- 2.4、 Mapper接口集成；

3、开始集成

- 需要的jar包



- pom.xml

```
<properties>
        <spring_data_version>4.3.8.RELEASE</spring_data_version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncodi
```

```
ng>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>


        <!-- Mybatis需要的jar包:
        cglib,commons-logging,log4j,ognl,
        log4j-core,javassist,slf4j-api,slf4j-log4j12, -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/cglib/cglib -->
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib</artifactId>
            <version>3.2.5</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/commons-logging/commons
-logging -->
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.2</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/log4j/log4j -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/ognl/ognl -->
        <dependency>
            <groupId>ognl</groupId>
            <artifactId>ognl</artifactId>
            <version>3.1.15</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.logging.log4
j/log4j-core -->
        <dependency>
```

```
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.javassist/javassist
 -->
        <dependency>
            <groupId>org.javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.22.0-CR2</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.25</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.25</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12
 -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.25</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-j
ava -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.18</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springframework/spr
ing-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
```

```
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring_data_version}</version>
        </dependency>

        <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>${spring_data_version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring_data_version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring_data_version}</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.1</version>
    </dependency>

    <dependency>
        <groupId>com.mchange</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.2.1</version>
    </dependency>

    <dependency>
        <groupId>com.mchange</groupId>
        <artifactId>mchange-commons-java</artifactId>
        <version>0.2.3.4</version>
    </dependency>

    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.1</version>
    </dependency>

    <dependency>
        <groupId>aopalliance</groupId>
        <artifactId>aopalliance</artifactId>
        <version>1.0</version>
    </dependency>

    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.5</version>
    </dependency>
```

```
    </dependencies>
```

- 配置文件集成

Mybatis的SqlMapConfig.xml

```
<?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>

    <!-- 设置全局参数 -->
    <settings>
        <!-- lazyLoadingEnabled: 延迟加载的开关，默认是false -->
        <setting name="lazyLoadingEnabled" value="true"/>
        <!-- aggressiveLazyLoading: 默认为true，一旦为true上面的懒加载开关失效
 -->
        <setting name="aggressiveLazyLoading" value="false"/>

        <!-- cacheEnabled: 二级缓存的总开关 默认是false-->
        <setting name="cacheEnabled" value="true"/>
    </settings>

    <!-- 定义别名 -->
    <typeAliases>
        <!-- 批量定义别名 -->
        <!-- name: 指定需要别名定义的包的名称 它的别名就是类名（类名的首字母大小写
都可）-->
        <package name="com.lanou"></package>
    </typeAliases>

    <!-- 注意: 与spring集成后，数据源和事务交给spring来管理 -->

    <!-- 加载mapper文件 -->
    <mappers>
        <mapper resource="UserMapper.xml"/>
    </mappers>
</configuration>
```

Spring的spring-config.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- 引用java配置文件 -->
    <context:property-placeholder location="db.properties"/>

    <!-- 1，配置数据源，使用dbcp连接池 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}"/>
        <property name="jdbcUrl" value="${db.url}"/>
        <property name="user" value="${db.username}"/>
        <property name="password" value="${db.password}"/>

        <property name="initialPoolSize" value="${db.initialPoolSize}"/>

        <!--最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。Default: 0 -->
        <property name="maxIdleTime" value="${db.maxIdleTime}"/>
        <!--连接池中保留的最大连接数。Default: 15 -->
        <property name="maxPoolSize" value="${db.maxPoolSize}"/>
        <property name="minPoolSize" value="${db.minPoolSize}"/>

        <!--当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。Default: 3 -->
        <property name="acquireIncrement" value="${db.acquireIncrement}"/>
        <!--两次连接中间隔时间，单位毫秒。Default: 1000 -->
        <property name="acquireRetryDelay" value="${db.acquireRetryDelay}"/>
        <!--定义在从数据库获取新连接失败后重复尝试的次数。Default: 30 -->
        <property name="acquireRetryAttempts" value="${db.acquireRetryAttempts}"/>
        <!--获取连接失败将会引起所有等待连接池来获取连接的线程抛出异常。
            但是数据源仍有效保留，并在下次调用getConnection()的时候继续尝试获取连接。
            如果设为true，那么在尝试
            获取连接失败后该数据源将申明已断开并永久关闭。Default: false -->
        <property name="breakAfterAcquireFailure" value="${db.breakAfterAcquireFailure}"/>
    </bean>

    <!--2，Srping对sqlSessionFactory进行管理配置-->
```

```
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFa
ctoryBean">
        <!-- mybatis的配置文件路径 -->
        <property name="configLocation" value="classpath:SqlMapConfig.x
ml"/>
        <!-- SqlSessionFactory需要数据源信息，之前是写在sqlmapconfig.xml，
        现在需要重新指定 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--3，配置事务对象（切面）-->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransact
ionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--4，配置事务属性-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="add*"/>
            <tx:method name="delete*"/>
            <tx:method name="update*"/>
            <tx:method name="find*"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!--5,配置spring aop-->
    <aop:config>
        <!--配置切入点-->
        <aop:pointcut id="pointcut"
                      expression="execution(* com.lanou.dao.impl.*.*(..
))"/>
        <!--配置通知-->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"/>

    </aop:config>

    <!--由spring管理原始dao的实际-->
    <!--方式一-->
    <!--<bean id="userDao"-->
    <!--class="com.lanou.dao.impl.UserDaoImpl">-->
    <!--<property name="sqlSessionFactory" ref="sqlSessionFactory"/>-->
    <!--</bean>-->

    <!--方式二-->
    <!--mapper代理开发方式之批量mapper配置
    1,通过MapperScannerConfigurer创建代理对象，这样就不不用定义那么dao的接口对象
了
    2,bean的名字默认为mapper接口类名的首字母小写-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
```

```
        <!--指定批量mapper（dao）配置的包名-->
        <property name="basePackage" value="com.lanou.dao"/>
        <!--指定使用的sqlSessionFactory-->
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFac
tory"/>
    </bean>
</beans>
```

Spring配置文件中的属性文件db.properties

```
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/mybatis
db.username=root
db.password=1
#
db.initialPoolSize=20
db.maxIdleTime=60
db.maxPoolSize=200
db.minPoolSize=50
#
db.acquireIncrement=3
db.acquireRetryDelay=1000
db.acquireRetryAttempts=30
db.breakAfterAcquireFailure=false
```

Mybatis映射文件UserMapper.xml

```
<?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：此时用mapper代理方式，它的值必须等于对应mapper接口的全限定名  --
>
<mapper namespace="com.lanou.dao.UserDao">

    <!--根据用户id来查询用户信息-->
    <select id="findUserById" parameterType="int"
            resultType="com.lanou.domain.User">
        SELECT * FROM USER WHERE id = #{id}
    </select>
</mapper>
```

dao层的接口对象

```java
public interface UserDao {
    // 1、 根据用户ID来查询用户信息;
    User findUserById(int id);

    // 2、 根据用户名称来模糊查询用户信息列表;
    List<User> findUsersByName(String name);

    // 3、 添加用户;
    void insertUser(User user);
}

//测试类
public class UserDaoTes {

    private ApplicationContext context;

    @Before
    public void init() {
        //读取spring的上下文，然后封装到ctx
        context = new ClassPathXmlApplicationContext(
                "spring-config.xml");
    }

    @Test
    public void testFindUserById() {
        //创建userdao对象
        UserDao userDao = (UserDao) context.getBean("userDao");
        //调用userdao对象的方法
        User user = userDao.findUserById(1);
        System.out.println(user);
    }
}
```

Mybatis的逆向工程（自学）

1、什么是逆向工程

- 简单点说，就是通过数据库中的单表，自动生成java代码。
- Mybatis官方提供了逆向工程，可以针对单表自动生成mybatis代码（mapper.java\mapper.xml\po类）
- 企业开发中，逆向工程是个很常用的工具。

2、下载逆向工程

- 在Pom.xml文件中添加 mybatis-generator-1.3.2依赖

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.lanou</groupId>
    <artifactId>MyBatisGenerator-02</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>MyBatisGenerator-02 Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <finalName>MyBatisGenerator-02</finalName>
        <plugins>
            <plugin>
                <groupId>org.mybatis.generator</groupId>
                <artifactId>mybatis-generator-maven-plugin</artifactId>
                <version>1.3.2</version>
                <configuration>
                    <verbose>true</verbose>
                    <overwrite>true</overwrite>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

- 添加generator.properties属性文件

```
jdbc.driverLocation=/Users/lilyxiao/workspaceweb-ssm/MyBatisGenerator02/lib/mysql-connector-java-5.1.18-bin.jar
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis
jdbc.username=root
jdbc.password=1
```

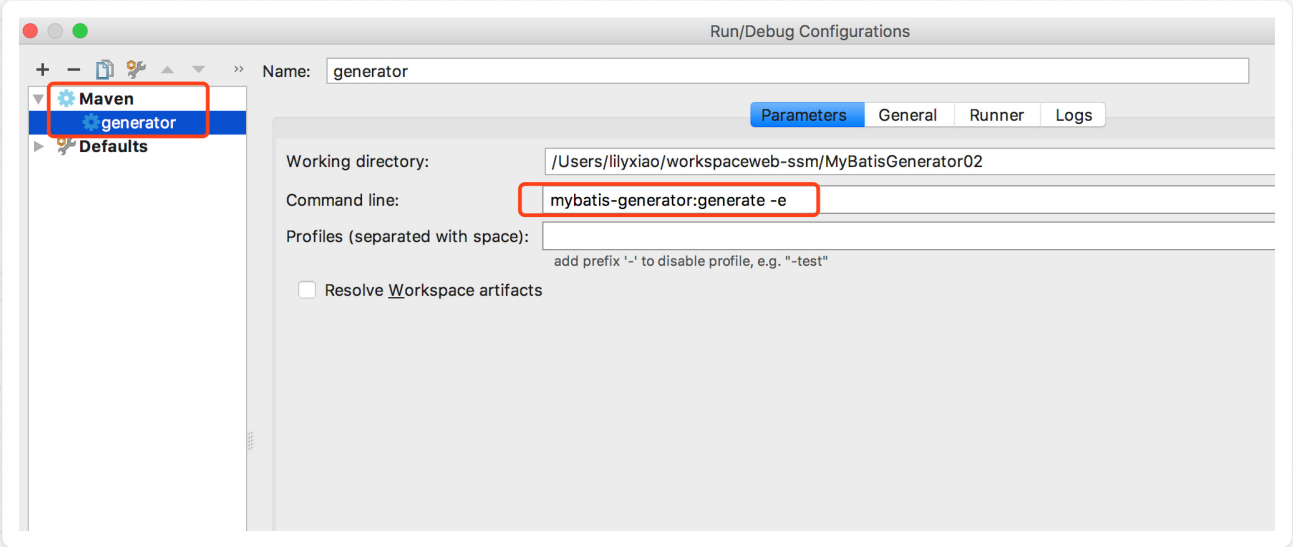
- 添加generatorConfig.xml

```
<?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>
    <!--导入属性配置-->
    <properties resource="generator.properties"/>

    <!--指定特定数据库的jdbc驱动jar包的位置-->
    <classPathEntry location="${jdbc.driverLocation}"/>

    <context id="default" targetRuntime="MyBatis3">

        <!-- optional，旨在创建class时，对注释进行控制 -->
        <commentGenerator>
            <property name="suppressDate" value="true"/>
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>

        <!--jdbc的数据库连接 -->
        <jdbcConnection
                driverClass="${jdbc.driver}"
                connectionURL="${jdbc.url}"
                userId="${jdbc.username}"
                password="${jdbc.password}">
        </jdbcConnection>


        <!-- 非必需，类型处理器，在数据库类型和java类型之间的转换控制-->
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>


        <!-- Model模型生成器,用来生成含有主键key的类，记录类 以及查询Example类
            targetPackage     指定生成的model生成所在的包名
            targetProject     指定在该项目下所在的路径
        -->
        <javaModelGenerator targetPackage="com.lanou.domain"
                            targetProject="src/main/java">

            <!-- 是否允许子包，即targetPackage.schemaName.tableName -->
            <property name="enableSubPackages" value="false"/>
            <!-- 是否对model添加 构造函数 -->
            <property name="constructorBased" value="true"/>
            <!-- 是否对类CHAR类型的列的数据进行trim操作 -->
            <property name="trimStrings" value="true"/>
```

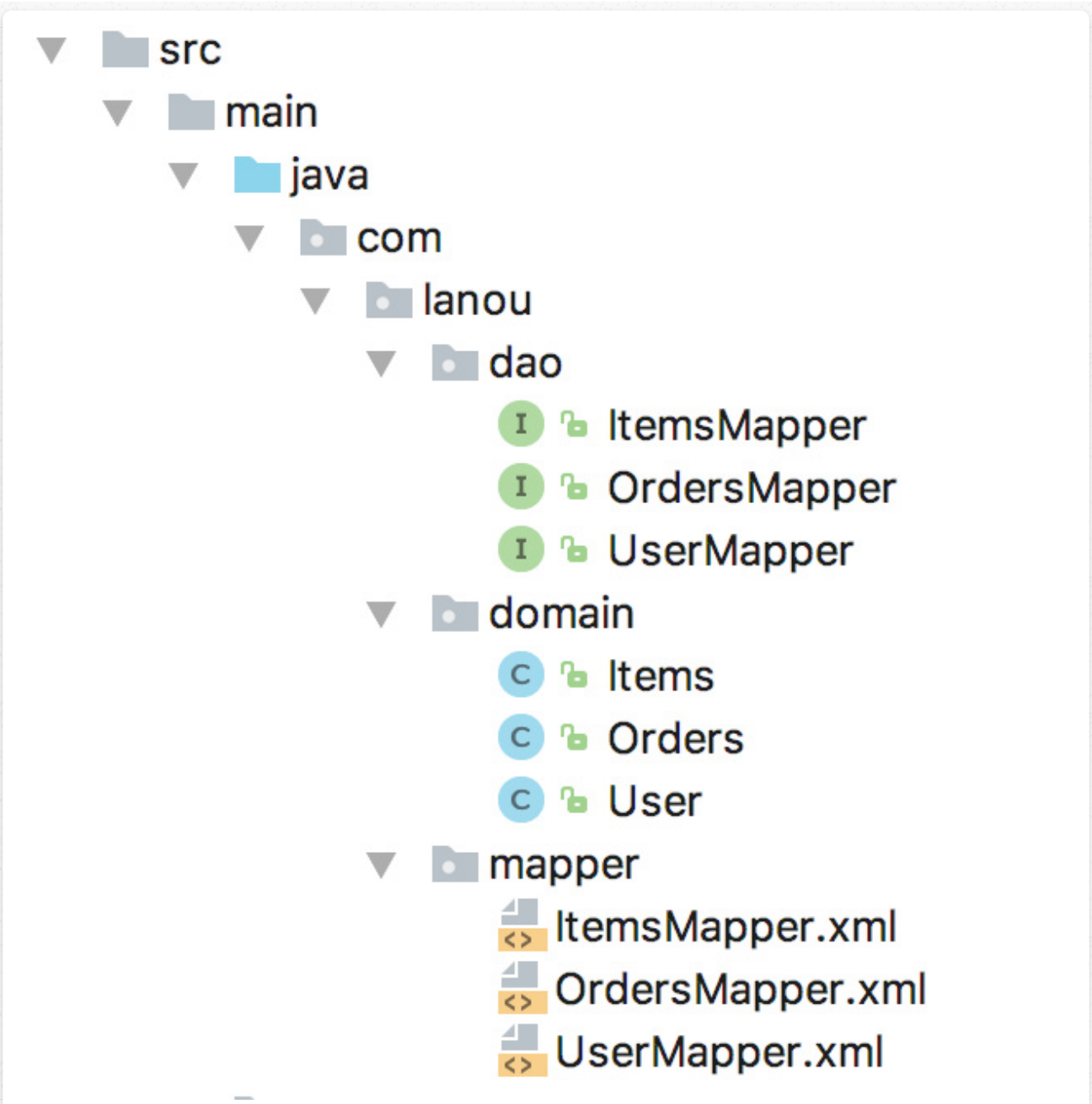
```
            <!-- 建立的Model对象是否 不可改变  即生成的Model对象不会有 setter方
法，只有构造方法 -->
            <property name="immutable" value="false"/>
        </javaModelGenerator>

        <!--Mapper映射文件生成所在的目录 为每一个数据库的表生成对应的SqlMap文件 -
->
        <sqlMapGenerator targetPackage="com.lanou.mapper"
                         targetProject="src/main/java">
            <property name="enableSubPackages" value="false"/>
        </sqlMapGenerator>

        <!-- 客户端代码，生成易于使用的针对Model对象和XML配置文件 的代码
                type="ANNOTATEDMAPPER",生成Java Model 和基于注解的Mapper对
象
                type="MIXEDMAPPER",生成基于注解的Java Model 和相应的Mapper对
象
                type="XMLMAPPER",生成SQLMap XML文件和独立的Mapper接口
        -->
        <javaClientGenerator targetPackage="com.lanou.dao"
                             targetProject="src/main/java"
                             type="XMLMAPPER">
            <property name="enableSubPackages" value="true"/>
        </javaClientGenerator>


        <!--数据库中的表对应关系-->
        <table tableName="User" domainObjectName="User"
               enableCountByExample="false" enableUpdateByExample="fals
e"
               enableDeleteByExample="false" enableSelectByExample="fal
se"
               selectByExampleQueryId="false">
        </table>

        <table tableName="Items" domainObjectName="Items"
               enableCountByExample="false" enableUpdateByExample="fals
e"
               enableDeleteByExample="false" enableSelectByExample="fal
se"
               selectByExampleQueryId="false">
        </table>
        <table tableName="Orders" domainObjectName="Orders"
               enableCountByExample="false" enableUpdateByExample="fals
e"
               enableDeleteByExample="false" enableSelectByExample="fal
se"
               selectByExampleQueryId="false">
        </table>
        <table tableName="OrdersDetail" domainObjectName="OrdersDetail"
               enableCountByExample="false" enableUpdateByExample="fals
e"
```

```
                enableDeleteByExample="false" enableSelectByExample="fal
se"
                selectByExampleQueryId="false">
        </table>
    </context>
</generatorConfiguration>
```

- 利用maven运行mybatis-generator-maven-plugin插件任务，注意里面配置的命令为：`mybatis-generator:generate -e`



- 运行generator任务，生成结果如下：



- 参考网址：http://blog.csdn.net/for_my_life/article/details/51228098