# Mybatis框架

## 01.mybatis的概述

mybatis是一个优秀的基于java的持久层框架，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。

mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。

采用ORM思想解决了实体和数据库映射的问题，对jdbc进行了封装，屏蔽了jdbc api底层访问细节，使我们不用与jdbc api打交道，就可以完成对数据库的持久化操作。

### 1.1 mybatis介绍

- mybatis是一个**持久层**框架，用java编写的。
- 它封装了jdbc操作的很多细节，使开发者只需要关注sql语句本身，而无需关注注册驱动，创建连接等繁杂过程
- 它使用了ORM思想实现了结果集的封装。

> ORM：Object Relational Mappging 对象关系映射 简单的说： 就是把数据库表和实体类及实体类的属性对应起来 让我们可以操作实体类就实现操作数据库表。

实体类中的属性和数据库表的字段名称保持一致。 user User  id id  user_name user_name

### 1.2 JDBC程序回顾

```
public static void main(String[] args) {
        Connection connection;
        connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        try {
            //加载数据库驱动
            Class.forName("com.mysql.jdbc.Driver");
            //通过驱动管理类获取数据库链接
            connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf-8", "root", "root");
            //定义sql语句 ?表示占位符
            String sql = "select * from user where username = ?";
            //获取预处理statement
            preparedStatement = connection.prepareStatement(sql);
            //设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值
```

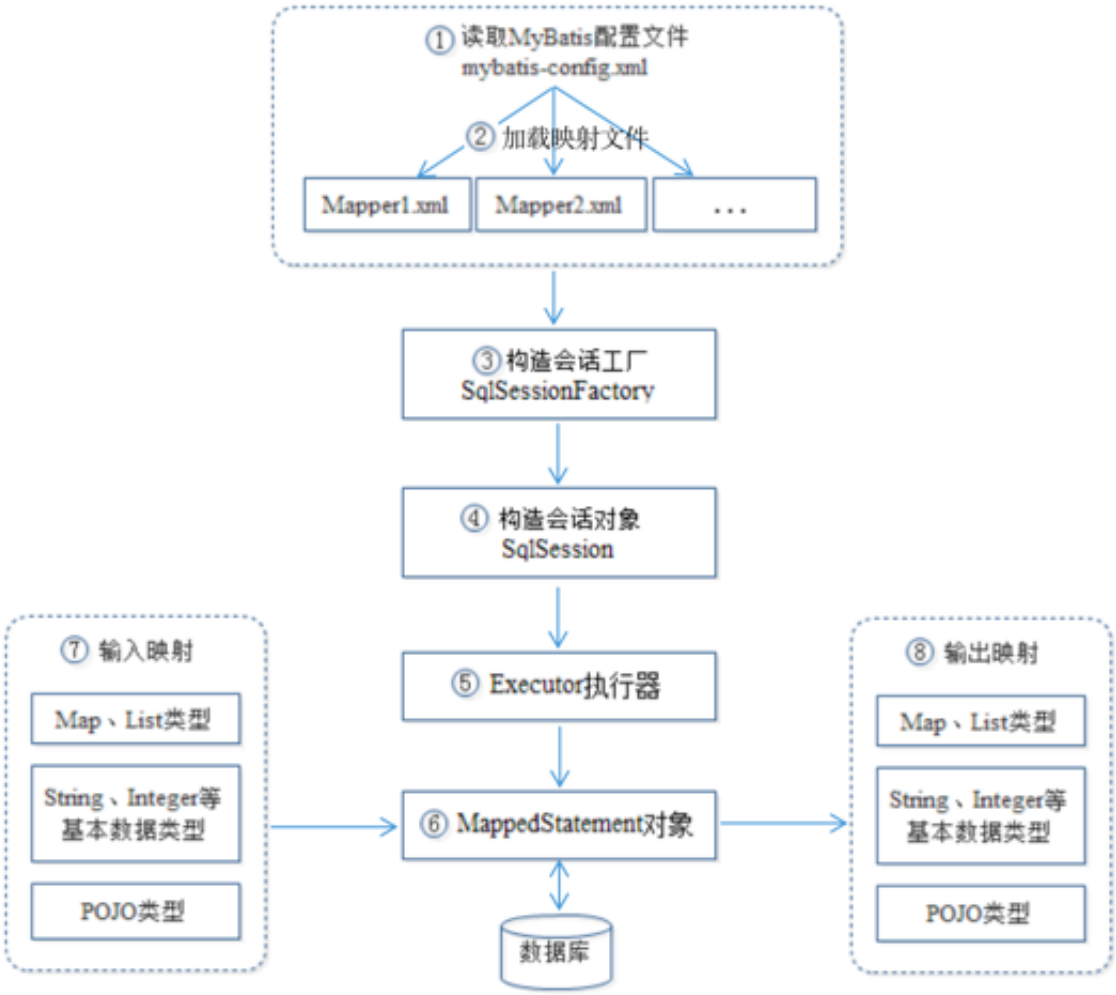
```
            preparedStatement.setString(1, "王五");
            //向数据库发出sql执行查询，查询出结果集
            resultSet = preparedStatement.executeQuery();
            //遍历查询结果集
            while (resultSet.next()) {
                System.out.println(resultSet.getString("id") + " " +
resultSet.getString("username"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //释放资源
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (preparedStatement != null) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

上边使用jdbc的原始方法（未经封装）实现了查询数据库表记录的操作。

### 1.3 jdbc问题分析

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
2. Sql语句在代码中硬编码，造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。
3. 使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。
4. 对结果集解析存在硬编码（查询列名），sql变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成pojo对象解析比较方便。

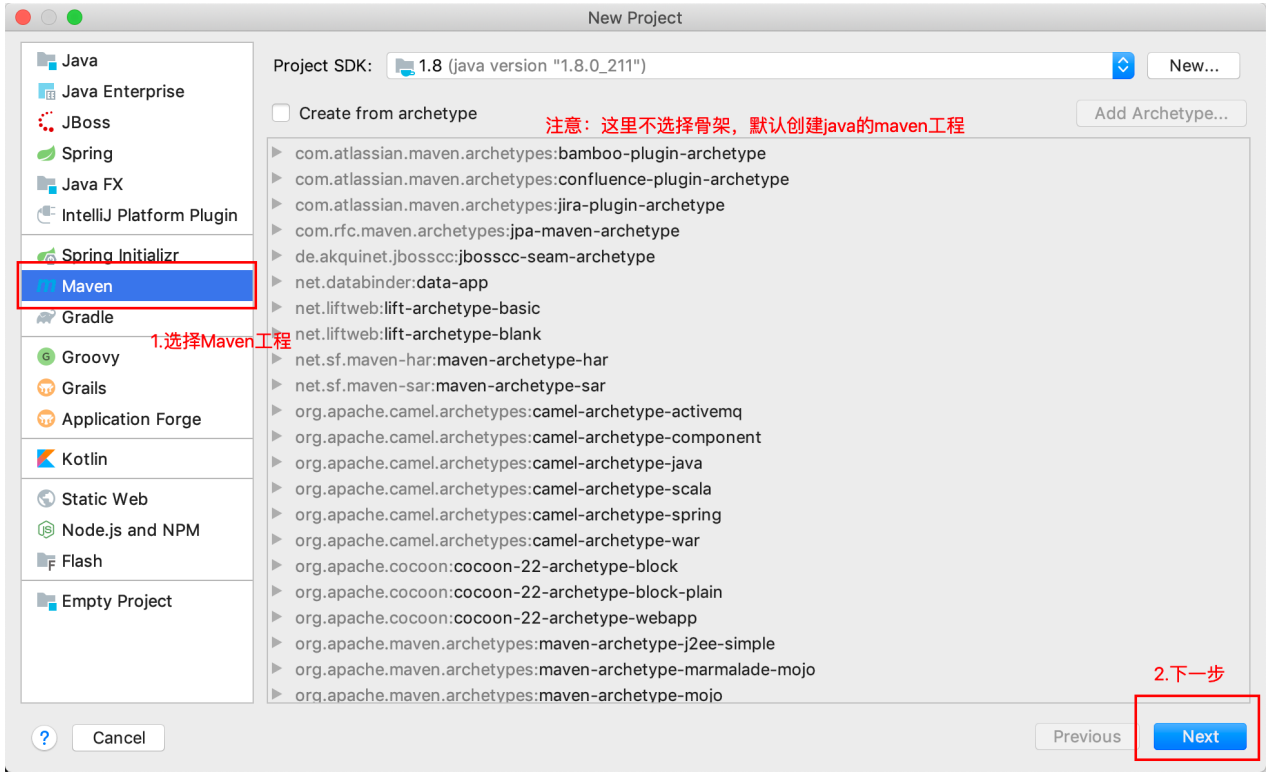
### 1.4 原理

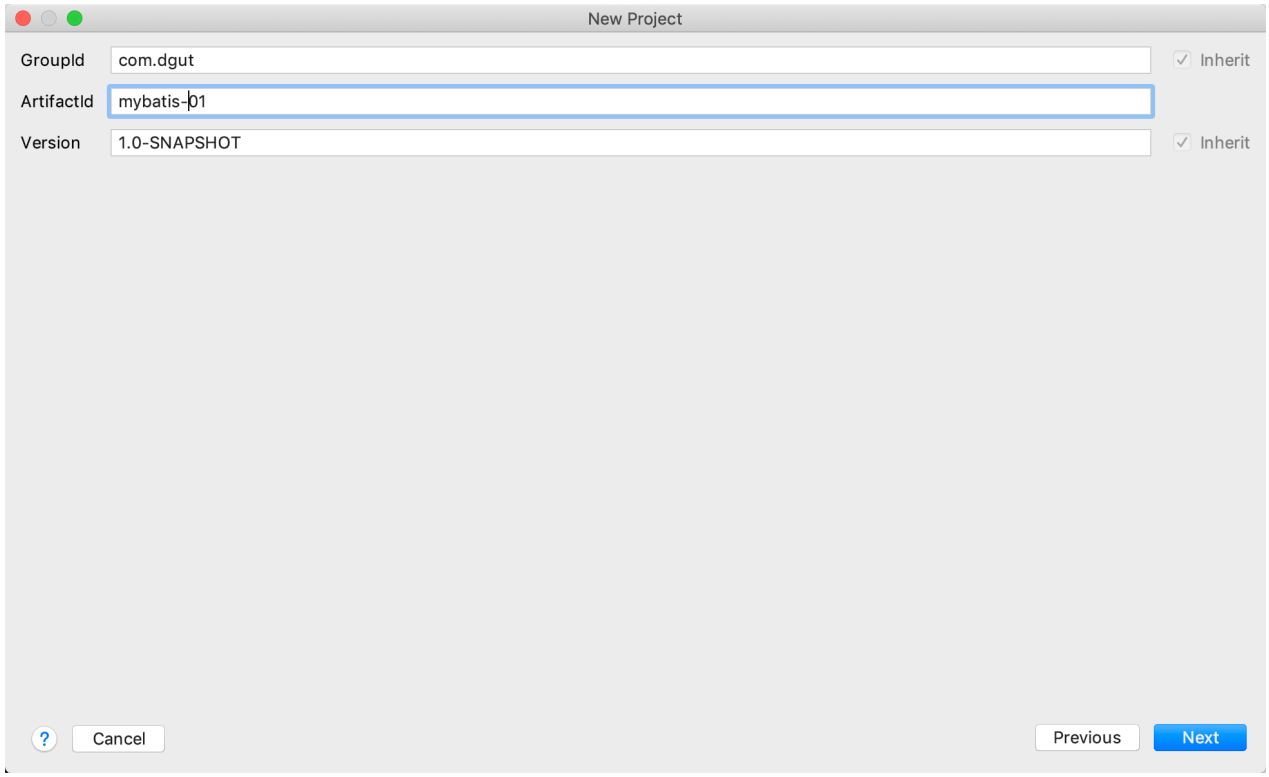



# 02.mybatis的环境搭建

## 2.1 搭建步骤

### 2.1.1 创建maven工程，并导入坐标




- 在pom.xml文件中添加Mybatis3.5.3的坐标

```
<dependencies>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.3</version>
        </dependency>

        <dependency>
```

```
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.17</version>
        </dependency>

        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>

    </dependencies>
```

#### 2.1.2 创建实体类和dao的接口

- 创建实体类
  - 创建com.dgut.domain包，并且新建一个**User**类

```
public class User {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String 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 + '\'' +
                '}';
    }
}
```

- 编写持久层接口**IUserDao** ，创建com.dgut.dao包，并在包下创建**IUserDao**接口

```
public interface IUserDao {
    public List<User> findAll();
}
```

#### 2.1.3 创建映射配置文件

- 在**resource**目录下创建com/dgut/dao目录,并且创建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.dgut.dao.IUserDao">
    <select id="findAll" resultType="com.dgut.domain.User">
    select * from user
  </select>
</mapper>
```

#### 2.1.4 创建Mybatis的主配置文件

- Resource目录下创建并且编写**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">
<configuration>
  <!-- 配置环境 -->
    <environments default="development">
        <!-- 配置mysql环境 -->
        <environment id="development">
            <!-- 配置事务 -->
            <transactionManager type="JDBC"/>
             <!-- 配置数据源连接池 -->
            <dataSource type="POOLED">
```

```
                <!-- 这里用得是mysql8的驱动包 -->
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="xieman123"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="com/dgut/dao/iUserDao.xml"/>
    </mappers>

</configuration>
```

#### 2.1.5 编写测试类

- 在test目录创建UserTest类

```
    @Test
    public void test1() throws IOException {
        //1.读取配置文件
        InputStream inputStream =
Resources.getResourceAsStream("sqlMapperConfig.xml");
        //2.创建SqlSessionFactory的构建者对象,使用构建者创建工厂对象
SqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        //3.使用SqlSessionFactory生产SqlSession对象
        SqlSession session = sqlSessionFactory.openSession();
        //4.使用SqlSession创建dao接口的代理对象
        IUserDao userDao = session.getMapper(IUserDao.class);
        //5.使用代理对象执行查询所有方法
        List<User> users = userDao.findAll();
        for (User user : users){
            System.out.println(user);
        }
        //6.释放资源
        session.close();
        inputStream.close();
    }
```

### 2.2 小结

通过快速入门示例，我们发现使用mybatis是非常容易的一件事情，因为只需要编写Dao接口并且按照mybatis要求编写两个配置文件，就可以实现功能。远比我们之前的jdbc方便多了。我们使用注解之后，将变得更为简单，只需要编写一个mybatis配置文件就够了。

> 说明 1.创建IUserDao.xml 和 IUserDao.java时名称是为了和我们之前的知识保持一致。 在Mybatis中它把持久层的操作接口名称和映射文件也叫做：Mapper 所以：IUserDao 和 IUserMapper是一样的

```
2. 在idea中创建目录的时候，它和包是不一样的
   包在创建时：com.dgut.dao它是三级结构
   目录在创建时：com.dgut.dao是一级目录
3. 映射配置文件的mapper标签namespace属性的取值必须是dao接口的全限定类名
4. 映射配置文件的操作配置（select），id属性的取值必须是dao接口的方法名
```

*当我们遵从了第三，四点之后，我们在开发中就无须再写dao的实现类。*

### 2.3 基于注解的mybatis使用

- 在持久层接口中添加注解

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

- 修改**SqlMapConfig.xml**

```
<!-- 告知mybatis映射配置的位置 -->
<mappers>
  <mapper class="com.dgut.dao.IUserDao"/>
 </mappers>
```

- 在使用基于注解的Mybatis配置时，请移除xml的映射配置（IUserDao.xml）。

### 2.4 日志

由于MyBatis默认使用log4j输出日志信息，所以如果要查看控制台的输出SQL语句，那么就需要在classpath路径下配置其日志文件。在resourses下新建一个log4j.properties文件

```
# 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
```

# 03.CRUD操作

### 3.1 根据ID查找Find

- 3.1.1 在持久层接口中添加**findById**方法

```
 /**
     * 根据ID查找
     * @param id
     * @return 用户对象
     */
    public User findById(Integer id);
```

- 3.1.2 在用户的映射配置文件中配置

```
<!--
        resultType属性：用于指定结果集的类型。
      parameterType属性：用于指定传入参数的类型。
    sql语句中使用#{}字符：  它代表占位符，相当于原来jdbc部分所学的?，都是用于执行语句时替换实际的数据。 具体的数据是由#{}里面的内容决定的。
         #{}中内容的写法：  由于数据类型是基本类型，所以此处可以随意写。
       -->
    <select id="findById" resultType="com.dgut.domain.User" >
        select * from user where id = #{abc}
    </select>
```

- 3.1.3 在测试类添加测试

```
public class MybatisTest {
    private  InputStream inputStream;
    private SqlSession session;
    private IUserDao userDao;
```

```
    @Before
    public void init() throws IOException {
        //1.读取配置文件
        inputStream =
Resources.getResourceAsStream("sqlMapperConfig.xml");
        //2.创建SqlSessionFactory的构建者对象,使用构建者创建工厂对象
SqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        //3.使用SqlSessionFactory生产SqlSession对象
        session = sqlSessionFactory.openSession();
        //4.使用SqlSession创建dao接口的代理对象
        userDao = session.getMapper(IUserDao.class);
    }

    @After
    public void destory() throws IOException {
        //6.释放资源
        session.close();
        inputStream.close();
    }

    @Test
    public void testFindAll() throws IOException {

        //5.使用代理对象执行查询所有方法
        List<User> users = userDao.findAll();
        for (User user : users){
            System.out.println(user);
        }
    }

    @Test
    public void testFindById() throws IOException {
        //5.使用代理对象执行查询所有方法
        User user = userDao.findById(41);
        System.out.println(user);
    }
}
```

### 3.2 保存操作save

- 3.2.1 在持久层接口中添加新增方法

```
/**
     * 插入用户
     * @param user 用户
     * @return 影响行数
     */
    public int inserUser(User user);
```

- 3.2.2 在用户的映射配置文件中配置

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

- 3.2.3 在测试类添加测试

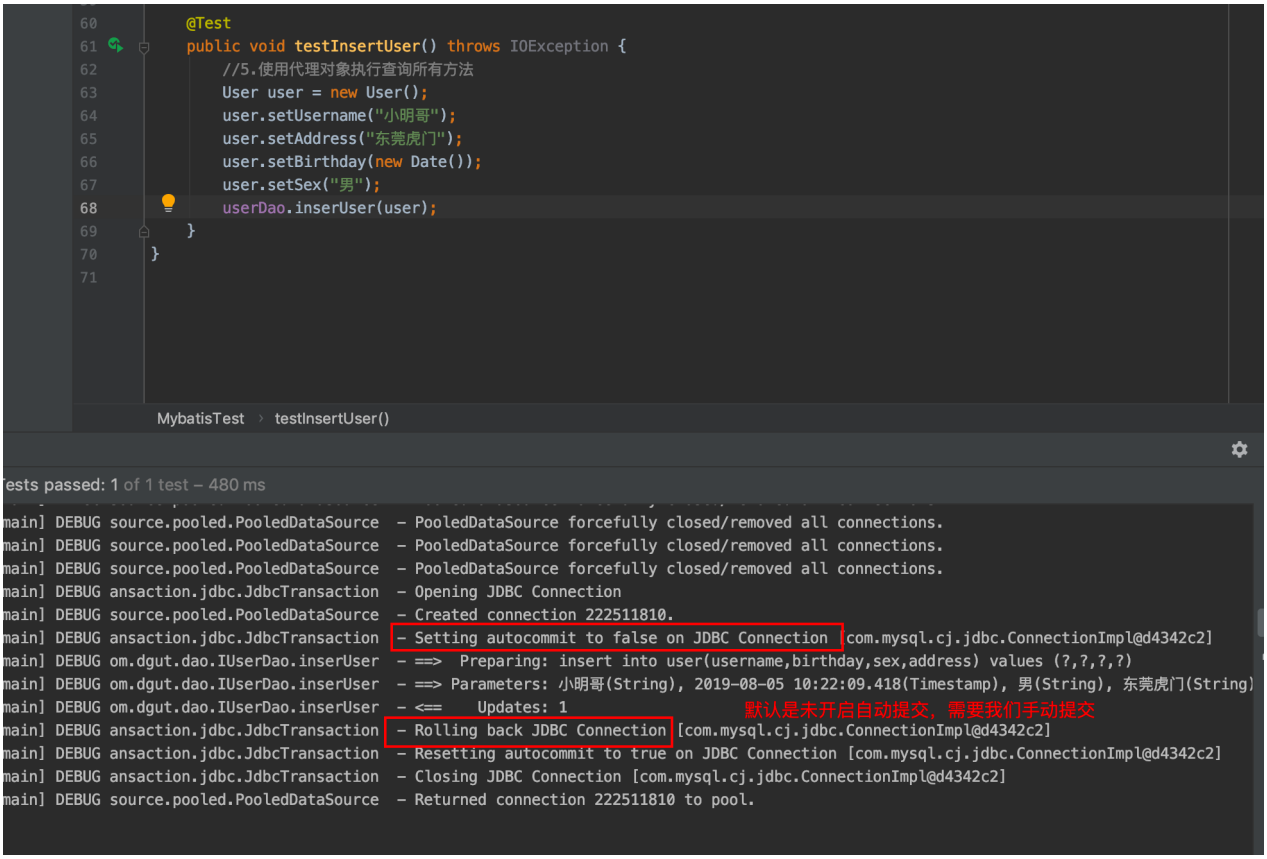
```
@Test
  public void testInsertUser() throws IOException {
      //5.使用代理对象执行查询所有方法
      User user = new User();
      user.setUsername("小明哥");
      user.setAddress("东莞虎门");
      user.setBirthday(new Date());
      user.setSex("男");
      userDao.inserUser(user);
  }
```

- 3.2.4 结果测试成功，但是数据未保存，

打开Mysql数据库发现并没有添加任何记录，原因是什么?

这一点和jdbc是一样的，我们在实现增删改时一定要去控制事务的提交，那么在mybatis中如何控制事务提交呢?

可以使用:session.commit();来实现事务提交原因:




- 3.2.5 在测试类添加 **session.commit();**

```
@Test
    public void testInsertUser() throws IOException {
        //5.使用代理对象执行查询所有方法
        User user = new User();
        user.setUsername("小明哥");
        user.setAddress("东莞虎门");
        user.setBirthday(new Date());
        user.setSex("男");
        userDao.inserUser(user);
        session.commit();
    }
```

- 3.2.6 中文保存数据库问题 修改SqlMapperConfig.xml 中的URL

```
<property name="url" value="jdbc:mysql://localhost:3306/mybatis?
characterEncoding=UTF-8"/>
```

- 3.2.7 问题扩展：新增用户**id**的返回值
  - 新增用户后，同时还要返回当前新增用户的id值，因为id是由数据库的自动增长来实现的，所以就相当于我们要在新增后将自动增长auto_increment的值返回。

```
<!-- 配置保存时获取插入的id -->
<selectKey keyColumn="id" keyProperty="id" resultType="int">
     select last_insert_id();
</selectKey>
```

```
<!--第二种方式-->
<insert id="inserUser" parameterType="user" useGeneratedKeys="true"
keyColumn="id" keyProperty="id">
        insert  into user(username,birthday,sex,address) values (#
{username},#{birthday},#{sex},#{address})
    </insert>
```

### 3.3 更新用户update

- 3.3.1 在持久层接口中添加updateUser方法

```
/**
     * 更新用户
     * @param user
     * @return 影响数据库记录的行数
     */
    int updateUser(User user);
```

- 3.3.2 在用户的映射配置文件中配置

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

- 3.3.3 在测试类添加测试

```
@Test
    public void testUpdateUser() throws Exception {
        //1.根据id查询
        User user = userDao.findById(50);
        // 2.更新操作
        user.setAddress("北京市顺义区");
        int res = userDao.updateUser(user);
        session.commit();
        System.out.println(res);
    }
```

### 3.4 删除用户delete

- 3.4.1 在持久层接口中添加deleteUser方法

```
    /**
     * 根据ID删除用户
     */
    int deleteUser(Integer id);
```

- 3.4.2 在用户的映射配置文件中配置

```
  <delete id="deleteUser" >
        delete from user where id = #{uid}
    </delete>
```

- 3.4.3 在测试类添加测试

```
@Test public void testDeleteUser() throws Exception {
        //6.执行操作
        int res = userDao.deleteUser(52);
        System.out.println(res);
        session.commit();
    }
```

### 3.5 用户模糊查询（第一种方式，掌握）

- 3.5.1 在持久层接口中添加findByName方法

```
    /**
     * 根据名称模糊查询
     */
    List<User> findByName(String username);
```

- 3.5.2 在用户的映射配置文件中配置

```
 <select id="findByName" resultType="com.dgut.domain.User"
parameterType="String">
        select * from user where username like #{username}
    </select>
```

- 3.5.3 在测试类添加测试

```
  @Test public void testFindByName() throws Exception {
        //6.执行操作
        List<User> users = userDao.findByName("%王%");
        System.out.println(users);
    }
```

### 3.6 用户模糊查询(第二种方式，了解)

- 3.6.1 在持久层接口中添加findByName方法

```
    /**
     * 根据名称模糊查询
     */
    List<User> findByName2(String username);
```

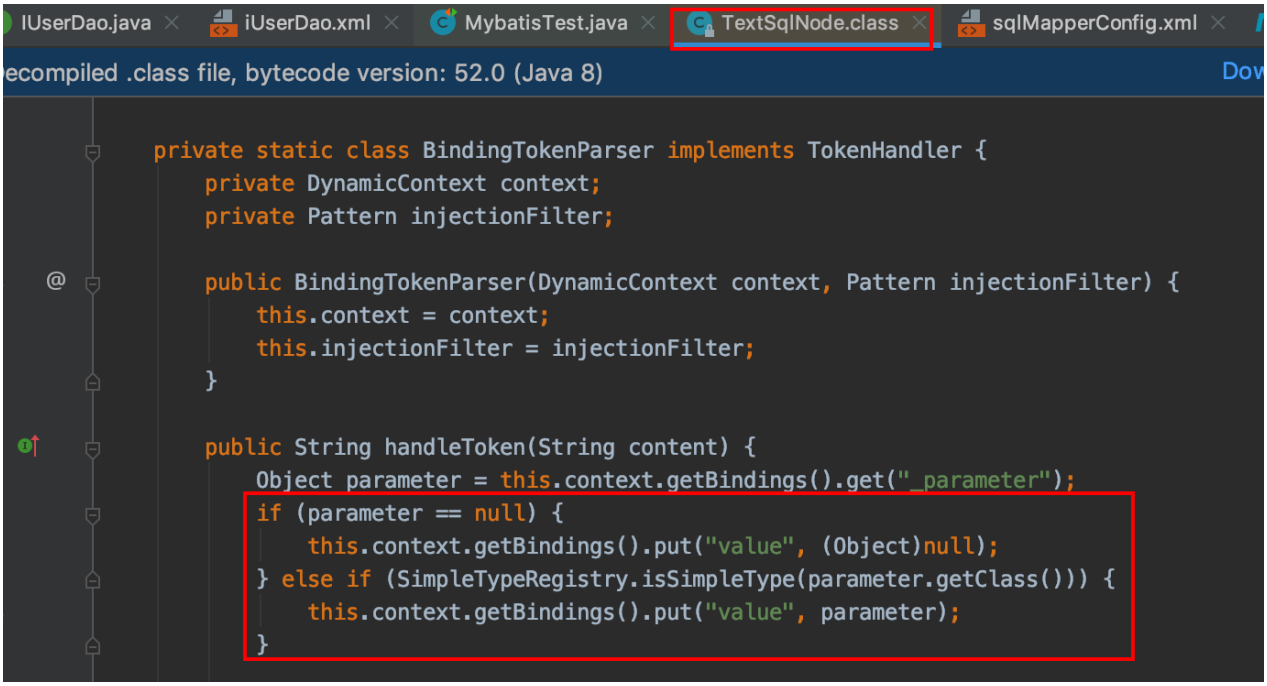
- 3.6.2 在用户的映射配置文件中配置

```
    <select id="findByName2" parameterType="string"
resultType="com.dgut.domain.User">
        select * from user where username like '%${value}%'
    </select>
```

- 3.6.3 在测试类添加测试

```
  @Test public void testFindByName2() throws Exception {
        List<User> users = userDao.findByName2("王");
        System.out.println(users);
    }
```

- 3.6.4 模糊查询的${value}源码分析




> 注意
>
> **#{}**表示一个占位符号
>
> 通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换，#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

> **${}表示拼接sql串**
>
> 通过可以将$parameterType$传入的内容拼接在$sql$中且不进行$jdbc$类型转换，{}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

```
  <select id="findUserByName" parameterType="java.lang.String"
resultType="com.dgut.domain.User">
        <!-- 拼接 MySQL,引起 SQL 注入 -->
        SELECT * FROM user WHERE username LIKE '%${value}%'
    </select>
java:
 public List<User>  findUserByName(String username);

test:
    List<User> users = mapper.findUserByName("' or '1'='1");
```

### 3.7 查询使用聚合函数

- 3.7.1 在持久层接口中添加findTotal方法

```
    /**
     * 查询总记录条数
     */
    int findTotal();
```

- 3.7.2 在用户的映射配置文件中配置

```
<!-- 查询总记录条数 -->
   <select id="findTotal" resultType="int">
     select count(*) from user;
   </select>
```

- 3.7.3 在测试类添加测试

```
  @Test
    public void testFindTotal() throws Exception {
        int res = userDao.findTotal();
        System.out.println(res);
    }
```

### 3.8 Mybatis与JDBC编程的比较

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决： 在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

2. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决： 将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

3. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数对应。

解决： Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

4. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决： Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

# 04.Mybatis的参数深入

### 4.1 parameterType配置参数

- 4.1.1使用说明

> 我们在上一章节中已经介绍了SQL语句传参，使用标签的parameterType属性来设定。该属性的取值可以是基本类型，引用类型（例如:String类型），还可以是实体类类型（POJO类）。同时也可以使用实体类的包装类，本章节将介绍如何使用实体类的包装类作为参数传递。

- 4.1.2 注意事项

> 基本类型和String我们可以直接写类型名称，也可以使用包名.类名的方式，例如：java.lang.String。
>
> 实体类类型，目前我们只能使用全限定类名。
>
> 究其原因，是mybaits在加载时已经把常用的数据类型注册了别名，从而我们在使用时可以不写包名，而我们的是实体类并没有注册别名，所以必须写全限定类名。

这是一些为常见的 Java 类型内建的相应的类型别名。它们都是不区分大小写的，注意对基本类型名称重复采取的特殊命名风格。

| 别名 | 映射的类型 |
| --- | --- |
| _byte | byte |
| _long | long |
| _short | short |
| _int | int |
| _integer | int |
| _double | double |
| _float | float |
| _boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

### 4.2 传递pojo包装对象

> 开发中通过pojo传递查询条件，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。
>
> Pojo类中包含pojo。
>
> 需求：根据用户名查询用户信息，查询条件放到QueryVo的属性中。

- 4.2.1 编写**QueryVo**

```
public class QueryVO {
    private String username;
    private String sex;

    public String getSex() {
        return sex;
    }

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

```
    }

    public String getUsername() {
        return username;
    }

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

- 4.2.2 编写持久层接口

```
<select id="findByVo" parameterType="com.dgut.domain.QueryVO"
resultType="com.dgut.domain.User">
        select * from user where username like #{username} and sex = #{sex};
    </select>
```

- 4.2.3 编写测试类

```
@Test
    public void testfindByVo(){
        QueryVO vo = new QueryVO();
        vo.setUsername("%王%");
        vo.setSex("男");
        List user = userDao.findByVo(vo);
        System.out.println(user);
    }
```

> 能不能更简单点?
>
> 提示： #{arg1} 或#{param1}.... @Param的使用

# 05.Mybatis的输出结果封装

> resultType属性可以指定结果集的类型，它支持基本类型和实体类类型。
>
> 我们在前面的CRUD案例中已经对此属性进行过应用了。
>
> 需要注意的是，它和parameterType一样，如果注册过类型别名的，可以直接使用别名。没有注册过的必须使用全限定类名。例如：我们的实体类此时必须是全限定类名
>
> 同时，当是实体类名称是，还有一个要求，实体类中的属性名称必须和查询语句中的***列名保持一致***，否则无法实现封装。

### 5.1 resultType配置结果类型

这里考虑实体类属性和数据库表的列名已经不一致的情况

- 5.1.1 创建实体类User2

```
public class User2 {
    private Integer userId;
    private String userName;
    private Date userBirthday;
    private String userSex;
    private String userAddress;

    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public Date getUserBirthday() {
        return userBirthday;
    }

    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 "User2{" +
                "userId=" + userId +
                ", userName='" + userName + '\'' +
                ", userBirthday=" + userBirthday +
                ", userSex='" + userSex + '\'' +
                ", userAddress='" + userAddress + '\'' +
                '}';
    }
}
```

- 5.1.2 编写dao接口

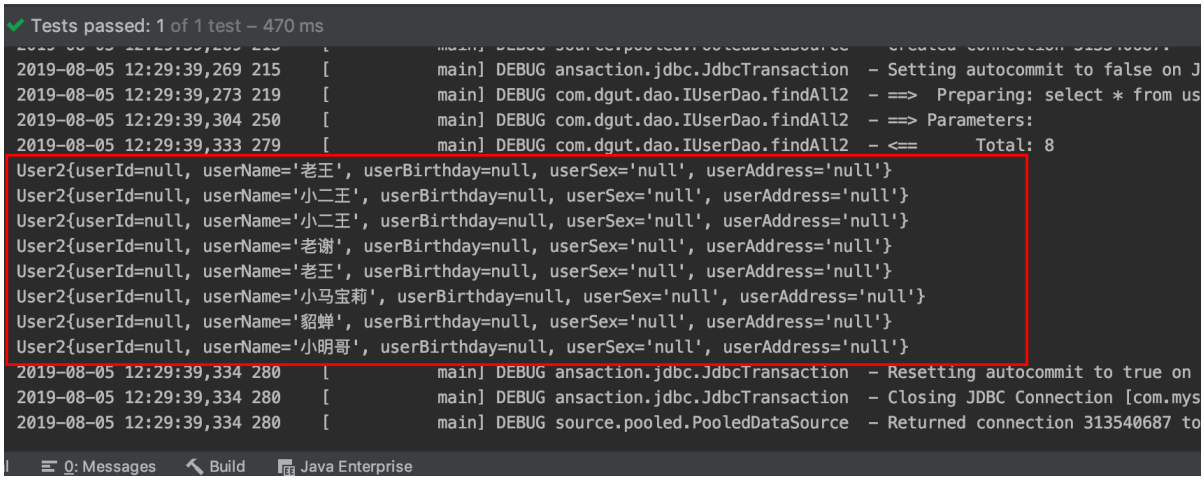
```
/**
 * 查找用户
 */
public List<User2> findAll2();
```

- 5.1.3 映射配置

```
 <select id="findAll2" resultType="com.dgut.domain.User2">
        select * from user
  </select>
```

- 5.1.4 测试查询结果

```
@Test
    public void testFindAll2() throws IOException {
        //5.使用代理对象执行查询所有方法
        List<User2> users = userDao.findAll2();
        for (User2 user : users) {
            System.out.println(user);
        }
    }
```



- 5.1.5 修改mapper文件

```
<select id="findAll2" resultType="com.dgut.domain.User2">
       select id as userId,username as userName,birthday as userBirthday,
sex as userSex,address as userAddress from user
 </select>
```

> 思考：
>
> 如果我们的查询很多，都使用别名的话写起来岂不是很麻烦，有没有别的解决办法呢?

### 5.2 resultMap结果类型

> resultMap标签可以建立查询的列名和实体类的属性名称不一致时建立对应关系。从而实现封装。
>
> 在select标签中使用resultMap属性指定引用即可。同时resultMap可以实现将查询结果映射为复杂类型的pojo，比如在查询结果映射对象中包括pojo和list实现一对一查询和一对多查询。

- 5.2.1 定义**resultMap**

```
<!--
       建立User实体和数据库表的对应关系
        type属性：指定实体类的全限定类名
          id属性：给定一个唯一标识，是给查询select标签引用用的。
       -->
   <resultMap id="user2Map" type="com.dgut.domain.User2">
       <!-- id标签：用于指定主键字段
        result标签：用于指定非主键字段
        column属性：用于指定数据库列名
         property属性：用于指定实体类属性名称
         -->
       <!--主键映射-->
       <id column="id" property="userId"/>
       <!--普通属性映射-->
       <result column="username" property="userName"/>
       <result column="sex" property="userSex"/>
       <result column="address" property="userAddress"/>
       <result column="birthday" property="userBirthday"/>
   </resultMap>
```

- 5.2.2 映射配置

```
<select id="findAll2" resultMap="user2Map">
       <!-- select id as userId,username as userName,birthday as
userBirthday, sex as userSex,address as userAddress from user-->
        select * from user
  </select>
```

- 5.2.3 测试结果

```
@Test
  public void testFindAll2() throws IOException {
      //5.使用代理对象执行查询所有方法
      List<User2> users = userDao.findAll2();
      for (User2 user : users) {
          System.out.println(user);
      }
  }
```

# 06.SqlMapConfig.xml配置文件

### 6.1 配置内容

- **SqlMapConfig.xml**中配置的内容和顺序

```
-properties（属性）
  --property
-settings（全局配置参数）
  --setting
-typeAliases（类型别名）
  --typeAliase
  --package
-typeHandlers（类型处理器）
-objectFactory（对象工厂）
-plugins（插件）
-environments（环境集合属性对象）
  --environment（环境子属性对象）
    ---transactionManager（事务管理）
    ---dataSource（数据源）
-mappers（映射器）
  --mapper --package
```

### 6.2 properties（属性）

> 在使用properties标签配置时，我们可以采用两种方式指定属性配置。

- 6.2.1 内部引用（了解）

```
<?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>
        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useUnicode=true&amp;characterEncoding=UTF-8&amp;serverTimezone=GMT%2B8"/>
        <property name="username" value="root"/>
        <property name="password" value="xieman123"/>
    </properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="com/dgut/dao/iUserDao.xml"/>
    </mappers>

</configuration>
```

- 6.2.2 外部引用
  - 新建jdbc.properties文件

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/mybatis?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=GMT%2B8
jdbc.username = root
jdbc.password = xieman123
```

  - 修改配置文件

```
<?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">

<configuration>
    <properties resource="jdbc.properties">

    </properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <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 resource="com/dgut/dao/iUserDao.xml"/>
    </mappers>
</configuration>
```

### 6.3 typeAliases（类型别名）

> 在SqlMapConfig.xml中配置：

```
<typeAliases>
      <!-- <typeAlias type="com.dgut.domain.User" alias="user"/>-->
       <package name="com.dgut.domain"/>
   </typeAliases>
```

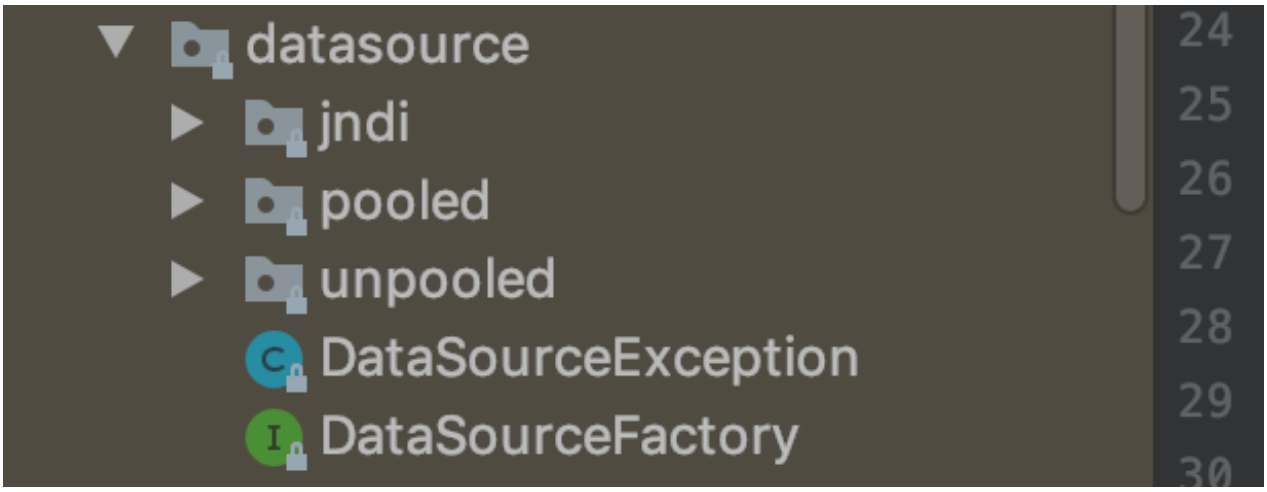
### 6.4 mappers（映射器）

```
 <mappers>
       <!--使用相对于类路径的资源 -->
      <!-- <mapper resource="com/dgut/dao/iUserDao.xml"/>-->
   <!-- 注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。 -
->
       <package name="com.dgut.dao"/>
   </mappers>
```

# 07.Mybatis连接池与事务深入

### 7.1 Mybatis的连接池技术

在Mybatis中我们将它的数据源dataSource分为以下几类：




可以看出Mybatis将它自己的数据源分为三类：

1. UNPOOLED 不使用连接池的数据源： 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。
2. POOLED 使用连接池的数据源： 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
3. JNDI 使用JNDI实现的数据源： 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。

> 在这三种数据源中，我们一般采用的是POOLED数据源（很多时候我们所说的数据源就是为了更好的管理数据库连接，也就是我们所说的连接池技术）。

**7.2 Mybatis的事务控制**

Mybatis中事务的提交方式，本质上就是调用JDBC的setAutoCommit()来实现事务控制。

**设置Mybatis自动提交事务的设置**

```
//3.使用SqlSessionFactory生产SqlSession对象
session = sqlSessionFactory.openSession(true);
```

# 08.动态sql

Mybatis的映射文件中，前面我们的SQL都是比较简单的，有些时候业务逻辑复杂时，我们的SQL是动态变化的，此时在前面的学习中我们的SQL就不能满足要求了。

**8.1 <if> 标签**

我们根据实体类的不同取值，使用不同的SQL语句来进行查询。比如在

id如果不为空时可以根据id查询，如果username不同空时还要加入sex作为条件。

这种情况在我们的多条件组合查询中经常会碰到。

- 编写dao接口

```
/**
  * 多条件查询
  */
 public List<User> findByUser(User user);
```

- 编写映射文件

```
<select id="findByUser" resultType="com.dgut.domain.User"
parameterType="com.dgut.domain.User">
        select * from user where 1 = 1
        <if test="username != null">
            and username like #{username}
        </if>
        <if test="sex != null">
            and sex = #{sex}
        </if>
    </select>
```

> 注意：标签的test属性中写的是对象的属性名，如果是包装类的对象要使用OGNL表达式的写法。
>
> 另外要注意where 1=1 的作用~！

- 编写测试代码

```
@Test
    public void findByUser() throws IOException {
        User u = new User();
        u.setUsername("%王%");
        u.setSex("男");
        //5.使用代理对象执行查询所有方法
        List<User> users = userDao.findByUser(u);
        for (User user : users) {
            System.out.println(user);
        }
    }
```

### 8.2 <where>标签

为了简化上面where 1=1的条件拼装，我们可以采用<where>标签来简化开发。

```
<select id="findByUser" resultType="com.dgut.domain.User"
parameterType="com.dgut.domain.User">
        select * from user
        <where>
        <if test="username != null">
            and username like #{username}
        </if>
        <if test="sex != null">
            and sex = #{sex}
        </if>
        </where>
    </select>
```

### 8.3<foreach>标签

- 需求

> 传入多个id查询用户信息，用下边两个sql实现：
>
> SELECT * FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)
>
> SELECT * FROM USERS WHERE username LIKE '%张%' AND id IN (10,89,16)
>
> 这样我们在进行范围查询时，就要将一个集合中的值，作为参数动态添加进来。 这样我们将如何进行参数的传递?

- 8.3.1 在**QueryVo**中加入一个**List**集合用于封装参数

```
public class QueryVO {
    private String username;
    private String sex;
    List<Integer> ids;

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

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

    public String getSex() {
        return sex;
    }

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

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

```
    }

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

- 8.3.2 编写dao

```
/**
     * 多条件查询
     */
    public List<User> findByQvo(QueryVO vo);
```

- 8.3.3 编写配置文件

```
<select id="findByQvo" resultType="com.dgut.domain.User"
parameterType="com.dgut.domain.QueryVO">
        select * from user
        <where>
            <if test="username != null">
                and username like #{username}
            </if>
            <if test="sex != null">
                and sex = #{sex}
            </if>
            <if test="ids != null and ids.size() > 0">
                <foreach collection="ids" open=" and id in (" close=")"
item="abc" separator=",">
                    #{abc}
                </foreach>
            </if>
        </where>
    </select>
```

> 标签用于遍历集合，它的属性：
>
> collection:代表要遍历的集合元素，注意编写时不要写#{}
>
> open:代表语句的开始部分
>
> close:代表结束部分
>
> item:代表遍历集合的每个元素，生成的变量名
>
> sperator:代表分隔符

- 8.3.4 编写测试类

```
@Test
    public void findByQVC() throws IOException {
        QueryVO u = new QueryVO();
        u.setUsername("%王%");
        List list = new ArrayList();
        list.add(41);
        list.add(43);
        u.setIds(list);
        //5.使用代理对象执行查询所有方法
        List<User> users = userDao.findByQvo(u);
        for (User user : users) {
            System.out.println(user);
        }
    }
```

> 自学**choose**、**when**、**otherwise**、**trim**

# 09.Mybatis中简化编写的SQL片段

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

## 9.1定义代码片段

```
<!-- 抽取重复的语句代码片段 -->
<sql id="defaultSql"> select * from user </sql>
```

## 9.2 引用代码片段

```
<!-- 配置查询所有操作 -->
<select id="findAll" resultType="user">
  <include refid="defaultSql"></include>
  </select>
  <!-- 根据id查询 -->
  <select id="findById" resultType="User" parameterType="int">
  <include refid="defaultSql"></include>
      where id = #{id}
</select>
```

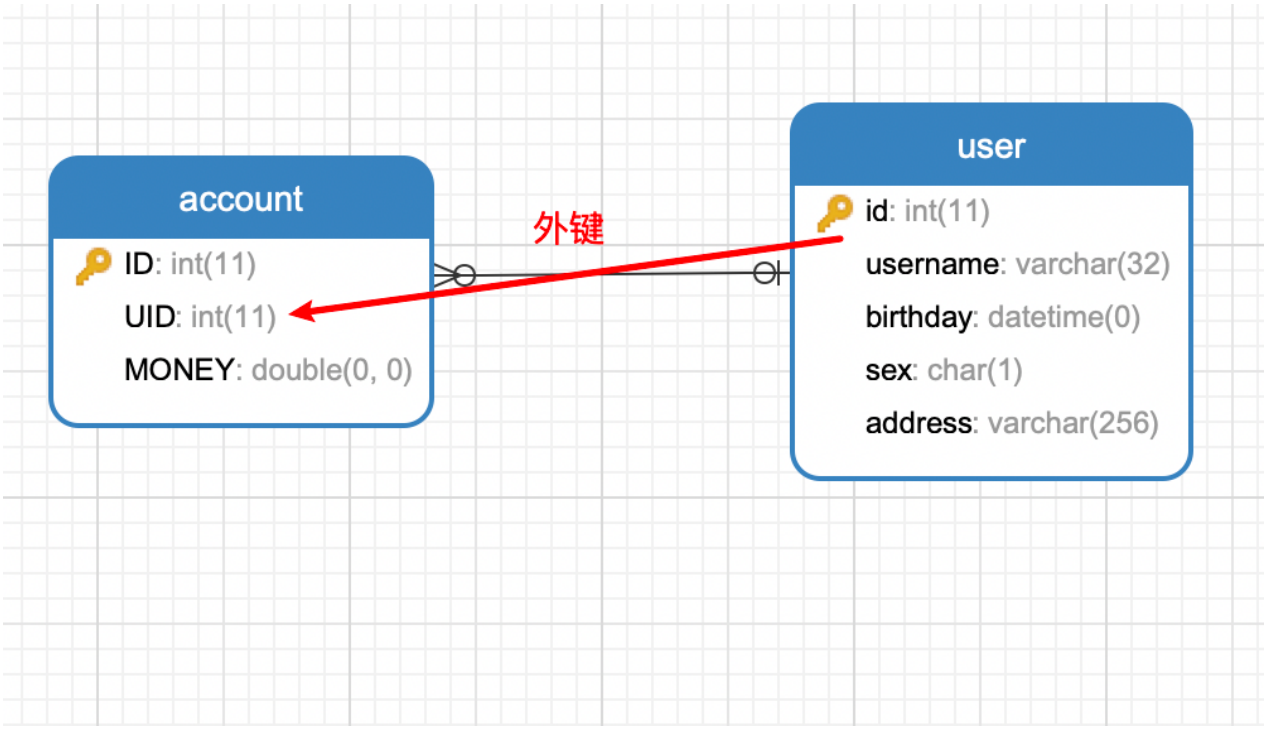
# 10.Mybatis 多表查询

- mybatis中的多表查询,表之间的关系有几种： 一对多 多对一 一对一 多对多
- 举例：
  - 用户和订单就是一对多 订单和用户就是多对一 一个用户可以下多个订单 一个订单属于同一个用户
  - 人和身份证号就是一对一 一个人只能有一个身份证号 一个身份证号只能属于一个人

    - 老师和学生之间就是多对多 一个学生可以被多个老师教过 一个老师可以教多个学生
- mybatis中的多表查询： 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（多个账户也可以属于同一个用户） 步骤：
- 1、建立两张表：用户表，账户表 让用户表和账户表之间具备一对多的关系：需要使用外键在账户表中添加
- 2、建立两个实体类：用户实体类和账户实体类 让用户和账户的实体类能体现出来一对多的关系
- 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：
    1. 当我们查询账户时，可以同时得到账户的所属用户信息(一对一)
    2. 当我们查询用户时，可以同时得到用户下所包含的账户信息（一对多）

**9.1 一对一查询(多对一)**

例子：用户为User 表，账户为Account 表。一个用户（User）可以有多个账户（Account）。具体关系如下：




- 9.1.1 新建Account类

```
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;
    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 +
                ", user=" + user +
                '}';
    }
}
```

- 9.1.2 新建接口Dao类IAccountDao

```
public interface IAccountDao {
    /**
     * 查询所有账户，同时获取账户的所属用户名称以及它的地址信息
     */
    List<Account> findAll();
}
```

- 9.1.3 新建**AccountDao.xml**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.dgut.dao.IAccountDao">
    <resultMap id="accountMap" type="com.dgut.domain.Account">
        <id property="id" column="id"></id>
        <result property="uid" column="uid"></result>
        <result property="money" column="money"></result>
        <association property="user" javaType="com.dgut.domain.User">
            <id column="id" property="id"/>
            <result property="username" column="username"/>
            <result property="birthday" column="birthday"/>
            <result property="sex" column="sex"/>
            <result property="address" column="address"/>
        </association>
    </resultMap>
 <select id="findAll" resultMap="accountMap">
        select a.*,u.username,u.birthday,u.sex,u.address from `user` u
,account a WHERE a.uid = u.id
 </select>

</mapper>
```

- 9.1.4 **AccountTest**类中加入测试方法

```
 @Test
    public void testFindAllAccount(){
        List list = accountDao.findAll();
        System.out.println(list);
    }
```

### 9.2 一对多查询

> 需求： 查询所有用户信息及用户关联的账户信息。
>
> 分析： 用户信息和他的账户信息为一对多关系，并且查询过程中如果用户没有账户信息，此时也要将用户信息查询出来，我们想到了左外连接查询比较合适。

- 9.2.1 编写SQL语句

```
SELECT user.*,account.ID as uid,account.MONEY from user  LEFT JOIN account
ON `user`.id = account.UID
```

- 9.2.2 修改User实体类，加入accounts列表

```
 private List<Account> accounts;

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

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

- 9.2.3 修改IUserDao接口，新增findALL方法

```
/**
* 查询所有用户，同时获取出每个用户下的所有账户信息
*/
public List<User> findALL();
```

- 9.2.4 修改IUserDao.xml

```
<resultMap id="userAccountMap" type="com.dgut.domain.User">
        <id column="id" property="id"/>
        <result property="username" column="username"/>
        <result property="address" column="address"/>
        <result property="sex" column="sex"/>
        <result property="birthday" column="birthday"/>
       <!-- collection是用于建立一对多中集合属性的对应关系 ofType用于指定集合元素的
数据类型 -->
        <collection property="accounts" ofType="com.dgut.domain.Account">
            <id property="id" column="uid"/>
            <result property="uid" column="uid"/>
            <result property="money" column="money"/>
        </collection>
    </resultMap>

    <select id="findALL" resultMap="userAccountMap">
        SELECT user.*,account.ID as uid,account.MONEY from user  LEFT JOIN
account ON user.id = account.UID
    </select>
```

> collection 部分定义了用户关联的账户信息。表示关联查询结果集 property="accList"： 关联查询的结果集存储在User对象的上哪个属性。 ofType="account"： 指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。

- 9.2.5 测试

```
@Test
    public void testFindALL(){
        List<User> users =  userDao.findALL();
        for (User user : users){
            System.out.println(user);
            System.out.println(user.getAccounts()+"\n");
        }
    }
```

### 9.3 多对多

> 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，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来多对多的关系 各自包含对方一个集合引用 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： 当我们查询用户时，可以同时得到用户所包含的角色信息 当我们查询角色时，可以同时得到角色的所赋予的用户信息

- 9.3.1 需求

  实现查询所有角色并且加载它所分配的用户信息。

  分析：

  查询角色我们需要用到Role表，但角色分配的用户的信息我们并不能直接找到用户信息，而是要通过中间表(USER_ROLE表)才能关联到用户信息。

| ID | ROLE_NAME | ROLE_DESC |
|---|---|---|
| 1 | 学生 | 普通学生 |
| 2 | 班长 | 班里的老大 |
| 3 | 学习委员 | 班里的学霸 |

| id | username | birthday | sex | address |
|---|---|---|---|---|
| 41 | 老王 | 2018-02-27 17:47:08 | 男 | 东莞南城 |
| 42 | 小二王 | 2018-03-02 15:09:37 | | 东莞松山湖 |
| 43 | 小二王 | 2018-03-04 11:34:34 | 女 | 东莞虎门 |
| 45 | 老谢 | 2018-03-04 12:04:06 | 男 | 东莞东城 |
| 46 | 老王 | 2018-03-07 17:37:26 | 男 | 东莞清溪 |
| 48 | 小马宝莉 | 2018-03-08 11:44:00 | 女 | 东莞塘厦 |
| 50 | 貂蝉 | 2019-08-04 21:26:24 | ? | 深圳华强北 |
| 51 | 小明哥 | 2019-08-05 10:29:03 | 男 | 东莞虎门 |
| 53 | 小明哥 | 2019-08-05 13:51:24 | 男 | 东莞虎门 |
| 54 | 小明哥 | 2019-08-05 13:51:35 | 男 | 东莞虎门 |

| UID | RID |
|---|---|
| 41 | 1 |
| 45 | 1 |
| 41 | 2 |

- 9.3.2 编写SQL语句

```
 SELECT role.id as rid,role.ROLE_NAME,role.ROLE_DESC,user.*
FROM role LEFT JOIN user_role ON role.ID = user_role.RID LEFT JOIN `user`
ON user_role.UID = `user`.id
```

- 9.3.3 编写Role实体

```
public class Role {
    private Integer id;
    private String roleName;
    private String roleDes;
    private List<User> users;

    public Integer getId() {
        return id;
    }

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

    public String getRoleName() {
        return roleName;
    }

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

    public String getRoleDes() {
        return roleDes;
    }

    public void setRoleDes(String roleDes) {
        this.roleDes = roleDes;
    }

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

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

    @Override
    public String toString() {
```

```
        return "Role{" +
                "id=" + id +
                ", roleName='" + roleName + '\'' +
                ", roleDes='" + roleDes + '\''
                +
                '}';
    }
}
```

- 9.3.4 编写IRoleDao接口

```
  /**
     * 查询所有角色
     * @return
     */
public interface IRoleDao {
    public List<Role> findAllRole();
}
```

- 9.3.5 编写配置文件 IRoleDao

```
<?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.dgut.dao.IRoleDao">

    <resultMap id="roleMap" type="com.dgut.domain.Role">
        <id property="id" column="rid"/>
        <result property="roleName" column="role_name"/>
        <result property="roleDes" column="role_desc"/>
        <collection property="users" ofType="com.dgut.domain.User">
            <id property="id" column="id"/>
            <result property="username" column="username"/>
            <result property="birthday" column="birthday"/>
            <result property="sex" column="sex"/>
            <result property="address" column="address"/>
        </collection>
    </resultMap>

    <select id="findAllRole" resultMap="roleMap">
       SELECT role.id as rid,role.ROLE_NAME,role.ROLE_DESC,user.*
FROM role LEFT JOIN user_role ON role.ID = user_role.RID LEFT JOIN `user`
ON user_role.UID = `user`.id
    </select>
</mapper>
```

- 9.3.6 编写测试文件

```
@Test
    public void testAllRole(){
        List<Role> roles = roleDao.findAllRole();
        for (Role r : roles){
            System.out.println(r);
            System.out.println(r.getUsers());
        }
    }
```

- 9.3.7 测试结果

```
Role{id=1, roleName='学生', roleDes='普通学生'}
[User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='东莞南城'}, User{id=45, username='老谢', birthday=Sun
Role{id=2, roleName='班长', roleDes='班里的老大'}
[User{id=41, username='老王', birthday=Tue Feb 27 17:47:08 CST 2018, sex='男', address='东莞南城'}]
Role{id=3, roleName='学习委员', roleDes='班里的学霸'}
[]
```

- 9.3.8 练习

> 实现查询所有用户信息并关联查询出每个用户的角色列表。

```
SELECT user.*,role.ID as rid,role.ROLE_NAME,role.ROLE_DESC FROM `user`
LEFT JOIN user_role on `user`.id = user_role.UID LEFT JOIN role ON role.id
 = user_role.RID
```