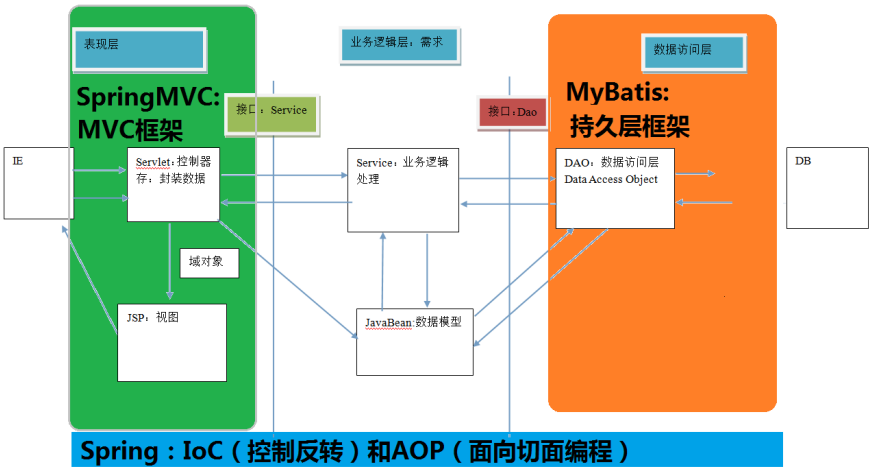
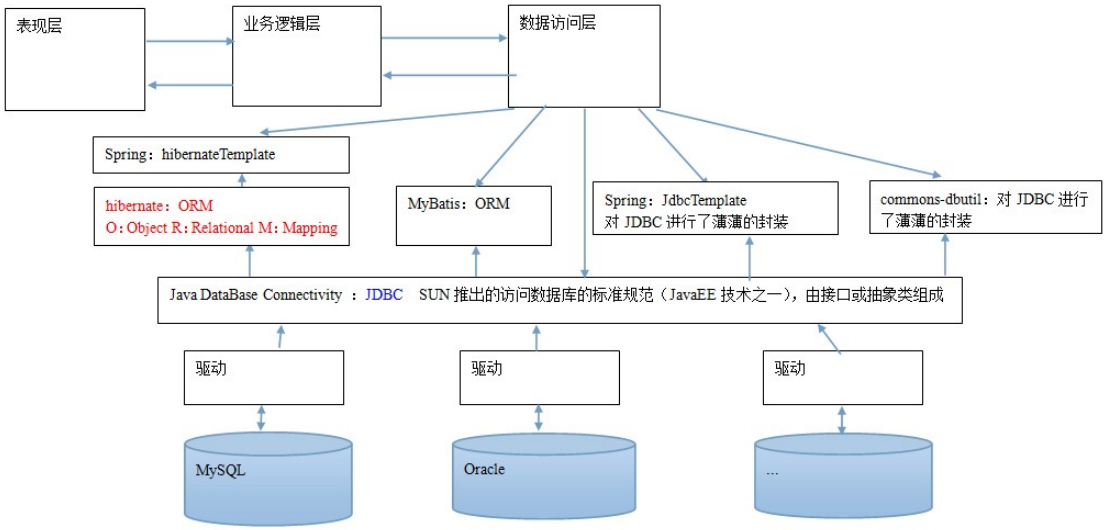


> `mybatis`是持久层的框架，持久层总图如下：



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`

> `Mybatis` 有中文文档

回顾

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

`MyBatis` 讲解

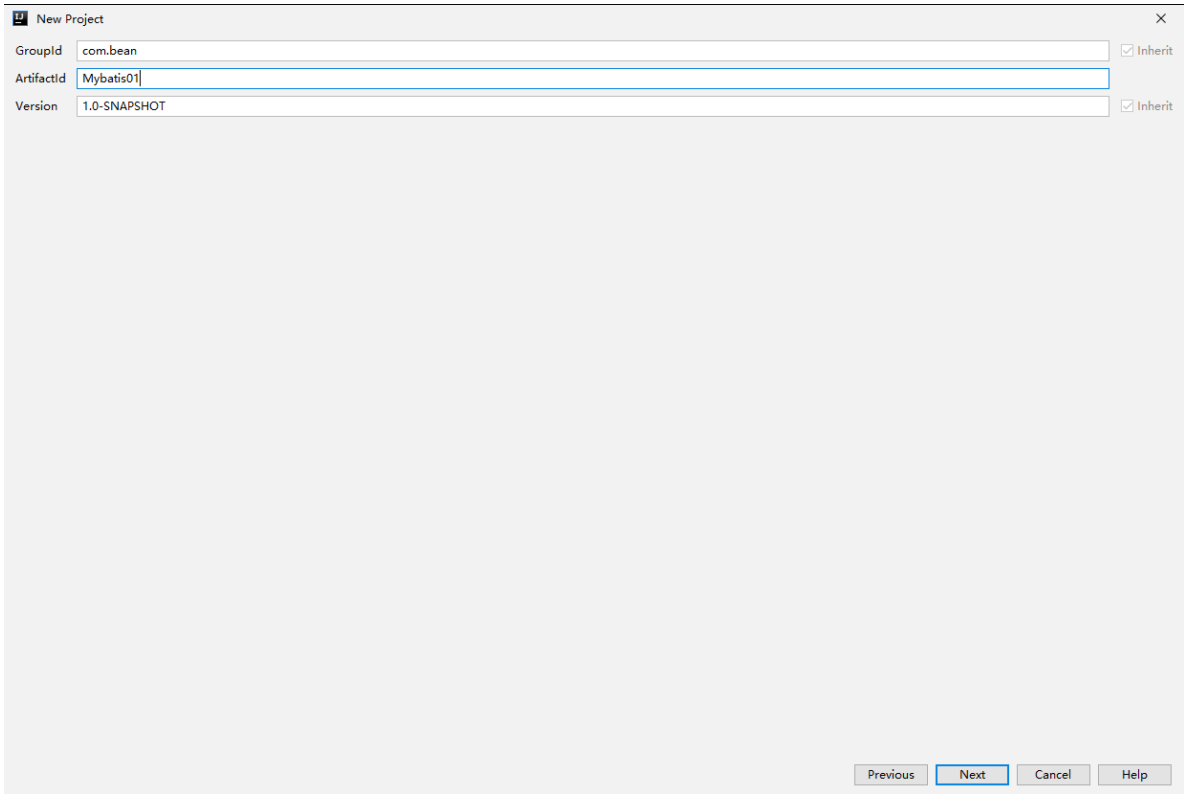
- 优秀的持久层框架。
- 可以使用 `XML` 或注解来配置
- 几乎避免了所有的代码
- 本来是 `apache` 的一个开源项目 `iBatis` ,2010年由 `apache software foundation` 迁移到了 `google code` ，并且修改名字为 `mybatis`

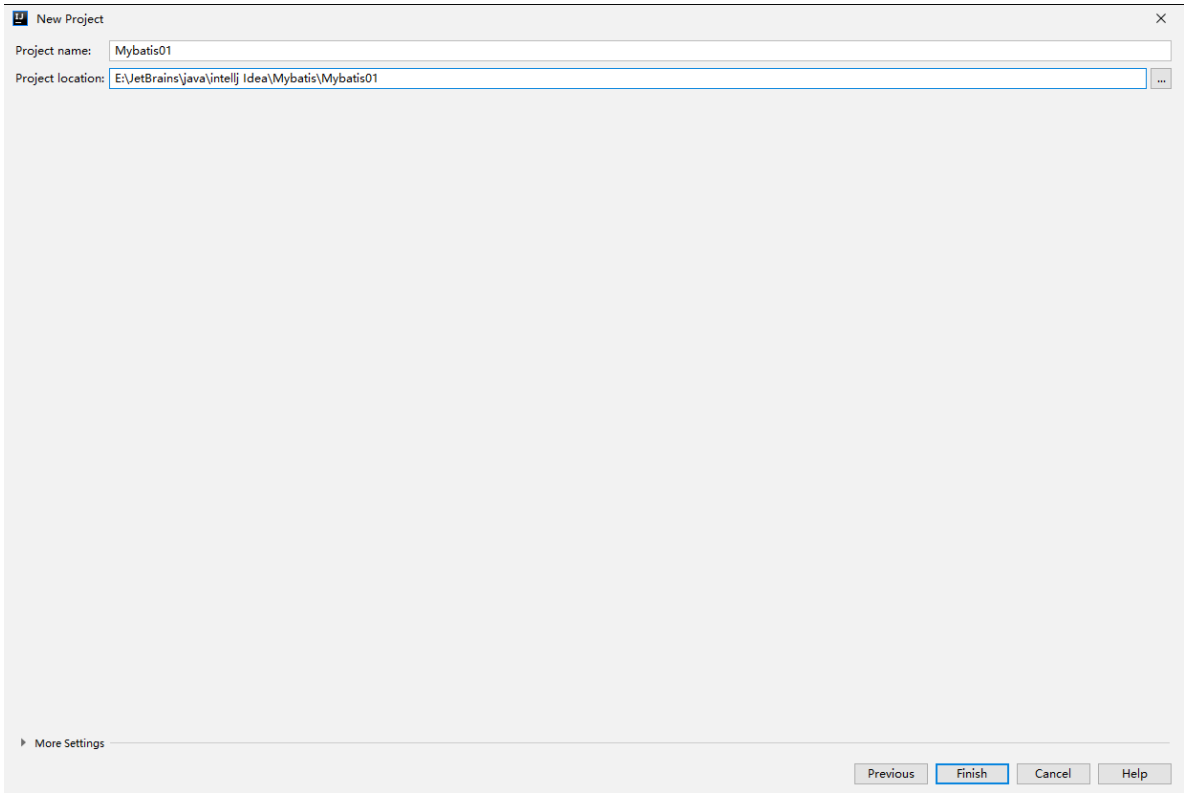
获取 `Mybatis`

- `Maven`
- `Github`

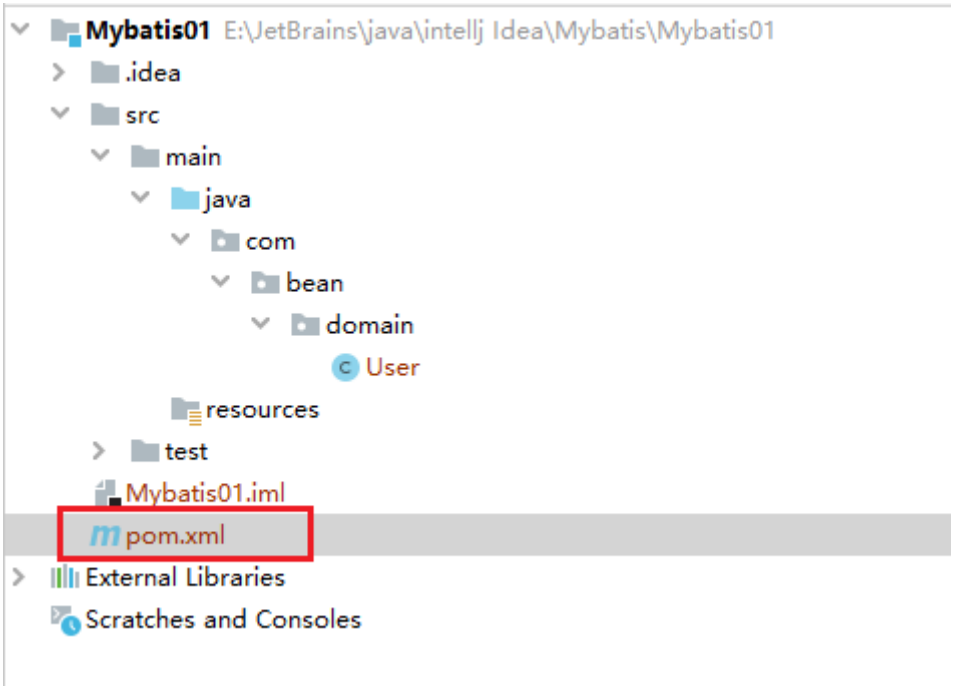
进行基本的环境搭建，跑个流程

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
com.bean
Inherit
ArtifactId
Mybatis01
Version
1.0-SNAPSHOT
Inherit
Previous
Next
Cancel
Help


New Project
Project name:
Mybatis01
Project location:
E:\JetBrains\java\intellj Idea\Mybatis\Mybatis01
More Settings
Previous
Finish
Cancel
Help

2. 填写配置文件




```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.bean</groupId>
    <artifactId>Mybatis01</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--    设定导出文件格式-->
    <packaging>jar</packaging>

<!--     导入包-->
    <dependencies>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
    </dependencies>
```

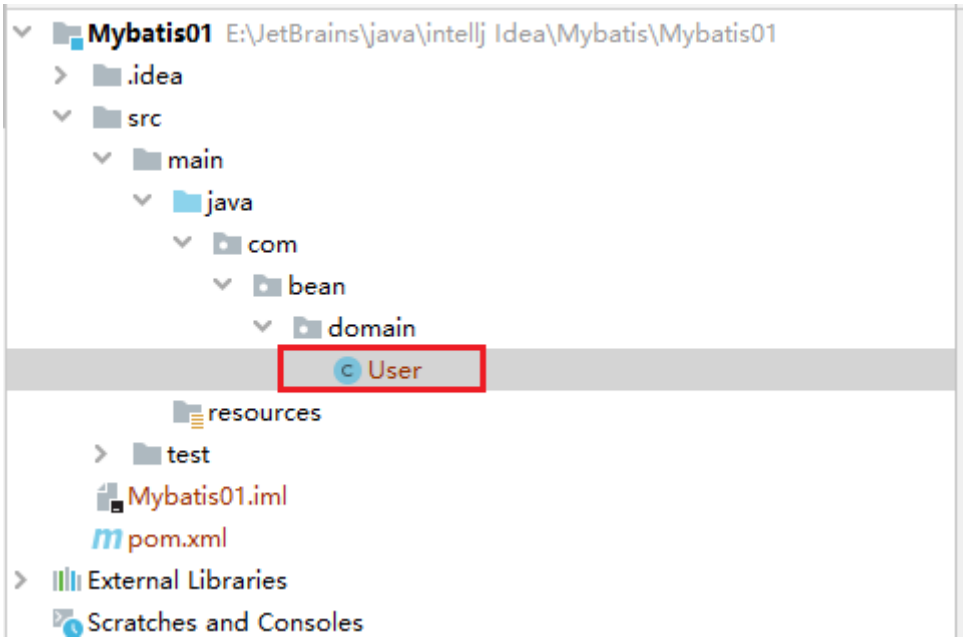
```
</project>
```

3. 数据库建表

```
CREATE DATABASE Mybatis;

use Mybatis;

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL auto_increment,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` datetime default NULL COMMENT '生日',
  `sex` char(1) default NULL COMMENT '性别',
  `address` varchar(256) default NULL COMMENT '地址',
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;



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');

```

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

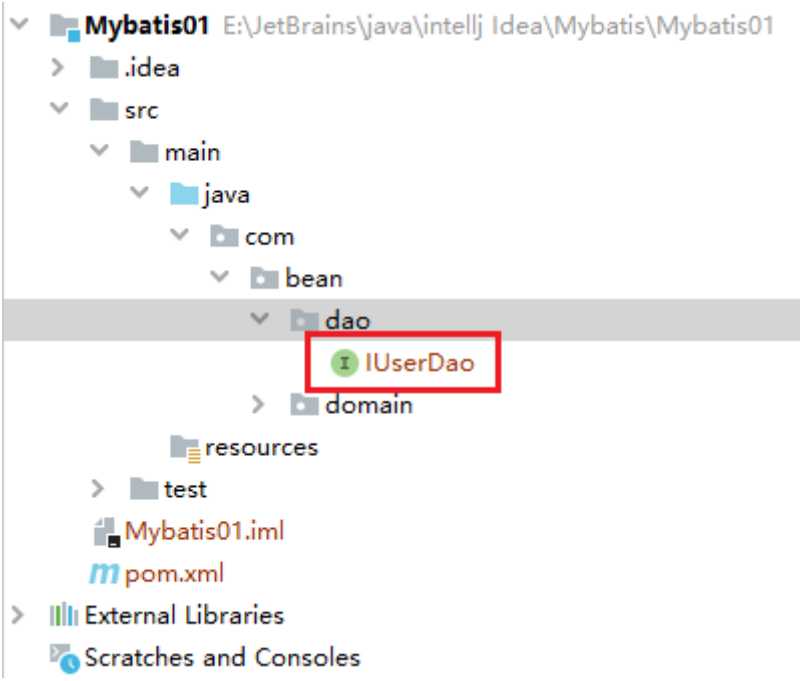
4. 建立domain，将表与类对应




```java
package com.bean.domain;

import java.io.Serializable;
import java.util.Date;

public class User implements Serializable {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    public User(Integer id, String username, Date birthday, String sex, String address) {
        this.id = id;
        this.username = username;
        this.birthday = birthday;
        this.sex = sex;
        this.address = address;
    }

    public User() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

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

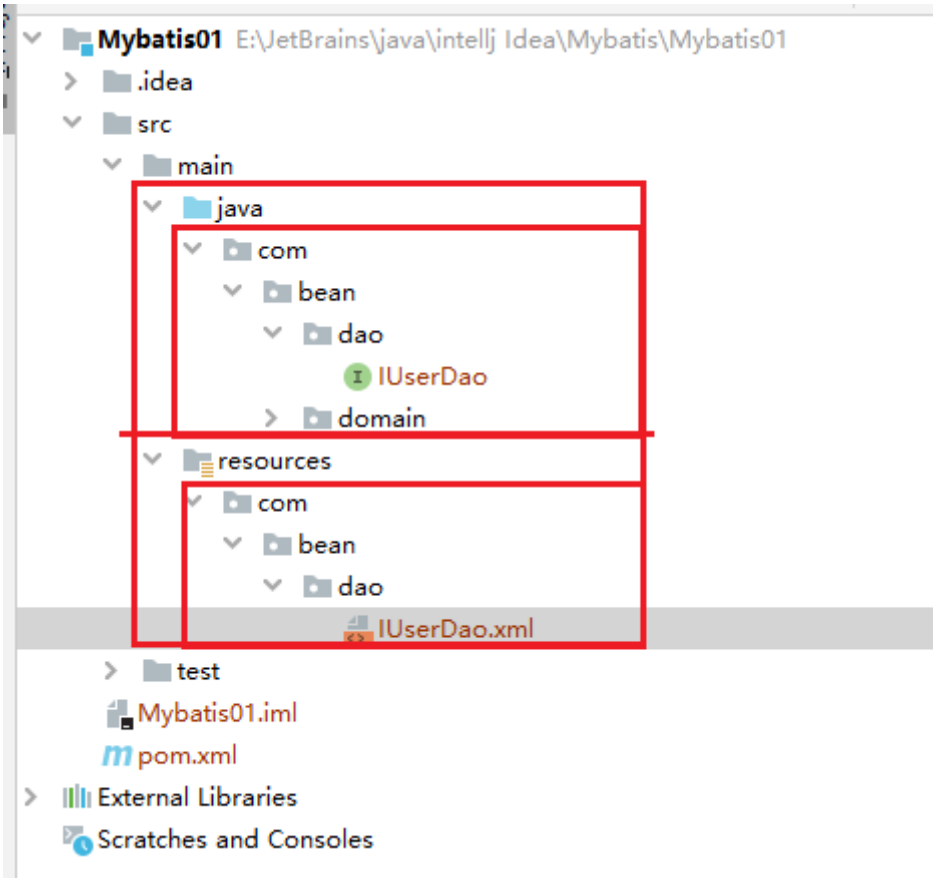
```
40
41      public Date getBirthday() {
42          return birthday;
43      }
44
45      public void setBirthday(Date birthday) {
46          this.birthday = birthday;
47      }
48
49      public String getSex() {
50          return sex;
51      }
52
53      public void setSex(String sex) {
54          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



```java
package com.bean.dao;

import com.bean.domain.User;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {

    /**
     * 查询所有用户
     * @return  List<User>
     */
    List<User> findAll();
}
```

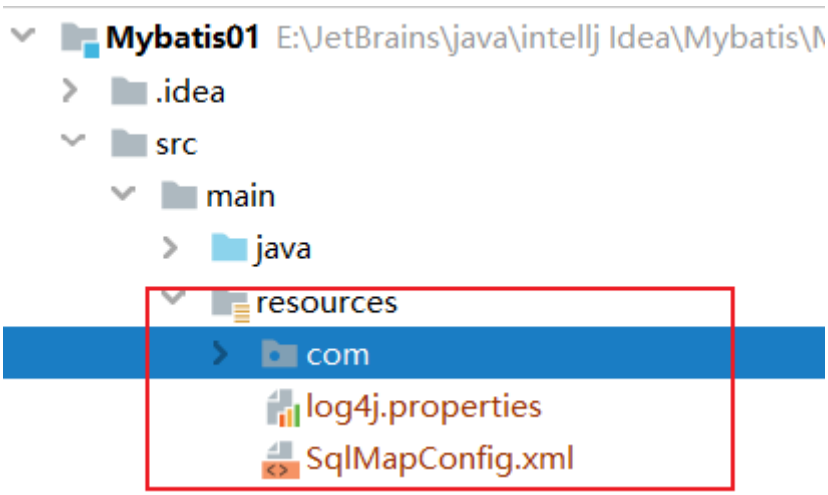
6. 编写持久层接口的映射文件 IUserDao.xml




```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- 上面的配置一定是要写上的-->

<!--配置mapper-->
<mapper namespace="com.bean.dao.IUserDao">
    <!--
        这个id不能瞎写，是IUserDao里面的方法名
        这个resultType是表对应的类，结果类型
        里面的语句是要执行的语句
      -->
```

```
    <select id="findAll" resultType="com.bean.domain.User">
        select * from user
    </select>
</mapper>
```

7. 编写 SqlMapConfig.xml 配置文件




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<!--注意上面的配置-->

<!--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-->

<configuration>
    <!--    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-->
    <environments default="mysql">
        <!--  配置mysql的环境，对mysql进行解释  -->
        <environment id="mysql">
            <!--配置事务的类型-->
            <transactionManager type="JDBC"></transactionManager>
            <!--配置连接数据库的信息，用的是数据源(连接池)，
                这里的driver，url，username，password应该都熟悉吧
            -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>


    <!--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-->
    <mappers>
        <mapper resource="com/bean/dao/IUserDao.xml"/>
    </mappers>
</configuration>
```

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

> - 一般我们使用 `resouce` 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

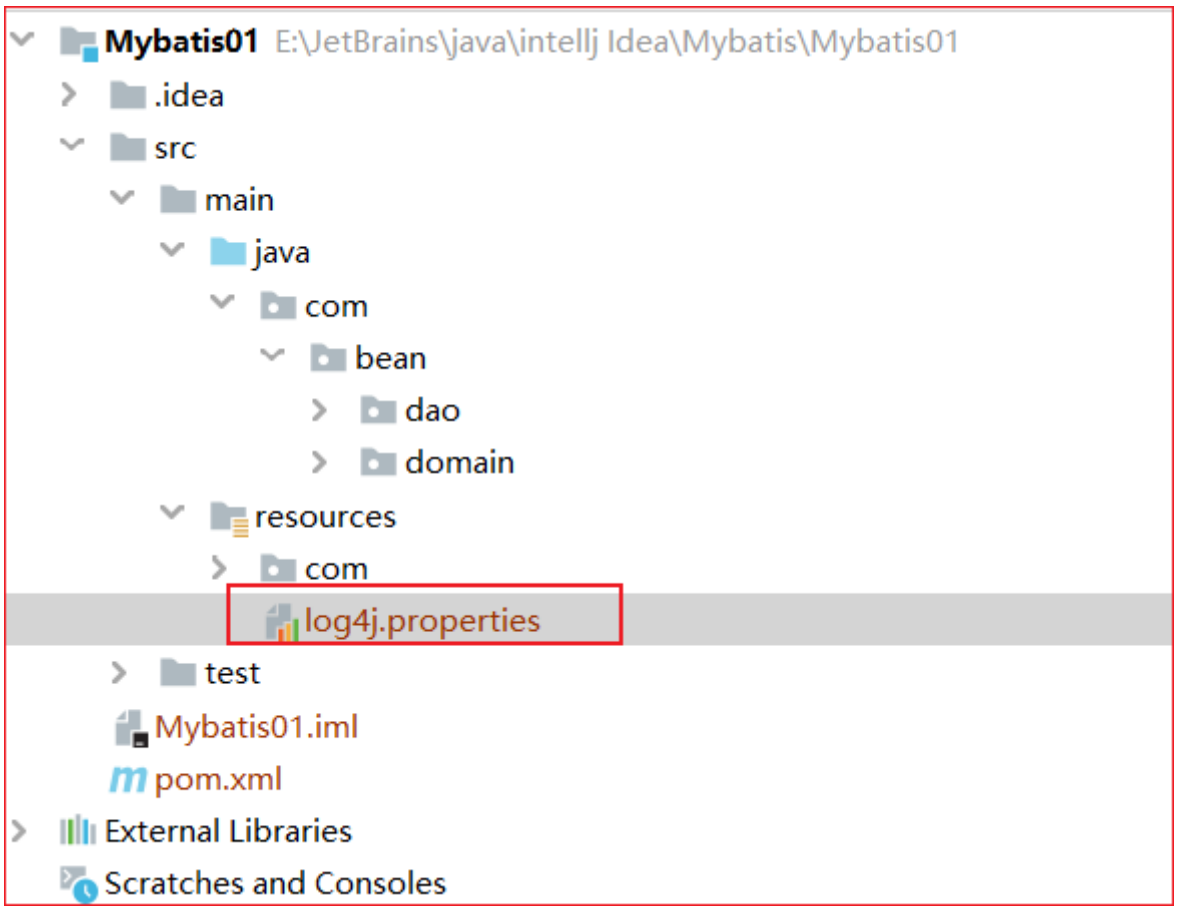
1. 在 `Mybatis` 中的 `dao` 层主要叫做 `Mapper` ，所以这里的 `IUserDao` 也叫做 `IUserMapper`
2. idea中创建目录和包是不一样的
    1. 包(Package)： `com.bean.dao` 是三级结构
    2. 目录(Directory)： `com.bean.dao` 是一级目录
3. `mybatis` 的映射配置文件位置必须和 `dao` 接口的包结构相同，也就是上面第六步的注意事项
4. 映射配置文件的 `mapper` 标签 `namespace` 属性的取值必须为 `dao` 接口的全限定类名
5. 映射配置文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

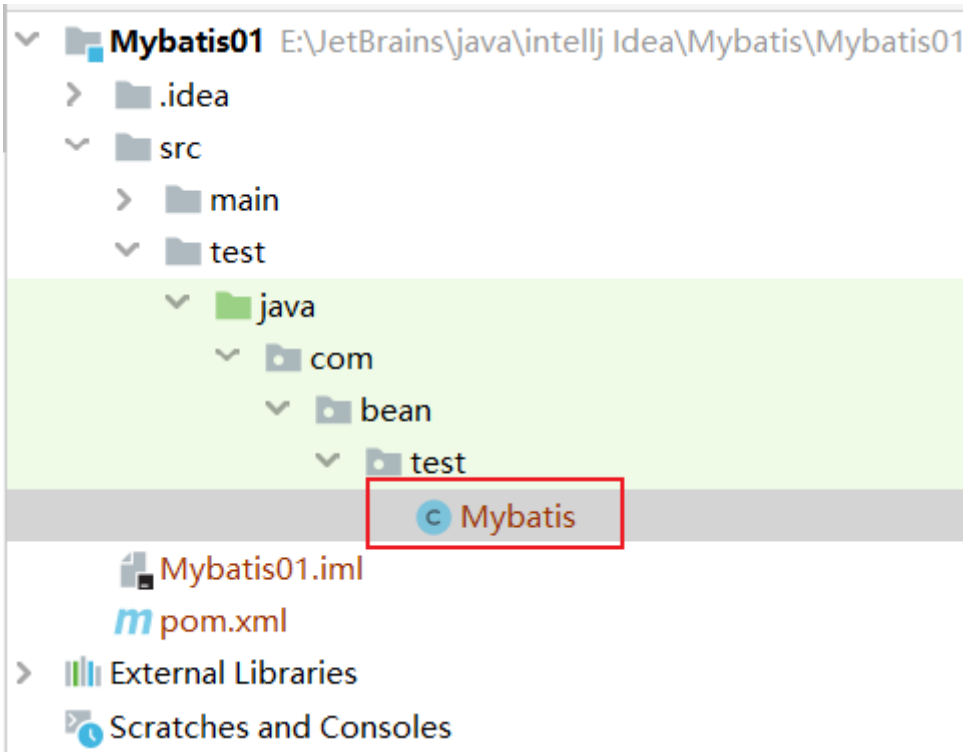
1. 首先往项目里拷一个文件： `log4j.properties` ，先拷下来，其他的再说

> 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




```
# Set root category priority to INFO and its only appender to CONSOLE.
#log4j.rootCategory=INFO, CONSOLE            debug   info   warn error fatal
log4j.rootCategory=debug, CONSOLE, LOGFILE

# Set the enterprise logger category to FATAL and its only appender to CONSOLE.
log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n

# LOGFILE is set to be a File appender using a PatternLayout.
log4j.appender.LOGFILE=org.apache.log4j.FileAppender
log4j.appender.LOGFILE.File=d:\axis.log
log4j.appender.LOGFILE.Append=true
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
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

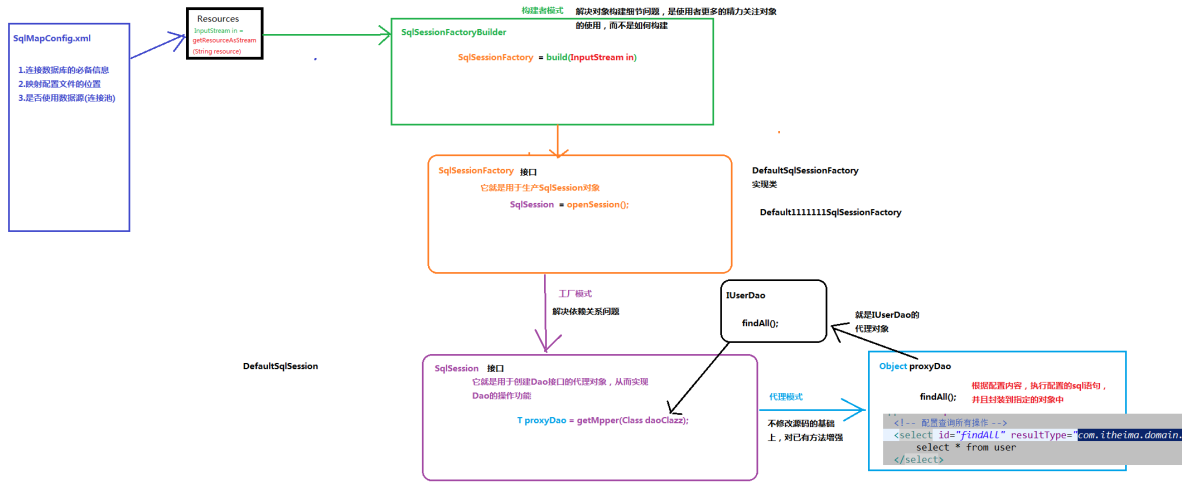
> 因为我们要做的是不写实现类实现接口，所以先把流程走一遍

```java
package com.bean.test;

public class Mybatis {
    /**
     * 入门案例
     * @param args
     */
    public static void main(String[] args) {
        //1. 读取配置文件
        //2. 创建sqlSessionFactory工厂
        //3. 使用工厂生产SqlSession对象
        //4. 使用Sqlsession创建Dao接口的代理对象
        //5. 使用代理对象执行方法
        //6. 释放资源
    }
}
```

4. 编写代码

```java
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

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

```

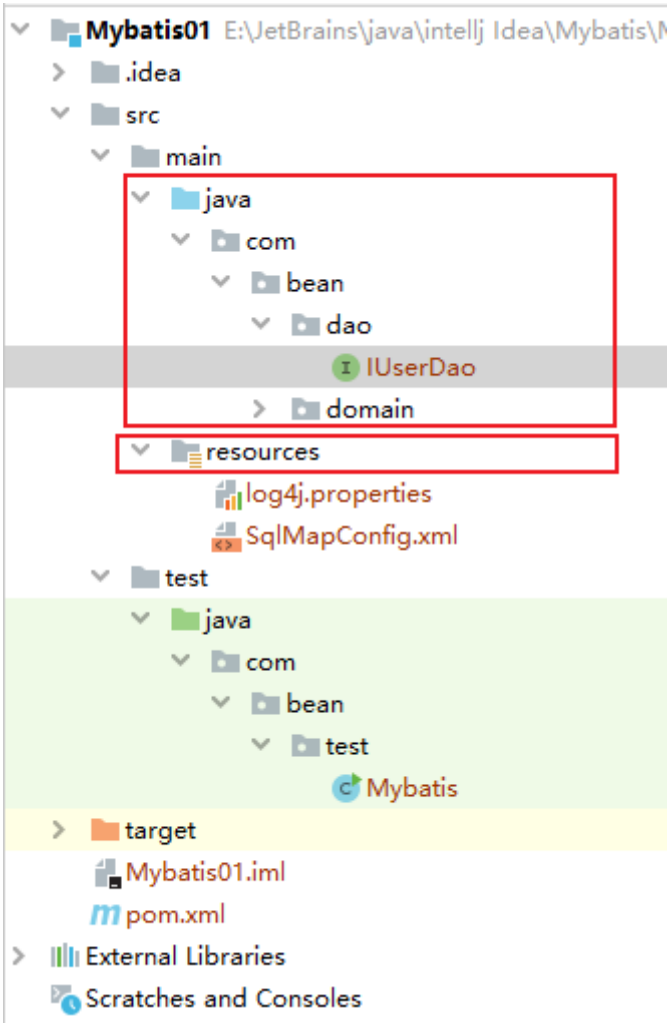
```java
public class Mybatis {
    /**
     * 入门案例
     * @param args
     */
    public static void main(String[] args) throws IOException {
        //1. 读取配置文件，这里读取的文件就是刚才mybatis的配置文件
        InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");

        //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
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(in);

        //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();
    }
}
```



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



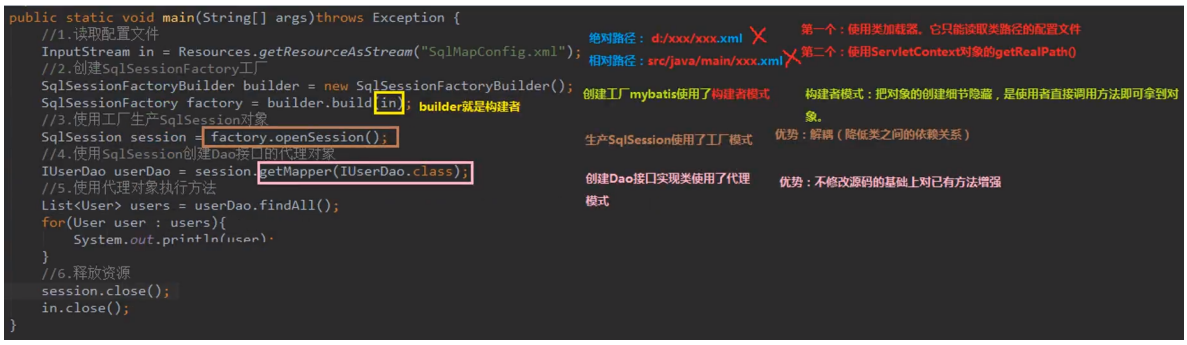
2. 在 `IUserDao` 接口中的语句中编写注解

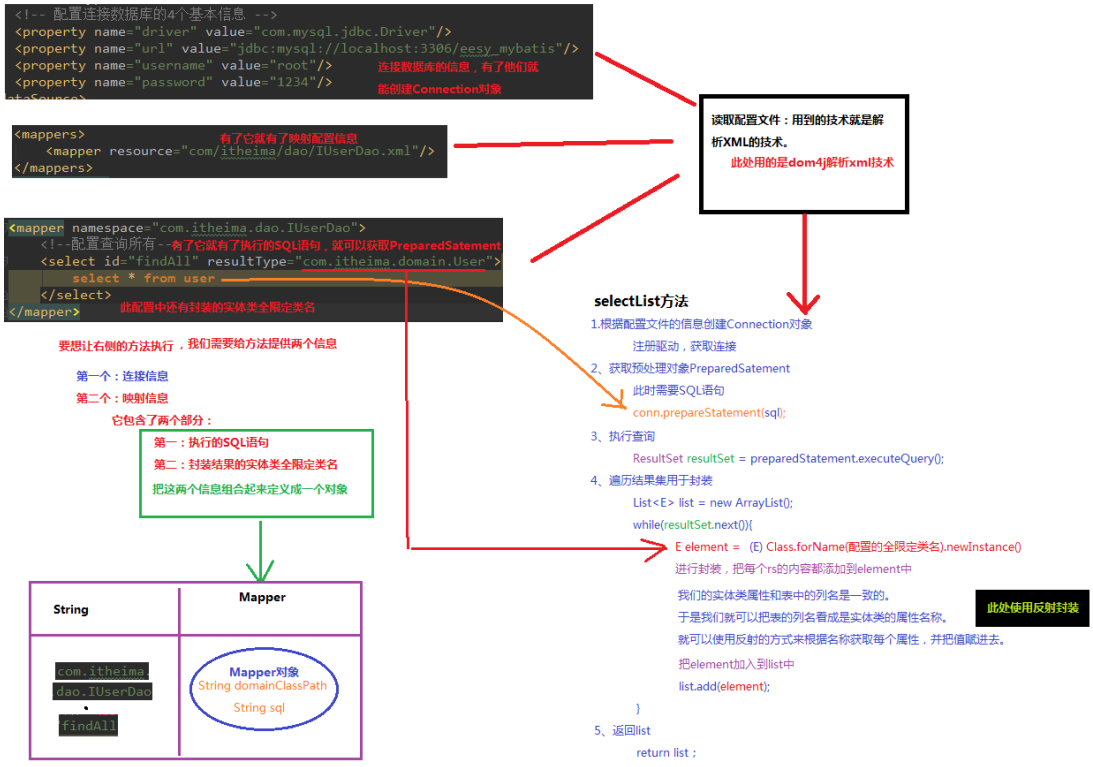
3. 在 `SqlMapConfig.xml` 中更改 `IUserDao.xml` 的映射，由 `resource` 改为 `class`

mybatis 入门案例中的设计模式分析

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

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

public class Mybatis {
    /**
     * 入门案例
     * @param args
```

```java
     */
    public static void main(String[] args) throws IOException {
        //1. 读取配置文件，这里读取的文件就是刚才mybatis的配置文件
        InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");

        //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
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(in);

        //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();
    }
}
```





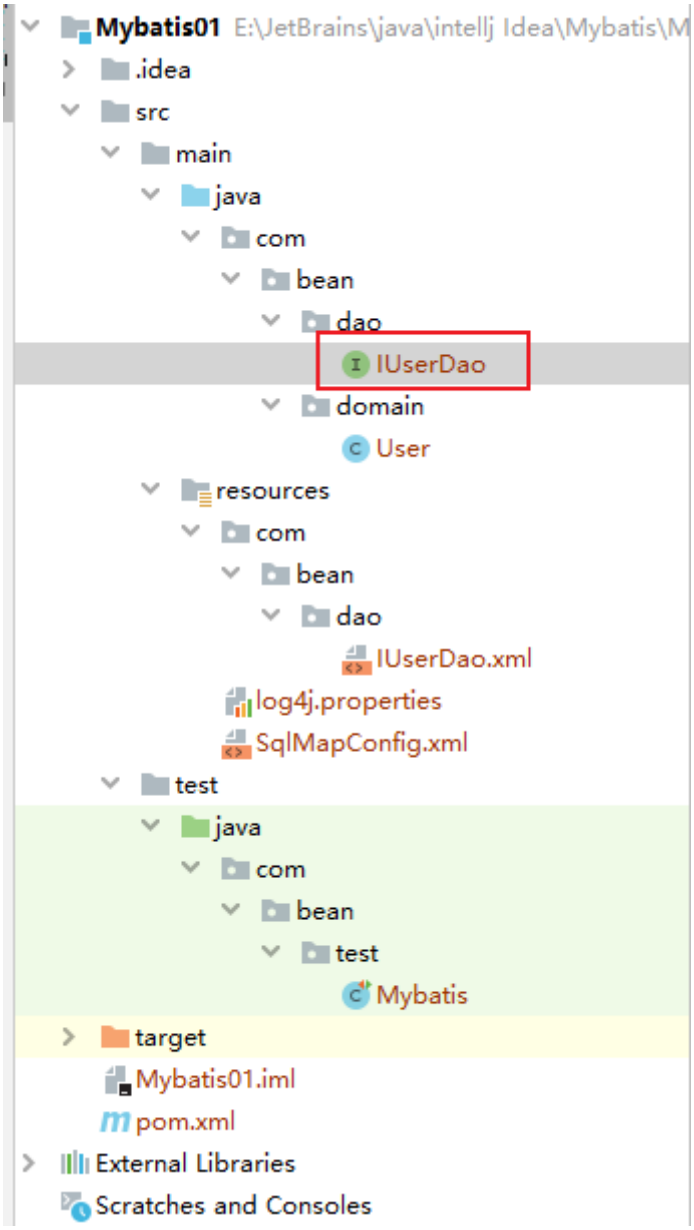
mybatis深入讲解

mybatis的crud

> 基于代理 dao 的方式实现 CRUD

保存操作

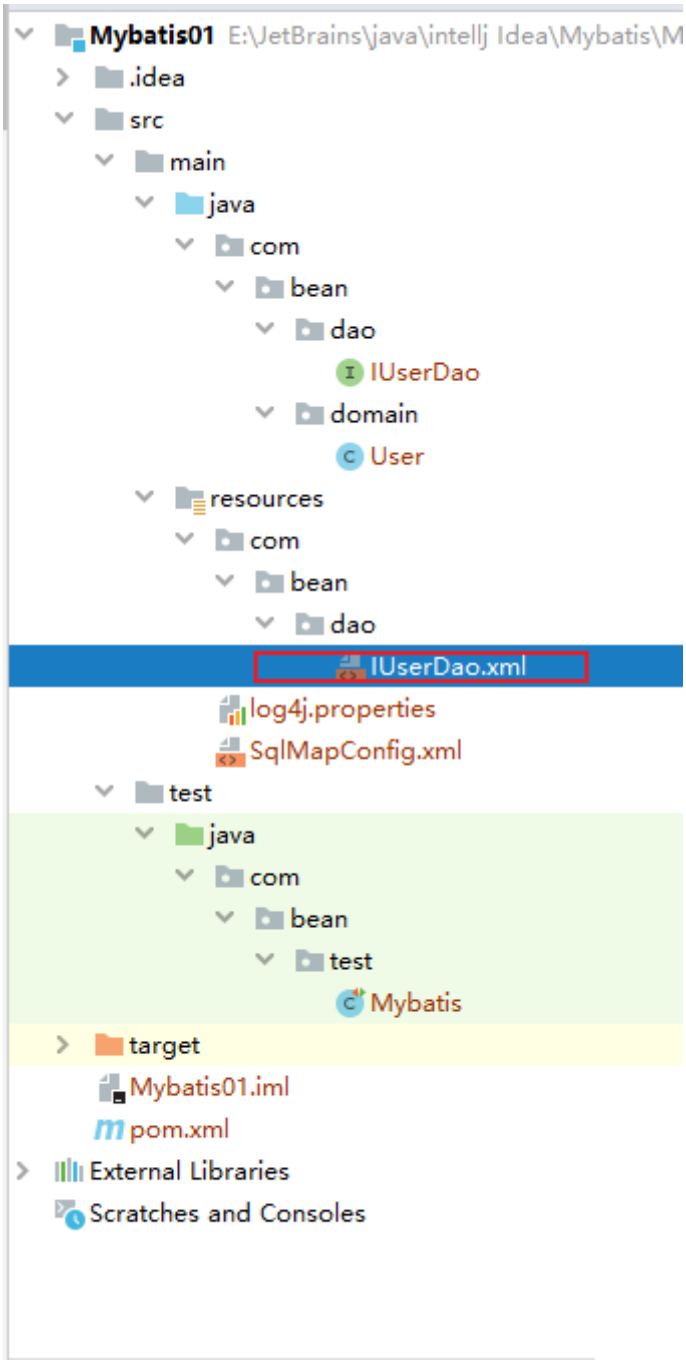
1. 进行环境搭建（我们现在不用注解，先使用xml方式）
2. 编写 IUserDao 接口




```java
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.Select;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {

    /**
     * 查询所有用户
     * @return  List<User>
     */
    List<User> findAll();

    /**
     * 向数据库中插入新用户
     * @param user
     */
    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




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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findAll" resultType="com.bean.domain.User">
        select * from user
    </select>

    <!--
        1. 新的选项，既然是插入功能就要使用insert
        2. 我们的接口参数要选择，就是parameterType
        3. 使用插入的语法，原来的?占位符替换为#{内容}
        4. #{}里的内容取决于getter，比如getId()->Id->id，如果是别的就对应别的
    -->
```

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

4. 在测试类中编写方法

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession session = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);
//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        session = factory.openSession();
//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = session.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        6. 关闭资源
        session.close();
        inputStream.close();
    }

    @Test   //查询所有方法
    public void findAll(){
//        5. 使用代理执行方法
        List<User> users = mapper.findAll();
        for (User user : users) {
            System.out.println(user);
        }
```

```
51      }
52
53      @Test   //保存新User
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 `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       */
```

```
    List<User> findAll();

    /**
     * 向数据库中插入新用户
     * @param user
     */
    void saveUser(User user);

    /**
     * 更新用户数据
     * @param user
     */
    void updateUser(User user);
}
```

3. 编写 `IUserDao.xml` 映射文件

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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findAll" resultType="com.bean.domain.User">
        select * from user
    </select>

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

    <!--
        1. 更新操作使用update
        2. parameterType还是User
        3. 占位符#{}
    -->
    <update id="updateUser" parameterType="com.bean.domain.User">
        update user set username=#{username},address=#{address},
                        sex=#{sex},birthday=#{birthday} where id=#{id}
    </update>
</mapper>
```

4. 编写测试代码

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.io.IOException;
```

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession session = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        session = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = session.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        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
        session.commit();

//        6. 关闭资源
        session.close();
        inputStream.close();
    }

    @Test   //查询所有方法
    public void findAll(){
//        5. 使用代理执行方法
        List<User> users = mapper.findAll();
        for (User user : users) {
            System.out.println(user);
        }
    }

    @Test   //保存新User
    public void saveUser(){
//        创建User对象
        User user = new User();
        user.setUsername("mybatis");
        user.setAddress("山东省");
        user.setBirthday(new Date());
        user.setSex("男");

//        调用方法
        mapper.saveUser(user);

//        事务的提交统一放到After上面了
```

```
    }

    @Test
    public void updateUser(){
//        这里应当是先查出来User对象，不过这里省事创建得了
        User user = new User();
        user.setId(49);
        user.setUsername("updateUser");
        user.setAddress("山东省");
        user.setBirthday(new Date());
        user.setSex("女");

//        更改内容
        mapper.updateUser(user);

//        事务的提交统一放到After上面了
    }

}
```

删除操作

1. 环境搭建
2. 编写 `IUserDao` 接口

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.Select;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {

    /**
     * 查询所有用户
     * @return  List<User>
     */
    List<User> findAll();

    /**
     * 向数据库中插入新用户
     * @param user
     */
    void saveUser(User user);

    /**
     * 更新用户数据
     * @param user
     */
    void updateUser(User user);

    /**
     * 删除用户数据只需要一个id就可以了
```

```
     * @param id
     */
    void deleteUser(Integer id);
}
```

3. 编写 `IUserDao.xml` 映像文件

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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findAll" resultType="com.bean.domain.User">
        select * from user
    </select>

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


    <update id="updateUser" parameterType="com.bean.domain.User">
        update user set username=#{username},address=#{address},
                        sex=#{sex},birthday=#{birthday} where id=#{id}
    </update>

    <!--
        1. 删除操作使用delete
        2. parameterType现在变为了int类型，但是写int,INT，Integer,INTEGER,java.long.Integer都可以
        3. 占位符#{}
        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
    -->
    <delete id="deleteUser" parameterType="java.lang.Integer">
        delete from user where id=#{id}
    </delete>
</mapper>
```

4. 编写测试类

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

```

```java
public class Mybatis {
    InputStream inputStream = null;
    SqlSession session = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        session = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = session.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        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
        session.commit();

//        6. 关闭资源
        session.close();
        inputStream.close();
    }

    @Test   //查询所有方法
    public void findAll(){
//        5. 使用代理执行方法
        List<User> users = mapper.findAll();
        for (User user : users) {
            System.out.println(user);
        }
    }

    @Test   //保存新User
    public void saveUser(){
//        创建User对象
        User user = new User();
        user.setUsername("mybatis");
        user.setAddress("山东省");
        user.setBirthday(new Date());
        user.setSex("男");

//        调用方法
        mapper.saveUser(user);

//        事务的提交统一放到After上面了
    }

    @Test
    public void updateUser(){
```

```
//        这里应当是先查出来User对象，不过这里省事创建得了
        User user = new User();
        user.setId(49);
        user.setUsername("updateUser");
        user.setAddress("山东省");
        user.setBirthday(new Date());
        user.setSex("女");

//        更改内容
        mapper.updateUser(user);

//        事务的提交统一放到After上面了
    }

    @Test
    public void deleteUser(){
//        直接执行函数得了，把新添加的删了
        mapper.deleteUser(49);
//        事务提交，放到了After中
    }

}
```

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

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.Select;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {

    /**
     * 通过id寻找用户
     * @param id
     * @return User对象
     */
    User findUserById(int id);


    /**
     * 通过某个名称来模糊查询用户们
     * @param name
     * @return  List<User>
     */
```

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

2. IUserDao.xml

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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findUserById" parameterType="INT" resultType="com.bean.domain.User">
        select * from user where id=#{id}
    </select>

    <select id="findUserByName" parameterType="STRING" resultType="com.bean.domain.User">
        select * from user where username like #{username}
    </select>
</mapper>
```

3. 测试类

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        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
    User
58          for (User user : users) {
59              System.out.println(user);
60          }
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

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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findCount" resultType="int">
        select count(id) from user;
    </select>
</mapper>
```

3. 测试类

```java
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        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
        sqlSession.commit();
//        6. 关闭资源
        sqlSession.close();
```

```
        inputStream.close();
    }

    @Test   //通过UserName来模糊查询
    public void findCount(){
        int count = mapper.findCount();
        System.out.println(count);
    }

}
```

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，那么我们就会得到0，因为没有插入

- `IUserDao.xml`

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

<mapper namespace="com.bean.dao.IUserDao">
    <insert id="insertUser" parameterType="com.bean.domain.User">
        <!-- 配置当User插入之后获取最后的id值
            keyProperty：User类中要获取到的值
            keyColumn：数据库表中的值，与User类对应
            resultType：返回类型
            order：在插入语句 之前/之后 执行
                AFTER：之后
                BEFORE：之前
            select last_insert_id()：查询最后的ID
         -->
        <selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER">
                select last_insert_id()
        </selectKey>
        insert into user (username,address,sex,birthday) values(#{username},#{address},#{sex},#{birthday})
    </insert>
</mapper>
```

- 测试类

```
package com.bean.test;

```

```java
import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        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
        sqlSession.commit();
//        6. 关闭资源
        sqlSession.close();
        inputStream.close();
    }

    @Test
    public void findByName(){
        User user = new User();
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("Address");
        user.setUsername("new User");

        System.out.println("保存之前："+user);

        mapper.insertUser(user);

```

```
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PreparesStatement
>
> 2.
>
> - 其实还可以使用： `select * from user where username like '%${value}%'`
> - 其实还可以使用： `List<User> users = mapper.findByName("王");`
> -
>
> ```
> 2019-11-14 10:16:22,504 776    [           main] DEBUG m.bean.dao.IUserDao.findByName  - ==>  Preparing: select * from user where username like '%王%'
> 2019-11-14 10:16:22,557 829    [           main] DEBUG m.bean.dao.IUserDao.findByName  - ==> Parameters:
> ```
>
> > 1. 使用的是Statement
> > 2. 注意xml中只能使用 `'%${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

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

<mapper namespace="com.bean.dao.IUserDao">

    <!--  通过pojo包装对象来获得User
        parameterType="com.bean.domain.QueryVo"：参数变为了QueryVo
        resultType="com.bean.domain.User"：返回值还是User对象

        select * from user where username like user.username：对于这个user.username
            1. 使用了OGNL表达式：user.username ==> QueryVo.username.username ==> QueryVo.getUsername().getusername()
            2. 获取的是User里面的username
      -->
    <select id="findUserByVo" parameterType="com.bean.domain.QueryVo" resultType="com.bean.domain.User">
        select * from user where username like user.username
    </select>
</mapper>
```

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.QueryVo;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.io.IOException;
```

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {
//        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
        sqlSession.commit();
//        6. 关闭资源
        sqlSession.close();
        inputStream.close();
    }

    @Test
    public void findByName(){
        QueryVo vo = new QueryVo();
        User u = new User();
        u.setUsername("%王%");
        vo.setUser(u);

        List<User> users = mapper.findUserByVo(vo);

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

    }

}
```

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
```

```
    public void setUserBirthday(Date userBirthday) {
        this.userBirthday = userBirthday;
    }

    public String getUserSex() {
        return userSex;
    }

    public void setUserSex(String userSex) {
        this.userSex = userSex;
    }

    public String getUserAddress() {
        return userAddress;
    }

    public void setUserAddress(String userAddress) {
        this.userAddress = userAddress;
    }

    @Override
    public String toString() {
        return "User{" +
                "userId=" + userId +
                ", userName='" + userName + '\'' +
                ", userBirthday=" + userBirthday +
                ", userSex='" + userSex + '\'' +
                ", userAddress='" + userAddress + '\'' +
                '}';
    }
}
```

---

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

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

<mapper namespace="com.bean.dao.IUserDao">
    <!--通过起别名的方式来实现-->
    <select id="findAll" resultType="com.bean.domain.User">
        select id as userId,username as userName,
                birthday as userBirthday,sex as userSex,address as userAddress
        from user;
    </select>

</mapper>
```

- 测试类

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
```

```java
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {

//        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
        sqlSession.commit();

//        6. 关闭资源
        sqlSession.close();
        inputStream.close();
    }

    @Test
    public void findAll(){

        List<User> users = mapper.findAll();

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

    }

}
```

2. 追求开发速度

- `IUserDao.xml` 中

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

<mapper namespace="com.bean.dao.IUserDao">

    <!--  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resultMap
        id：主键
            property：用户类对应的字段名称
            column：表中对应的字段名称
        result：其他字段
            property：用户类对应的字段名称
            column：表中对应的字段名称
    -->
    <resultMap id="userMap" type="com.bean.domain.User">
        <id property="userId" column="id"></id>
        <result property="userName" column="username"></result>
        <result property="userBirthday" column="birthday"></result>
        <result property="userSex" column="sex"></result>
        <result property="userAddress" column="address"></result>
    </resultMap>

    <!--使用resultMap，映射-->
    <select id="findAll" resultMap="userMap">
        select * from user;
    </select>

</mapper>
```

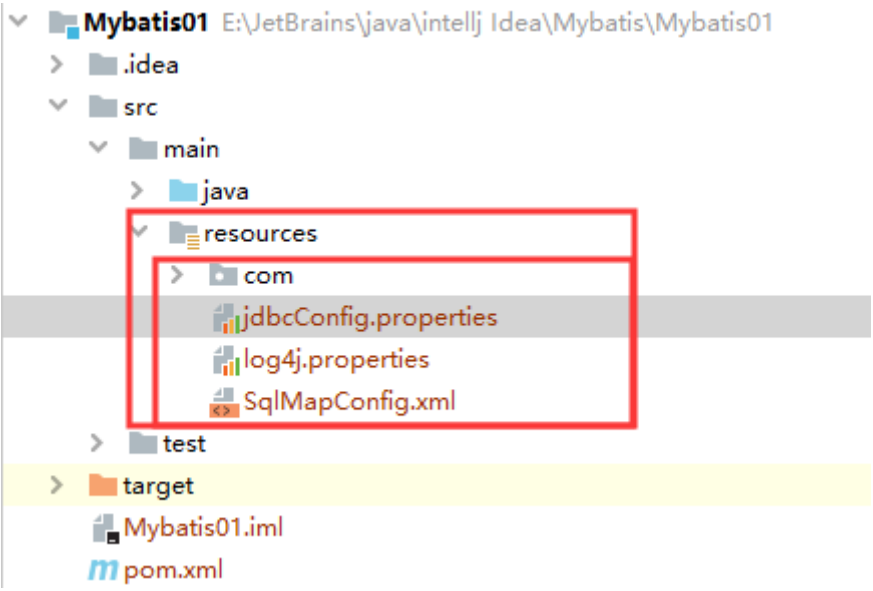
- 测试类中

```java
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    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`

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

- `SqlMapConfig.xml`

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

<configuration>
    <!--配置properties：可以在标签内部配置连接数据库的的信息，也可以通过属性引用外部配置文件的信息
        resource：常用
            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
        url：
            是要求按照URL的写法来写地址
            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
            URL的写法：
                协议  主机  端口  URI
                http://localhost:8080/tomcat/demo
                协议不仅仅只有一个http协议
            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
    -->
    <properties
url="file:///E:/JetBrains/java/intellj%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>

    <environments default="mysql">

        <environment id="mysql">

            <transactionManager type="JDBC"></transactionManager>

            <!--这里也是改变的地方，其中value值全部都从properties文件中读取-->
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
```

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

        </environment>

    </environments>

    <mappers>
        <mapper class="com.bean.dao.IUserDao"/>
    </mappers>
</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

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

<configuration>


<!--typeAliases：用于起别名
    type：参数的全限定类名
    alias：别名
    别名一旦起了，就不再限制大小写，比如：user  User  USER
-->
    <typeAliases>
        <typeAlias type="com.bean.domain.User" alias="user"></typeAlias>
    </typeAliases>

    <environments default="mysql">

        <environment id="mysql">

            <transactionManager type="JDBC"></transactionManager>

            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>

        </environment>
```

```
    </environments>

    <mappers>
        <mapper class="com.bean.dao.IUserDao"/>
    </mappers>
</configuration>
```

- `IUserDao.xml`

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

<mapper namespace="com.bean.dao.IUserDao">



    <select id="findAll" resultType="user">
        select * from user;
    </select>

</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

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

<configuration>


<!--typeAliases：用于起别名
    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
-->
    <typeAliases>
        <package name="com.bean.domain"></package>
    </typeAliases>

    <environments default="mysql">

        <environment id="mysql">

            <transactionManager type="JDBC"></transactionManager>

            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>

```

```
        </environment>

    </environments>

    <mappers>
        <mapper class="com.bean.dao.IUserDao"/>
    </mappers>
</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`

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

<configuration>


    <typeAliases>
        <package name="com.bean.domain"></package>
    </typeAliases>

    <environments default="mysql">

        <environment id="mysql">

            <transactionManager type="JDBC"></transactionManager>

            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/Mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>

        </environment>

    </environments>

    <mappers>
        <package name="com.bean.dao"></package>
    </mappers>
</configuration>
```

最终配置

- `jdbcConfig.properties`

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

- `SqlMapConfig.xml`

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

<configuration>

    <!--读取好配置文件-->
    <properties
url="file:///E:/JetBrains/java/intellj%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>

<!--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：
    User类
        别名：User
        别名不区分大小写
-->
    <typeAliases>
        <package name="com.bean.domain"></package>
    </typeAliases>

    <!--配置环境们，并且设置默认为mysql环境
        名字是不是mysql无所谓，但是要进行解释-->
    <environments default="mysql">
        <!--对mysql进行解释-->
        <environment id="mysql">
            <!--事务类型：JDBC-->
            <transactionManager type="JDBC"></transactionManager>
            <!--数据源，类型：连接池-->
            <dataSource type="POOLED">
                <!--参数们：引用的properties下面的数值-->
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>

    </environments>
    <!--映射：映射到dao包下，找到IUserDao.xml-->
    <mappers>
        <package name="com.bean.dao"></package>
    </mappers>

</configuration>
```

- `IUserDao.xml`

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

<!--映射，为接口编写语句，接口为"com.bean.dao.IUserDao"-->
<mapper namespace="com.bean.dao.IUserDao">
    <!--查询方法，返回类型为User
        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
    -->
    <select id="findAll" resultType="user">
        select * from user;
    </select>


    <!--增加方法，参数为User对象
        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
        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
    -->
    <insert id="insertUser" parameterType="user">
        <!--使用selectKey找到插入之后的id
            keyProperty：id      用户类中的id
            keyColumn：表中的id
            resultType：返回类型
            order：在插入语句之前/之后执行
                AFTER
                BEFORE
            在插入之前执行方法返回0
        -->
        <selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER">
            <!--选择最后一个id-->
            select last_insert_id()
        </selectKey>
       insert into user(username,address,sex,birthday)
                    values(#{username},#{address},#{sex},#{birthday})
    </insert>


    <!--更新方法，参数为User对象
        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
    -->
    <update id="updateUser" parameterType="user">
        update user set username=#{username},address=#{address},
                        sex=#{sex},birthday=#{birthday} where id=#{id}
    </update>


    <!--删除操作，参数为int类型-->
    <delete id="deleteById" parameterType="int">
        delete from user where id=#{id}
    </delete>


    <!--根据id查询，参数为int类型-->
    <select id="findUserById" parameterType="int">
        select * from user where id=#{id}
    </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。
>
> - 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

```
package com.bean.dao;

import com.bean.domain.User;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {

    /**
     * 实现动态SQL查询，动态的条件查询
     * @param user  id,username,address,sex,等等，条件随机组合
     * @return
     */
    List<User> findUserByCondition(User user);
}
```

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

```

```xml
<mapper namespace="com.bean.dao.IUserDao">

    <select id="findUserByCondition" resultType="com.bean.domain.User"
parameterType="com.bean.domain.User">
        select * from user where 1=1
        <!--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-->
        <if test="username != null">
            <!--进行与上面的内容拼接-->
            and username=#{username}
        </if>
    </select>

</mapper>
```

```java
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {

//        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
        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"
    parameterType="com.bean.domain.User">
9           select * from user
10          <where>
11              <if test="username != null">and username=#{username}</if>
12              <if test="sex != null">and sex=#{sex}</if>
13          </where>
14      </select>
15
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

```java
package com.bean.domain;

import java.util.List;

public class QueryVo {
    private User user;
    private List<Integer> ids;      //这里就是id的集合

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public List<Integer> getIds() {
        return ids;
    }

    public void setIds(List<Integer> ids) {
        this.ids = ids;
    }
}
```

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

<mapper namespace="com.bean.dao.IUserDao">
    <select id="findUserByIds" parameterType="com.bean.domain.QueryVo"
resultType="com.bean.domain.User">
        select * from user
        <where>
            <!--假如id不为空-->
            <if test="ids != null and ids.size()>0">
                <!--令集合内容为QueryVo.ids，语句为id in (id),#{uid}=ids[item]，以逗号分隔-->
                <foreach collection="ids" open="id in (" close=")" item="uid" separator=",">
                    <!--select * from user where id in (ids[0],ids[1],....,ids[item-1])-->
                    #{uid}
                </foreach>
            </if>
```

```
        </where>
    </select>

</mapper>
```

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.QueryVo;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Mybatis {
    InputStream inputStream = null;
    SqlSession sqlSession = null;
    IUserDao mapper;

    @Before //在测试之前调用
    public void init() throws IOException {
//        1. 读取配置文件
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//        2. 构建SqlSessionFactory工厂
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

//        3. 使用SqlSessionFactory工厂构建SqlSession对象
        sqlSession = factory.openSession();

//        4. 使用SqlSession对象创建Dao接口的代理对象
        mapper = sqlSession.getMapper(IUserDao.class);

    }

    @After  //在测试之后调用
    public void destory() throws IOException {

//        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
        sqlSession.commit();

//        6. 关闭资源
        sqlSession.close();
        inputStream.close();
    }

    @Test
```

```
    public void findUserByIds(){

        List<Integer> list = new ArrayList<Integer>();
        list.add(41);
        list.add(42);
        list.add(43);

        QueryVo vo = new QueryVo();
        vo.setIds(list);

        List<User> users = mapper.findUserByIds(vo);

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

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

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

<mapper namespace="com.bean.dao.IUserDao">
    <!--这里用来定义重复的语句-->
    <sql id="defaultSql">select * from user</sql>

    <select id="findUserByCondition" resultType="com.bean.domain.User" parameterType="com.bean.domain.User">
        <!--语句引入-->
        <include refid="defaultSql"></include>
        <where>
            <if test="username != null">and username=#{username}</if>
            <if test="sex != null">and sex=#{sex}</if>
        </where>
    </select>

    <select id="findUserByIds" parameterType="com.bean.domain.QueryVo" resultType="com.bean.domain.User">
        <include refid="defaultSql"></include>
        <where>
            <if test="ids != null and ids.size()>0">
                <foreach collection="ids" open="id in (" close=")" item="id" separator=",">
                    #{id}
                </foreach>
            </if>
        </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种添加
> 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

```sql
DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL auto_increment,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` datetime default NULL COMMENT '生日',
  `sex` char(1) default NULL COMMENT '性别',
  `address` varchar(256) default NULL COMMENT '地址',
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

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');
```

> 外键是UID，引用user表的ID

```sql
DROP TABLE IF EXISTS `account`;

CREATE TABLE `account` (
  `ID` int(11) NOT NULL COMMENT '编号',
  `UID` int(11) default NULL COMMENT '用户编号',
  `MONEY` double default NULL COMMENT '金额',
  PRIMARY KEY  (`ID`),
  KEY `FK_Reference_8` (`UID`),
  CONSTRAINT `FK_Reference_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert  into `account`(`ID`,`UID`,`MONEY`) values (1,46,1000),(2,45,1000),(3,46,2000);
```

- 两个domain

```java
package com.bean.domain;

import java.io.Serializable;

public class Account implements Serializable {
    private Integer id;
    private Integer uid;
    private Double money;

    public Account() {
    }

    public Account(Integer id, Integer uid, Double money) {
```

```
        this.id = id;
        this.uid = uid;
        this.money = money;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getUid() {
        return uid;
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                '}';
    }
}
```

```
package com.bean.domain;

import java.io.Serializable;
import java.util.Date;

public class User implements Serializable {
    //注释中是数据库中的名称
    private Integer id; //id
    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）
    private Date birthday;  //birthday
    private String sex;     //sex
    private String address;     //address

    public User() {
    }

    public User(Integer id, String username, Date birthday, String sex, String address) {
```

```
        this.id = id;
        this.username = username;
        this.birthday = birthday;
        this.sex = sex;
        this.address = address;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

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

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- 两个dao

```java
package com.bean.dao;

import com.bean.domain.Account;

import java.util.List;

public interface IAccountDao {
    /**
     * 查询所有账户
     * @return
     */
    List<Account> findAll();
}
```

```java
package com.bean.dao;

import com.bean.domain.User;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {
    /**
     * 查询所有用用户
     * @return
     */
    List<User> findAll();

    /**
     * 根据ID查询用户
     * @return
     */
    User findUserById();
}
```

- 两个xml

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

<mapper namespace="com.bean.dao.IAccountDao">
    <select id="findAll" resultType="com.bean.domain.Account">
        select * from account
    </select>
</mapper>
```

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

```
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.bean.dao.IUserDao">


    <select id="findAll" resultType="com.bean.domain.User">
        select * from user
    </select>

    <select id="findUserById" resultType="com.bean.domain.User">
        select * from user
        <where>
            <if test="id!=null">id=#{id}</if>
        </where>
    </select>

</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：
>
> ```
> select u.*,a.id as aid,a.uid,a.money from account a,user u where u.id=a.uid;
> # 主要在于user内的id与account内的id冲突，所以要给acount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

```
package com.bean.domain;

import java.io.Serializable;

public class Account implements Serializable {
```

```
    private Integer id;
    private Integer uid;
    private Double money;

    /*从表实体对主表实体进行的引用，并编写getter setter方法
    * 其实话虽然比较绕口，但是实际上就是这样
    * */
    private User user;

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getUid() {
        return uid;
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                '}';
    }
}
```

- 主表(User)内容不变

```
package com.bean.domain;

import java.io.Serializable;
import java.util.Date;

```

```java
public class User implements Serializable {
    //注释中是数据库中的名称
    private Integer id; //id
    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）
    private Date birthday;  //birthday
    private String sex;     //sex
    private String address;     //address

    public User() {
    }

    public User(Integer id, String username, Date birthday, String sex, String address) {
        this.id = id;
        this.username = username;
        this.birthday = birthday;
        this.sex = sex;
        this.address = address;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

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

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

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

```
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- Account的接口

```
package com.bean.dao;

import com.bean.domain.Account;

import java.util.List;

public interface IAccountDao {
    /**
     * 查询所有账户
     * @return
     */
    List<Account> findAll();

    /**
     * 查询所有账户，连带User表中的内容
     * @return
     */
    List<Account> findAllAccountAndUser();
}
```

- `IAccount.xml` 文件实现

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

<mapper namespace="com.bean.dao.IAccountDao">
    <select id="findAll" resultType="com.bean.domain.Account">
        select * from account
    </select>

    <!--下面是本次内容-->

    <!--定义一个ResultMap，目的是为了把User和Account的内容整合到一起-->
    <resultMap id="accountUserMap" type="account">
        <!--这里关于为什么是aid解释一下：
            column是表中的id值，虽然在表中的的确是id没错，
            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
```

```
        -->
        <id property="id" column="aid"></id>
        <result property="uid" column="uid"></result>
        <result property="money" column="money"></result>
        <!--
            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
            一对一关系映射：association
            association
                property：映射的类
                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
                javaType：向哪个对象封装
        -->
        <association property="user" column="uid" javaType="user">
            <id property="id" column="id"></id>
            <result property="username" column="username"></result>
            <result property="birthday" column="birthday"></result>
            <result property="sex" column="sex"></result>
            <result property="address" column="address"></result>
        </association>
    </resultMap>


    <select id="findAllAccountAndUser" resultMap="accountUserMap" resultType="account">
        <!--注意，这里写的是aid所以resultMap中才使用的aid-->
        select u.*,a.id as aid,a.uid,a.money from account a,user u where u.id=a.uid;
    </select>
</mapper>
```

- 测试类

```
package com.bean.test;

import com.bean.dao.IAccountDao;
import com.bean.dao.IUserDao;
import com.bean.domain.Account;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.annotation.Resource;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class AccountTest {

    InputStream resourceAsStream;
    SqlSession sqlSession;
    IAccountDao accountDao;
    @Before
    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：

```
select * from user u left join account a on u.id = a.uid;
# 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

```
package com.bean.domain;

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

public class User implements Serializable {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    /*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
    * 注意一件事：accounts就是后面IUser.xml中实现resultType中collection的accounts
    * */
    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
```

```
    }

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

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- `IUserDao`

```
package com.bean.dao;

import com.bean.domain.User;

import java.util.List;

/**
 * IUserDao接口就是我们常说的持久层接口，也可以写作UserDao或者UserMapper
 */
public interface IUserDao {
    /**
     * 查询所有用用户，顺带查询所有的从表信息
```

```
     * @return
     */
    List<User> findAll();

    /**
     * 根据ID查询用户
     * @return
     */
    User findUserById();
}
```

- IUserDao.xml

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

<mapper namespace="com.bean.dao.IUserDao">




    <select id="findUserById" resultType="com.bean.domain.User">
        select * from user
        <where>
            <if test="id!=null">id=#{id}</if>
        </where>
    </select>


    <!--一对多的查询，type为Account-->
    <resultMap id="userAccountMap" type="user">
        <id property="id" column="id"></id>
        <result property="username" column="username"></result>
        <result property="birthday" column="birthday"></result>
        <result property="sex" column="sex"></result>
        <result property="address" column="address"></result>
        <!--
            collection：一对多关系映射
                property：accounts就是User类中实现从表关系映射集合的实现类
                ofType：集合中每个元素的类型
        -->
        <collection property="accounts" ofType="account">
            <id property="id" column="id"></id>
            <result property="uid" column="uid"></result>
            <result property="money" column="money"></result>
        </collection>
    </resultMap>


<!--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-->
    <select id="findAll" resultMap="userAccountMap">
        select * from user u left join account a on u.id = a.uid;
    </select>

```

```
</mapper>
```

- 测试类 `UserTest`

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class UserTest {

    InputStream resourceAsStream;
    SqlSession sqlSession;
    IUserDao userDao;
    @Before
    public void Before() throws IOException {
        resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(resourceAsStream);
        sqlSession = factory.openSession();
        userDao = sqlSession.getMapper(IUserDao.class);
    }

    @After
    public void After() throws IOException {
        sqlSession.commit();
        sqlSession.close();
        resourceAsStream.close();
    }

    @Test
    public void findAll(){
        List<User> users = userDao.findAll();
        for (User user : users) {
            System.out.println("--- 每一个UserAccount ---");
            System.out.println(user);
            System.out.println(user.getAccounts());
        }
    }
}
```

- 结果

```
--- 每一个UserAccount ---
```

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

```sql
DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL auto_increment,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` datetime default NULL COMMENT '生日',
  `sex` char(1) default NULL COMMENT '性别',
  `address` varchar(256) default NULL COMMENT '地址',
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

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）

```sql
DROP TABLE IF EXISTS `role`;

CREATE TABLE `role` (
  `ID` int(11) NOT NULL COMMENT '编号',
  `ROLE_NAME` varchar(30) default NULL COMMENT '角色名称',
  `ROLE_DESC` varchar(60) default NULL COMMENT '角色描述',
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

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）

```sql
DROP TABLE IF EXISTS `user_role`;

CREATE TABLE `user_role` (
  `UID` int(11) NOT NULL COMMENT '用户编号',
  `RID` int(11) NOT NULL COMMENT '角色编号',
  PRIMARY KEY  (`UID`,`RID`),
  KEY `FK_Reference_10` (`RID`),
  CONSTRAINT `FK_Reference_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),
  CONSTRAINT `FK_Reference_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

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

```sql
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
            left join user u on ur.uid=u.id;
# 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`

```java
package com.bean.domain;

import java.io.Serializable;
import java.util.List;

public class Role implements Serializable {

    private int id;
    private String roleName;
    private String roleDesc;

    /*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*/
    List<User> users;

    public List<User> getUsers() {
        return users;
    }

    public void setUsers(List<User> users) {
        this.users = users;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```

```
    }

    public String getRoleName() {
        return roleName;
    }

    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }

    public String getRoleDesc() {
        return roleDesc;
    }

    public void setRoleDesc(String roleDesc) {
        this.roleDesc = roleDesc;
    }

    @Override
    public String toString() {
        return "Role{" +
                "id=" + id +
                ", roleName='" + roleName + '\'' +
                ", roleDesc='" + roleDesc + '\'' +
                '}';
    }
}
```

- `user.class`

```
package com.bean.domain;

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

public class User implements Serializable {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    /*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*/
    private List<Role> roles;

    public List<Role> getRoles() {
        return roles;
    }

    public void setRoles(List<Role> roles) {
        this.roles = roles;
    }

    public Integer getId() {
        return id;
    }
```

```
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`

```java
package com.bean.dao;

import com.bean.domain.Role;

import java.util.List;

public interface IRoleDao {

    List<Role> findAll();
}
```

- IRoleDao.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.bean.dao.IRoleDao">

    <resultMap id="roleMap" type="role">
        <id property="id" column="rid"></id>
        <result property="roleName" column="role_name"></result>
        <result property="roleDesc" column="role_desc"></result>
        <collection property="users" ofType="user">
            <id property="id" column="id"></id>
            <result property="username" column="username"></result>
            <result property="birthday" column="birthday"></result>
            <result property="sex" column="sex"></result>
            <result property="address" column="address"></result>
        </collection>
    </resultMap>

    <select id="findAll" resultMap="roleMap">
        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
                            left join user u on ur.uid=u.id;
    </select>
</mapper>
```

- RoleTest.class

```java
package com.bean.test;

import com.bean.dao.IRoleDao;
import com.bean.domain.Role;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.annotation.Resource;
import java.io.IOException;
```

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

public class RoleTest {
    InputStream inputStream;
    SqlSessionFactoryBuilder builder;
    SqlSession sqlSession;
    IRoleDao roleDao;

    @Before
    public void Before() throws IOException {
        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);
        sqlSession = factory.openSession();
        roleDao = sqlSession.getMapper(IRoleDao.class);
    }

    @After
    public void After() throws IOException {
        sqlSession.commit();
        sqlSession.close();
        inputStream.close();
    }

    @Test
    public void findAll() {
        List<Role> roles = roleDao.findAll();
        for (Role role : roles) {
            System.out.println("---  所有角色及角色赋予的人 ---");
            System.out.println(role);
            System.out.println(role.getUsers());
        }
    }

}
```

- 结果

```
---  所有角色及角色赋予的人 ---
Role{id=1, roleName='院长', roleDesc='管理整个学院'}
[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'}]
---  所有角色及角色赋予的人 ---
Role{id=2, roleName='总裁', roleDesc='管理整个公司'}
[User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='北京'}]
---  所有角色及角色赋予的人 ---
Role{id=3, roleName='校长', roleDesc='管理整个学校'}
[]
```

- 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`

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

<configuration>

    <properties url="file:///E:/JetBrains/java/intellj%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>

<!--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：https://mybatis.org/mybatis-3/zh/configuration.html#settings
    settings：mybatis的全部设置
        lazyLoadingEnabled：延迟加载的全局开关
            true：开启延迟加载
            false：关闭延迟加载
        aggressiveLazyLoading：是否取消按需加载
            true：确认关闭按需加载
            false：不确认关闭按需加载（按需加载）
-->
    <settings>
        <setting name="lazyLoadingEnabled" value="true"/>
        <setting name="aggressiveLazyLoading" value="false"/>
    </settings>

    <typeAliases>
        <package name="com.bean.domain"></package>
    </typeAliases>

    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>

    </environments>
    <mappers>
        <package name="com.bean.dao"></package>
    </mappers>

</configuration>
```

- `Account.class`

```java
package com.bean.domain;

import java.io.Serializable;

```

```java
public class Account implements Serializable {
    private int id;
    private int uid;
    private int money;

    /*把User放进来*/
    private User user;

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getUid() {
        return uid;
    }

    public void setUid(int uid) {
        this.uid = uid;
    }

    public int getMoney() {
        return money;
    }

    public void setMoney(int money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                '}';
    }
}
```

- `IAccountDao.xml`

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

```
<mapper namespace="com.bean.dao.IAccountDao">

    <resultMap id="accountMap" type="account">
        <id property="id" column="id"></id>
        <result property="uid" column="uid"></result>
        <result property="money" column="money"></result>
<!--
    associaion：进行一对一关系映射
        property：映射的类
        column：用哪个字段来按需获取
        javaType：向哪个对象进行封装
        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
-->
        <association property="user" column="uid" javaType="user"
select="com.bean.dao.IUserDao.findById">
<!--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
    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
            <id property="id" column="id"></id>
            <result property="username" column="username"></result>
            <result property="birthday" column="birthday"></result>
            <result property="sex" column="sex"></result>
            <result property="address" column="address"></result>
-->
        </association>
    </resultMap>

    <!--查询所有账户信息并且返回对应的用户信息-->
    <select id="findAll" resultMap="accountMap">
        select * from account;
    </select>

</mapper>
```

- IUserDao.xml

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

<mapper namespace="com.bean.dao.IUserDao">

    <select id="findUserById" parameterType="int" resultType="com.bean.domain.User" >
        select * from user where id=#{id};
    </select>

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

    <select id="findAll" resultMap="userMap">
        select * from user u left join account a on u.id=a.uid;
    </select>

```

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

- `IAccountDao.interface`

```
package com.bean.dao;

import com.bean.domain.Account;

import java.util.List;

public interface IAccountDao {

    /**
     * 查询所有账户信息，实现按需加载对应的账户信息
     * @return
     */
    List<Account> findAll();

}
```

- `AccountTest.class`

```
package com.bean.test;

import com.bean.dao.IAccountDao;
import com.bean.domain.Account;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class AccountTest {
    InputStream resourceAsStream;
    SqlSession sqlSession;
    IAccountDao accountDao;

    @Before
    public void Before() throws IOException {

        resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(resourceAsStream);

```

```
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
 .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
 .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
 .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`

```java
package com.bean.dao;

import com.bean.domain.Account;

import java.util.List;

public interface IAccountDao {
    List<Account> findAccountByUid(Integer uid);
}
```

- `IUserDao`

```java
package com.bean.dao;

import com.bean.domain.User;

import java.util.List;


public interface IUserDao {

    /**
     * 查询所有User对象，顺便实现延迟加载Account
     * @return
     */
    List<User> findAll();
}
```

- `User.class`

```java
package com.bean.domain;

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

public class User implements Serializable {
    private int id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    /*加入Account*/
    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }


```

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

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- IUserDao.xml

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

<mapper namespace="com.bean.dao.IUserDao">

    <resultMap id="userMap" type="com.bean.domain.User">
        <id property="id" column="id"></id>
        <result property="username" column="username"></result>
        <result property="birthday" column="birthday"></result>
        <result property="sex" column="sex"></result>
        <result property="address" column="address"></result>

```

```
        <!--collection：一对多关系映射
            property：实现从表关系映射的集合类
            ofType：集合中的每一个元素的类型
            column：实现延迟加载的查询方法的参数
            select：实现延迟加载的查询方法
            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
        -->
        <collection property="accounts" ofType="account" column="id" select="com.bean.dao.IAccountDao.findAccountByUid"></collection>

    </resultMap>

    <select id="findAll" resultMap="userMap">
        select * from user
    </select>

</mapper>
```

- `IAccountDao.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.bean.dao.IAccountDao">

    <select id="findAccountByUid" parameterType="int" resultType="com.bean.domain.Account">
        select * from account where uid=#{uid}
    </select>

</mapper>
```

- `UserTest.class`

```
package com.bean.test;

import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class UserTest {

    InputStream resourceAsStream;
    SqlSession sqlSession;
    IUserDao userDao;
    @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

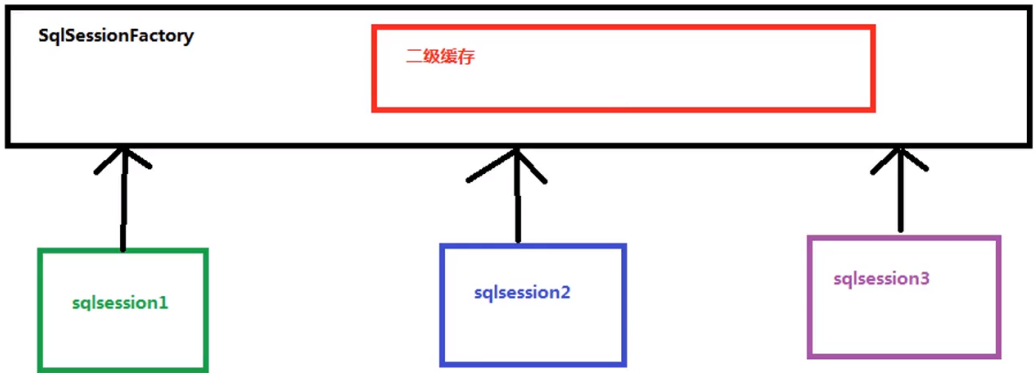
- 减少和数据库的交互次数，提高执行效率

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
  - 数据的正确与否队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，读数据度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`

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

```
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>


    <properties
url="file:///E:/JetBrains/java/intellj%20Idea/Mybatis/Mybatis01/src/main/resources/jdbcConfig.properties"></properties>

    <settings>
        <!--延迟加载-->
        <setting name="lazyLoadingEnabled" value="true"/>
        <setting name="aggressiveLazyLoading" value="false"/>

        <!--二级缓存开启-->
        <setting name="cacheEnabled" value="true"/>
    </settings>

    <typeAliases>
        <package name="com.bean.domain"></package>
    </typeAliases>




    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>

    </environments>
    <mappers>
        <package name="com.bean.dao"></package>
    </mappers>

</configuration>
```

- IUserDao.xml

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

<mapper namespace="com.bean.dao.IUserDao">

    <!--开启二级缓存-->
    <cache/>

    <!--设置可以使用二级缓存-->
    <select id="findAll" resultType="user" useCache="true">
        select * from user
```

```
    </select>

</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`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--引入外部配置文件-->
    <properties resource="jdbcConfig.properties"></properties>

    <!--配置别名-->
    <typeAliases>
        <package name="com.bean.domain"/>
    </typeAliases>

    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--指定带有注解dao接口所在的位置-->
```

```
    <mappers>
        <package name="com.bean.dao"/>
    </mappers>
</configuration>
```

- `jdbcConfig.properties`

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

- 加入 `log4j` 文件

- `com.bean.domain.User.class`

```
package com.bean.domain;

import java.io.Serializable;
import java.util.Date;

public class User implements Serializable {
    private int id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    public User() {
    }

    public User(int id, String username, Date birthday, String sex, String address) {
        this.id = id;
        this.username = username;
        this.birthday = birthday;
        this.sex = sex;
        this.address = address;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

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

    public Date getBirthday() {
        return birthday;
```

```
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- `com.bean.dao.IUserDao.interface`

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface IUserDao {
/*
* mybatis中对于注解开发一共有四个注解
* @Select @Insert @Update @Delete
* 以@Select举例：
*   @Select(value="语句")，当然，参数只有一个的时候可以省略value
* */


    /**
     * 查询所有用户
     * @return
     */
```

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

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`

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import java.util.List;

public interface IUserDao {

    /**
     * 查询所有方法
     * @return
     */
    @Select("select * from user")
    List<User> findAll();

    /**
     * 插入用户
     */
    @Insert("insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address})")
    void insertUser(User user);

    /**
     * 更新用户
```

```
     * @param user
     */
    @Update("update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id}")
    void updateUser(User user);

    /**
     * 删除操作
     * @param id
     */
    @Delete("delete from user where id=#{id}")
    void deleteUser(Integer id);

    /**
     * 根据id查询用户
     * @return
     */
    @Select("select * from user where id=#{id}")
    User findUserById(Integer id);

    /**
     * 根据名称进行模糊查询
     * @param username
     * @return
     */
    @Select("select * from user where username like #{username}")
    List<User> findUsersLikeUsername(String username);

    /**
     * 使用聚合函数查询总人数
     * @return
     */
    @Select("select count(id) from user")
    int findCount();

}
```

- `UserTest.class`

```
import com.bean.dao.IUserDao;
import com.bean.domain.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class UserTest {

    InputStream inputStream;
```

```
    SqlSession session;
    IUserDao userDao;
    @Before
    public void Before() throws IOException {

        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);
        session = factory.openSession();
        userDao = session.getMapper(IUserDao.class);
    }

    @After
    public void After() throws IOException {
        session.commit();
        session.close();
        inputStream.close();
    }


    @Test
    public void findAll(){
        List<User> users = userDao.findAll();
        for (User user : users) {
            System.out.println(user);
        }
    }

    @Test
    public void insertUser(){
        User user = new User();
        user.setUsername("annotation");
        user.setSex('男');
        user.setBirthday(new Date());
        user.setAddress("annotationAddress");
        userDao.insertUser(user);
    }

    @Test
    public void updateUser(){
        User user = new User();
        user.setId(49);
        user.setUsername("annotation");
        user.setAddress("annotationAddress");
        user.setSex('男');
        user.setBirthday(new Date());
        userDao.updateUser(user);
    }

    @Test
    public void deleteUser(){
        userDao.deleteUser(49);
    }

    @Test
    public void findUserById(){
        User user = userDao.findUserById(48);
        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 = {
```

```
            @Result(id = true,property = "id",column = "id"),
            @Result(property = "username",column = "username"),
            @Result(property = "address",column = "address"),
            @Result(property = "sex",column = "sex")
        }
    )
    List<User> findAll();

    /**
     * 根据id查询用户
     * @return
     */
    @Select("select * from user where id=#{id}")
    @ResultMap("userMap")
    User findUserById(Integer id);

    /**
     * 根据名称进行模糊查询
     * @param username
     * @return
     */
    @Select("select * from user where username like #{username}")
    @ResultMap("userMap")
    List<User> findUsersLikeUsername(String username);

}
```

- 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`

```
package com.bean.domain;

public class Account {
    private Integer id;
    private Integer uid;
    private Integer money;


    /*要实现一对一的加载，就需要User数据*/
```

```
    private User user;

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }


    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getUid() {
        return uid;
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public Integer getMoney() {
        return money;
    }

    public void setMoney(Integer money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                '}';
    }
}
```

- `IUserDao.interface`

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.*;

import java.util.List;

public interface IUserDao {

```

```
    /**
     * 查询所有方法
     * @return
     */
    @Select("select * from user")
    @Results(
        id="userMap",
        value = {
            @Result(id = true,property = "id",column = "id"),
            @Result(property = "username",column = "username"),
            @Result(property = "address",column = "address"),
            @Result(property = "sex",column = "sex")
        }
    )
    List<User> findAll();

    /**
     * 根据id查询用户
     * @return
     */
    @Select("select * from user where id=#{id}")
    @ResultMap("userMap")
    User findUserById(Integer id);

    /**
     * 根据名称进行模糊查询
     * @param username
     * @return
     */
    @Select("select * from user where username like #{username}")
    @ResultMap("userMap")
    List<User> findUsersLikeUsername(String username);

}
```

- `IAccountDao.interface`

```
package com.bean.dao;

import com.bean.domain.Account;
import org.apache.ibatis.annotations.*;
import org.apache.ibatis.mapping.FetchType;

import java.util.List;

public interface IAccountDao {

    /*首先实现数据的封装，将User也封装到这个里面*/
    /*
    @Results：定义类属性与数据库字段的对应关系
        - id：唯一标识，@ResultMap可以根据这个找到
        - value：类属性与表字段的映射
            - @Result：单个字段的对应关系
                - id：是否是主键，默认为false
                - 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
                - 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
```

```
                - one：一对一（mybatis中不存在多对一）的时候使用的那个一
                    - @One
                        - select：查询使用的方法，要求的是全限定类名
                        - fetchType：如何进行加载
                            - FetchType.EAGER：立即加载
                            - FetchType.LAZY：懒加载
                            - FetchType.DEFAULT：默认，可能是随意选一个
    @ResultMap：引用@Results
        - value：引用的@Results值，假如只有一个可以去掉value和括号
    */


    /**
     * 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
     * @return
     */
    @Select("select * from account")
    @Results(
            id = "accountMap",
            value = {
                    @Result(id = true,property = "id",column = "id"),
                    @Result(property = "uid",column = "uid"),
                    @Result(property = "money",column = "money"),
                    @Result(
                            property = "user",
                            column = "uid",
                            one = @One(select =
"com.bean.dao.IUserDao.findUserById",fetchType = FetchType.EAGER)
                    )
            }
    )
    List<Account> findAll();
}
```

- `AccountTest.class`

```
import com.bean.dao.IAccountDao;
import com.bean.domain.Account;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class AccountTest {

    InputStream inputStream;
    SqlSession session;
    IAccountDao accountDao;
    @Before
```

```
    public void Before() throws IOException {

        inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);
        session = factory.openSession();
        accountDao = session.getMapper(IAccountDao.class);
    }

    @After
    public void After() throws IOException {
        session.commit();
        session.close();
        inputStream.close();
    }


    @Test
    public void findAll(){
        List<Account> accounts = accountDao.findAll();
        for (Account account : accounts) {
            System.out.println("-------------");
            System.out.println(account);
            System.out.println(account.getUser());
        }
    }

}
```

一对多的操作

- `User.class`

```
package com.bean.domain;

import java.util.Date;
import java.util.List;

public class User {
    private Integer id;
    private String username;
    private Date birthday;
    private Character sex;
    private String address;


    /*首先我们使用一对多的映射，将Account的集合封装进来*/
    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }
```

```
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  }
```

- `IUserDao.interface`

```
1   package com.bean.dao;
```

```java

import com.bean.domain.User;
import org.apache.ibatis.annotations.*;
import org.apache.ibatis.mapping.FetchType;

import java.util.List;

public interface IUserDao {

    /*
    @Results：定义类属性与数据库字段的对应关系
        - id：唯一标识，@ResultMap可以根据这个找到
        - value：类属性与表字段的映射
            - @Result：单个字段的对应关系
                - id：是否是主键，默认为false
                - 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
                - 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
                - many
                    - @Many
                        - select：要执行的方法的全类名
                        - fetchType
                            - FetchType.LAZY：懒加载
                            - FetchType.EAGER：立即加载
                            - FetchType.DEFAULT：默认，可能是随意选一个

    @ResultMap：引用@Results
        - value：引用的@Results值，假如只有一个可以去掉value和括号
    */

    /**
     * 查询所有方法，实现一对多功能，并且实现懒加载
     * @return
     */
    @Select("select * from user")
    @Results(
            id="userMap",
            value = {
                    @Result(id = true,property = "id",column = "id"),
                    @Result(property = "username",column = "username"),
                    @Result(property = "address",column = "address"),
                    @Result(property = "sex",column = "sex"),
                    @Result(
                        property = "accounts",
                        column = "id",
                        many = @Many(
                            select = "com.bean.dao.IAccountDao.findAccontByUid",
                            fetchType = FetchType.LAZY
                        )
                    )
            }
    )
    List<User> findAll();

    /**
     * 根据id查询用户
     * @return
     */
    @Select("select * from user where id=#{id}")
```

```java
    @ResultMap("userMap")
    User findUserById(Integer id);

    /**
     * 根据名称进行模糊查询
     * @param username
     * @return
     */
    @Select("select * from user where username like #{username}")
    @ResultMap("userMap")
    List<User> findUsersLikeUsername(String username);

}
```

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` 中配置开启缓二级缓存
> 2. 使用注解 `@CacheNamespace`

1. 在 `SqlMapConfig.xml` 中配置二级缓存

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

<configuration>
    <properties resource="jdbcConfig.properties"></properties>

    <!--配置开启二级缓存-->
    <settings>
        <setting name="cacheEnabled" value="true"/>
    </settings>

    <typeAliases><package name="com.bean.domain"/></typeAliases>

    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <mappers><package name="com.bean.dao"/></mappers>
```

```
</configuration>
```

2. 使用注解 `@CacheNameSpace(blocking=true)`

```
package com.bean.dao;

import com.bean.domain.User;
import org.apache.ibatis.annotations.*;
import org.apache.ibatis.mapping.FetchType;

import java.util.List;

/*在这里配置二级缓存，把blocking配置为true*/
@CacheNamespace(blocking = true)
public interface IUserDao {


    @Select("select * from user")
    @Results(
            id="userMap",
            value = {
                    @Result(id = true,property = "id",column = "id"),
                    @Result(property = "username",column = "username"),
                    @Result(property = "address",column = "address"),
                    @Result(property = "sex",column = "sex"),
                    @Result(
                        property = "accounts",
                        column = "id",
                        many = @Many(
                            select = "com.bean.dao.IAccountDao.findAccontByUid",
                            fetchType = FetchType.LAZY
                        )
                    )
            }
    )
    List<User> findAll();

    @Select("select * from user where id=#{id}")
    @ResultMap("userMap")
    User findUserById(Integer id);

    @Select("select * from user where username like #{username}")
    @ResultMap("userMap")
    List<User> findUsersLikeUsername(String username);

}
```