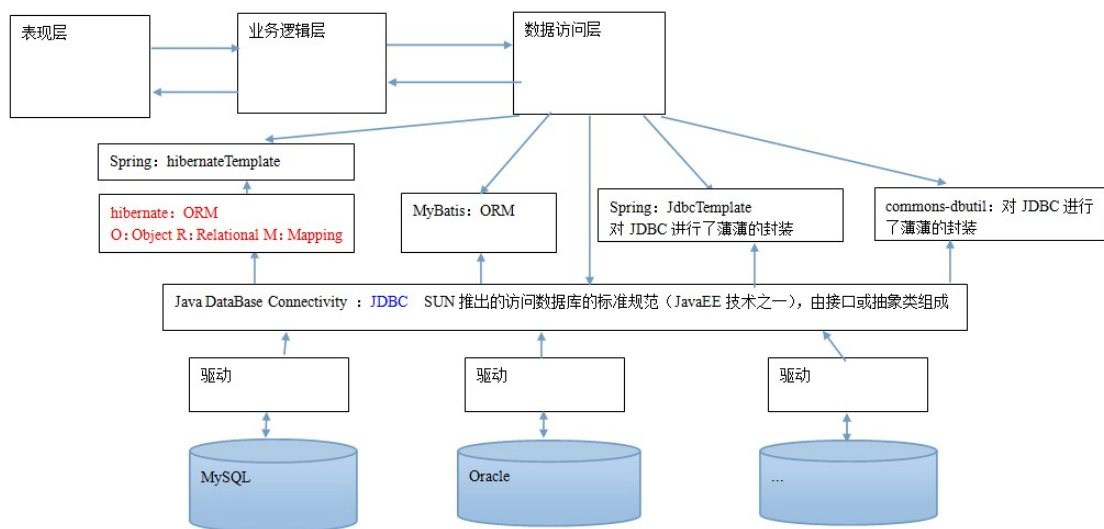


mybatis 是持久层的框架，持久层总图如下：



搭建 Mybatis 开发环境

首先maven工程报错了，那么可能是仓库没配好，看：<https://www.cnblogs.com/phpdragon/p/7216626.html>

环境

- JDK1.8
- Mysql 5.7
- maven 3.6.1

回顾

- JDBC
 - MYSQL
 - JAVA基础
 - Maven
 - Junit
-

MyBatis 讲解

- 优秀的持久层框架。
- 可以使用 XML 或注解来配置
- 几乎避免了所有的代码
- 本来是 apache 的一个开源项目 iBatis ,2010年由 apache software foundation 迁移到了 google code , 并且修改名字为 mybatis

获取 Mybatis

- Maven
- Github

进行基本的环境搭建，跑个流程

1. 首先打开idea选择maven工程，不需要create，直接点击下一步，填写好名字

New Project

GroupId: ☒ Inherit

ArtifactId:

Version: ☒ Inherit

Previous Next Cancel Help

New Project

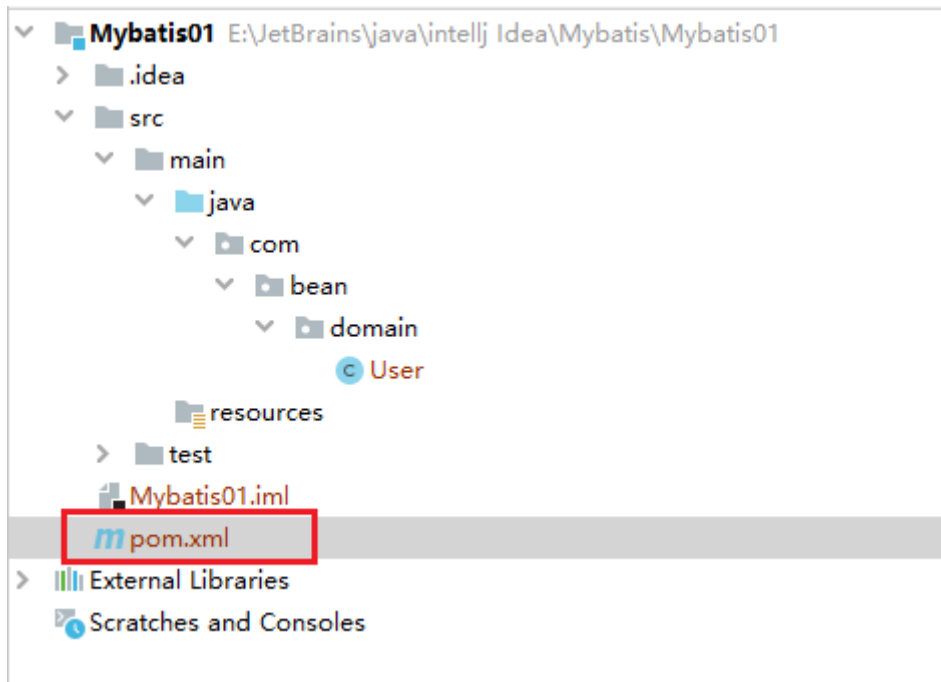
Project name:

Project location: ...

More Settings

Previous Finish Cancel Help

2. 填写配置文件



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.bean</groupId>
8     <artifactId>Mybatis01</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <!-- 设定导出文件格式-->
11    <packaging>jar</packaging>
12
13    <!-- 导入包-->
14    <dependencies>
15        <dependency>
16            <groupId>org.mybatis</groupId>
17            <artifactId>mybatis</artifactId>
18            <version>3.4.5</version>
19        </dependency>
20        <dependency>
21            <groupId>junit</groupId>
22            <artifactId>junit</artifactId>
23            <version>4.10</version>
24            <scope>test</scope>
25        </dependency>
26        <dependency>
27            <groupId>mysql</groupId>
28            <artifactId>mysql-connector-java</artifactId>
29            <version>5.1.6</version>
30            <scope>runtime</scope>
31        </dependency>
32        <dependency>
33            <groupId>log4j</groupId>
34            <artifactId>log4j</artifactId>
35            <version>1.2.12</version>
36        </dependency>
37    </dependencies>
```

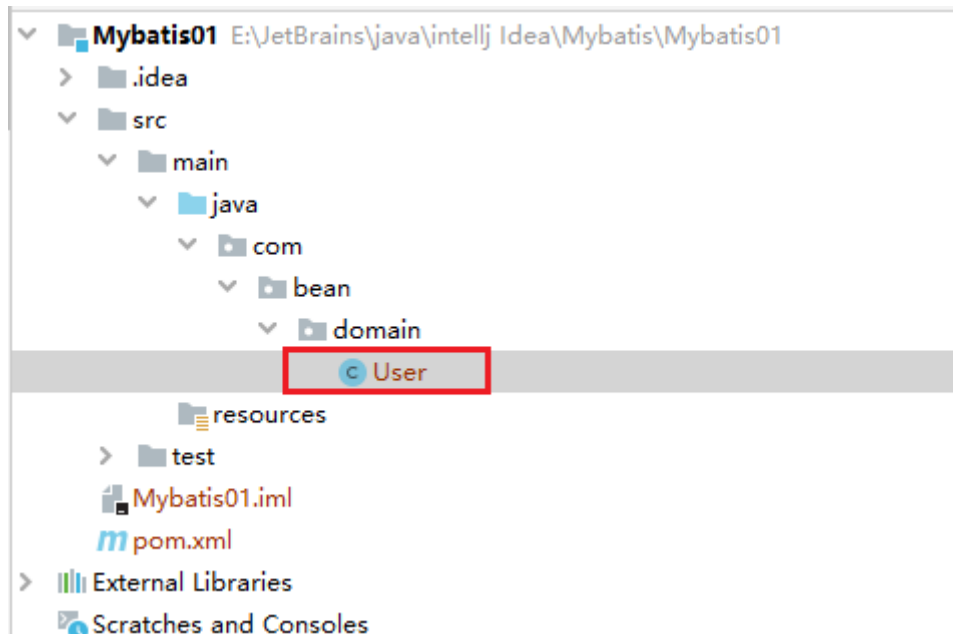
```
38
39 </project>
```

3. 数据库建表

```
1 CREATE DATABASE Mybatis;
2
3 use Mybatis;
4
5 DROP TABLE IF EXISTS `user`;
6
7 CREATE TABLE `user` (
8     `id` int(11) NOT NULL auto_increment,
9     `username` varchar(32) NOT NULL COMMENT '用户名称',
10    `birthday` datetime default NULL COMMENT '生日',
11    `sex` char(1) default NULL COMMENT '性别',
12    `address` varchar(256) default NULL COMMENT '地址',
13    PRIMARY KEY (`id`)
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
15
16
17
18 insert into `user`(`id`,`username`,`birthday`,`sex`,`address`) values (41,'老王','2018-02-
19 27 17:47:08','男','北京'),(42,'小二王','2018-03-02 15:09:37','女','北京金燕龙'),(43,'小二
    王','2018-03-04 11:34:34','女','北京金燕龙'),(45,'传智播客','2018-03-04 12:04:06','男','北京金燕
    龙'),(46,'老王','2018-03-07 17:37:26','男','北京'),(48,'小马宝莉','2018-03-08 11:44:00','女','北
    京修正');
```

	id	username	birthday	sex	address
▶	41	老王	2018-02-27 17:47:08	男	北京
	42	小二王	2018-03-02 15:09:37	女	北京金燕龙
	43	小二王	2018-03-04 11:34:34	女	北京金燕龙
	45	传智播客	2018-03-04 12:04:06	男	北京金燕龙
	46	老王	2018-03-07 17:37:26	男	北京
	48	小马宝莉	2018-03-08 11:44:00	女	北京修正

4. 建立domain, 将表与类对应



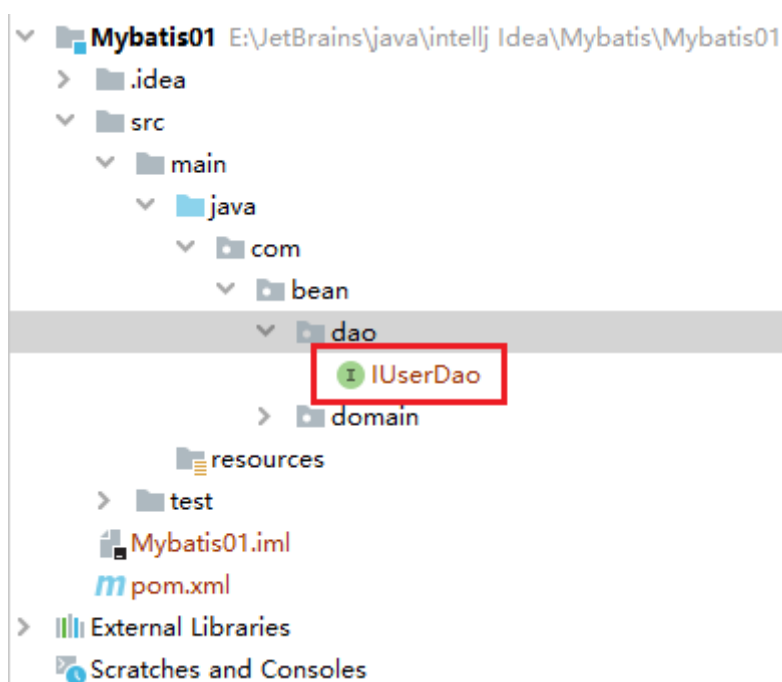
```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.Date;
5
6 public class User implements Serializable {
7
8     private Integer id;
9     private String username;
10    private Date birthday;
11    private String sex;
12    private String address;
13
14    public User(Integer id, String username, Date birthday, String sex, String address) {
15        this.id = id;
16        this.username = username;
17        this.birthday = birthday;
18        this.sex = sex;
19        this.address = address;
20    }
21
22    public User() {
23    }
24
25    public Integer getId() {
26        return id;
27    }
28
29    public void setId(Integer id) {
30        this.id = id;
31    }
32
33    public String getUsername() {
34        return username;
35    }
36
37    public void setUsername(String username) {
38        this.username = username;
39    }
}
```

```

40
41     public Date getBirthday() {
42         return birthday;
43     }
44
45     public void setBirthday(Date birthday) {
46         this.birthday = birthday;
47     }
48
49     public String getSex() {
50         return sex;
51     }
52
53     public void setSex(String sex) {
54         this.sex = sex;
55     }
56
57     public String getAddress() {
58         return address;
59     }
60
61     public void setAddress(String address) {
62         this.address = address;
63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "id=" + id +
69             ", username='" + username + '\'' +
70             ", birthday=" + birthday +
71             ", sex='" + sex + '\'' +
72             ", address='" + address + '\'' +
73             '}';
74     }
75 }

```

5. 编写持久层接口 IUserDao

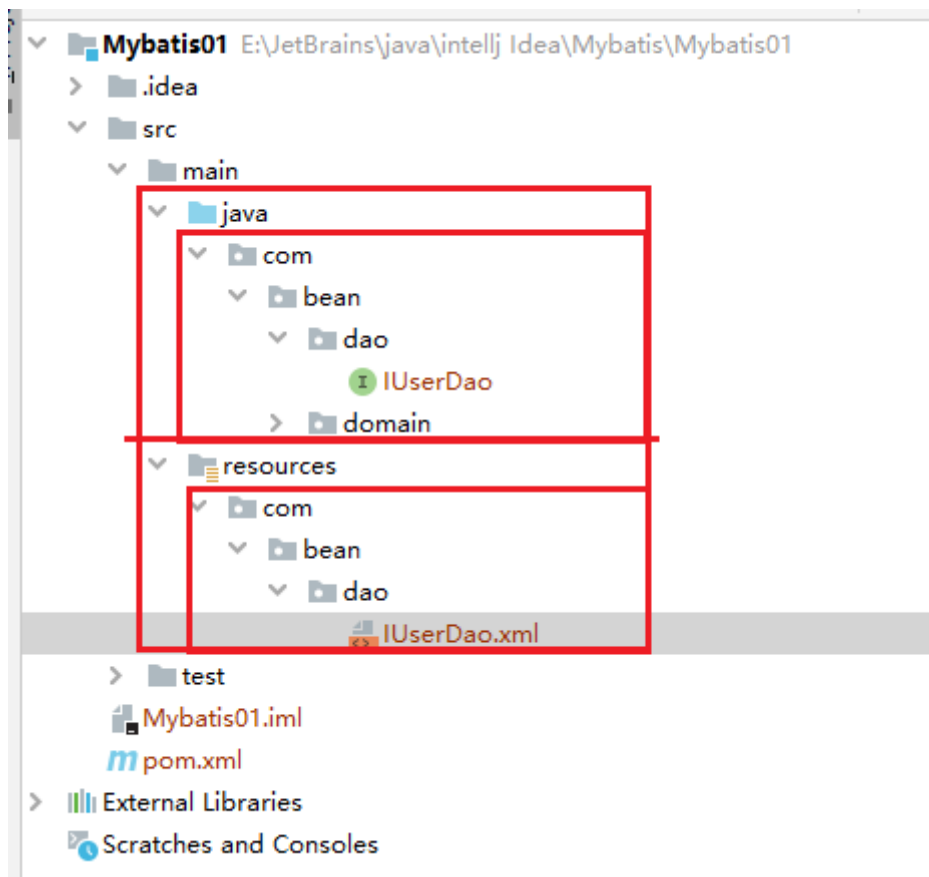


```

1 package com.bean.dao;
2
3 import com.bean.domain.User;
4
5 import java.util.List;
6
7 /**
8  * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
9  */
10 public interface IUserDao {
11
12     /**
13      * 查询所有用户
14      * @return List<User>
15      */
16     List<User> findAll();
17 }

```

6. 编写持久层接口的映射文件 IUserDao.xml



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <!-- 上面的配置一定是要写上的-->
6
7 <!--配置mapper-->
8 <mapper namespace="com.bean.dao.IUserDao">
9     <!--
10         这个id不能瞎写，是IUserDao里面的方法名
11         这个resultType是表对应的类，结果类型
12         里面的语句是要执行的语句
13     -->

```

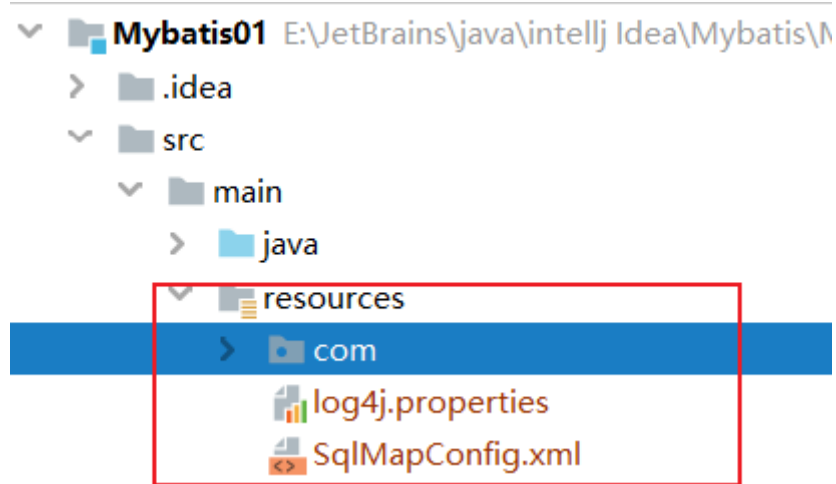


```

14     <select id="findAll" resultType="com.bean.domain.User">
15         select * from user
16     </select>
17 </mapper>

```

7. 编写 SqlMapConfig.xml 配置文件



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5  <!--注意上面的配置-->
6
7  <!--这个文件名字叫什么无所谓，但是配置的是什么很重要，配置的是啥呢，配置的是mybatis的环境-->
8
9  <configuration>
10     <!-- 首先来配置mybatis的环境，叫不叫mysql无所谓，但是下面必须对这个名字解释-->
11     <environments default="mysql">
12         <!-- 配置mysql的环境，对mysql进行解释 -->
13         <environment id="mysql">
14             <!--配置事务的类型-->
15             <transactionManager type="JDBC"></transactionManager>
16             <!--配置连接数据库的信息，用的是数据源(连接池)，
17             这里的driver, url, username, password应该都熟悉吧
18             -->
19             <dataSource type="POOLED">
20                 <property name="driver" value="com.mysql.jdbc.Driver"/>
21                 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
22                 <property name="username" value="root"/>
23                 <property name="password" value="root"/>
24             </dataSource>
25         </environment>
26     </environments>
27
28     <!--接下来指定映射配置文件的位置，映射配置文件是指的每个dao的独立的配置文件-->
29     <mappers>
30         <mapper resource="com/bean/dao/IUserDao.xml"/>
31     </mappers>
32 </configuration>
33

```

注意一件事：<mapper></mapper> 的配置可以是 resource ， 可以是 class

- 一般我们使用 `resource` 来进行配置的时候使用的是 `xml` 文档
- 使用 `class` 进行配置的时候使用的是注解开发

环境搭建的注意事项

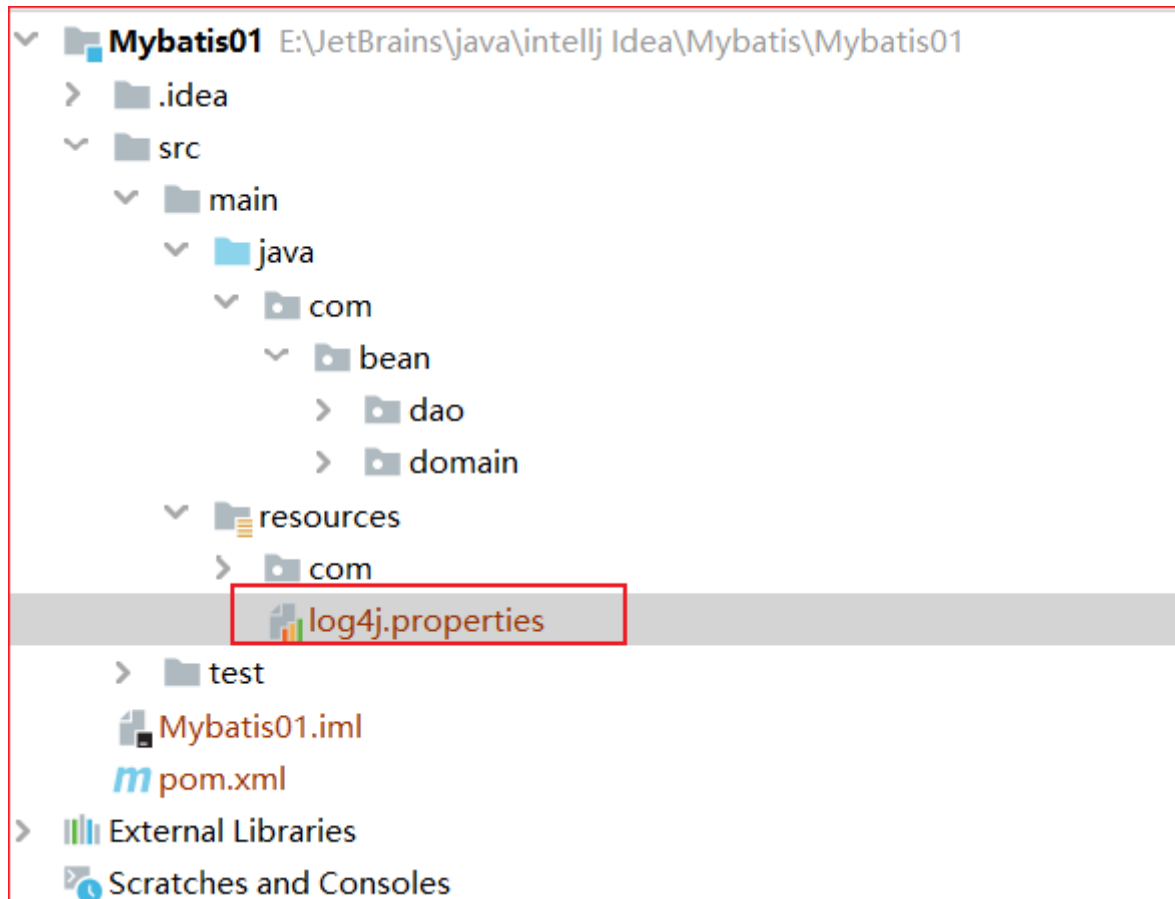
1. 在 `Mybatis` 中的 `dao` 层主要叫做 `Mapper` , 所以这里的 `IUserDao` 也叫做 `IUserMapper`
2. idea中创建目录和包是不一样的
 1. 包(Package): `com.bean.dao` 是三级结构
 2. 目录(Directory): `com.bean.dao` 是一级目录
3. `mybatis` 的映射配置文件位置必须和 `dao` 接口的包结构相同, 也就是上面第六步的注意事项
4. 映射配置文件的 `mapper` 标签 `namespace` 属性的取值必须为 `dao` 接口的全限定类名
5. 映射配置文件的操作配置(select), id属性必须是 `dao` 接口的方法名

当我们在遵从了3, 4, 5点之后, 我们在开发中就无需写 `dao` 的实现类, 也就是说写完接口我们的操作就结束了

Mybatis 的入门

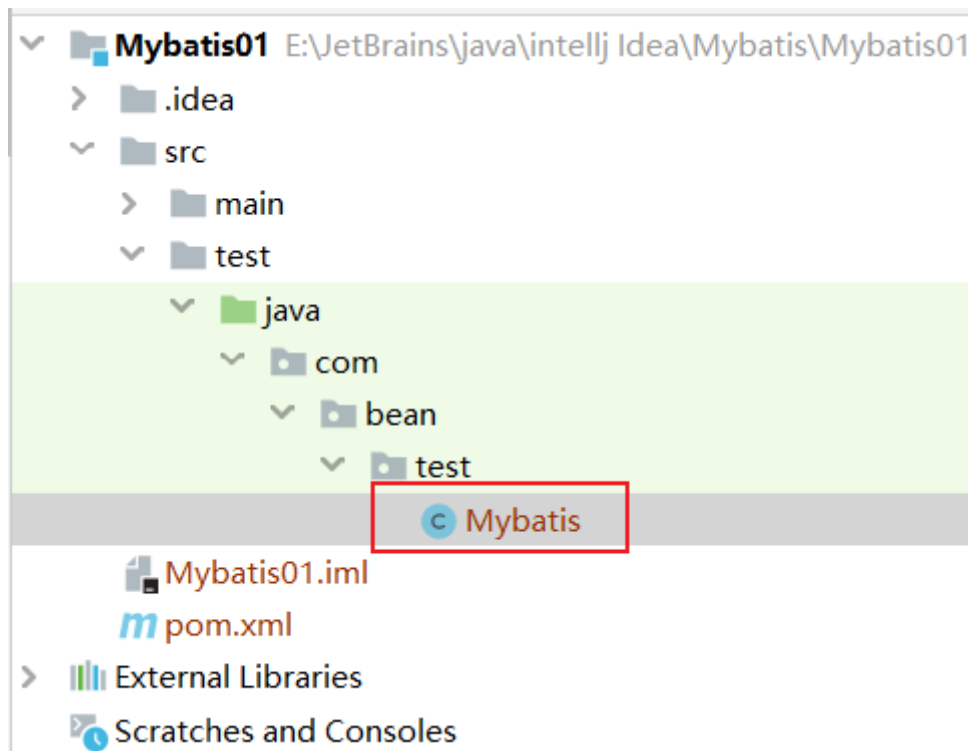
1. 首先往项目里拷一个文件: `log4j.properties` , 先拷下来, 其他的再说

读取配置文件用到的技术就是解析xml的技术, 这里用的就是log4j解析xml技术



```
1  # Set root category priority to INFO and its only appender to CONSOLE.
2  #log4j.rootCategory=INFO, CONSOLE          debug   info   warn error fatal
3  log4j.rootCategory=debug, CONSOLE, LOGFILE
4
5  # Set the enterprise logger category to FATAL and its only appender to CONSOLE.
6  log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE
7
8  # CONSOLE is set to be a ConsoleAppender using a PatternLayout.
9  log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
10 log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
11 log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x -
    %m\n
12
13 # LOGFILE is set to be a File appender using a PatternLayout.
14 log4j.appender.LOGFILE=org.apache.log4j.FileAppender
15 log4j.appender.LOGFILE.File=d:\axis.log
16 log4j.appender.LOGFILE.Append=true
17 log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
18 log4j.appender.LOGFILE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x -
    %m\n
```

2. 在Test中建立一个测试类，叫做 `MybatisTest.class`



3. 走过流程

因为我们要做的是不写实现类实现接口，所以先把流程走一遍

```
1 package com.bean.test;
2
3 public class Mybatis {
4     /**
5      * 入门案例
6      * @param args
7      */
8     public static void main(String[] args) {
9         //1. 读取配置文件
10        //2. 创建sqlSessionFactory工厂
11        //3. 使用工厂生产SqlSession对象
12        //4. 使用SqlSession创建Dao接口的代理对象
13        //5. 使用代理对象执行方法
14        //6. 释放资源
15    }
16 }
```

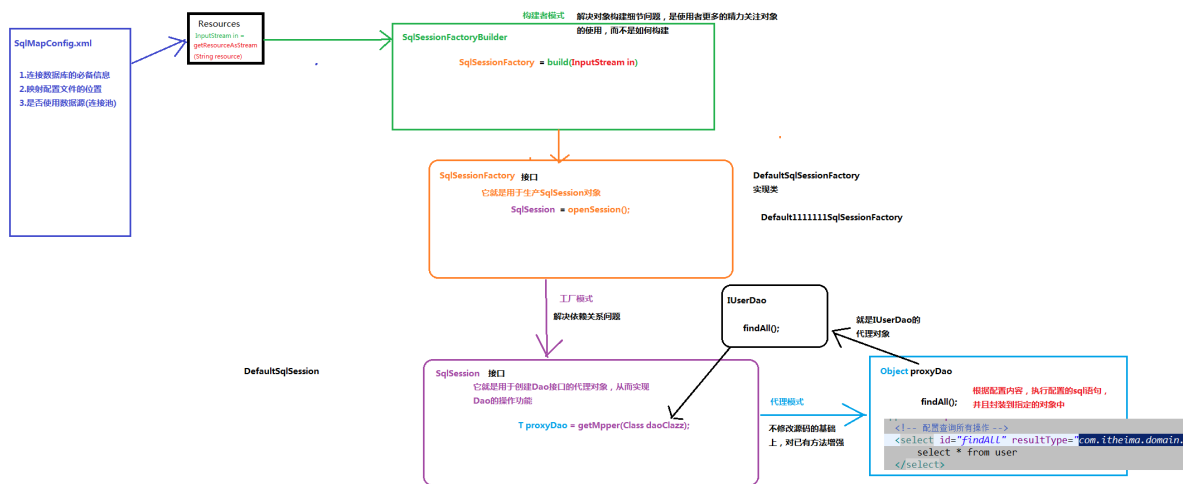
4. 编写代码

```
1 package com.bean.test;
2
3 import com.bean.dao.IUserDao;
4 import com.bean.domain.User;
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.util.List;
13
```

```

14     public class Mybatis {
15         /**
16          * 入门案例
17          * @param args
18          */
19         public static void main(String[] args) throws IOException {
20             //1. 读取配置文件，这里读取的文件就是刚才mybatis的配置文件
21             InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
22
23             //2. 创建sqlSessionFactory工厂，这个东西是一个接口，所以我们找他的实现类
24             SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
25             SqlSessionFactory factory = builder.build(in);
26
27             //3. 使用工厂生产SqlSession对象
28             SqlSession session = factory.openSession();
29
30             //4. 使用SqlSession创建Dao接口的代理对象
31             IUserDao userDao = session.getMapper(IUserDao.class);
32
33             //5. 使用代理对象执行方法
34             List<User> users = userDao.findAll();
35             for (User user : users) {
36                 System.out.println(user);
37             }
38             //6. 释放资源
39             session.close();
40             in.close();
41         }
42     }

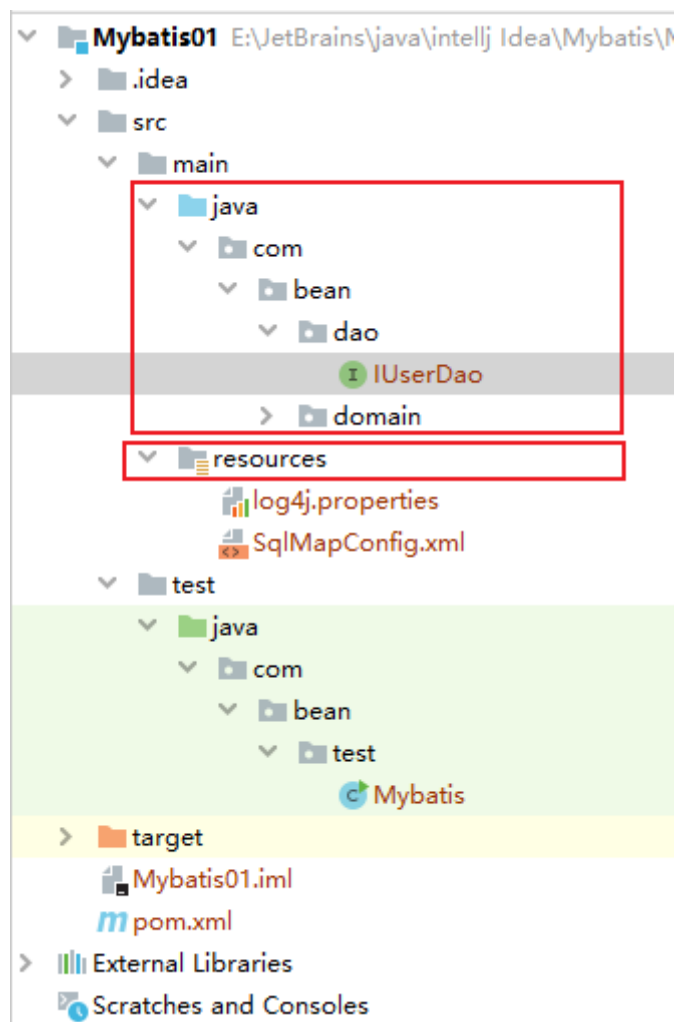
```



使用注解代替持久层接口的映射文件

- 使用注解可以代替 `IUserDao.xml`，也就是环境搭建的第6步，替换IUserDao.xml

1. 把IUserDao.xml删掉



2. 在 `IUserDao` 接口中的语句中编写注解



3. 在 `SqlMapConfig.xml` 中更改 `IUserDao.xml` 的映射，由 `resource` 改为 `class`

```
6
7 <!-- 这个文件名字叫什么无所谓，但是配置的是啥呢，配置的是mybatis的环境-->
8
9 <configuration>
10 <!-- 首先来配置mybatis的环境，叫不叫mysql无所谓，但是下面必须对这个名字解释-->
11 <environments default="mysql">
12 <!-- 配置mysql的环境，对mysql进行解释 -->
13 <environment id="mysql">
14 <!-- 配置事务的类型-->
15 <transactionManager type="JDBC"></transactionManager>
16 <!-- 配置连接数据库的信息，用的是数据源(连接池)，
17 这里的driver, url, username, password应该都熟悉吧
18 -->
19 <dataSource type="POOLED">
20 <property name="driver" value="com.mysql.jdbc.Driver"/>
21 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
22 <property name="username" value="root"/>
23 <property name="password" value="root"/>
24 </dataSource>
25 </environment>
26 </environments>
27
28
29 <!-- 接下来指定映射配置文件的位置，映射配置文件是指的每个dao的独立的配置文件-->
30 <mappers>
31 <mapper resource="com/bean/dao/IUserDao.xml"/>
32 </mappers>
33 </configuration>
```

```
6
7 <!-- 这个文件名字叫什么无所谓，但是配置的是啥呢，配置的是mybatis的环境-->
8
9 <configuration>
10 <!-- 首先来配置mybatis的环境，叫不叫mysql无所谓，但是下面必须对这个名字解释-->
11 <environments default="mysql">
12 <!-- 配置mysql的环境，对mysql进行解释 -->
13 <environment id="mysql">
14 <!-- 配置事务的类型-->
15 <transactionManager type="JDBC"></transactionManager>
16 <!-- 配置连接数据库的信息，用的是数据源(连接池)，
17 这里的driver, url, username, password应该都熟悉吧
18 -->
19 <dataSource type="POOLED">
20 <property name="driver" value="com.mysql.jdbc.Driver"/>
21 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
22 <property name="username" value="root"/>
23 <property name="password" value="root"/>
24 </dataSource>
25 </environment>
26 </environments>
27
28
29 <!-- 接下来指定映射配置文件的位置，映射配置文件是指的每个dao的独立的配置文件-->
30 <mappers>
31 <mapper class="com.bean.dao.IUserDao"/>
32 </mappers>
33 </configuration>
```

mybatis 入门案例中的设计模式分析

```
1 package com.bean.test;
2
3 import com.bean.dao.IUserDao;
4 import com.bean.domain.User;
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.util.List;
13
14 public class Mybatis {
15     /**
16      * 入门案例
17      * @param args
```

```

18      */
19      public static void main(String[] args) throws IOException {
20          //1. 读取配置文件，这里读取的文件就是刚才mybatis的配置文件
21          InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
22
23          //2. 创建sqlSessionFactory工厂，这个东西是一个接口，所以我们找他的实现类
24          SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
25          SqlSessionFactory factory = builder.build(in);
26
27          //3. 使用工厂生产SqlSession对象
28          SqlSession session = factory.openSession();
29
30          //4. 使用SqlSession创建Dao接口的代理对象
31          IUserDao userDao = session.getMapper(IUserDao.class);
32
33          //5. 使用代理对象执行方法
34          List<User> users = userDao.findAll();
35          for (User user : users) {
36              System.out.println(user);
37          }
38          //6. 释放资源
39          session.close();
40          in.close();
41      }
42  }

```

```

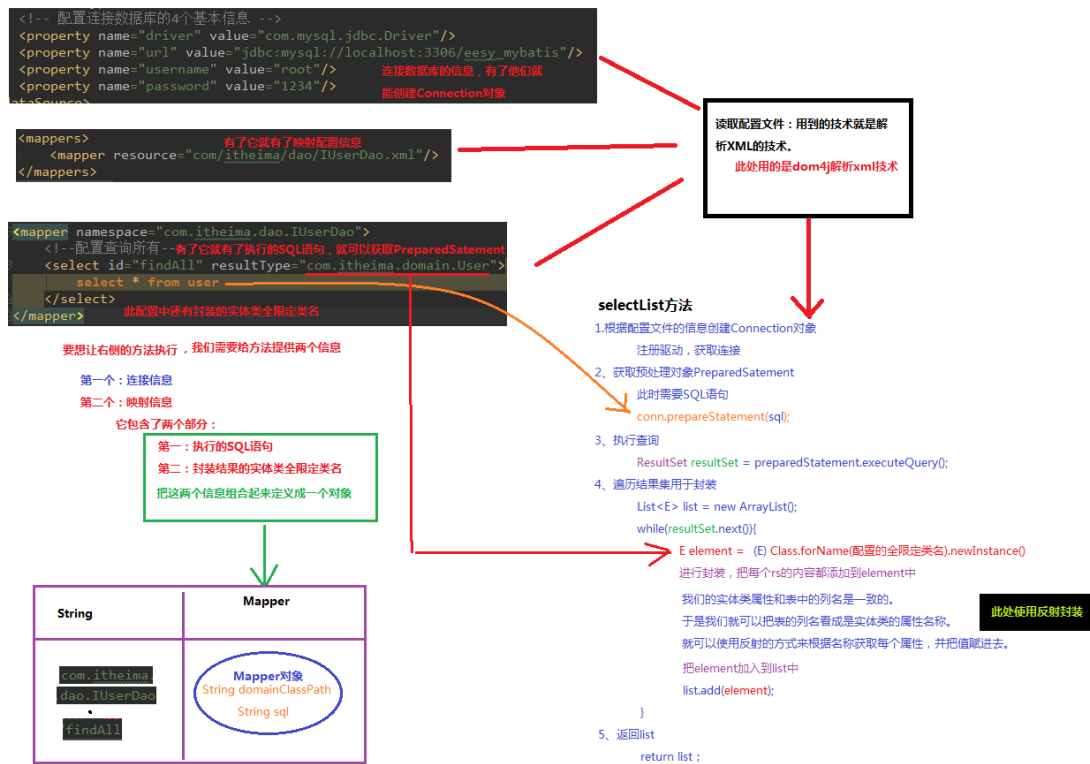
public static void main(String[] args) throws Exception {
    //1. 读取配置文件
    InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2. 创建SqlSessionFactory工厂
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    SqlSessionFactory factory = builder.build(in); builder就是构建者
    //3. 使用工厂生产SqlSession对象
    SqlSession session = factory.openSession();
    //4. 使用SqlSession创建Dao接口的代理对象
    IUserDao userDao = session.getMapper(IUserDao.class);
    //5. 使用代理对象执行方法
    List<User> users = userDao.findAll();
    for (User user : users){
        System.out.println(user);
    }
    //6. 释放资源
    session.close();
    in.close();
}

```

绝对路径: d:/xxx/xxx.xml ✗ 第一个: 使用类加载器。它只能读取类路径的配置文件
 相对路径: src/java/main/xxx.xml ✗ 第二个: 使用ServletContext对象的getRealPath()

创建工厂mybatis使用了构建者模式 构建者模式: 把对象的创建细节隐藏, 是使用者直接调用方法即可拿到对象。
 生产SqlSession使用了工厂模式 优势: 解耦 (降低类之间的依赖关系)

创建Dao接口实现类使用了代理模式 优势: 不修改源码的基础上对已有方法增强



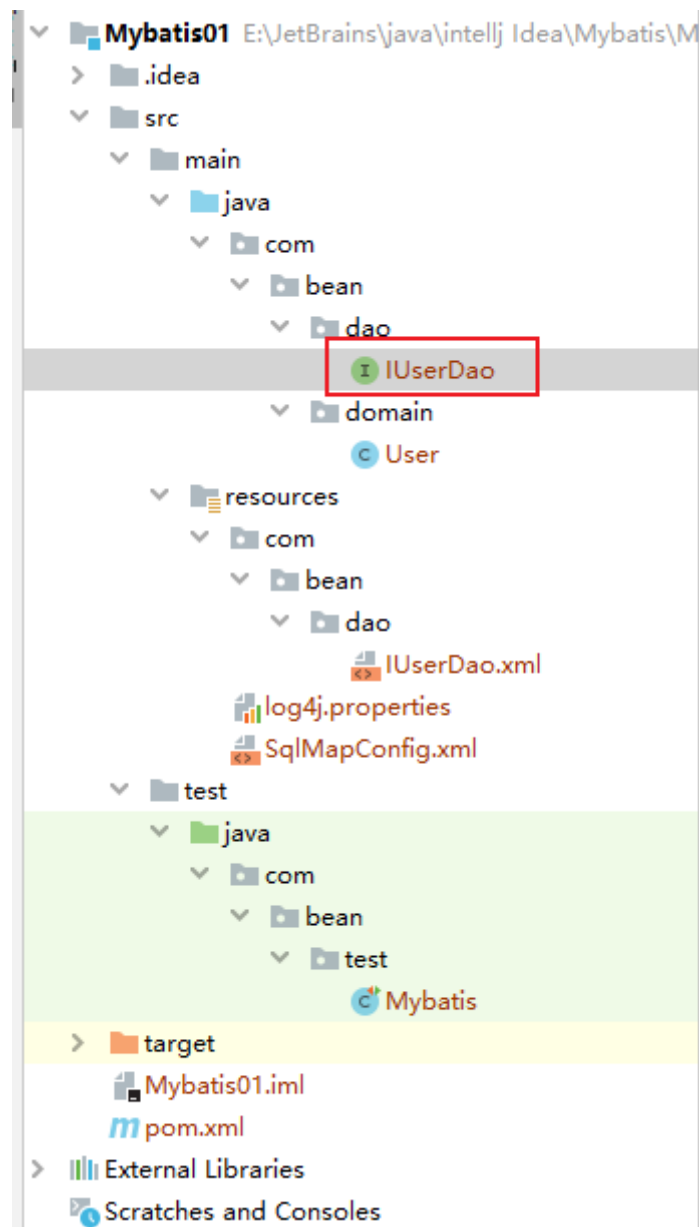
mybatis 深入讲解

mybatis 的 crud

基于代理 `dao` 的方式实现 `CRUD`

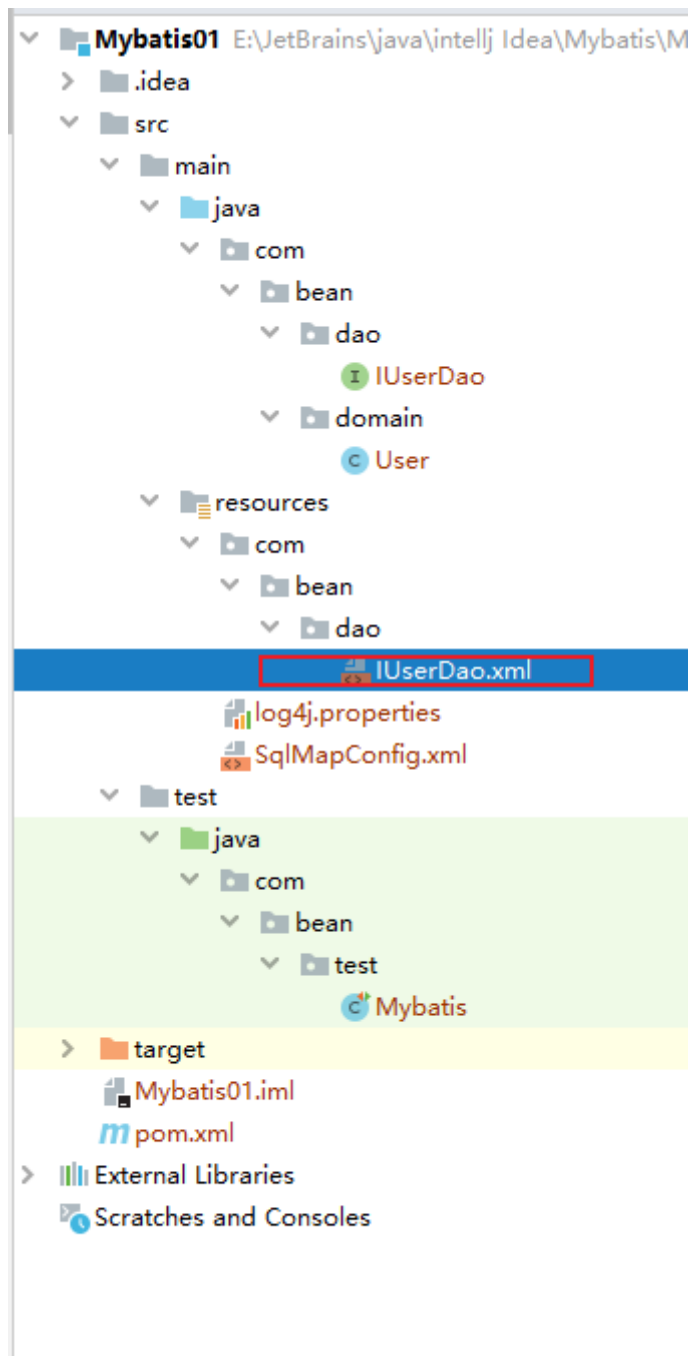
保存操作

1. 进行环境搭建（我们现在不用注解，先使用xml方式）
2. 编写 `IUserDao` 接口



```
1 package com.bean.dao;
2
3 import com.bean.domain.User;
4 import org.apache.ibatis.annotations.Select;
5
6 import java.util.List;
7
8 /**
9  * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
10  */
11 public interface IUserDao {
12
13     /**
14      * 查询所有用户
15      * @return List<User>
16      */
17     List<User> findAll();
18
19     /**
20      * 向数据库中插入新用户
21      * @param user
22      */
23     void saveUser(User user);
```

3. 在 `IUserDao.xml` 映射文件中添加插入新用户这个方法的映射



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <select id="findAll" resultType="com.bean.domain.User">
8          select * from user
9      </select>
10
11      <!--
12          1. 新的选项，既然是插入功能就要使用insert
13          2. 我们的接口参数要选择，就是parameterType
14          3. 使用插入的语法，原来的?占位符替换为#{内容}
15          4. #{ }里的内容取决于getter，比如getId()->id->id，如果是别的就对应别的
16      -->

```

```

17     <insert id="saveUser" parameterType="com.bean.domain.User">
18         insert into user(username,address,sex,birthday) values(#{username},#{address},#
            {sex},#{birthday})
19     </insert>
20 </mapper>

```

4. 在测试类中编写方法

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17
18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession session = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         //      1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27         //      2. 构建SqlSessionFactory工厂
28         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
29         SqlSessionFactory factory = builder.build(inputStream);
30         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
31         session = factory.openSession();
32         //      4. 使用SqlSession对象创建Dao接口的代理对象
33         mapper = session.getMapper(IUserDao.class);
34
35     }
36
37     @After //在测试之后调用
38     public void destory() throws IOException {
39         //      6. 关闭资源
40         session.close();
41         inputStream.close();
42     }
43
44     @Test //查询所有方法
45     public void findAll(){
46         //      5. 使用代理执行方法
47         List<User> users = mapper.findAll();
48         for (User user : users) {
49             System.out.println(user);
50         }

```

```

51     }
52
53     @Test    //保存新用户
54     public void saveUser(){
55         //        创建User对象
56         User user = new User();
57         user.setUsername("mybatis");
58         user.setAddress("山东省");
59         user.setBirthday(new Date());
60         user.setSex("男");
61
62         //        调用方法
63         mapper.saveUser(user);
64
65         //        因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚
66         session.commit();
67     }
68
69 }

```

- 注意点

1. 在编写映射文件的时候注意

1. 插入方法不是 `select`，而是 `insert`
2. 接口参数使用 `parameterType` 选择
3. 占位符的替换，替换为`#{}`
4. 占位符中的内容取决于getter

2. 在编写测试类中注意

1. `mybatis` 中的事务是手动提交的，所以需要使用 `SqlSession.commit()` 来进行提交

更新操作

1. 环境搭建
2. 写 `I UserDao` 接口

```

1     package com.bean.dao;
2
3     import com.bean.domain.User;
4     import org.apache.ibatis.annotations.Select;
5
6     import java.util.List;
7
8     /**
9      * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
10     */
11     public interface IUserDao {
12
13         /**
14          * 查询所有用户
15          * @return List<User>
16          */

```

```

17     List<User> findAll();
18
19     /**
20      * 向数据库中插入新用户
21      * @param user
22      */
23     void saveUser(User user);
24
25     /**
26      * 更新用户数据
27      * @param user
28      */
29     void updateUser(User user);
30 }

```

3. 编写 IUserDao.xml 映射文件

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <select id="findAll" resultType="com.bean.domain.User">
8          select * from user
9      </select>
10
11     <insert id="saveUser" parameterType="com.bean.domain.User">
12         insert into user(username,address,sex,birthday)
13             values(#{username},#{address},#{sex},#{birthday})
14     </insert>
15
16     <!--
17         1. 更新操作使用update
18         2. parameterType还是User
19         3. 占位符#{ }
20     -->
21     <update id="updateUser" parameterType="com.bean.domain.User">
22         update user set username=#{username},address=#{address},
23             sex=#{sex},birthday=#{birthday} where id=#{id}
24     </update>
25 </mapper>

```

4. 编写测试代码

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;

```

```

14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17
18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession session = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         // 1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27
28         // 2. 构建SqlSessionFactory工厂
29         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
30         SqlSessionFactory factory = builder.build(inputStream);
31
32         // 3. 使用SqlSessionFactory工厂构建SqlSession对象
33         session = factory.openSession();
34
35         // 4. 使用SqlSession对象创建Dao接口的代理对象
36         mapper = session.getMapper(IUserDao.class);
37
38     }
39
40     @After //在测试之后调用
41     public void destory() throws IOException {
42         // 因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         session.commit();
44
45         // 6. 关闭资源
46         session.close();
47         inputStream.close();
48     }
49
50     @Test //查询所有方法
51     public void findAll(){
52         // 5. 使用代理执行方法
53         List<User> users = mapper.findAll();
54         for (User user : users) {
55             System.out.println(user);
56         }
57     }
58
59     @Test //保存新用户
60     public void saveUser(){
61         // 创建User对象
62         User user = new User();
63         user.setUsername("mybatis");
64         user.setAddress("山东省");
65         user.setBirthday(new Date());
66         user.setSex("男");
67
68         // 调用方法
69         mapper.saveUser(user);
70
71         // 事务的提交统一放到After上面了

```

```

72     }
73
74     @Test
75     public void updateUser(){
76         //         这里应当是先查出来User对象，不过这里省事创建得了
77         User user = new User();
78         user.setId(49);
79         user.setUsername("updateUser");
80         user.setAddress("山东省");
81         user.setBirthday(new Date());
82         user.setSex("女");
83
84         //         更改内容
85         mapper.updateUser(user);
86
87         //         事务的提交统一放到After上面了
88     }
89
90 }

```

删除操作

1. 环境搭建
2. 编写 IUserDao 接口

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4  import org.apache.ibatis.annotations.Select;
5
6  import java.util.List;
7
8  /**
9   * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
10  */
11  public interface IUserDao {
12
13      /**
14       * 查询所有用户
15       * @return List<User>
16       */
17      List<User> findAll();
18
19      /**
20       * 向数据库中插入新用户
21       * @param user
22       */
23      void saveUser(User user);
24
25      /**
26       * 更新用户数据
27       * @param user
28       */
29      void updateUser(User user);
30
31      /**
32       * 删除用户数据只需要一个id就可以了

```



```

33     * @param id
34     */
35     void deleteUser(Integer id);
36 }

```

3. 编写 IUserDao.xml 映像文件

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <select id="findAll" resultType="com.bean.domain.User">
8          select * from user
9      </select>
10
11     <insert id="saveUser" parameterType="com.bean.domain.User">
12         insert into user(username,address,sex,birthday)
13             values({username},{address},{sex},{birthday})
14     </insert>
15
16
17     <update id="updateUser" parameterType="com.bean.domain.User">
18         update user set username={username},address={address},
19             sex={sex},birthday={birthday} where id={id}
20     </update>
21
22     <!--
23         1. 删除操作使用delete
24         2. parameterType现在变为了int类型，但是写int,INT,Integer,INTEGER,java.lang.Integer都可以
25         3. 占位符#{ }
26         4. 当参数内容只有一个的时候，写什么都无所谓，哪怕写个a都无所谓
27     -->
28     <delete id="deleteUser" parameterType="java.lang.Integer">
29         delete from user where id={id}
30     </delete>
31 </mapper>

```

4. 编写测试类

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17

```

```

18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession session = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         //      1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27
28         //      2. 构建SqlSessionFactory工厂
29         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
30         SqlSessionFactory factory = builder.build(inputStream);
31
32         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
33         session = factory.openSession();
34
35         //      4. 使用SqlSession对象创建Dao接口的代理对象
36         mapper = session.getMapper(IUserDao.class);
37
38     }
39
40     @After //在测试之后调用
41     public void destory() throws IOException {
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         session.commit();
44
45         //      6. 关闭资源
46         session.close();
47         inputStream.close();
48     }
49
50     @Test //查询所有方法
51     public void findAll(){
52         //      5. 使用代理执行方法
53         List<User> users = mapper.findAll();
54         for (User user : users) {
55             System.out.println(user);
56         }
57     }
58
59     @Test //保存新用户
60     public void saveUser(){
61         //      创建User对象
62         User user = new User();
63         user.setUsername("mybatis");
64         user.setAddress("山东省");
65         user.setBirthday(new Date());
66         user.setSex("男");
67
68         //      调用方法
69         mapper.saveUser(user);
70
71         //      事务的提交统一放到After上面了
72     }
73
74     @Test
75     public void updateUser(){

```

```

76 //      这里应当是先查出来User对象，不过这里省事创建得了
77     User user = new User();
78     user.setId(49);
79     user.setUsername("updateUser");
80     user.setAddress("山东省");
81     user.setBirthday(new Date());
82     user.setSex("女");
83
84 //      更改内容
85     mapper.updateUser(user);
86
87 //      事务的提交统一放到After上面了
88 }
89
90 @Test
91 public void deleteUser(){
92 //      直接执行函数得了，把新添加的删了
93     mapper.deleteUser(49);
94 //      事务提交，放到了After中
95 }
96
97 }

```

根据id查询和模糊查询

- 根据id进行查询
- 根据名称进行模糊查询

1. 接口

```

1 package com.bean.dao;
2
3 import com.bean.domain.User;
4 import org.apache.ibatis.annotations.Select;
5
6 import java.util.List;
7
8 /**
9  * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
10  */
11 public interface IUserDao {
12
13     /**
14      * 通过id寻找用户
15      * @param id
16      * @return User对象
17      */
18     User findUserById(int id);
19
20
21     /**
22      * 通过某个名称来模糊查询用户们
23      * @param name
24      * @return List<User>
25      */

```

```
26     List<User> findUserByName(String name);
27 }
```

2. IUserDao.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <select id="findUserById" parameterType="INT" resultType="com.bean.domain.User">
8          select * from user where id=#{id}
9      </select>
10
11     <select id="findUserByName" parameterType="STRING" resultType="com.bean.domain.User">
12         select * from user where username like #{username}
13     </select>
14 </mapper>
```

3. 测试类

```
1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17
18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession sqlSession = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         // 1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27
28         // 2. 构建SqlSessionFactory工厂
29         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
30         SqlSessionFactory factory = builder.build(inputStream);
31
32         // 3. 使用SqlSessionFactory工厂构建SqlSession对象
33         sqlSession = factory.openSession();
34
35         // 4. 使用SqlSession对象创建Dao接口的代理对象
36         mapper = sqlSession.getMapper(IUserDao.class);
```

```

37
38     }
39
40     @After    //在测试之后调用
41     public void destory() throws IOException {
42         //        因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();
44         //        6. 关闭资源
45         sqlSession.close();
46         inputStream.close();
47     }
48
49     @Test    //通过Id来查询User对象
50     public void findUserById(){
51         User user = mapper.findUserById(48);
52         System.out.println(user);
53     }
54
55     @Test    //通过UserName来模糊查询
56     public void findUserByName(){
57         List<User> users = mapper.findUserByName("%王%");    //通过查询每一个username含有"王"的
58         User
59         for (User user : users) {
60             System.out.println(user);
61         }
62     }

```

使用聚合函数

- 查询总记录条数

1. 接口

```

1     package com.bean.dao;
2
3     import com.bean.domain.User;
4     import org.apache.ibatis.annotations.Select;
5
6     import java.util.List;
7
8     /**
9      * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
10     */
11     public interface IUserDao {
12
13         /**
14          * 使用聚合函数找到总人数的多少
15          * @return 总人数
16          */
17         int findCount();
18     }

```

2. IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <select id="findCount" resultType="int">
8          select count(id) from user;
9      </select>
10 </mapper>

```

3. 测试类

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17
18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession sqlSession = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         //      1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27
28         //      2. 构建SqlSessionFactory工厂
29         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
30         SqlSessionFactory factory = builder.build(inputStream);
31
32         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
33         sqlSession = factory.openSession();
34
35         //      4. 使用SqlSession对象创建Dao接口的代理对象
36         mapper = sqlSession.getMapper(IUserDao.class);
37
38     }
39
40     @After //在测试之后调用
41     public void destory() throws IOException {
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();
44         //      6. 关闭资源
45         sqlSession.close();

```

```

46         inputStream.close();
47     }
48
49     @Test    //通过UserName来模糊查询
50     public void findCount(){
51         int count = mapper.findCount();
52         System.out.println(count);
53     }
54
55 }

```

细节

1. 最后一个id

因为我们在查询的时候和删除的时候都需要id，但是我们在插入的时候，因为id是自增长的，所以不知道当前id号是多少，所以我们需要知道id号是多少

`select last_insert_id()`，通过这个语句我们可以在插入之后找到新插入的这个id号，但是假如没有插入直接就使用这个语句，那么我们会得到0，因为没有插入

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <insert id="insertUser" parameterType="com.bean.domain.User">
8          <!-- 配置当User插入之后获取最后的id值
9              keyProperty: User类中要获取到的值
10             keyColumn: 数据库表中的值，与User类对应
11             returnType: 返回类型
12             order: 在插入语句 之前/之后 执行
13                 AFTER: 之后
14                 BEFORE: 之前
15             select last_insert_id(): 查询最后的ID
16         -->
17         <selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER">
18             select last_insert_id()
19         </selectKey>
20         insert into user (username,address,sex,birthday) values(#{username},#{address},#
21             {sex},#{birthday})
22     </insert>
23 </mapper>

```

- 测试类

```

1  package com.bean.test;
2

```

```

3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.Date;
16 import java.util.List;
17
18 public class Mybatis {
19     InputStream inputStream = null;
20     SqlSession sqlSession = null;
21     IUserDao mapper;
22
23     @Before //在测试之前调用
24     public void init() throws IOException {
25         //      1. 读取配置文件
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27
28         //      2. 构建SqlSessionFactory工厂
29         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
30         SqlSessionFactory factory = builder.build(inputStream);
31
32         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
33         sqlSession = factory.openSession();
34
35         //      4. 使用SqlSession对象创建Dao接口的代理对象
36         mapper = sqlSession.getMapper(IUserDao.class);
37     }
38
39
40     @After //在测试之后调用
41     public void destory() throws IOException {
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();
44         //      6. 关闭资源
45         sqlSession.close();
46         inputStream.close();
47     }
48
49     @Test
50     public void findByName(){
51         User user = new User();
52         user.setSex("男");
53         user.setBirthday(new Date());
54         user.setAddress("Address");
55         user.setUsername("new User");
56
57         System.out.println("保存之前: "+user);
58
59         mapper.insertUser(user);
60

```



```

61 //      注意，在这里的时候并没有使用查询操作，而是直接打印
62      System.out.println("保存之后: "+user);
63  }
64
65  }

```

保存之前：User{id=null, username='new User', birthday=Thu Nov 14 10:57:53 CST 2019, sex='男', address='Address'}

保存之后：User{id=49, username='new User', birthday=Thu Nov 14 10:57:53 CST 2019, sex='男', address='Address'}

2. 占位符

我们在刚才使用聚合函数查询语句的时候：

1.

- 在 IUserDao.xml 中使用的是： `select * from user where username like #{username}`
- 在测试类中使用的是： `List<User> users = mapper.findByName("%王%");`
-

```

2019-11-14 10:04:56,601 738 [main] DEBUG m.bean.dao.IUserDao.findByName - ==> Preparing: select * from user where username like ?
2019-11-14 10:04:56,650 787 [main] DEBUG m.bean.dao.IUserDao.findByName - ==> Parameters: %王%(String)

```

从上面我们可以看出来，是先使用？，然后再赋值，使用的PreparedStatement

2.

- 其实还可以使用： `select * from user where username like '%${value}%'`
- 其实还可以使用： `List<User> users = mapper.findByName("王");`
-

```

2019-11-14 10:16:22,504 776 [main] DEBUG m.bean.dao.IUserDao.findByName - ==> Preparing: select * from user where username like '%王%'
2019-11-14 10:16:22,557 829 [main] DEBUG m.bean.dao.IUserDao.findByName - ==> Parameters:

```

- 使用的是Statement
- 注意xml中只能使用 `'%${value}%'`

mybatis 中的参数深入及结果集的深入

参数深入

- 传递简单类型
- 传递pojo对象
- 传递pojo包装对象

1. 传递简单类型

对于普通类型，聚合函数的占位符等等都是简单类型

2. 传递pojo对象

Mybatis中使用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

- OGNL表达式

- Object Graphic Navigation Language (对象 图 导航 语言)

他是通过对象的取值方法来获取数据, 在写法上把get省略了

- 比如, 获取用户的名称:
 - 类中写法: `user.getUsername()`
 - OGNL表达式: `user.username`
- mybatis中能直接写`username`而不用写`user.username`, 原因是:
 - `parameterType="com.bean.domain.User"` : `parameterType` 已经写好了

3. 传递pojo包装对象

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <!-- 通过pojo包装对象来获得User
9          parameterType="com.bean.domain.QueryVo": 参数变为了QueryVo
10         resultType="com.bean.domain.User": 返回值还是User对象
11
12         select * from user where username like user.username: 对于这个user.username
13             1. 使用了OGNL表达式: user.username ==> QueryVo.username.username ==>
14             QueryVo.getUsername().getusername()
15             2. 获取的是User里面的username
16         -->
17         <select id="findUserByVo" parameterType="com.bean.domain.QueryVo"
18             resultType="com.bean.domain.User">
19             select * from user where username like user.username
20         </select>
21     </mapper>
```

```
1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.QueryVo;
5  import com.bean.domain.User;
6  import org.apache.ibatis.io.Resources;
7  import org.apache.ibatis.session.SqlSession;
8  import org.apache.ibatis.session.SqlSessionFactory;
9  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
10 import org.junit.After;
11 import org.junit.Before;
12 import org.junit.Test;
13
14 import java.io.IOException;
```

```

15 import java.io.InputStream;
16 import java.util.Date;
17 import java.util.List;
18
19 public class Mybatis {
20     InputStream inputStream = null;
21     SqlSession sqlSession = null;
22     IUserDao mapper;
23
24     @Before //在测试之前调用
25     public void init() throws IOException {
26         // 1. 读取配置文件
27         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
28
29         // 2. 构建SqlSessionFactory工厂
30         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
31         SqlSessionFactory factory = builder.build(inputStream);
32
33         // 3. 使用SqlSessionFactory工厂构建SqlSession对象
34         sqlSession = factory.openSession();
35
36         // 4. 使用SqlSession对象创建Dao接口的代理对象
37         mapper = sqlSession.getMapper(IUserDao.class);
38
39     }
40
41     @After //在测试之后调用
42     public void destory() throws IOException {
43         // 因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
44         sqlSession.commit();
45         // 6. 关闭资源
46         sqlSession.close();
47         inputStream.close();
48     }
49
50     @Test
51     public void findByName(){
52         QueryVo vo = new QueryVo();
53         User u = new User();
54         u.setUsername("%王%");
55         vo.setUser(u);
56
57         List<User> users = mapper.findUserByVo(vo);
58
59         for (User user : users) {
60             System.out.println(user);
61         }
62
63     }
64
65 }

```

结果集的深入

我们现在使用的User类是数据库表中的各个元素，但是假如和数据库的元素对不上，比如

- 数据库中: `id` , `username` , `sex` , `address` , `birthday`
- 类中: `userId` , `userName` , `userSex` , `userAddress` , `userBirthday`

这样对不上的情况就会出现差错, 现在有两种解决方式:

- 追求查询效率: 使用起别名的方式从SQL语句上根本解决问题
- 追求开发效率: 使用映射来解决问题

-
- 新的User表, 和数据库中的不同

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.Date;
5
6 public class User implements Serializable {
7     //注释中是数据库中的名称
8     private Integer userId; //id
9     private String userName; //username (在windows系统下mysql数据库并不区分大小写, 所以这个大小
    无所谓)
10    private Date userBirthday; //birthday
11    private String userSex; //sex
12    private String userAddress; //address
13
14    public User() {
15    }
16
17    public User(Integer userId, String userName, Date userBirthday, String userSex, String
    userAddress) {
18        this.userId = userId;
19        this.userName = userName;
20        this.userBirthday = userBirthday;
21        this.userSex = userSex;
22        this.userAddress = userAddress;
23    }
24
25    public Integer getUserId() {
26        return userId;
27    }
28
29    public void setUserId(Integer userId) {
30        this.userId = userId;
31    }
32
33    public String getUserName() {
34        return userName;
35    }
36
37    public void setUserName(String userName) {
38        this.userName = userName;
39    }
40
41    public Date getUserBirthday() {
42        return userBirthday;
43    }
44}
```

```

45     public void setUserBirthday(Date userBirthday) {
46         this.userBirthday = userBirthday;
47     }
48
49     public String getUserSex() {
50         return userSex;
51     }
52
53     public void setUserSex(String userSex) {
54         this.userSex = userSex;
55     }
56
57     public String getUserAddress() {
58         return userAddress;
59     }
60
61     public void setUserAddress(String userAddress) {
62         this.userAddress = userAddress;
63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "userId=" + userId +
69             ", userName='" + userName + '\'' +
70             ", userBirthday=" + userBirthday +
71             ", userSex='" + userSex + '\'' +
72             ", userAddress='" + userAddress + '\'' +
73             '}';
74     }
75 }

```

1. 追求查询效率

- 使用别名，在SQL语句上根本解决问题， IUserDao.xml 如下

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <!--通过起别名的方式来实现-->
8      <select id="findAll" resultType="com.bean.domain.User">
9          select id as userId,username as userName,
10             birthday as userBirthday,sex as userSex,address as userAddress
11          from user;
12      </select>
13
14  </mapper>

```

- 测试类

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;

```

```

5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class Mybatis {
18     InputStream inputStream = null;
19     SqlSession sqlSession = null;
20     IUserDao mapper;
21
22     @Before //在测试之前调用
23     public void init() throws IOException {
24         //      1. 读取配置文件
25         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
26
27         //      2. 构建SqlSessionFactory工厂
28         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
29         SqlSessionFactory factory = builder.build(inputStream);
30
31         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
32         sqlSession = factory.openSession();
33
34         //      4. 使用SqlSession对象创建Dao接口的代理对象
35         mapper = sqlSession.getMapper(IUserDao.class);
36
37     }
38
39     @After //在测试之后调用
40     public void destory() throws IOException {
41
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();
44
45         //      6. 关闭资源
46         sqlSession.close();
47         inputStream.close();
48     }
49
50     @Test
51     public void findAll(){
52
53         List<User> users = mapper.findAll();
54
55         for (User user : users) {
56             System.out.println(user);
57         }
58
59     }
60
61 }

```

2. 追求开发速度

- IUserDao.xml 中

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <!-- 另一种方式，使用映射关系来将数据库和类进行对应，但是这样一来就不适用resultType了，使用
9      resultMap
10         id: 主键
11             property: 用户类对应的字段名称
12             column: 表中对应的字段名称
13         result: 其他字段
14             property: 用户类对应的字段名称
15             column: 表中对应的字段名称
16     -->
17     <resultMap id="userMap" type="com.bean.domain.User">
18         <id property="userId" column="id"></id>
19         <result property="userName" column="username"></result>
20         <result property="userBirthday" column="birthday"></result>
21         <result property="userSex" column="sex"></result>
22         <result property="userAddress" column="address"></result>
23     </resultMap>
24
25     <!--使用resultMap, 映射-->
26     <select id="findAll" resultMap="userMap">
27         select * from user;
28     </select>
29 </mapper>
```

- 测试类中

```
1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class Mybatis {
18     InputStream inputStream = null;
19     SqlSession sqlSession = null;
20     IUserDao mapper;
```

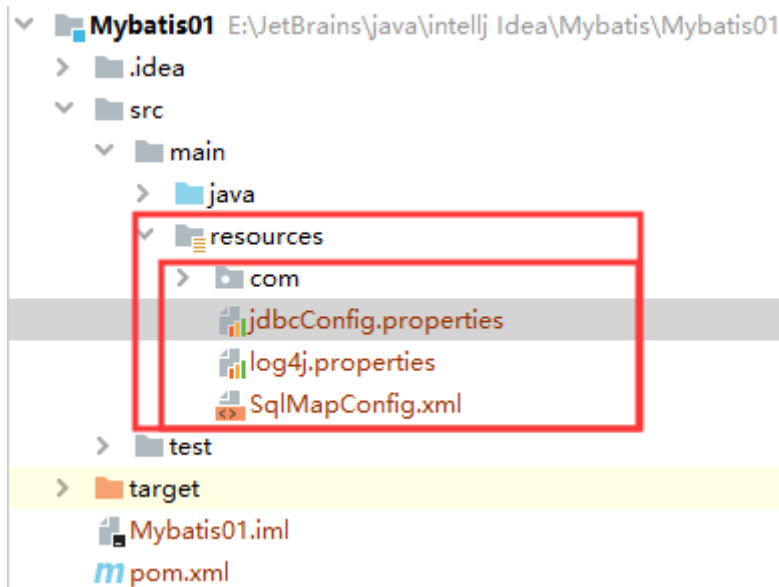
```

21
22     @Before //在测试之前调用
23     public void init() throws IOException {
24         //      1. 读取配置文件
25         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
26
27         //      2. 构建SqlSessionFactory工厂
28         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
29         SqlSessionFactory factory = builder.build(inputStream);
30
31         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
32         sqlSession = factory.openSession();
33
34         //      4. 使用SqlSession对象创建Dao接口的代理对象
35         mapper = sqlSession.getMapper(IUserDao.class);
36
37     }
38
39     @After //在测试之后调用
40     public void destory() throws IOException {
41
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();
44
45         //      6. 关闭资源
46         sqlSession.close();
47         inputStream.close();
48     }
49
50     @Test
51     public void findAll(){
52
53         List<User> users = mapper.findAll();
54
55         for (User user : users) {
56             System.out.println(user);
57         }
58
59     }
60
61 }

```

mybatis 中的配置

properties 标签



- jdbcConfig.properties

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatis
3 jdbc.username=root
4 jdbc.password=root
```

- SqlMapConfig.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7     <!--配置properties: 可以在标签内部配置连接数据库的信息，也可以通过属性引用外部配置文件的信息
8     resource: 常用
9         用于指定配置文件的具体位置，是按照类路径的写法来写，而且必须存在于类路径下
10    url:
11        是要求按照URL的写法来写地址
12        URL: Uniform Resource Locator 统一资源定位符，他是可以唯一标识一个资源的位置
13        URL的写法:
14            协议 主机 端口 URI
15            http://localhost:8080/tomcat/demo
16            协议不仅仅只有一个http协议
17        URI: Uniform Resource Identifier 统一资源标识符。只可以在应用中唯一定位一个资源
18    -->
19    <properties
20    url="file:///E:/JetBrains/java/intelliJ%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>
21
22    <environments default="mysql">
23        <environment id="mysql">
24
25            <transactionManager type="JDBC"></transactionManager>
26
27            <!--这里也是改变的地方，其中value值全部都从properties文件中读取-->
28            <dataSource type="POOLED">
29                <property name="driver" value="${jdbc.driver}"/>
30                <property name="url" value="${jdbc.url}"/>
```

```

31         <property name="username" value="${jdbc.username}" />
32         <property name="password" value="${jdbc.password}" />
33     </dataSource>
34
35 </environment>
36
37 </environments>
38
39 <mappers>
40     <mapper class="com.bean.dao.IUserDao" />
41 </mappers>
42 </configuration>

```

typeAliases 标签

刚才我们在使用 `parameterType` 的时候说了，可以使用各种类型的重载，比如

- `int` `INT` `INTEGER` `Integer`
- `string` `String` `STRING`

这是因为 `typeAliases` 标签的作用

- `SqlMapConfig.xml`

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7
8
9  <!--typeAliases: 用于起别名
10     type: 参数的全限定类名
11     alias: 别名
12     别名一旦起了，就不再限制大小写，比如: user  User  USER
13 -->
14     <typeAliases>
15         <typeAlias type="com.bean.domain.User" alias="user"></typeAlias>
16     </typeAliases>
17
18     <environments default="mysql">
19
20         <environment id="mysql">
21
22             <transactionManager type="JDBC"></transactionManager>
23
24             <dataSource type="POOLED">
25                 <property name="driver" value="com.mysql.jdbc.Driver" />
26                 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis" />
27                 <property name="username" value="root" />
28                 <property name="password" value="root" />
29             </dataSource>
30
31         </environment>

```

```

32
33     </environments>
34
35     <mappers>
36         <mapper class="com.bean.dao.IUserDao" />
37     </mappers>
38 </configuration>

```

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8
9
10     <select id="findAll" resultType="user">
11         select * from user;
12     </select>
13
14 </mapper>

```

上面的确实很好用，但是还有更好用的，因为使用 `typeAlias` 太多也很麻烦，所以有了 package

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7
8
9  <!-- typeAliases: 用于起别名
10      package: 用于指定包，包下的所有类都会自动配置好别名，并且别名就是类名
11  -->
12     <typeAliases>
13         <package name="com.bean.domain"></package>
14     </typeAliases>
15
16     <environments default="mysql">
17
18         <environment id="mysql">
19
20             <transactionManager type="JDBC"></transactionManager>
21
22             <dataSource type="POOLED">
23                 <property name="driver" value="com.mysql.jdbc.Driver" />
24                 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis" />
25                 <property name="username" value="root" />
26                 <property name="password" value="root" />
27             </dataSource>
28

```

```

29         </environment>
30
31     </environments>
32
33     <mappers>
34         <mapper class="com.bean.dao.IUserDao"/>
35     </mappers>
36 </configuration>

```

mappers 标签

`mappers` 标签里面和 `typeAliases` 是差不多的，只要指定好 `package` 就不需要再指定 `resource` 或者 `class`

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7
8
9      <typeAliases>
10         <package name="com.bean.domain"></package>
11     </typeAliases>
12
13     <environments default="mysql">
14
15         <environment id="mysql">
16
17             <transactionManager type="JDBC"></transactionManager>
18
19             <dataSource type="POOLED">
20                 <property name="driver" value="com.mysql.jdbc.Driver"/>
21                 <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
22                 <property name="username" value="root"/>
23                 <property name="password" value="root"/>
24             </dataSource>
25
26         </environment>
27     </environments>
28
29     <mappers>
30         <package name="com.bean.dao"></package>
31     </mappers>
32
33 </configuration>

```

最终配置

- `jdbcConfig.properties`

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatis
3 jdbc.username=root
4 jdbc.password=root
```

- SqlMapConfig.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7
8     <!-- 读取好配置文件 -->
9     <properties
10         url="file:///E:/JetBrains/java/intellij%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfi
11         g.properties"></properties>
12
13     <!-- 使用标签配置好了 com.bean.domain 包下面的类，给他们起了别名，比如：
14         User 类
15         别名：User
16         别名不区分大小写
17     -->
18     <typeAliases>
19         <package name="com.bean.domain"></package>
20     </typeAliases>
21
22     <!-- 配置环境们，并且设置默认为mysql环境
23         名字是不是mysql无所谓，但是要进行解释 -->
24     <environments default="mysql">
25         <!-- 对mysql进行解释 -->
26         <environment id="mysql">
27             <!-- 事务类型：JDBC -->
28             <transactionManager type="JDBC"></transactionManager>
29             <!-- 数据源，类型：连接池 -->
30             <dataSource type="POOLED">
31                 <!-- 参数们：引用的properties下面的数值 -->
32                 <property name="driver" value="${jdbc.driver}"/>
33                 <property name="url" value="${jdbc.url}"/>
34                 <property name="username" value="${jdbc.username}"/>
35                 <property name="password" value="${jdbc.password}"/>
36             </dataSource>
37         </environment>
38     </environments>
39
40     <!-- 映射：映射到dao包下，找到IUserDao.xml -->
41     <mappers>
42         <package name="com.bean.dao"></package>
43     </mappers>
44
45 </configuration>
```

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <!--映射，为接口编写语句，接口为"com.bean.dao.IUserDao"-->
7  <mapper namespace="com.bean.dao.IUserDao">
8      <!--查询方法，返回类型为User
9          这里因为在SqlMapConfig.xml中编写好了别名，所以这里不用写全类名了
10     -->
11     <select id="findAll" resultType="user">
12         select * from user;
13     </select>
14
15
16     <!--增加方法，参数为User对象
17         同样因为在SqlMapConfig.xml中编写好了别名，所以这里不用写全类名了
18         假如要知道类中的字段名和表中的字段名不一致的解决方法，看resultMap
19     -->
20     <insert id="insertUser" parameterType="user">
21         <!--使用selectKey找到插入之后的id
22             keyProperty: id      用户类中的id
23             keyColumn: 表中的id
24             resultType: 返回类型
25             order: 在插入语句之前/之后执行
26                 AFTER
27                 BEFORE
28             在插入之前执行方法返回0
29         -->
30         <selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER">
31             <!--选择最后一个id-->
32             select last_insert_id()
33         </selectKey>
34         insert into user(username,address,sex,birthday)
35             values({username},{address},{sex},{birthday})
36     </insert>
37
38
39     <!--更新方法，参数为User对象
40         同样因为在SqlMapConfig.xml中编写好了别名，所以这里不用写全类名了
41     -->
42     <update id="updateUser" parameterType="user">
43         update user set username={username},address={address},
44             sex={sex},birthday={birthday} where id={id}
45     </update>
46
47
48     <!--删除操作，参数为int类型-->
49     <delete id="deleteById" parameterType="int">
50         delete from user where id={id}
51     </delete>
52
53
54     <!--根据id查询，参数为int类型-->
55     <select id="findUserById" parameterType="int">
56         select * from user where id={id}
57     </select>

```

```
58
59
60      <!--模糊查询，参数为string
61          注意模糊查询的时候写参数的时候要写限定符，比如"%王%", 查询包含"王"的username
62      -->
63      <select id="findUserByName" parameterType="string">
64          select * from user where username like #{username}
65      </select>
66
67
68      <!--使用聚合函数-->
69      <select id="findCount">
70          select count(id) from user
71      </select>
72  </mapper>
```

mybatis 中的连接池以及事务控制

- mybatis 中连接池使用及分析
- mybatis 事务控制的分析

连接池

1. 连接池

- 我们在实际开发中都会使用连接池。
- 因为它可以减少我们获取连接所消耗的时间。

- 连接池就是一个用于存储连接的一个容器
- 容器其实就是一个集合对象，该集合必须是线程安全的，不能两个线程拿到同一个连接
- 该集合还必须实现队列的特性：先进先出

2. mybatis 中的连接池

mybatis 连接池提供了3种方式的配置：

- 配置的位置：

主配置文件 `SqlMapConfig.xml` 中的 `dataSource` 标签，`type` 属性就是表示采用何种连接池方式。

- type属性的取值：

`POOLED` 采用传统的 `javax.sql.DataSource` 规范中的连接池，`mybatis` 中有针对规范的实现

`UNPOOLED` 采用传统的获取连接的方式，虽然也实现 `Javax.sql.DataSource` 接口，但是并没有使用池的思想。

JNDI 采用服务器提供的JNDI技术实现，来获取 DataSource 对象，不同的服务器所能拿到 DataSource 是不一样的。

- 注意：如果不是 web 或者 maven 的war工程，是不能使用的。
- 我们课程中使用的是tomcat服务器，采用连接池就是 dbcp 连接池。

mybatis中的事务

- 什么是事务
- 事务的四大特性ACID
- 不考虑隔离性会产生的三个问题
- 问题的解决办法：四种隔离级别

mybatis 基于XML配置的动态SQL语句使用 条件查询

- 我们在查询出某些数据的时候经常会有不同的条件，比如输入姓名，年龄，性别，地址，等等。
- 这些条件可能有一些，可能没有，也可能都有，这个时候就需要动态SQL查询了，标签可以实现

- mappers配置文件中的几个标签：

- <if>
- <where>
- <foreach>
- <sql>

if 标签

```
1 package com.bean.dao;
2
3 import com.bean.domain.User;
4
5 import java.util.List;
6
7 /**
8  * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
9  */
10 public interface IUserDao {
11
12     /**
13      * 实现动态SQL查询，动态的条件查询
14      * @param user id,username,address,sex,等等，条件随机组合
15      * @return
16      */
17     List<User> findUserByCondition(User user);
18 }
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
```



```

6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <select id="findUserByCondition" resultType="com.bean.domain.User"
parameterType="com.bean.domain.User">
9          select * from user where 1=1
10         <!--这里的test内是判断条件，就是当参数里的username不为空的时候进行这里面的内容-->
11         <if test="username != null">
12             <!--进行与上面的内容拼接-->
13             and username=#{username}
14         </if>
15     </select>
16
17 </mapper>

```

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class Mybatis {
18     InputStream inputStream = null;
19     SqlSession sqlSession = null;
20     IUserDao mapper;
21
22     @Before //在测试之前调用
23     public void init() throws IOException {
24         //      1. 读取配置文件
25         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
26
27         //      2. 构建SqlSessionFactory工厂
28         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
29         SqlSessionFactory factory = builder.build(inputStream);
30
31         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
32         sqlSession = factory.openSession();
33
34         //      4. 使用SqlSession对象创建Dao接口的代理对象
35         mapper = sqlSession.getMapper(IUserDao.class);
36     }
37
38
39     @After //在测试之后调用
40     public void destory() throws IOException {
41
42         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
43         sqlSession.commit();

```

```

44
45     //        6. 关闭资源
46         sqlSession.close();
47         inputStream.close();
48     }
49
50     @Test
51     public void findUserByCondition(){
52
53         User u = new User();
54         u.setUsername("老王");
55
56         List<User> users = mapper.findUserByCondition(u);
57
58         for (User user : users) {
59             System.out.println(user);
60         }
61
62     }
63
64 }

```

注意一件事，我们在这里使用"与"的时候，不能使用符号 `&&`，而是要使用 `and`，因为要照顾到mysql

where标签

我们可以看到 `where 1=1`，这句话其实看起来很不舒服，所以我们用新的 `<where></where>` 来替换这个

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <select id="findUserByCondition" resultType="com.bean.domain.User"
9          parameterType="com.bean.domain.User">
10         select * from user
11         <where>
12             <if test="username != null">and username=#{username}</if>
13             <if test="sex != null">and sex=#{sex}</if>
14         </where>
15     </select>
16 </mapper>

```

foreach标签

foreach标签可以解决子查询的问题

比如 `select * from user where id in (41,42,43);` , 可以使用foreach标签来获得这样的效果

- `<foreach></foreach>`
 - collection: 集合内容
 - open: 开始的语句内容
 - close: 结束的语句内容
 - item: 集合内的每一个元素
 - separator: 每一个元素的分隔符

我们使用QueryVo来及逆行foreach标签的运行

```
1 package com.bean.domain;
2
3 import java.util.List;
4
5 public class QueryVo {
6     private User user;
7     private List<Integer> ids;    //这里就是id的集合
8
9     public User getUser() {
10         return user;
11     }
12
13     public void setUser(User user) {
14         this.user = user;
15     }
16
17     public List<Integer> getIds() {
18         return ids;
19     }
20
21     public void setIds(List<Integer> ids) {
22         this.ids = ids;
23     }
24 }
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.bean.dao.IUserDao">
7     <select id="findUserByIds" parameterType="com.bean.domain.QueryVo"
8         resultType="com.bean.domain.User">
9         select * from user
10         <where>
11             <!--假如id不为空-->
12             <if test="ids != null and ids.size()>0">
13                 <!--令集合内容为QueryVo.ids, 语句为id in (id),#{uid}=ids[item], 以逗号分隔-->
14                 <foreach collection="ids" open="id in (" close=")" item="uid" separator=",">
15                     <!--select * from user where id in (ids[0],ids[1],...,ids[item-1])-->
16                     #{uid}
17                 </foreach>
18             </if>
```

```

18         </where>
19     </select>
20
21 </mapper>

```

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.QueryVo;
5  import com.bean.domain.User;
6  import org.apache.ibatis.io.Resources;
7  import org.apache.ibatis.session.SqlSession;
8  import org.apache.ibatis.session.SqlSessionFactory;
9  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
10 import org.junit.After;
11 import org.junit.Before;
12 import org.junit.Test;
13
14 import java.io.IOException;
15 import java.io.InputStream;
16 import java.util.ArrayList;
17 import java.util.List;
18
19 public class Mybatis {
20     InputStream inputStream = null;
21     SqlSession sqlSession = null;
22     IUserDao mapper;
23
24     @Before //在测试之前调用
25     public void init() throws IOException {
26         //      1. 读取配置文件
27         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
28
29         //      2. 构建SqlSessionFactory工厂
30         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
31         SqlSessionFactory factory = builder.build(inputStream);
32
33         //      3. 使用SqlSessionFactory工厂构建SqlSession对象
34         sqlSession = factory.openSession();
35
36         //      4. 使用SqlSession对象创建Dao接口的代理对象
37         mapper = sqlSession.getMapper(IUserDao.class);
38
39     }
40
41     @After //在测试之后调用
42     public void destory() throws IOException {
43
44         //      因为mybatis将事务提交改为了手动提交，所以需要手动提交，否则会回滚，事务的提交到这里来了
45         sqlSession.commit();
46
47         //      6. 关闭资源
48         sqlSession.close();
49         inputStream.close();
50     }
51
52     @Test

```

```

53     public void findUserByIds(){
54
55         List<Integer> list = new ArrayList<Integer>();
56         list.add(41);
57         list.add(42);
58         list.add(43);
59
60         QueryVo vo = new QueryVo();
61         vo.setIds(list);
62
63         List<User> users = mapper.findUserByIds(vo);
64
65         for (User user : users) {
66             System.out.println(user);
67         }
68     }
69 }

```

sql与include标签

- sql与include标签是抽取重复内容的标签
 - sql用来定义重复的语句
 - include用来引入这个重复的语句

比如：`select * from user` 出现频率很高，就把他抽取出来

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7      <!--这里用来定义重复的语句-->
8      <sql id="defaultSql">select * from user</sql>
9
10     <select id="findUserByCondition" resultType="com.bean.domain.User"
11         parameterType="com.bean.domain.User">
12         <!--语句引入-->
13         <include refid="defaultSql"></include>
14         <where>
15             <if test="username != null">and username=#{username}</if>
16             <if test="sex != null">and sex=#{sex}</if>
17         </where>
18     </select>
19
20     <select id="findUserByIds" parameterType="com.bean.domain.QueryVo"
21         resultType="com.bean.domain.User">
22         <include refid="defaultSql"></include>
23         <where>
24             <if test="ids != null and ids.size()>0">
25                 <foreach collection="ids" open="id in (" close=")" item="id" separator=",">
26                     #{id}
27                 </foreach>
28             </if>
29         </where>

```

```
28     </select>
29
30 </mapper>
```

mybatis 中的多表操作

mybatis中的理念

- 一对多
- 多对一
- 一对一 (?)
- 多对多

关于一对一和多对一来讲

- 多对一：比如许多订单都只能有一个用户
- 一对一：比如一个身份证只能属于一个人

Mybatis中的特例：

- 但是还有一种特例，就是许多订单中的一个只属于一个人，这就是一对一，所以 Mybatis 种就把多对一看成一对一

mybatis 中的多表查询

示例和步骤

示例：

- 用户和账户
 - 一个用户可以有多个账户
 - 一个账户只能属于一个用户

步骤

1. 建立两张表：用户表，账户表
 - 让两个表之间具备一对多的关系：需要使用外键在账户表中添加
2. 建立两个实体类：用户实体类和账户实体类
 - 让用户和账户的实体类能体现出一对多的关系
3. 建立两个配置文件
 - 用户的配置文件
 - 账户的配置文件
4. 实现配置
 - 当查询账户时，可以同时得到用户下包含的账户信息
 - 当查询账户时，可以同时得到账户的所属用户信息

- 建立两张表

第一张表是user表，在之前早就创建了

```
1 DROP TABLE IF EXISTS `user`;
2
3 CREATE TABLE `user` (
4     `id` int(11) NOT NULL auto_increment,
5     `username` varchar(32) NOT NULL COMMENT '用户名称',
6     `birthday` datetime default NULL COMMENT '生日',
7     `sex` char(1) default NULL COMMENT '性别',
8     `address` varchar(256) default NULL COMMENT '地址',
9     PRIMARY KEY (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
11
12 insert into `user`(`id`,`username`,`birthday`,`sex`,`address`)
13     values (41,'老王','2018-02-27 17:47:08','男','北京'),
14           (42,'小二王','2018-03-02 15:09:37','女','北京金燕龙'),
15           (43,'小二王','2018-03-04 11:34:34','女','北京金燕龙'),
16           (45,'传智播客','2018-03-04 12:04:06','男','北京金燕龙'),
17           (46,'老王','2018-03-07 17:37:26','男','北京'),
18           (48,'小马宝莉','2018-03-08 11:44:00','女','北京修正');
```

外键是UID，引用user表的ID

```
1 DROP TABLE IF EXISTS `account`;
2
3 CREATE TABLE `account` (
4     `ID` int(11) NOT NULL COMMENT '编号',
5     `UID` int(11) default NULL COMMENT '用户编号',
6     `MONEY` double default NULL COMMENT '金额',
7     PRIMARY KEY (`ID`),
8     KEY `FK_Reference_8` (`UID`),
9     CONSTRAINT `FK_Reference_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
11
12 insert into `account`(`ID`,`UID`,`MONEY`) values (1,46,1000),(2,45,1000),(3,46,2000);
```

- 两个domain

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4
5 public class Account implements Serializable {
6     private Integer id;
7     private Integer uid;
8     private Double money;
9
10     public Account() {
11     }
12
13     public Account(Integer id, Integer uid, Double money) {
```

```

14         this.id = id;
15         this.uid = uid;
16         this.money = money;
17     }
18
19     public Integer getId() {
20         return id;
21     }
22
23     public void setId(Integer id) {
24         this.id = id;
25     }
26
27     public Integer getUid() {
28         return uid;
29     }
30
31     public void setUid(Integer uid) {
32         this.uid = uid;
33     }
34
35     public Double getMoney() {
36         return money;
37     }
38
39     public void setMoney(Double money) {
40         this.money = money;
41     }
42
43     @Override
44     public String toString() {
45         return "Account{" +
46             "id=" + id +
47             ", uid=" + uid +
48             ", money=" + money +
49             '}';
50     }
51 }

```

```

1  package com.bean.domain;
2
3  import java.io.Serializable;
4  import java.util.Date;
5
6  public class User implements Serializable {
7      //注释中是数据库中的名称
8      private Integer id; //id
9      private String username; //username (在windows系统下mysql数据库并不区分大小写，所以这个大小
    无所谓)
10     private Date birthday; //birthday
11     private String sex; //sex
12     private String address; //address
13
14     public User() {
15     }
16
17     public User(Integer id, String username, Date birthday, String sex, String address) {

```



```

18         this.id = id;
19         this.username = username;
20         this.birthday = birthday;
21         this.sex = sex;
22         this.address = address;
23     }
24
25     public Integer getId() {
26         return id;
27     }
28
29     public void setId(Integer id) {
30         this.id = id;
31     }
32
33     public String getUsername() {
34         return username;
35     }
36
37     public void setUsername(String username) {
38         this.username = username;
39     }
40
41     public Date getBirthday() {
42         return birthday;
43     }
44
45     public void setBirthday(Date birthday) {
46         this.birthday = birthday;
47     }
48
49     public String getSex() {
50         return sex;
51     }
52
53     public void setSex(String sex) {
54         this.sex = sex;
55     }
56
57     public String getAddress() {
58         return address;
59     }
60
61     public void setAddress(String address) {
62         this.address = address;
63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "id=" + id +
69             ", username='" + username + '\'' +
70             ", birthday=" + birthday +
71             ", sex='" + sex + '\'' +
72             ", address='" + address + '\'' +
73             '}';
74     }
75 }

```

- 两个dao

```
1 package com.bean.dao;
2
3 import com.bean.domain.Account;
4
5 import java.util.List;
6
7 public interface IAccountDao {
8     /**
9      * 查询所有账户
10     * @return
11     */
12     List<Account> findAll();
13 }
```

```
1 package com.bean.dao;
2
3 import com.bean.domain.User;
4
5 import java.util.List;
6
7 /**
8  * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
9  */
10 public interface IUserDao {
11     /**
12     * 查询所有用户
13     * @return
14     */
15     List<User> findAll();
16
17     /**
18     * 根据ID查询用户
19     * @return
20     */
21     User findUserById();
22 }
```

- 两个xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.bean.dao.IAccountDao">
7     <select id="findAll" resultType="com.bean.domain.Account">
8         select * from account
9     </select>
10 </mapper>
```

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

```

3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6      <mapper namespace="com.bean.dao.IUserDao">
7
8
9      <select id="findAll" resultType="com.bean.domain.User">
10         select * from user
11     </select>
12
13     <select id="findUserById" resultType="com.bean.domain.User">
14         select * from user
15         <where>
16             <if test="id!=null">id=#{id}</if>
17         </where>
18     </select>
19
20 </mapper>

```

完成一对一的操作

现在我们有这个需求：查询所有账户，同时查询用户。返回的信息中要有 `user` 信息和 `account` 信息

重点是：每一个Account中都有一个User信息对应，所以这是一对一的关系

因为我们要求的信息在两张表中：

- `account` : `ID`, `UID`, `MONEY`
- `user` : `id`, `username`, `birthday`, `sex`, `address`

所以查询语句应为：

```

1  select u.*,a.id as aid,a.uid,a.money from account a,user u where u.id=a.uid;
2  # 主要在于user内的id与account内的id冲突，所以要给account内的id起一个别名

```

我们发现，这里涉及到的是两张表，并且是在 `Mybatis` 中这里是一对一的关系，如果想要查询两张表中的各种内容，需要将两张表中的内容整合到一起，这里有两种方法：

- 编写一个继承类，继承Account，并且在Account上面添加User的数据，最后在 `toString` 的时候改变输出
- 在从表中实现对主表的引用，就是说在从表中引用主表中的内容

- 我们讲第二种方法，因为在实际开发过程中不可能要查一个就实现一个新的继承类，这不现实

- 在从表(Account)中实现对主表实体的引用

```

1  package com.bean.domain;
2
3  import java.io.Serializable;
4
5  public class Account implements Serializable {

```

```

6     private Integer id;
7     private Integer uid;
8     private Double money;
9
10    /*从表实体对主表实体进行的引用，并编写getter setter方法
11    * 其实话虽然比较绕口，但是实际上就是这样
12    * */
13    private User user;
14
15    public User getUser() {
16        return user;
17    }
18
19    public void setUser(User user) {
20        this.user = user;
21    }
22
23    public Integer getId() {
24        return id;
25    }
26
27    public void setId(Integer id) {
28        this.id = id;
29    }
30
31    public Integer getUid() {
32        return uid;
33    }
34
35    public void setUid(Integer uid) {
36        this.uid = uid;
37    }
38
39    public Double getMoney() {
40        return money;
41    }
42
43    public void setMoney(Double money) {
44        this.money = money;
45    }
46
47    @Override
48    public String toString() {
49        return "Account{" +
50            "id=" + id +
51            ", uid=" + uid +
52            ", money=" + money +
53            '}';
54    }
55 }

```

- 主表(User)内容不变

```

1     package com.bean.domain;
2
3     import java.io.Serializable;
4     import java.util.Date;
5

```

```
6 public class User implements Serializable {
7     //注释中是数据库中的名称
8     private Integer id; //id
9     private String username; //username (在windows系统下mysql数据库并不区分大小写，所以这个大小
    无所谓)
10    private Date birthday; //birthday
11    private String sex; //sex
12    private String address; //address
13
14    public User() {
15    }
16
17    public User(Integer id, String username, Date birthday, String sex, String address) {
18        this.id = id;
19        this.username = username;
20        this.birthday = birthday;
21        this.sex = sex;
22        this.address = address;
23    }
24
25    public Integer getId() {
26        return id;
27    }
28
29    public void setId(Integer id) {
30        this.id = id;
31    }
32
33    public String getUsername() {
34        return username;
35    }
36
37    public void setUsername(String username) {
38        this.username = username;
39    }
40
41    public Date getBirthday() {
42        return birthday;
43    }
44
45    public void setBirthday(Date birthday) {
46        this.birthday = birthday;
47    }
48
49    public String getSex() {
50        return sex;
51    }
52
53    public void setSex(String sex) {
54        this.sex = sex;
55    }
56
57    public String getAddress() {
58        return address;
59    }
60
61    public void setAddress(String address) {
62        this.address = address;
```

```

63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "id=" + id +
69             ", username='" + username + '\'' +
70             ", birthday=" + birthday +
71             ", sex='" + sex + '\'' +
72             ", address='" + address + '\'' +
73             '}';
74     }
75 }

```

- Account的接口

```

1  package com.bean.dao;
2
3  import com.bean.domain.Account;
4
5  import java.util.List;
6
7  public interface IAccountDao {
8      /**
9       * 查询所有账户
10      * @return
11      */
12      List<Account> findAll();
13
14      /**
15       * 查询所有账户，连带User表中的内容
16       * @return
17       */
18      List<Account> findAllAccountAndUser();
19  }

```

- IAccount.xml 文件实现

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IAccountDao">
7      <select id="findAll" resultType="com.bean.domain.Account">
8          select * from account
9      </select>
10
11      <!--下面是本次内容-->
12
13      <!--定义一个ResultMap，目的是为了把User和Account的内容整合到一起-->
14      <resultMap id="accountUserMap" type="account">
15          <!--这里关于为什么是aid解释一下：
16              column是表中的id值，虽然在表中的的确是id没错，
17              但是我们为了与User中的id区分，我们给Account中的id起了一个别名为aid，所以就是aid

```

```

18     -->
19     <id property="id" column="aid"></id>
20     <result property="uid" column="uid"></result>
21     <result property="money" column="money"></result>
22     <!--
23         上面的是将Account中的内容全部封装完了，接下来是重头戏：封装User
24         一对一关系映射：association
25         association
26             property: 映射的类
27             column: 用的哪个字段来获取的,在这里不写这个字段也可以，这个字段主要作用在按需查询的时候
28             javaType: 向哪个对象封装
29     -->
30     <association property="user" column="uid" javaType="user">
31         <id property="id" column="id"></id>
32         <result property="username" column="username"></result>
33         <result property="birthday" column="birthday"></result>
34         <result property="sex" column="sex"></result>
35         <result property="address" column="address"></result>
36     </association>
37 </resultMap>
38
39
40 <select id="findAllAccountAndUser" resultMap="accountUserMap" resultType="account">
41     <!--注意，这里写的是aid所以resultMap中才使用的aid-->
42     select u.*,a.id as aid,a.uid,a.money from account a,user u where u.id=a.uid;
43 </select>
44 </mapper>

```

• 测试类

```

1  package com.bean.test;
2
3  import com.bean.dao.IAccountDao;
4  import com.bean.dao.IUserDao;
5  import com.bean.domain.Account;
6  import com.bean.domain.User;
7  import org.apache.ibatis.io.Resources;
8  import org.apache.ibatis.session.SqlSession;
9  import org.apache.ibatis.session.SqlSessionFactory;
10 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
11 import org.junit.After;
12 import org.junit.Before;
13 import org.junit.Test;
14
15 import javax.annotation.Resource;
16 import java.io.IOException;
17 import java.io.InputStream;
18 import java.util.List;
19
20 public class AccountTest {
21
22     InputStream resourceAsStream;
23     SqlSession sqlSession;
24     IAccountDao accountDao;
25     @Before
26     public void Before() throws IOException {

```

```

27     resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
28     SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
29     SqlSessionFactory factory = builder.build(resourceAsStream);
30     sqlSession = factory.openSession();
31     accountDao = sqlSession.getMapper(IAccountDao.class);
32 }
33
34 @After
35 public void After() throws IOException {
36     sqlSession.commit();
37     sqlSession.close();
38     resourceAsStream.close();
39 }
40
41 @Test
42 public void AccountTest(){
43     List<Account> accounts = accountDao.findAll();
44     for (Account account : accounts) {
45         System.out.println(account);
46     }
47 }
48
49 @Test
50 public void findAllAccountAndUser(){
51     List<Account> accounts = accountDao.findAllAccountAndUser();
52     for (Account account : accounts) {
53         System.out.println("---每一个Account User---");
54         System.out.println(account.getUser());
55         System.out.println(account);
56     }
57 }
58 }
59 }

```

• 结果

```

1  ---每一个Account User---
2  User{id=46, username='老王', birthday=Wed Mar 07 17:37:26 CST 2018, sex='女', address='北京'}
3  Account{id=1, uid=46, money=1000.0}
4  ---每一个Account User---
5  User{id=45, username='传智播客', birthday=Sun Mar 04 12:04:06 CST 2018, sex='男', address='北京金燕龙'}
6  Account{id=2, uid=45, money=1000.0}
7  ---每一个Account User---
8  User{id=46, username='老王', birthday=Wed Mar 07 17:37:26 CST 2018, sex='女', address='北京'}
9  Account{id=3, uid=46, money=2000.0}

```

完成一对多的操作

首先注意一点，就是不是所有的 `User` 都有 `Account` 的，只有编号为45,46的才有 `Account`，而且46有两个 `Account`，这样就是一对多的关系

现在我们有这个需求：查询出所有 `User`，顺带查询出所有 `User` 包含的 `Account`

我们这里是一对多的关系，所以应该在主表中实现一对多的关系映射，也就是主表实体应该包含从表实体的集合引用

我们要求的信息在两张表中：

- `account` : `ID`, `UID`, `MONEY`
- `user` : `id`, `username`, `birthday`, `sex`, `address`

所以查询语句应为：

```
1 select * from user u left join account a on u.id = a.uid;
2 # 使用左外连接将user表中的所有数据全部显示出来，然后显示account中符合条件的数据
```

- `User` 类

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.Date;
5 import java.util.List;
6
7 public class User implements Serializable {
8
9     private Integer id;
10    private String username;
11    private Date birthday;
12    private String sex;
13    private String address;
14
15    /*下面我们实现一对多的关系映射：在主表(User)中实现从表(Account)实体的集合
16     * 注意一件事：accounts就是后面IUser.xml中实现resultType中collection的accounts
17     * */
18    private List<Account> accounts;
19
20    public List<Account> getAccounts() {
21        return accounts;
22    }
23
24    public void setAccounts(List<Account> accounts) {
25        this.accounts = accounts;
26    }
27
28    public Integer getId() {
29        return id;
30    }
31
32    public void setId(Integer id) {
33        this.id = id;
34    }
35
36    public String getUsername() {
37        return username;
```

```

38     }
39
40     public void setUsername(String username) {
41         this.username = username;
42     }
43
44     public Date getBirthday() {
45         return birthday;
46     }
47
48     public void setBirthday(Date birthday) {
49         this.birthday = birthday;
50     }
51
52     public String getSex() {
53         return sex;
54     }
55
56     public void setSex(String sex) {
57         this.sex = sex;
58     }
59
60     public String getAddress() {
61         return address;
62     }
63
64     public void setAddress(String address) {
65         this.address = address;
66     }
67
68     @Override
69     public String toString() {
70         return "User{" +
71             "id=" + id +
72             ", username='" + username + '\'' +
73             ", birthday=" + birthday +
74             ", sex='" + sex + '\'' +
75             ", address='" + address + '\'' +
76             '}';
77     }
78 }

```

- IUserDao

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4
5  import java.util.List;
6
7  /**
8   * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
9   */
10 public interface IUserDao {
11     /**
12      * 查询所有用户，顺带查询所有的从表信息

```

```

13      * @return
14      */
15      List<User> findAll();
16
17      /**
18       * 根据ID查询用户
19       * @return
20       */
21      User findUserById();
22  }

```

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8
9
10
11      <select id="findUserById" resultType="com.bean.domain.User">
12          select * from user
13          <where>
14              <if test="id!=null">id=#{id}</if>
15          </where>
16      </select>
17
18
19      <!--一对多的查询，type为Account-->
20      <resultMap id="userAccountMap" type="user">
21          <id property="id" column="id"></id>
22          <result property="username" column="username"></result>
23          <result property="birthday" column="birthday"></result>
24          <result property="sex" column="sex"></result>
25          <result property="address" column="address"></result>
26          <!--
27              collection: 一对多关系映射
28              property: accounts就是User类中实现从表关系映射集合的实现类
29              ofType: 集合中每个元素的类型
30          -->
31          <collection property="accounts" ofType="account">
32              <id property="id" column="id"></id>
33              <result property="uid" column="uid"></result>
34              <result property="money" column="money"></result>
35          </collection>
36      </resultMap>
37
38
39      <!--接下来实现需求：查询User表的所有信息，并且查询Account表中属于某个User的全部信息-->
40      <select id="findAll" resultMap="userAccountMap">
41          select * from user u left join account a on u.id = a.uid;
42      </select>
43

```

- 测试类 UserTest

```
1 package com.bean.test;
2
3 import com.bean.dao.IUserDao;
4 import com.bean.domain.User;
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9 import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class UserTest {
18
19     InputStream resourceAsStream;
20     SqlSession sqlSession;
21     IUserDao userDao;
22     @Before
23     public void Before() throws IOException {
24         resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
25         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
26         SqlSessionFactory factory = builder.build(resourceAsStream);
27         sqlSession = factory.openSession();
28         userDao = sqlSession.getMapper(IUserDao.class);
29     }
30
31     @After
32     public void After() throws IOException {
33         sqlSession.commit();
34         sqlSession.close();
35         resourceAsStream.close();
36     }
37
38     @Test
39     public void findAll(){
40         List<User> users = userDao.findAll();
41         for (User user : users) {
42             System.out.println("--- 每一个UserAccount ---");
43             System.out.println(user);
44             System.out.println(user.getAccounts());
45         }
46     }
47 }
```

- 结果

```
1 --- 每一个UserAccount ---
```

```
2 User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='北京'}
3 [Account{id=41, uid=null, money=null}]
4 --- 每一个UserAccount ---
5 User{id=42, username='小二王', birthday=Fri Mar 02 15:09:37 CST 2018, sex='女', address='北京金燕龙'}
6 [Account{id=42, uid=null, money=null}]
7 --- 每一个UserAccount ---
8 User{id=43, username='小二王', birthday=Sun Mar 04 11:34:34 CST 2018, sex='女', address='北京金燕龙'}
9 [Account{id=43, uid=null, money=null}]
10 --- 每一个UserAccount ---
11 User{id=45, username='传智播客', birthday=Sun Mar 04 12:04:06 CST 2018, sex='男', address='北京金燕龙'}
12 [Account{id=45, uid=45, money=1000.0}]
13 --- 每一个UserAccount ---
14 User{id=46, username='老王', birthday=Wed Mar 07 17:37:26 CST 2018, sex='女', address='北京'}
15 [Account{id=46, uid=46, money=1000.0}]
16 --- 每一个UserAccount ---
17 User{id=48, username='小马宝莉', birthday=Thu Mar 08 11:44:00 CST 2018, sex='女', address='北京修正'}
18 [Account{id=48, uid=null, money=null}]
19 --- 每一个UserAccount ---
20 User{id=49, username='new User', birthday=Thu Nov 14 10:57:53 CST 2019, sex='男', address='Address'}
21 [Account{id=49, uid=null, money=null}]
```

仔细分析一下结果，我们就会发现，不是所有的 `User` 都有 `Account`，这就是一对多

完成多对多的操作

示例：用户和角色

- 一个用户可以有多个角色
- 一个角色可以赋予多个用户

步骤：

1. 建立两张表：用户表和角色表
 - 让用户表和角色表具有多对多的关系。需要使用中间表，中间表中包含各自的主键，在中间表中是外键
2. 建立两个实体类：用户实体类和角色实体类
 - 让用户和角色的实体类能体现出现出来多对多的关系，各自包含对方的一个集合使用
3. 建立两个配置文件
 - 用户的配置文件
 - 角色的配置文件
4. 实现配置
 - 当查询角色时，可以同时得到用户所包含的角色信息
 - 当查询角色时，可以同时得到角色的所赋予的用户信息

- `user` 表

```

1 DROP TABLE IF EXISTS `user`;
2
3 CREATE TABLE `user` (
4     `id` int(11) NOT NULL auto_increment,
5     `username` varchar(32) NOT NULL COMMENT '用户名称',
6     `birthday` datetime default NULL COMMENT '生日',
7     `sex` char(1) default NULL COMMENT '性别',
8     `address` varchar(256) default NULL COMMENT '地址',
9     PRIMARY KEY (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
11
12 insert into `user`(`id`,`username`,`birthday`,`sex`,`address`) values (41,'老王','2018-02-27 17:47:08','男','北京'),(42,'小二王','2018-03-02 15:09:37','女','北京金燕龙'),(43,'小二王','2018-03-04 11:34:34','女','北京金燕龙'),(45,'传智播客','2018-03-04 12:04:06','男','北京金燕龙'),(46,'老王','2018-03-07 17:37:26','男','北京'),(48,'小马宝莉','2018-03-08 11:44:00','女','北京修正');

```

- **role** 表 (角色表)

```

1 DROP TABLE IF EXISTS `role`;
2
3 CREATE TABLE `role` (
4     `ID` int(11) NOT NULL COMMENT '编号',
5     `ROLE_NAME` varchar(30) default NULL COMMENT '角色名称',
6     `ROLE_DESC` varchar(60) default NULL COMMENT '角色描述',
7     PRIMARY KEY (`ID`)
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
9
10 insert into `role`(`ID`,`ROLE_NAME`,`ROLE_DESC`) values (1,'院长','管理整个学院'),(2,'总裁','管理整个公司'),(3,'校长','管理整个学校');

```

- **user_role** 表 (中间表)

```

1 DROP TABLE IF EXISTS `user_role`;
2
3 CREATE TABLE `user_role` (
4     `UID` int(11) NOT NULL COMMENT '用户编号',
5     `RID` int(11) NOT NULL COMMENT '角色编号',
6     PRIMARY KEY (`UID`,`RID`),
7     KEY `FK_Reference_10` (`RID`),
8     CONSTRAINT `FK_Reference_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),
9     CONSTRAINT `FK_Reference_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
11
12 insert into `user_role`(`UID`,`RID`) values (41,1),(45,1),(41,2);

```

分析一下现在的状态：

- 角色表中有几个角色
- 用户表中有几个用户
- 角色和用户表的中间表存放着 **user** 和 **role** 的对应关系

不是所有角色都有用户，也不是所有用户都有角色

现在出现了一个需求：查询所有角色并且返回该角色所赋予的用户

- 角色是主要查询的内容，用户只是顺带

根据这个需求，我们可以得到：

1. 查询角色信息
2. 根据左外连接，将 `role` 和 `user_role` 连接起来，消除笛卡尔积
3. 根据左外连接，将 `user_role` 和 `user` 连接起来，消除笛卡尔积

所以流程是：先查询角色信息，然后把角色信息和中间表连起来获取用户信息，然后根据用户信息查询所有用户

```
1 select r.id as rid,r.role_name,r.role_desc,u.* from role r left join user_role ur on
   r.id=ur.rid
2         left join user u on ur.uid=u.id;
3 # 不要中间表的信息
```

rid	role_name	role_desc	id	username	birthday	sex	address
1	院长	管理整个学院	41	老王	2018-02-27 17:47:08	男	北京
1	院长	管理整个学院	45	传智播客	2018-03-04 12:04:06	男	北京金燕龙
2	总裁	管理整个公司	41	老王	2018-02-27 17:47:08	男	北京
3	校长	管理整个学校	(Null)	(Null)	(Null)	(Null)	(Null)

- `role.class`

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.List;
5
6 public class Role implements Serializable {
7
8     private int id;
9     private String roleName;
10    private String roleDesc;
11
12    /*多对多其实是两个一对多，所以根据一对多，主表应该包含从表的集合*/
13    List<User> users;
14
15    public List<User> getUsers() {
16        return users;
17    }
18
19    public void setUsers(List<User> users) {
20        this.users = users;
21    }
22
23    public int getId() {
24        return id;
25    }
26
27    public void setId(int id) {
28        this.id = id;
29    }
30 }
```

```

29     }
30
31     public String getRoleName() {
32         return roleName;
33     }
34
35     public void setRoleName(String roleName) {
36         this.roleName = roleName;
37     }
38
39     public String getRoleDesc() {
40         return roleDesc;
41     }
42
43     public void setRoleDesc(String roleDesc) {
44         this.roleDesc = roleDesc;
45     }
46
47     @Override
48     public String toString() {
49         return "Role{" +
50             "id=" + id +
51             ", roleName='" + roleName + '\'' +
52             ", roleDesc='" + roleDesc + '\'' +
53             '}';
54     }
55 }

```

- `user.class`

```

1  package com.bean.domain;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import java.util.List;
6
7  public class User implements Serializable {
8
9      private Integer id;
10     private String username;
11     private Date birthday;
12     private String sex;
13     private String address;
14
15     /*多对多其实是两个一对多，所以根据一对多，主表应该包含从表的集合*/
16     private List<Role> roles;
17
18     public List<Role> getRoles() {
19         return roles;
20     }
21
22     public void setRoles(List<Role> roles) {
23         this.roles = roles;
24     }
25
26     public Integer getId() {
27         return id;
28     }

```



```

29
30     public void setId(Integer id) {
31         this.id = id;
32     }
33
34     public String getUsername() {
35         return username;
36     }
37
38     public void setUsername(String username) {
39         this.username = username;
40     }
41
42     public Date getBirthday() {
43         return birthday;
44     }
45
46     public void setBirthday(Date birthday) {
47         this.birthday = birthday;
48     }
49
50     public String getSex() {
51         return sex;
52     }
53
54     public void setSex(String sex) {
55         this.sex = sex;
56     }
57
58     public String getAddress() {
59         return address;
60     }
61
62     public void setAddress(String address) {
63         this.address = address;
64     }
65
66     @Override
67     public String toString() {
68         return "User{" +
69             "id=" + id +
70             ", username='" + username + '\'' +
71             ", birthday=" + birthday +
72             ", sex='" + sex + '\'' +
73             ", address='" + address + '\'' +
74             '}';
75     }
76 }

```

-
- `IRoleDao.interface`

```

1 package com.bean.dao;
2
3 import com.bean.domain.Role;
4
5 import java.util.List;
6
7 public interface IRoleDao {
8
9     List<Role> findAll();
10 }

```

- IRoleDao.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.bean.dao.IRoleDao">
6
7     <resultMap id="roleMap" type="role">
8         <id property="id" column="rid"></id>
9         <result property="roleName" column="role_name"></result>
10        <result property="roleDesc" column="role_desc"></result>
11        <collection property="users" ofType="user">
12            <id property="id" column="id"></id>
13            <result property="username" column="username"></result>
14            <result property="birthday" column="birthday"></result>
15            <result property="sex" column="sex"></result>
16            <result property="address" column="address"></result>
17        </collection>
18    </resultMap>
19
20    <select id="findAll" resultMap="roleMap">
21        select r.id as rid,r.role_name,r.role_desc,u.*
22        from role r left join user_role ur on r.id=ur.rid
23                left join user u on ur.uid=u.id;
24    </select>
25 </mapper>

```

- RoleTest.class

```

1 package com.bean.test;
2
3 import com.bean.dao.IRoleDao;
4 import com.bean.domain.Role;
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9 import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import javax.annotation.Resource;
14 import java.io.IOException;

```

```

15 import java.io.InputStream;
16 import java.util.List;
17
18 public class RoleTest {
19     InputStream inputStream;
20     SqlSessionFactoryBuilder builder;
21     SqlSession sqlSession;
22     IRoleDao roleDao;
23
24     @Before
25     public void Before() throws IOException {
26         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
27         builder = new SqlSessionFactoryBuilder();
28         SqlSessionFactory factory = builder.build(inputStream);
29         sqlSession = factory.openSession();
30         roleDao = sqlSession.getMapper(IRoleDao.class);
31     }
32
33     @After
34     public void After() throws IOException {
35         sqlSession.commit();
36         sqlSession.close();
37         inputStream.close();
38     }
39
40     @Test
41     public void findAll() {
42         List<Role> roles = roleDao.findAll();
43         for (Role role : roles) {
44             System.out.println("--- 所有角色及角色赋予的人 ---");
45             System.out.println(role);
46             System.out.println(role.getUsers());
47         }
48     }
49
50 }

```

• 结果

```

1 --- 所有角色及角色赋予的人 ---
2 Role{id=1, roleName='院长', roleDesc='管理整个学院'}
3 [User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='北京'},
  User{id=45, username='传智播客', birthday=Sun Mar 04 12:04:06 CST 2018, sex='男', address='北京金燕龙'}]
4 --- 所有角色及角色赋予的人 ---
5 Role{id=2, roleName='总裁', roleDesc='管理整个公司'}
6 [User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='北京'}]
7 --- 所有角色及角色赋予的人 ---
8 Role{id=3, roleName='校长', roleDesc='管理整个学校'}
9 []

```

- 从用户到角色和从角色到用户是类似的，只是 `sql` 语句改变一下而已

mybatis 中的延迟加载

- 情景：在一对多中，我们有一个用户，这个用户创建了100个账户

现在有两个问题：

1. 在查询用户时，要不要把关联的账户查出来？
2. 在查询账户时，要不要把关联的用户查出来？

分析：

1. 用户有100个账户，但是查询用户的时候不一定要全部查出来，这样浪费空间，而且没有必要
2. 每个账户对应1个用户，不把用户信息查出来那不知道是谁的信息，所以要查出来

我们可以看到，根据上面两个问题有两种不同的解决方法

- 扩展数据只有使用到对应数据的时候才发起查询，不用就不查询
- 扩展数据一调用方法，马上发起查询

对应的四中表的关系中：一对一，一对多，多对一，多对多

什么是立即加载

- 立即加载：一调用方法，马上发起查询
- 多对一，一对一：通常情况下采用立即加载

- mybatis 中没有多对一的概念
- 我们平常使用的加载就是立即加载

什么是延迟加载

- 延迟加载：只有使用到对应数据的时候才发起查询，不用就不查询。按需加载（懒加载）
- 一对多，多对多：通常情况下采用延迟加载

实现延迟加载

一对一实现延迟加载

需求：查询用户账户的时候延迟查询用户

我们现在就使用 `Account` 和 `User` 进行一对一延迟加载的操作，其实一对一得到时候应该是直接加载，但是这里讲一下延迟加载

首先开启延迟加载和按需加载，要在 mybatis 中的配置文件中进行设置

- SqlMapConfig.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7
8     <properties
9         url="file:///E:/JetBrains/java/intelliJ%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>
10
11 <!-- 首先注意顺序，settings必须放在这里，要找settings配置可以去mybatis中文网找：
12      https://mybatis.org/mybatis-3/zh/configuration.html#settings
13      settings: mybatis的全部设置
14      lazyLoadingEnabled: 延迟加载的全局开关
15          true: 开启延迟加载
16          false: 关闭延迟加载
17      aggressiveLazyLoading: 是否取消按需加载
18          true: 确认关闭按需加载
19          false: 不确认关闭按需加载（按需加载）
20 -->
21 <settings>
22     <setting name="lazyLoadingEnabled" value="true"/>
23     <setting name="aggressiveLazyLoading" value="false"/>
24 </settings>
25
26 <typeAliases>
27     <package name="com.bean.domain"></package>
28 </typeAliases>
29
30 <environments default="mysql">
31     <environment id="mysql">
32         <transactionManager type="JDBC"></transactionManager>
33         <dataSource type="POOLED">
34             <property name="driver" value="${jdbc.driver}"/>
35             <property name="url" value="${jdbc.url}"/>
36             <property name="username" value="${jdbc.username}"/>
37             <property name="password" value="${jdbc.password}"/>
38         </dataSource>
39     </environment>
40 </environments>
41
42 <mappers>
43     <package name="com.bean.dao"></package>
44 </mappers>
45
46 </configuration>
```

- Account.class

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4
```

```

5  public class Account implements Serializable {
6      private int id;
7      private int uid;
8      private int money;
9
10     /*把User放进来*/
11     private User user;
12
13     public User getUser() {
14         return user;
15     }
16
17     public void setUser(User user) {
18         this.user = user;
19     }
20
21     public int getId() {
22         return id;
23     }
24
25     public void setId(int id) {
26         this.id = id;
27     }
28
29     public int getUid() {
30         return uid;
31     }
32
33     public void setUid(int uid) {
34         this.uid = uid;
35     }
36
37     public int getMoney() {
38         return money;
39     }
40
41     public void setMoney(int money) {
42         this.money = money;
43     }
44
45     @Override
46     public String toString() {
47         return "Account{" +
48             "id=" + id +
49             ", uid=" + uid +
50             ", money=" + money +
51             '}';
52     }
53 }

```

- IAccountDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

```

```

5  <mapper namespace="com.bean.dao.IAccountDao">
6
7      <resultMap id="accountMap" type="account">
8          <id property="id" column="id"></id>
9          <result property="uid" column="uid"></result>
10         <result property="money" column="money"></result>
11     <!--
12         associaion: 进行一对一关系映射
13         property: 映射的类
14         column: 用哪个字段来按需获取
15         javaType: 向哪个对象进行封装
16         select: 查询用户的唯一标志, 需要的是column里面的内容, column就是需求
17     -->
18     <association property="user" column="uid" javaType="user"
19     select="com.bean.dao.IUserDao.findById">
20     <!--以下就不需要了, 因为没有查数据的时候是不可能进行封装的
21         查数据的时候使用的是IUserDao.findById, 也就是在IUserDao.xml中实现的findById接口
22         <id property="id" column="id"></id>
23         <result property="username" column="username"></result>
24         <result property="birthday" column="birthday"></result>
25         <result property="sex" column="sex"></result>
26         <result property="address" column="address"></result>
27     -->
28     </association>
29 </resultMap>
30
31 <!--查询所有账户信息并且返回对应的用户信息-->
32 <select id="findAll" resultMap="accountMap">
33     select * from account;
34 </select>
35 </mapper>

```

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <select id="findUserById" parameterType="int" resultType="com.bean.domain.User" >
9          select * from user where id=#{id};
10     </select>
11
12     <resultMap id="userMap" type="user">
13         <id property="id" column="id"></id>
14         <result property="username" column="username"></result>
15         <result property="birthday" column="birthday"></result>
16         <result property="sex" column="sex"></result>
17         <result property="address" column="address"></result>
18     </resultMap>
19
20     <select id="findAll" resultMap="userMap">
21         select * from user u left join account a on u.id=a.uid;
22     </select>
23

```

```
24      <!--实现findById这个方法-->
25      <select id="findById" resultType="user">
26          select * from user where id=#{id}
27      </select>
28  </mapper>
```

- `IAccountDao.interface`

```
1  package com.bean.dao;
2
3  import com.bean.domain.Account;
4
5  import java.util.List;
6
7  public interface IAccountDao {
8
9      /**
10       * 查询所有账户信息，实现按需加载对应的账户信息
11       * @return
12       */
13      List<Account> findAll();
14
15  }
```

- `AccountTest.class`

```
1  package com.bean.test;
2
3  import com.bean.dao.IAccountDao;
4  import com.bean.domain.Account;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class AccountTest {
18     InputStream resourceAsStream;
19     SqlSession sqlSession;
20     IAccountDao accountDao;
21
22     @Before
23     public void Before() throws IOException {
24
25         resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
26         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
27         SqlSessionFactory factory = builder.build(resourceAsStream);
28     }
```



```

29         sqlSession = factory.openSession();
30         accountDao = sqlSession.getMapper(IAccountDao.class);
31     }
32
33     @After
34     public void After() throws IOException {
35         sqlSession.commit();
36         sqlSession.close();
37         resourceAsStream.close();
38     }
39
40     @Test
41     public void find() {
42         List<Account> accounts = accountDao.findAll();
43     }
44
45     @Test
46     public void findAccount() {
47         List<Account> accounts = accountDao.findAll();
48         for (Account account : accounts) {
49             /*这里注意了，这里只输出了account，但是没有输出user的内容*/
50             System.out.println("--- 只有account的内容 ---");
51             System.out.println(account);
52         }
53     }
54
55     @Test
56     public void findAccountAndUser() {
57         List<Account> accounts = accountDao.findAll();
58         for (Account account : accounts) {
59             /*这里是输出了User*/
60             System.out.println("--- user与account的内容 ---");
61             System.out.println(account.getUser());
62             System.out.println(account);
63         }
64     }
65
66 }

```

上面的代码结果是这样的：

- 查询并输出 `user` 和 `account`

```

2019-11-20 15:54:50,040 655 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection
2019-11-20 15:54:50,371 986 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1227074340.
2019-11-20 15:54:50,372 987 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:54:50,374 989 [ main] DEBUG m.bean.dao.IAccountDao.findAll - ==> Preparing: select * from account;
2019-11-20 15:54:50,414 1029 [ main] DEBUG m.bean.dao.IAccountDao.findAll - ==> Parameters:
2019-11-20 15:54:50,501 1116 [ main] DEBUG m.bean.dao.IAccountDao.findAll - <== Total: 3
--- user与account的内容 ---
2019-11-20 15:54:50,502 1117 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Preparing: select * from user where id=?
2019-11-20 15:54:50,503 1118 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Parameters: 46(Integer)
2019-11-20 15:54:50,512 1127 [ main] DEBUG com.bean.dao.IUserDao.findById - <== Total: 1
User{id=46, username='老王', birthday=Wed Mar 07 17:37:26 CST 2018, sex='女', address='北京'}
Account{id=1, uid=46, money=1000}
--- user与account的内容 ---
2019-11-20 15:54:50,512 1127 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Preparing: select * from user where id=?
2019-11-20 15:54:50,513 1128 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Parameters: 45(Integer)
2019-11-20 15:54:50,518 1133 [ main] DEBUG com.bean.dao.IUserDao.findById - <== Total: 1
User{id=45, username='传智播客', birthday=Sun Mar 04 12:04:06 CST 2018, sex='男', address='北京金燕龙'}
Account{id=2, uid=45, money=1000}
--- user与account的内容 ---
User{id=46, username='老王', birthday=Wed Mar 07 17:37:26 CST 2018, sex='女', address='北京'}
Account{id=3, uid=46, money=2000}
2019-11-20 15:54:50,519 1134 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:54:50,520 1135 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:54:50,520 1135 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1227074340 to pool.

```

- 查询并输出 `account`

```

Account{id=1, uid=46, money=1000}
--- 只有account的内容 ---
2019-11-20 15:53:23,053 987 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Preparing: select * from user where id=?
2019-11-20 15:53:23,053 987 [ main] DEBUG com.bean.dao.IUserDao.findById - ==> Parameters: 45(Integer)
2019-11-20 15:53:23,057 991 [ main] DEBUG com.bean.dao.IUserDao.findById - <== Total: 1
Account{id=2, uid=45, money=1000}
--- 只有account的内容 ---
Account{id=3, uid=46, money=2000}
2019-11-20 15:53:23,058 992 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:53:23,059 993 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc
.jdbc.JDBC4Connection@4923ab24]

```

- 只查询，不输出

```

2019-11-20 15:51:27,257 576 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection
2019-11-20 15:51:27,668 987 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1227074340.
2019-11-20 15:51:27,669 988 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:51:27,673 992 [ main] DEBUG m.bean.dao.IAccountDao.findAll - ==> Preparing: select * from account;
2019-11-20 15:51:27,756 1075 [ main] DEBUG m.bean.dao.IAccountDao.findAll - ==> Parameters:
2019-11-20 15:51:27,874 1193 [ main] DEBUG m.bean.dao.IAccountDao.findAll - <== Total: 3
2019-11-20 15:51:27,875 1194 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:51:27,876 1195 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc
.jdbc.JDBC4Connection@4923ab24]
2019-11-20 15:51:27,877 1196 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1227074340 to pool.

```

此时，我们可以说是完全地实现了我们的需求：不需要就不查询

一对多实现延迟加载

需求：查询用户的时候延迟查询用户的账户

- `IAccountDao`

```

1 package com.bean.dao;
2
3 import com.bean.domain.Account;
4
5 import java.util.List;
6
7 public interface IAccountDao {
8     List<Account> findAccountByUid(Integer uid);
9 }

```

- IUserDao

```
1 package com.bean.dao;
2
3 import com.bean.domain.User;
4
5 import java.util.List;
6
7
8 public interface IUserDao {
9
10     /**
11      * 查询所有User对象，顺便实现延迟加载Account
12      * @return
13      */
14     List<User> findAll();
15 }
```

- User.class

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.Date;
5 import java.util.List;
6
7 public class User implements Serializable {
8     private int id;
9     private String username;
10    private Date birthday;
11    private String sex;
12    private String address;
13
14    /*加入Account*/
15    private List<Account> accounts;
16
17    public List<Account> getAccounts() {
18        return accounts;
19    }
20
21    public void setAccounts(List<Account> accounts) {
22        this.accounts = accounts;
23    }
24
25    public int getId() {
26        return id;
27    }
28
29    public void setId(int id) {
30        this.id = id;
31    }
32
33    public String getUsername() {
34        return username;
35    }
36 }
```

```

37     public void setUsername(String username) {
38         this.username = username;
39     }
40
41     public Date getBirthday() {
42         return birthday;
43     }
44
45     public void setBirthday(Date birthday) {
46         this.birthday = birthday;
47     }
48
49     public String getSex() {
50         return sex;
51     }
52
53     public void setSex(String sex) {
54         this.sex = sex;
55     }
56
57     public String getAddress() {
58         return address;
59     }
60
61     public void setAddress(String address) {
62         this.address = address;
63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "id=" + id +
69             ", username='" + username + '\'' +
70             ", birthday=" + birthday +
71             ", sex='" + sex + '\'' +
72             ", address='" + address + '\'' +
73             '}';
74     }
75 }

```

- IUserDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.IUserDao">
7
8      <resultMap id="userMap" type="com.bean.domain.User">
9          <id property="id" column="id"></id>
10         <result property="username" column="username"></result>
11         <result property="birthday" column="birthday"></result>
12         <result property="sex" column="sex"></result>
13         <result property="address" column="address"></result>
14     </resultMap>

```

```

15         <!--collection: 一对多关系映射
16             property: 实现从表关系映射的集合类
17             ofType: 集合中的每一个元素的类型
18             column: 实现延迟加载的查询方法的参数
19             select: 实现延迟加载的查询方法
20             注意: 这个时候就不要写javaType了, 因为javaType的使用是需要参数的, 而这里直接调用了查询方法,
                所以不需要参数
21         -->
22         <collection property="accounts" ofType="account" column="id"
                select="com.bean.dao.IAccountDao.findAccountByUid"></collection>
23
24     </resultMap>
25
26     <select id="findAll" resultMap="userMap">
27         select * from user
28     </select>
29
30 </mapper>

```

- IAccountDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5  <mapper namespace="com.bean.dao.IAccountDao">
6
7      <select id="findAccountByUid" parameterType="int" resultType="com.bean.domain.Account">
8          select * from account where uid=#{uid}
9      </select>
10
11  </mapper>

```

- UserTest.class

```

1  package com.bean.test;
2
3  import com.bean.dao.IUserDao;
4  import com.bean.domain.User;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSession;
7  import org.apache.ibatis.session.SqlSessionFactory;
8  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import java.io.IOException;
14 import java.io.InputStream;
15 import java.util.List;
16
17 public class UserTest {
18
19     InputStream resourceAsStream;
20     SqlSession sqlSession;
21     IUserDao userDao;
22     @Before

```

```

23     public void Before() throws IOException {
24         resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
25         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
26         SqlSessionFactory factory = builder.build(resourceAsStream);
27         sqlSession = factory.openSession();
28         userDao = sqlSession.getMapper(IUserDao.class);
29     }
30
31     @After
32     public void After() throws IOException {
33         sqlSession.commit();
34         sqlSession.close();
35         resourceAsStream.close();
36     }
37
38     @Test
39     public void find(){
40         List<User> users = userDao.findAll();
41     }
42
43     @Test
44     public void findUser(){
45         List<User> users = userDao.findAll();
46         for (User user : users) {
47             System.out.println("---查询user---");
48             System.out.println(user);
49         }
50     }
51
52
53     @Test
54     public void findUserAndAccount(){
55         List<User> users = userDao.findAll();
56         for (User user : users) {
57             System.out.println("---查询user和account---");
58             System.out.println(user);
59             System.out.println(user.getAccounts());
60         }
61     }
62 }

```

mybatis 中的缓存

什么是缓存

- 存在于内存中的临时数据

为什么使用缓存

- 减少和数据库的交互次数，提高执行效率

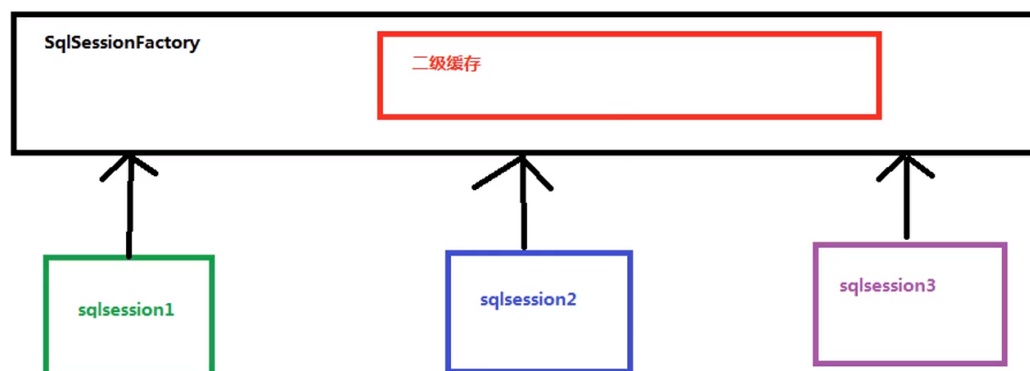
缓存可以作用在什么数据

- 适用于缓存的数据
 - 经常查询并且不经常改变的
 - 数据的正确与否对最终结果影响不大的
- 不适用于缓存
 - 经常改变的数据
 - 数据的正确与否对最终结果影响很大的
 - 例如：商品的库存，银行的汇率，股市的牌价

因为使用缓存之后就会产生与数据库的数据分离，数据库的数据改变之后缓存内容也不会改变，读数据库缓存的时候假如缓存和数据差别影响很大就不适用

mybatis 中的一级缓存和二级缓存

- 一级缓存
 - 指的是 Mybatis 中 `SqlSession` 对象的缓存
 - 当我们执行查询之后，查询的结果会同时存入到 `SqlSession` 为我们提供一块区域中
 - 该区域的结构是一个 `Map`，当我们再次查询同样的内容，会先去 `SqlSession` 中查询是否有，有则直接拿出来用
 - `SqlSession` 对象消失时，mybatis 的一级缓存也就消失了
 - `sqlSession.clearCache()`：清空缓存，也就是清空 `sqlSession` 中的一级缓存
 - mybatis 中的一级缓存当调用 `sqlSession` 的修改，添加，删除，`commit()`，`close()` 等方法就会清空一级缓存，也就是当数据和缓存不同步时就会清空缓存
- 二级缓存
 - 指的是 mybatis 中 `sqlSessionFactory` 对象的缓存，由同一个 `SqlSessionFactory` 对象创建的 `SqlSessionFactory` 共享其缓存



- 二级缓存的使用步骤
 1. 让 mybatis 框架支持二级缓存（在 `SqlMapConfig.xml` 中配置）
 2. 让当前的映射文件支持二级缓存（在 `IUserDao.xml` 中配置）
 3. 让当前的操作支持二级缓存（在 `select` 标签中配置）
- 二级缓存中存放的是数据而不是地址，每来一个类型那这些数据，都会创建对应的类型对象填到相应对象里面去。所以判断缓存中是否相等的时候是 `false`，因为这两个对象不是同一个了
- `SqlMapConfig.xml`

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  
```

```

4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6      <configuration>
7
8
9          <properties
url="file:///E:/JetBrains/java/intellj%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfi
g.properties"></properties>
10
11          <settings>
12              <!--延迟加载-->
13              <setting name="lazyLoadingEnabled" value="true"/>
14              <setting name="aggressiveLazyLoading" value="false"/>
15
16              <!--二级缓存开启-->
17              <setting name="cacheEnabled" value="true"/>
18          </settings>
19
20          <typeAliases>
21              <package name="com.bean.domain"></package>
22          </typeAliases>
23
24
25
26
27          <environments default="mysql">
28              <environment id="mysql">
29                  <transactionManager type="JDBC"></transactionManager>
30                  <dataSource type="POOLED">
31                      <property name="driver" value="${jdbc.driver}"/>
32                      <property name="url" value="${jdbc.url}"/>
33                      <property name="username" value="${jdbc.username}"/>
34                      <property name="password" value="${jdbc.password}"/>
35                  </dataSource>
36              </environment>
37
38          </environments>
39          <mappers>
40              <package name="com.bean.dao"></package>
41          </mappers>
42
43      </configuration>

```

- IUserDao.xml

```

1      <?xml version="1.0" encoding="UTF-8"?>
2      <!DOCTYPE mapper
3          PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4          "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6      <mapper namespace="com.bean.dao.IUserDao">
7
8          <!--开启二级缓存-->
9          <cache/>
10
11          <!--设置可以使用二级缓存-->
12          <select id="findAll" resultType="user" useCache="true">
13              select * from user

```



```
14     </select>
15
16 </mapper>
```

mybatis 中的注解开发

首先注意一件事：mybatis 中的注解开发是替代sql语句，对于 mybatis 配置文件 SqlMapConfig.xml 没有注解

环境搭建

重新开始创建项目，开始环境搭建

- 选择maven工程
- 写 SqlMapConfig.xml
- 写 User.class
- 写 IUser.interface
- 写注解

-
- SqlMapConfig.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5  <configuration>
6      <!--引入外部配置文件-->
7      <properties resource="jdbcConfig.properties"></properties>
8
9      <!--配置别名-->
10     <typeAliases>
11         <package name="com.bean.domain"/>
12     </typeAliases>
13
14     <environments default="mysql">
15         <environment id="mysql">
16             <transactionManager type="JDBC"></transactionManager>
17             <dataSource type="POOLED">
18                 <property name="driver" value="${jdbc.driver}"/>
19                 <property name="url" value="${jdbc.url}"/>
20                 <property name="username" value="${jdbc.username}"/>
21                 <property name="password" value="${jdbc.password}"/>
22             </dataSource>
23         </environment>
24     </environments>
25
26     <!--指定带有注解dao接口所在的位置-->
```

```
27     <mappers>
28         <package name="com.bean.dao"/>
29     </mappers>
30 </configuration>
```

- jdbcConfig.properties

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatis
3 jdbc.username=root
4 jdbc.password=root
```

- 加入 log4j 文件

-
- com.bean.domain.User.class

```
1 package com.bean.domain;
2
3 import java.io.Serializable;
4 import java.util.Date;
5
6 public class User implements Serializable {
7     private int id;
8     private String username;
9     private Date birthday;
10    private String sex;
11    private String address;
12
13    public User() {
14    }
15
16    public User(int id, String username, Date birthday, String sex, String address) {
17        this.id = id;
18        this.username = username;
19        this.birthday = birthday;
20        this.sex = sex;
21        this.address = address;
22    }
23
24    public int getId() {
25        return id;
26    }
27
28    public void setId(int id) {
29        this.id = id;
30    }
31
32    public String getUsername() {
33        return username;
34    }
35
36    public void setUsername(String username) {
37        this.username = username;
38    }
39
40    public Date getBirthday() {
41        return birthday;
```

```

42     }
43
44     public void setBirthday(Date birthday) {
45         this.birthday = birthday;
46     }
47
48     public String getSex() {
49         return sex;
50     }
51
52     public void setSex(String sex) {
53         this.sex = sex;
54     }
55
56     public String getAddress() {
57         return address;
58     }
59
60     public void setAddress(String address) {
61         this.address = address;
62     }
63
64     @Override
65     public String toString() {
66         return "User{" +
67             "id=" + id +
68             ", username='" + username + '\'' +
69             ", birthday=" + birthday +
70             ", sex='" + sex + '\'' +
71             ", address='" + address + '\'' +
72             '}';
73     }
74 }

```

- com.bean.dao.IUserDao.interface

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4  import org.apache.ibatis.annotations.Select;
5
6  import java.util.List;
7
8  public interface IUserDao {
9      /*
10      * mybatis中对于注解开发一共有四个注解
11      * @Select @Insert @Update @Delete
12      * 以@Select举例:
13      *   @Select(value="语句"), 当然, 参数只有一个的时候可以省略value
14      * */
15
16
17      /**
18      * 查询所有用户
19      * @return
20      */

```

```

21     @Select("select * from user")
22     List<User> findAll();
23 }

```

细节

- 当使用注解进行开发的时候，不论是不是使用了 `xml` 实现 `sql` 语句，只要存在 `xml` 就会报错

- 比如现在使用注解实现 `IUserDao.interface` 中的语句，只要存在 `IUserDao.xml`，就会报错
- 就算现在在 `SqlMapConfig.xml` 没配置 `xml` 文件，也会报错
- 除非调整到不相关的路径下去，因为 `java.com.bean.dao.IUserDao.interface` 和 `resource.com.bean.dao.IUserDao.xml` 在 `mybatis` 的路径是对应的
- 删掉解决 100% 的问题

单表CRUD操作(代理Dao的方式)

- Select
- Insert
- Update
- Delete

- `IUserDao.interface`

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4  import org.apache.ibatis.annotations.Delete;
5  import org.apache.ibatis.annotations.Insert;
6  import org.apache.ibatis.annotations.Select;
7  import org.apache.ibatis.annotations.Update;
8
9  import java.util.List;
10
11  public interface IUserDao {
12
13      /**
14       * 查询所有方法
15       * @return
16       */
17      @Select("select * from user")
18      List<User> findAll();
19
20      /**
21       * 插入用户
22       */
23      @Insert("insert into user(username,birthday,sex,address) values(#{username},#{
24      {birthday}},#{sex},#{address})")
25      void insertUser(User user);
26
27      /**
28       * 更新用户

```

```

28     * @param user
29     */
30     @Update("update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{
address} where id=#{id}")
31     void updateUser(User user);
32
33     /**
34     * 删除操作
35     * @param id
36     */
37     @Delete("delete from user where id=#{id}")
38     void deleteUser(Integer id);
39
40     /**
41     * 根据id查询用户
42     * @return
43     */
44     @Select("select * from user where id=#{id}")
45     User findUserById(Integer id);
46
47     /**
48     * 根据名称进行模糊查询
49     * @param username
50     * @return
51     */
52     @Select("select * from user where username like #{username}")
53     List<User> findUsersLikeUsername(String username);
54
55     /**
56     * 使用聚合函数查询总人数
57     * @return
58     */
59     @Select("select count(id) from user")
60     int findCount();
61
62 }

```

- `UserTest.class`

```

1  import com.bean.dao.IUserDao;
2  import com.bean.domain.User;
3  import org.apache.ibatis.io.Resources;
4  import org.apache.ibatis.session.SqlSession;
5  import org.apache.ibatis.session.SqlSessionFactory;
6  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
7  import org.junit.After;
8  import org.junit.Before;
9  import org.junit.Test;
10
11  import java.io.IOException;
12  import java.io.InputStream;
13  import java.util.Date;
14  import java.util.List;
15  import java.util.Properties;
16
17  public class UserTest {
18
19      InputStream inputStream;

```

```

20     SqlSession session;
21     IUserDao userDao;
22     @Before
23     public void Before() throws IOException {
24
25         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
26         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
27         SqlSessionFactory factory = builder.build(inputStream);
28         session = factory.openSession();
29         userDao = session.getMapper(IUserDao.class);
30     }
31
32     @After
33     public void After() throws IOException {
34         session.commit();
35         session.close();
36         inputStream.close();
37     }
38
39
40     @Test
41     public void findAll(){
42         List<User> users = userDao.findAll();
43         for (User user : users) {
44             System.out.println(user);
45         }
46     }
47
48     @Test
49     public void insertUser(){
50         User user = new User();
51         user.setUsername("annotation");
52         user.setSex('男');
53         user.setBirthday(new Date());
54         user.setAddress("annotationAddress");
55         userDao.insertUser(user);
56     }
57
58     @Test
59     public void updateUser(){
60         User user = new User();
61         user.setId(49);
62         user.setUsername("annotation");
63         user.setAddress("annotationAddress");
64         user.setSex('男');
65         user.setBirthday(new Date());
66         userDao.updateUser(user);
67     }
68
69     @Test
70     public void deleteUser(){
71         userDao.deleteUser(49);
72     }
73
74     @Test
75     public void findUserById(){
76         User user = userDao.findUserById(48);
77         System.out.println(user);

```

```

78     }
79
80     @Test
81     public void findUsersLikeUsername(){
82         List<User> users = userDao.findUsersLikeUsername("%王%");
83         for (User user : users) {
84             System.out.println(user);
85         }
86     }
87
88     @Test
89     public void findCount(){
90         System.out.println(userDao.findCount());
91     }
92
93 }

```

细节

- 在数据库中表的字段名和实际的类的字段名对应不起来的时候

使用注解解决方案:

- @Results
- @Result
- @ResultMap

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4  import org.apache.ibatis.annotations.*;
5
6  import java.util.List;
7
8  public interface IUserDao {
9
10     /*
11         @Results: 定义类属性与数据库字段的对应关系
12             - id: 唯一标识, @ResultMap可以根据这个找到
13             - value: 类属性与表字段的映射
14             - Result: 单个字段的对应关系
15                 - id: 是否是主键, 默认为false
16                 - property: 类中的属性
17                 - column: 数据库中字段名称
18         @ResultMap: 引用@Results
19             - value: 引用的@Results值, 假如只有一个可以去掉value和括号
20     */
21     /**
22     * 查询所有方法
23     * @return
24     */
25     @Select("select * from user")
26     @Results(
27         id="userMap",
28         value = {

```

```

29         @Result(id = true, property = "id", column = "id"),
30         @Result(property = "username", column = "username"),
31         @Result(property = "address", column = "address"),
32         @Result(property = "sex", column = "sex")
33     }
34 )
35 List<User> findAll();
36
37 /**
38  * 根据id查询用户
39  * @return
40  */
41 @Select("select * from user where id=#{id}")
42 @ResultMap("userMap")
43 User findUserById(Integer id);
44
45 /**
46  * 根据名称进行模糊查询
47  * @param username
48  * @return
49  */
50 @Select("select * from user where username like #{username}")
51 @ResultMap("userMap")
52 List<User> findUsersLikeUsername(String username);
53
54 }

```

- 当然，现在是能对应起来的，所以写不写这个都无所谓，对应不起来的时候写这个就很有用
- 而且注意一件事，`@Results` 注解不能抽离出去然后所有的都使用 `@ResultMap` 来引用，只有在一个方法上生效之后其余的才可以引用

多表查询操作

一对一操作

使用一对一操作就需要使用注解配置，以下这三个主要的功能并不是进行字段与数据对齐，而是进行数据的封装

- `@Results`
 - `@Result`
 - `@ResultMap`
- `Account.class`

```

1 package com.bean.domain;
2
3 public class Account {
4     private Integer id;
5     private Integer uid;
6     private Integer money;
7
8
9     /*要实现一对一的加载，就需要User数据*/

```



```

10     private User user;
11
12     public User getUser() {
13         return user;
14     }
15
16     public void setUser(User user) {
17         this.user = user;
18     }
19
20
21     public Integer getId() {
22         return id;
23     }
24
25     public void setId(Integer id) {
26         this.id = id;
27     }
28
29     public Integer getUid() {
30         return uid;
31     }
32
33     public void setUid(Integer uid) {
34         this.uid = uid;
35     }
36
37     public Integer getMoney() {
38         return money;
39     }
40
41     public void setMoney(Integer money) {
42         this.money = money;
43     }
44
45     @Override
46     public String toString() {
47         return "Account{" +
48             "id=" + id +
49             ", uid=" + uid +
50             ", money=" + money +
51             '}';
52     }
53 }

```

- IUserDao.interface

```

1     package com.bean.dao;
2
3     import com.bean.domain.User;
4     import org.apache.ibatis.annotations.*;
5
6     import java.util.List;
7
8     public interface IUserDao {
9

```

```

10      /**
11       * 查询所有方法
12       * @return
13       */
14      @Select("select * from user")
15      @Results(
16          id="userMap",
17          value = {
18              @Result(id = true,property = "id",column = "id"),
19              @Result(property = "username",column = "username"),
20              @Result(property = "address",column = "address"),
21              @Result(property = "sex",column = "sex")
22          }
23      )
24      List<User> findAll();
25
26      /**
27       * 根据id查询用户
28       * @return
29       */
30      @Select("select * from user where id=#{id}")
31      @ResultMap("userMap")
32      User findUserById(Integer id);
33
34      /**
35       * 根据名称进行模糊查询
36       * @param username
37       * @return
38       */
39      @Select("select * from user where username like #{username}")
40      @ResultMap("userMap")
41      List<User> findUsersLikeUsername(String username);
42
43  }

```

- IAccountDao.interface

```

1  package com.bean.dao;
2
3  import com.bean.domain.Account;
4  import org.apache.ibatis.annotations.*;
5  import org.apache.ibatis.mapping.FetchType;
6
7  import java.util.List;
8
9  public interface IAccountDao {
10
11      /**首先实现数据的封装，将User也封装到这个里面*/
12      /**
13       @Results: 定义类属性与数据库字段的对应关系
14       - id: 唯一标识，@ResultMap可以根据这个找到
15       - value: 类属性与表字段的映射
16       - @Result: 单个字段的对应关系
17       - id: 是否是主键，默认为false
18       - property: 类中的属性，如果使用多表查询，那么对应着的是封装到哪个对象中
19       - column: 数据库中字段名称，如果使用多表查询，那么对应着的是查询所需要的参数

```

```

20         - one: 一对一（mybatis中不存在多对一）的时候使用的那个一
21         - @One
22         - select: 查询使用的方法，要求的是全限定类名
23         - fetchType: 如何进行加载
24             - FetchType.EAGER: 立即加载
25             - FetchType.LAZY: 懒加载
26             - FetchType.DEFAULT: 默认，可能是随意选一个
27     @ResultMap: 引用@Results
28         - value: 引用的@Results值，假如只有一个可以去掉value和括号
29     */
30
31
32     /**
33      * 查询所有内容，顺便实现延迟懒加载用户信息内容，实现一对一操作
34      * @return
35      */
36     @Select("select * from account")
37     @Results(
38         id = "accountMap",
39         value = {
40             @Result(id = true, property = "id", column = "id"),
41             @Result(property = "uid", column = "uid"),
42             @Result(property = "money", column = "money"),
43             @Result(
44                 property = "user",
45                 column = "uid",
46                 one = @One(select =
47                     "com.bean.dao.IUserDao.findUserById", fetchType = FetchType.EAGER)
48             )
49         }
50     )
51     List<Account> findAll();

```

- AccountTest.class

```

1     import com.bean.dao.IAccountDao;
2     import com.bean.domain.Account;
3     import org.apache.ibatis.io.Resources;
4     import org.apache.ibatis.session.SqlSession;
5     import org.apache.ibatis.session.SqlSessionFactory;
6     import org.apache.ibatis.session.SqlSessionFactoryBuilder;
7     import org.junit.After;
8     import org.junit.Before;
9     import org.junit.Test;
10
11     import java.io.IOException;
12     import java.io.InputStream;
13     import java.util.List;
14
15     public class AccountTest {
16
17         InputStream inputStream;
18         SqlSession session;
19         IAccountDao accountDao;
20         @Before

```

```

21     public void Before() throws IOException {
22
23         inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
24         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
25         SqlSessionFactory factory = builder.build(inputStream);
26         session = factory.openSession();
27         accountDao = session.getMapper(IAccountDao.class);
28     }
29
30     @After
31     public void After() throws IOException {
32         session.commit();
33         session.close();
34         inputStream.close();
35     }
36
37
38     @Test
39     public void findAll(){
40         List<Account> accounts = accountDao.findAll();
41         for (Account account : accounts) {
42             System.out.println("-----");
43             System.out.println(account);
44             System.out.println(account.getUser());
45         }
46     }
47
48 }

```

一对多的操作

- User.class

```

1     package com.bean.domain;
2
3     import java.util.Date;
4     import java.util.List;
5
6     public class User {
7         private Integer id;
8         private String username;
9         private Date birthday;
10        private Character sex;
11        private String address;
12
13
14        /*首先我们使用一对多的映射，将Account的集合封装进来*/
15        private List<Account> accounts;
16
17        public List<Account> getAccounts() {
18            return accounts;
19        }
20
21        public void setAccounts(List<Account> accounts) {
22            this.accounts = accounts;
23        }

```

```

24
25     public Integer getId() {
26         return id;
27     }
28
29     public void setId(Integer id) {
30         this.id = id;
31     }
32
33     public String getUsername() {
34         return username;
35     }
36
37     public void setUsername(String username) {
38         this.username = username;
39     }
40
41     public Date getBirthday() {
42         return birthday;
43     }
44
45     public void setBirthday(Date birthday) {
46         this.birthday = birthday;
47     }
48
49     public Character getSex() {
50         return sex;
51     }
52
53     public void setSex(Character sex) {
54         this.sex = sex;
55     }
56
57     public String getAddress() {
58         return address;
59     }
60
61     public void setAddress(String address) {
62         this.address = address;
63     }
64
65     @Override
66     public String toString() {
67         return "User{" +
68             "id=" + id +
69             ", username='" + username + '\'' +
70             ", birthday=" + birthday +
71             ", sex='" + sex + '\'' +
72             ", address='" + address + '\'' +
73             '}';
74     }
75 }

```

- UserDao.interface

```

1     package com.bean.dao;

```

```

2
3 import com.bean.domain.User;
4 import org.apache.ibatis.annotations.*;
5 import org.apache.ibatis.mapping.FetchType;
6
7 import java.util.List;
8
9 public interface IUserDao {
10
11     /*
12     @Results: 定义类属性与数据库字段的对应关系
13         - id: 唯一标识, @ResultMap可以根据这个找到
14         - value: 类属性与表字段的映射
15         - @Result: 单个字段的对应关系
16             - id: 是否是主键, 默认为false
17             - property: 类中的属性, 如果使用多表查询, 那么对应着的是封装到哪个对象中
18             - column: 数据库中字段名称, 如果使用多表查询, 那么对应着的是查询所需要的参数
19             - many
20                 - @Many
21                     - select: 要执行的方法的全类名
22                     - fetchType
23                         - FetchType.LAZY: 懒加载
24                         - FetchType.EAGER: 立即加载
25                         - FetchType.DEFAULT: 默认, 可能是随意选一个
26
27     @ResultMap: 引用@Results
28         - value: 引用的@Results值, 假如只有一个可以去掉value和括号
29     */
30
31     /**
32     * 查询所有方法, 实现一对多功能, 并且实现懒加载
33     * @return
34     */
35     @Select("select * from user")
36     @Results(
37         id="userMap",
38         value = {
39             @Result(id = true,property = "id",column = "id"),
40             @Result(property = "username",column = "username"),
41             @Result(property = "address",column = "address"),
42             @Result(property = "sex",column = "sex"),
43             @Result(
44                 property = "accounts",
45                 column = "id",
46                 many = @Many(
47                     select = "com.bean.dao.IAccountDao.findAccontByUid",
48                     fetchType = FetchType.LAZY
49                 )
50             )
51         }
52     )
53     List<User> findAll();
54
55     /**
56     * 根据id查询用户
57     * @return
58     */
59     @Select("select * from user where id=#{id}")

```

```

60     @ResultMap("userMap")
61     User findUserById(Integer id);
62
63     /**
64      * 根据名称进行模糊查询
65      * @param username
66      * @return
67      */
68     @Select("select * from user where username like #{username}")
69     @ResultMap("userMap")
70     List<User> findUsersLikeUsername(String username);
71
72 }

```

缓存的配置

- 二级缓存的配置

一级缓存不用配置就可以开启，现在用注解配置二级缓存：

1. 在 `SqlMapConfig.xml` 中配置开启二级缓存
2. 使用注解 `@CacheNamespace`

1. 在 `SqlMapConfig.xml` 中配置二级缓存

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7      <properties resource="jdbcConfig.properties"></properties>
8
9      <!--配置开启二级缓存-->
10     <settings>
11         <setting name="cacheEnabled" value="true"/>
12     </settings>
13
14     <typeAliases><package name="com.bean.domain"/></typeAliases>
15
16     <environments default="mysql">
17         <environment id="mysql">
18             <transactionManager type="JDBC"></transactionManager>
19             <dataSource type="POOLED">
20                 <property name="driver" value="${jdbc.driver}"/>
21                 <property name="url" value="${jdbc.url}"/>
22                 <property name="username" value="${jdbc.username}"/>
23                 <property name="password" value="${jdbc.password}"/>
24             </dataSource>
25         </environment>
26     </environments>
27
28     <mappers><package name="com.bean.dao"/></mappers>

```

2. 使用注解 @CacheNamespace(blocking=true)

```

1  package com.bean.dao;
2
3  import com.bean.domain.User;
4  import org.apache.ibatis.annotations.*;
5  import org.apache.ibatis.mapping.FetchType;
6
7  import java.util.List;
8
9  /*在这里配置二级缓存，把blocking配置为true*/
10 @CacheNamespace(blocking = true)
11 public interface IUserDao {
12
13
14     @Select("select * from user")
15     @Results(
16         id="userMap",
17         value = {
18             @Result(id = true,property = "id",column = "id"),
19             @Result(property = "username",column = "username"),
20             @Result(property = "address",column = "address"),
21             @Result(property = "sex",column = "sex"),
22             @Result(
23                 property = "accounts",
24                 column = "id",
25                 many = @Many(
26                     select = "com.bean.dao.IAccountDao.findAccontByUid",
27                     fetchType = FetchType.LAZY
28                 )
29             )
30         }
31     )
32     List<User> findAll();
33
34     @Select("select * from user where id=#{id}")
35     @ResultMap("userMap")
36     User findUserById(Integer id);
37
38     @Select("select * from user where username like #{username}")
39     @ResultMap("userMap")
40     List<User> findUsersLikeUsername(String username);
41
42 }

```