

1. 【入门篇】

1.1 初识 MybatisPlus

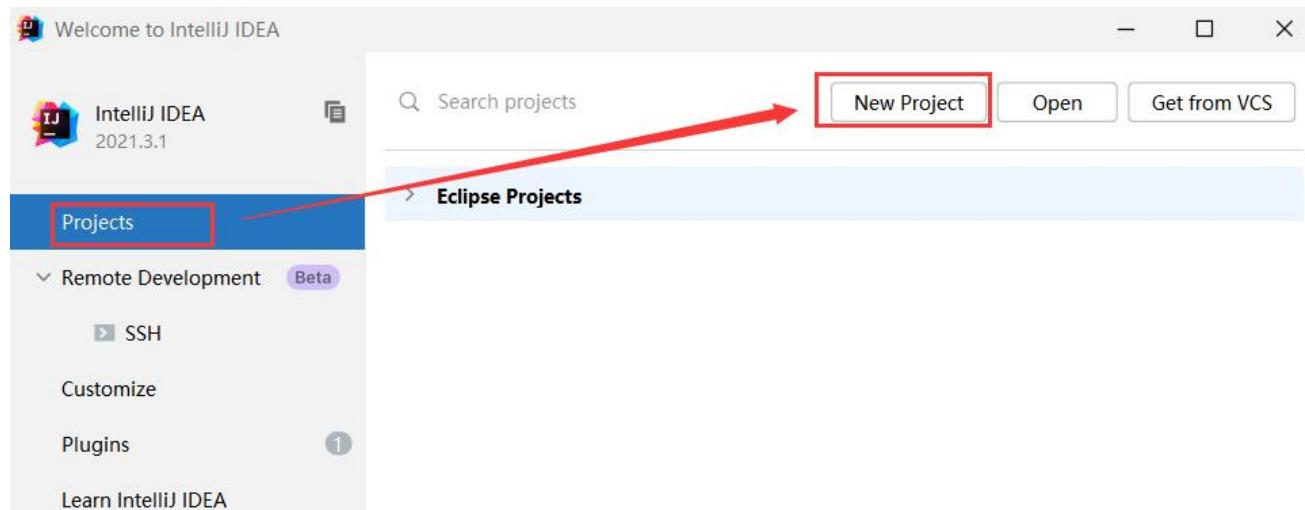
1.1.1 Mybatis 框架回顾

MybatisPlus，从名称上来看，我们就发现，他和 Mybatis 长得很像，其实 MybatisPlus 就是 Mybatis 的孪生兄弟，在学习 MybatisPlus 之前，我们先来回顾一下 Mybatis 框架的搭建流程。

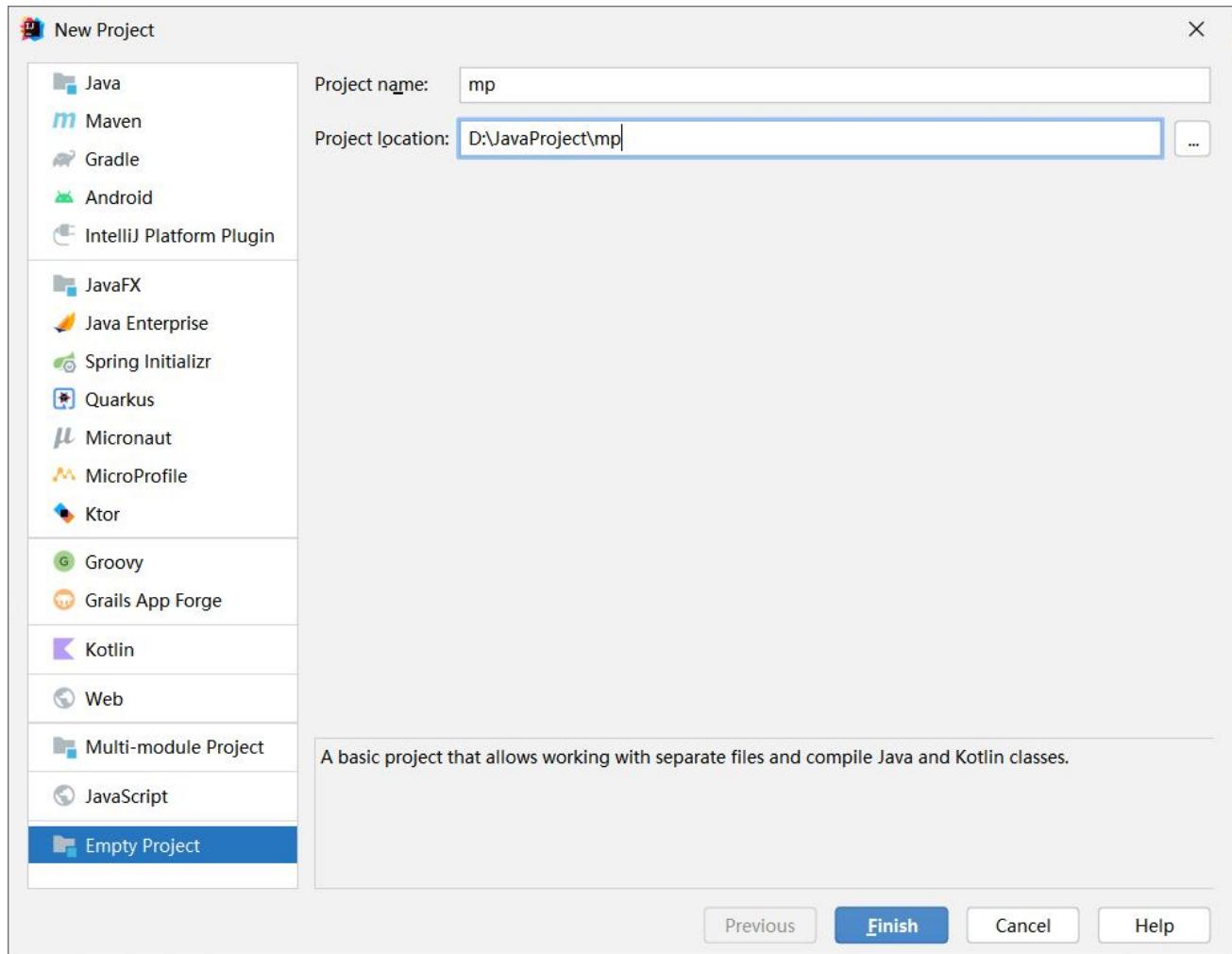
什么是 Mybatis 框架呢，他是一个持久层框架，目的是简化持久层的开发。在这里我们就使用 springboot 整合 Mybatis，实现 Mybatis 框架的搭建。

【1】工程结构创建

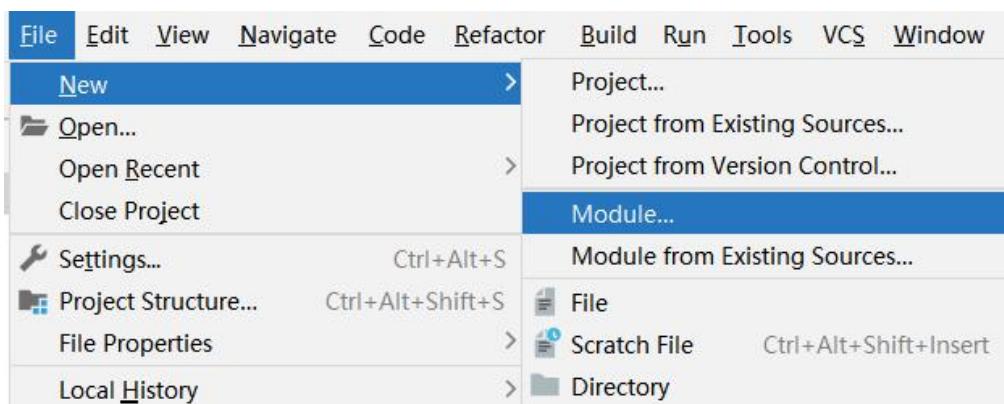
我们首先创建一个空的工程

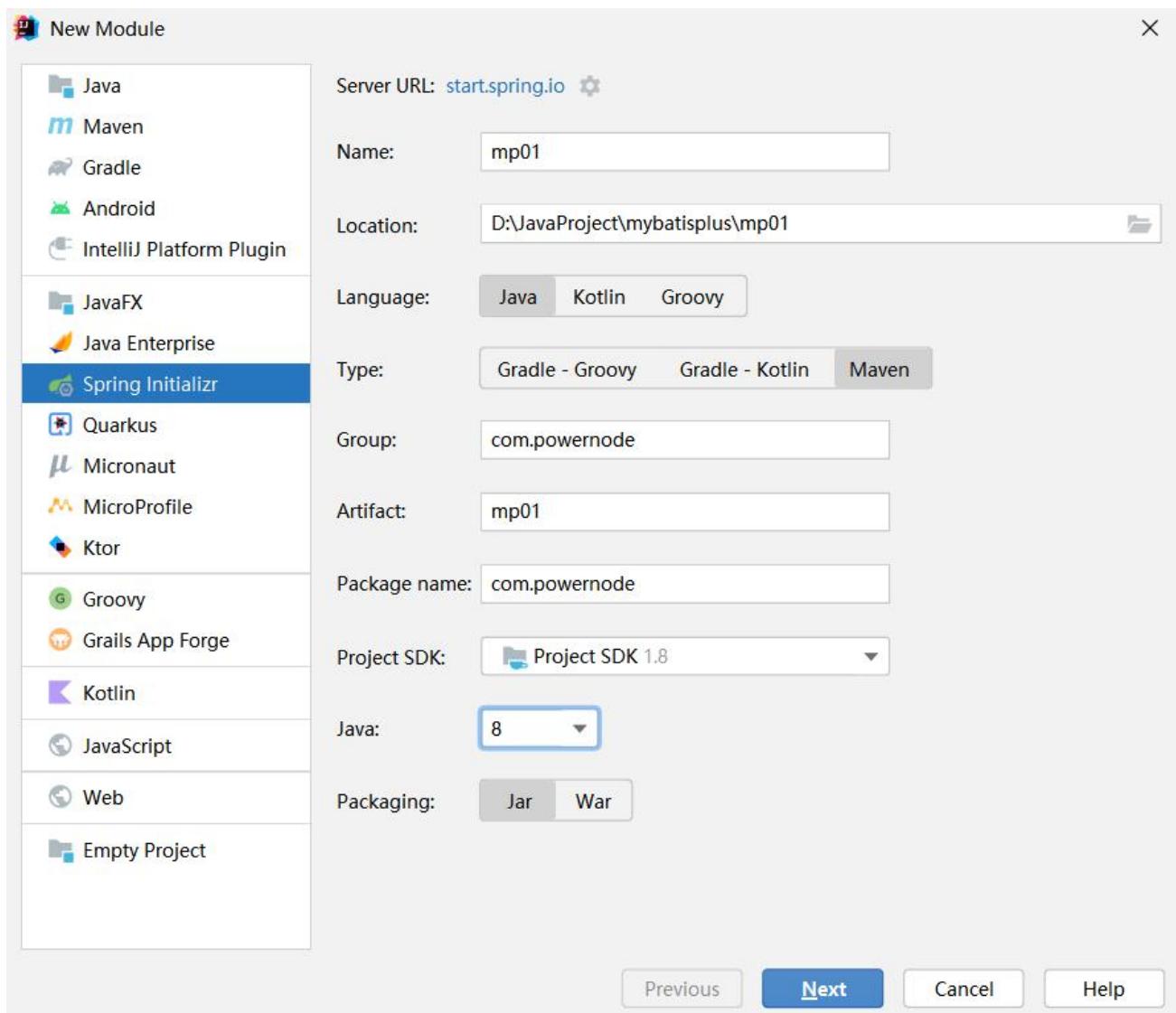


工程名称是 mp

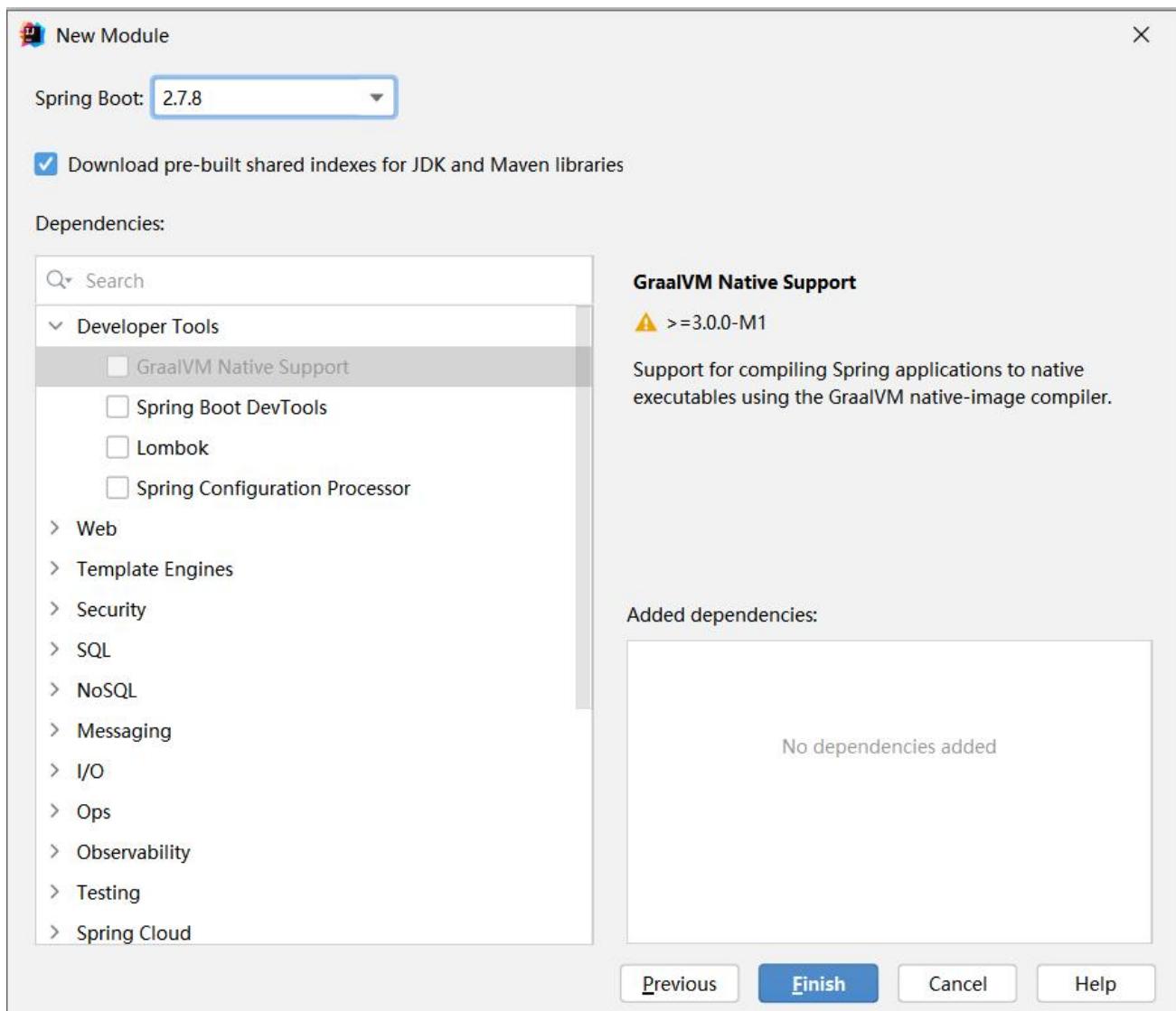


创建 springboot 模块





这里我们选择 springboot2.7.8 的版本，并不勾选依赖，随后通过 pom.xml 手动添加

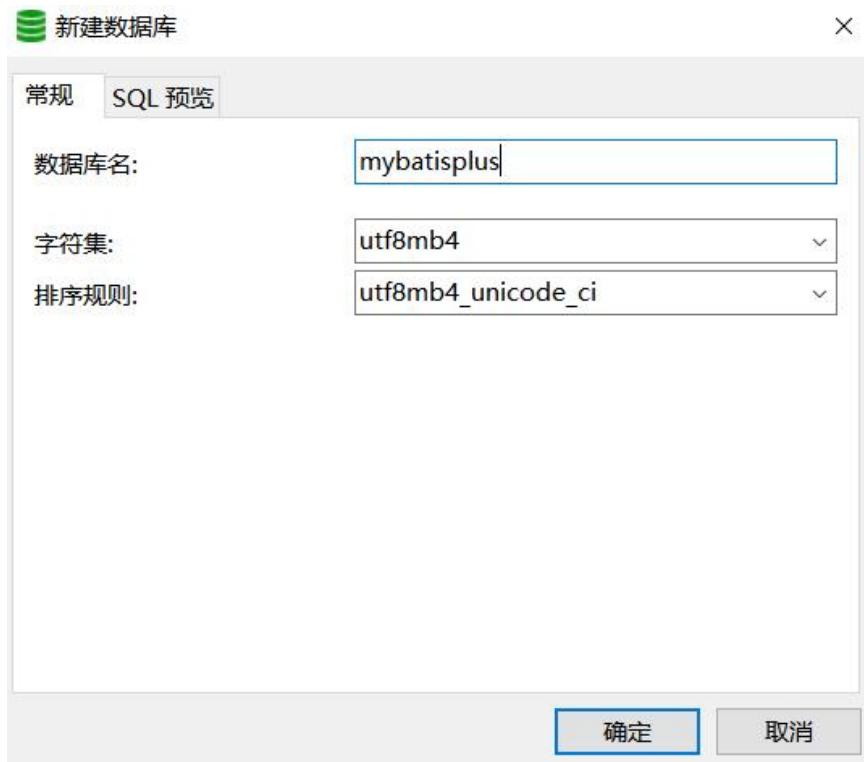


创建后的结构如下



【2】创建数据库，表结构，导入表数据

创建数据库 mybatisplus



建表语句

```
DROP TABLE IF EXISTS user;
CREATE TABLE user(
    id BIGINT(20) NOT NULL COMMENT '主键 ID',
    name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
    age INT(11) NULL DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
    PRIMARY KEY (id)
);
```

插入数据

```
DELETE FROM user;
INSERT INTO user (id, name, age, email) VALUES
(1, 'Jone', 18, 'test1@baomidou.com'),
(2, 'Jack', 20, 'test2@baomidou.com'),
(3, 'Tom', 28, 'test3@baomidou.com'),
(4, 'Sandy', 21, 'test4@baomidou.com'),
(5, 'Billie', 24, 'test5@baomidou.com');
```

【3】引入相关依赖

```
<!--mysql 驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>

<!--mybatis 起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.3</version>
</dependency>

<!--web 环境起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!--lombok 插件依赖-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

【4】创建 domain 包，编写实体类，实体类的数据类型和表字段类型一致



Lombok 可以极大的省略类的代码量，使代码更加的简洁。

Lombok 相关常用注解

`@Data` 注解在类上，提供类的 get、set、equals、hashCode、toString 等方法

`@AllArgsConstructor` 注解在类上，提供满参构造

`@NoArgsConstructor` 注解在类上，提供空参构造

```
package com.powernode.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    //主键
    private Long id;
    //姓名
    private String name;
    //年龄
    private Integer age;
    //邮箱
    private String email;
}
```

【5】 创建 mapper 包，编写 Mapper 接口

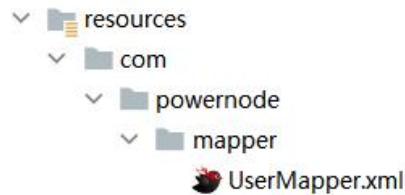


@Mapper 作用在接口上，扫描到该注解后，会根据接口创建该接口的实现类对象

```
import com.powernode.domain.User;
import org.apache.ibatis.annotations.Mapper;
import java.util.List;

@Mapper
public interface UserMapper {
    public List<User> selectList();
}
```

【6】 编写映射文件

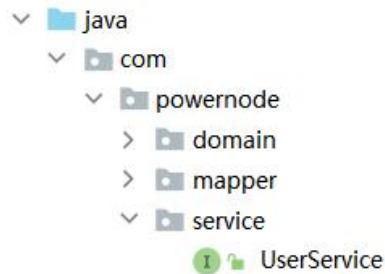


```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.powernode.mapper.UserMapper">

    <select id="selectAll" resultType="com.powernode.domain.User">
        select * from user
    </select>

</mapper>
```

【7】 创建 service 包，编写 Service



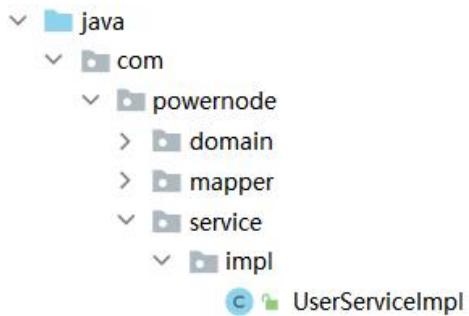
```
package com.powernode.service;

import com.powernode.domain.User;

import java.util.List;

public interface UserService {
    List<User> selectAll();
}
```

【8】编写ServiceImpl



```
package com.powernode.service.impl;

import com.powernode.domain.User;
import com.powernode.mapper.UserMapper;
import com.powernode.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

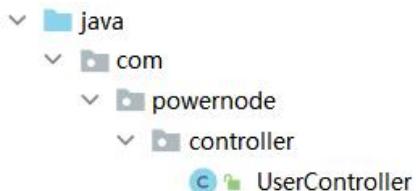
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    @Override
```

```
public List<User> selectAll() {  
    return userMapper.selectAll();  
}  
}
```

【9】 编写 Controller



```
package com.powernode.controller;  
  
import com.powernode.domain.User;  
import com.powernode.service.UserService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class UserController {  
  
    @Autowired  
    private UserService userService;  
  
    @RequestMapping("/selectAll")  
    public List<User> selectAll(){  
        List<User> users = userService.selectAll();  
        return users;  
    }  
}
```

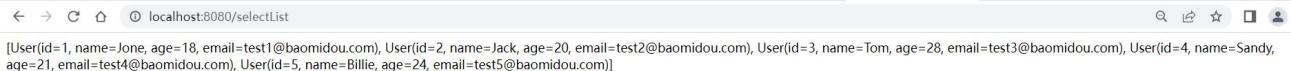
【10】 将配置文件改为 yml 格式，编写配置文件



```
spring:
  datasource:
    password: root
    username: root
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false
```

【11】 测试结果

请求 controller 接口，发现响应了数据，至此 Mybatis 环境开发完毕



1.1.2 Mybatis 框架的开发效率问题分析

我们来思考一下，Mybatis 框架的开发效率怎么样？

开发效率也就是我们使用这款框架开发的速度快不快，是否简单好用易上手。从这个角度思考，每当我们需要编写一个 SQL 需求的时候，我们需要做几步

【1】Mapper 接口提供一个抽象方法

【2】Mapper 接口对应的映射配置文件提供对应的标签和 SQL 语句

【3】在 Service 中依赖 Mapper 实例对象

【4】调用 Mapper 实例中的方法

【5】在 Controller 中依赖 Service 实例对象

【6】调用 Service 实例中的方法

通过上面的发现，对于一个 SQL 需求，无论是单表还是多表，我们是需要完成如上几步，才能实现 SQL 需求的开发

但是在开发中，有一些操作是通用逻辑，这些通用逻辑是可以被简化的，例如：

【1】对于 dao，是否可以由框架帮我们提供好单表的 Mapper 抽象方法，和对应的 SQL 实现，不需要程序员去实现这些

【2】对于 service，使用可以有框架直接帮我们提供好一些 service 的抽象方法，和对应的实现，不需要程序员去实现这些

【3】一些其他的企业开发中所需要的操作

分析到这里我们发现，其实核心框架并没有发生变化，依然还是 Mybatis，只不过我们希望对于 Mybatis 进行一些封装和进化，让它更加的好用，更加的易用。

MybatisPlus 它来了，他是 Mybatis 的一款增强工具。

1.1.3 MybatisPlus 的介绍

MyBatis-Plus ([opens new window](#)) (简称 MP) 是一个 MyBatis ([opens new window](#)) 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

愿景

我们的愿景是成为 MyBatis 最好的搭档，就像 魂斗罗 中的 1P、2P，基友搭配，效率翻倍。



TO BE THE BEST PARTNER OF MYBATIS

1.1.4 MybatisPlus 的特性讲解

特性

无侵入: 只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑

损耗小: 启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作

强大的 CRUD 操作: 内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现

单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求

支持 Lambda 形式调用: 通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错

支持主键自动生成: 支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题

支持 ActiveRecord 模式: 支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强大的 CRUD 操作

支持自定义全局通用操作: 支持全局通用方法注入（Write once, use anywhere）

内置代码生成器：采用代码或者 Maven 插件可快速生成 Mapper 、 Model 、 Service 、 Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用

内置分页插件：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询

分页插件支持多种数据库：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer 等多种数据库

内置性能分析插件：可输出 SQL 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出慢查询

内置全局拦截插件：提供全表 delete 、 update 操作智能分析阻断，也可自定义拦截规则，预防误操作

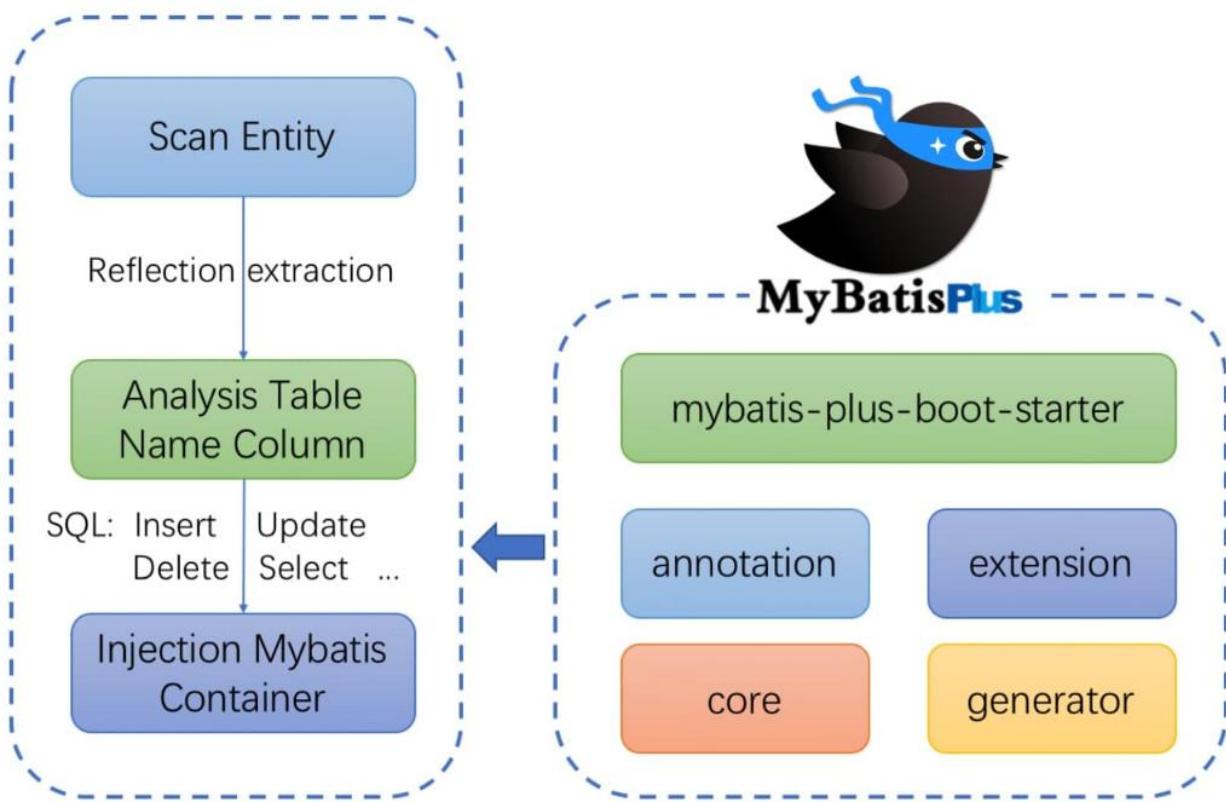
支持数据库

任何能使用 MyBatis 进行 CRUD，并且支持标准 SQL 的数据库，具体支持情况如下，如果不在下列表查看分页部分教程 PR 您的支持。

MySQL, Oracle, DB2, H2, HSQL, SQLite, PostgreSQL, SQLServer, Phoenix, Gauss , ClickHouse, Sybase, OceanBase, Firebird, Cubrid, Goldilocks, csiidb

达梦数据库，虚谷数据库，人大金仓数据库，南大通用(华库)数据库，南大通用数据库，神通数据库，瀚高数据库

1.1.5 MybatisPlus 的架构模型



1.1.6 小结

本章节主要介绍了传统的 Mybatis 框架的开发效率问题，通过发现问题，我们期望得到一款增强工具，在不改变 Mybatis 核心原理的同时，解决 Mybatis 开发效率问题，并提供其他数据库所需要的功能。

通过了解 MybatisPlus，我们得知，它是一款国产的增强工具，极大的简化了 Mybatis 框架操作，降低了 Mybatis 框架的学习成本，提高了开发效率，在国内十分流行。

1.2 入门案例

1.2.1 准备相关开发环境

IDEA 2021.3.1

PostMan 10.6.7

Navicat 15.0.9

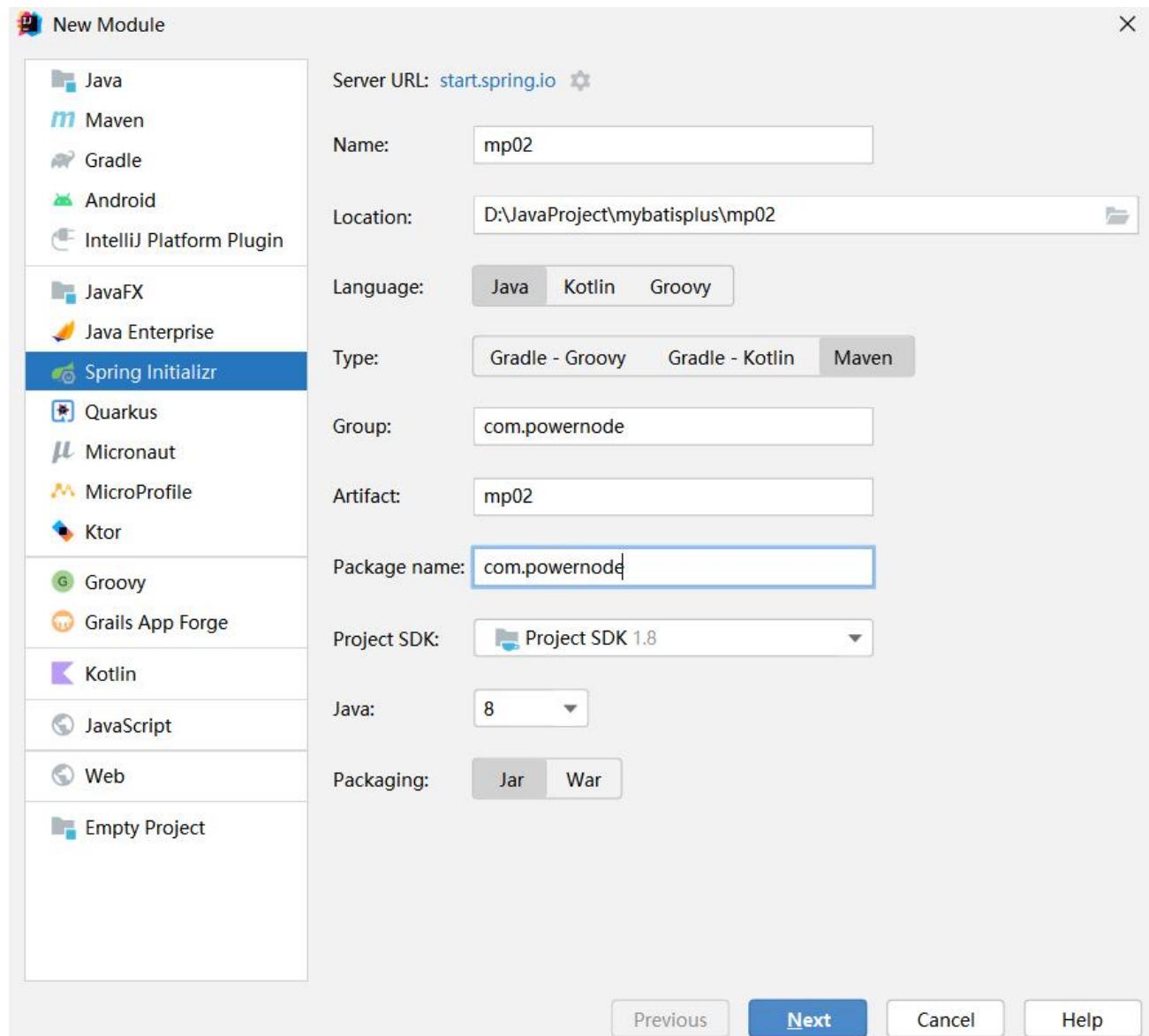
Mysql 5.7.27

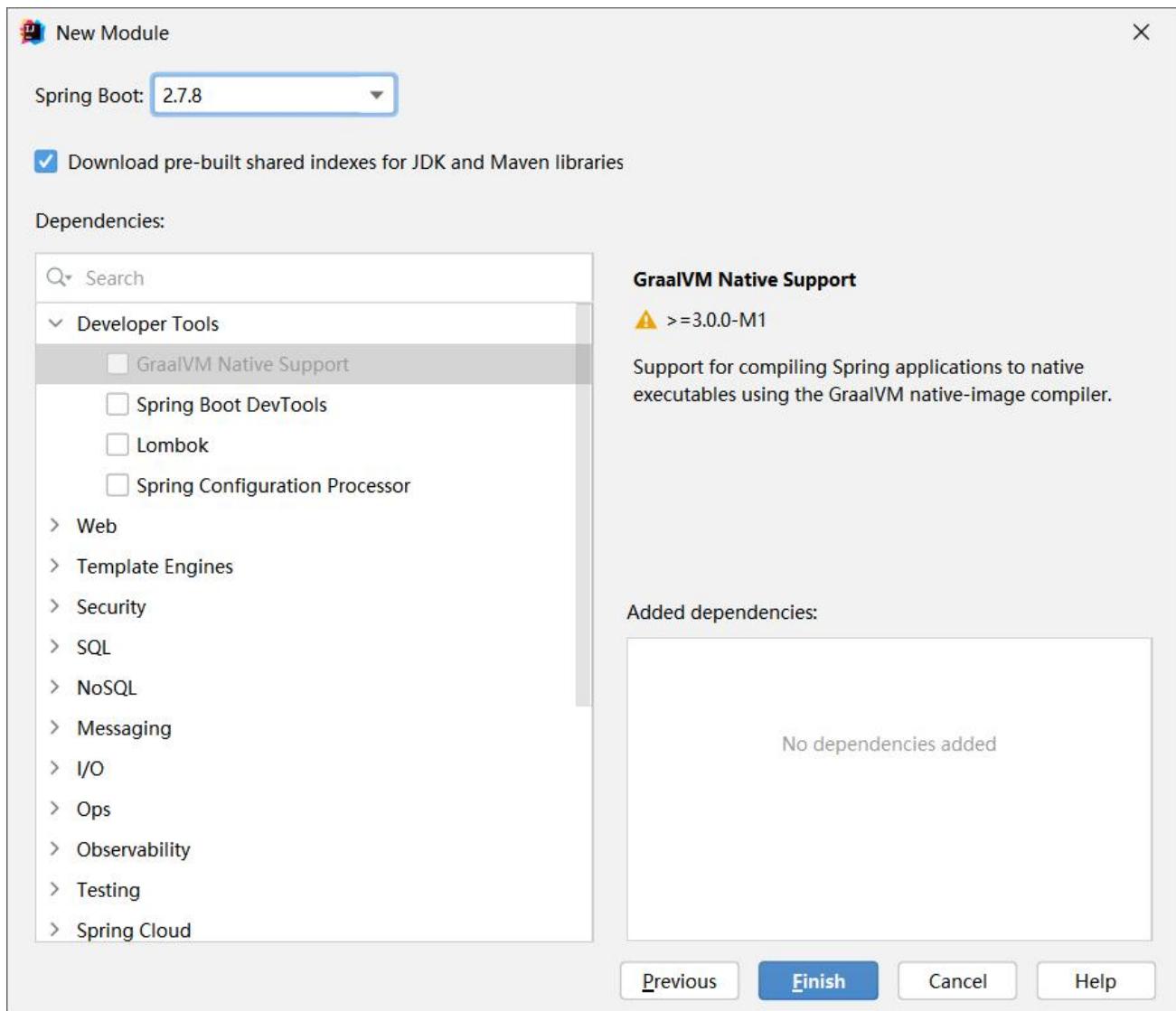
JDK8 1.8.0_311

以上环境各位同学之前一定接触过，没有安装的自行安装，在这里就不再演示安装了

另外本课程需要有 Mybatis 和 springboot 相关的基础，如果没有这部分基础的同学，需要先回顾相关课程，然后再继续本课程的学习

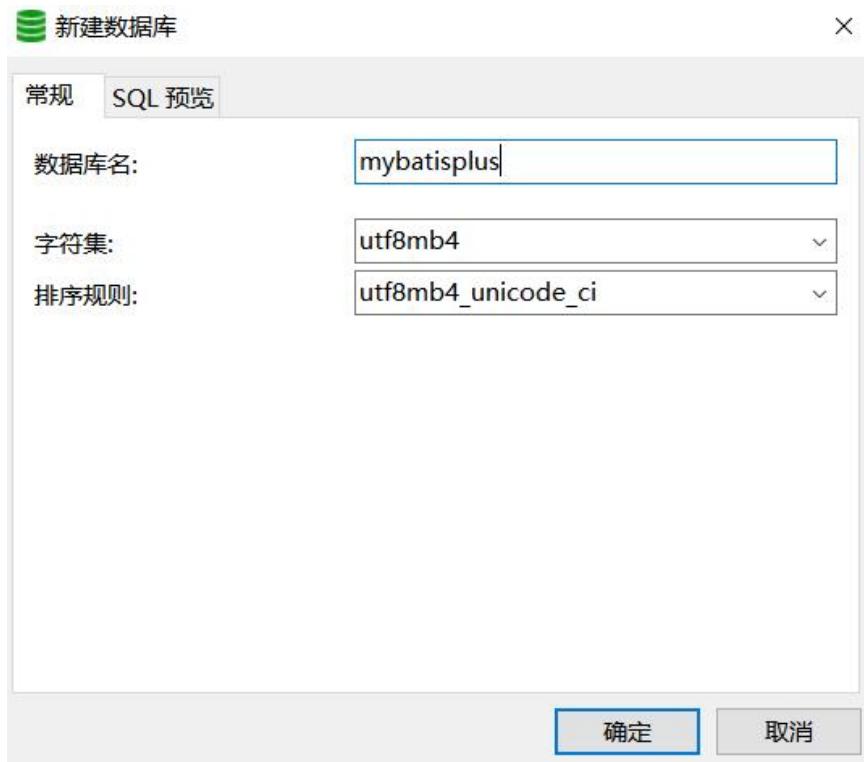
1.2.2 搭建 springboot 工程





1.2.3 创建表结构

创建数据库 mybatisplus



建表语句

```
DROP TABLE IF EXISTS user;
CREATE TABLE user(
    id BIGINT(20) NOT NULL COMMENT '主键 ID',
    name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
    age INT(11) NULL DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
    PRIMARY KEY (id));
```

1.2.4 添加表数据

插入数据

```
DELETE FROM user;
INSERT INTO user (id, name, age, email) VALUES
(1, 'Jone', 18, 'test1@baomidou.com'),
(2, 'Jack', 20, 'test2@baomidou.com'),
(3, 'Tom', 28, 'test3@baomidou.com'),
(4, 'Sandy', 21, 'test4@baomidou.com'),
```

```
(5, 'Billie', 24, 'test5@baomidou.com');
```

1.2.5 引入相关依赖

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.16</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

1.2.6 创建实体类

```
package com.powernode.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@AllArgsConstructor
```

```
@NoArgsConstructor
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

1.2.7 集成 MybatisPlus

【1】编写 Mapper 接口

```
package com.powernode.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.powernode.domain.User;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface UserMapper extends BaseMapper<User> { }
```

【2】编写 Service 接口

```
package com.powernode.service;

import com.powernode.domain.User;

import java.util.List;

public interface UserService {
    List<User> selectList();
}
```

【3】编写 ServiceImpl

```
@Service
public class UserServiceImpl implements UserService { }
```

```
@Autowired  
UserMapper userMapper;  
  
public List<User> selectList(){  
    return userMapper.selectList(null);  
}  
}
```

【4】 编写 Controller

```
import com.powernode.domain.User;  
import com.powernode.service.UserService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class UserController {  
    @Autowired  
    UserService userService;  
  
    @RequestMapping("/selectList")  
    public String selectList() {  
        List<User> all;  
        all = userService.selectList();  
        return all.toString();  
    }  
}
```

【5】 编写配置文件

```
spring:  
  datasource:  
    password: root  
    username: root  
    driver-class-name: com.mysql.cj.jdbc.Driver
```

```
url:  
jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&  
useSSL=false
```

1.2.8 测试效果



1.2.9 单元测试

到此，模拟开发的三层我们就测试完了，因为 Controller 和 Service 都是之前的知识，所以后面我们测试 MybatisPlus 的代码，通过单元测试的方式来编写

```
@SpringBootTest  
class Mp02ApplicationTests {  
  
    @Autowired  
    private UserMapper userMapper;  
  
    @Test  
    void selectList() {  
        List<User> userList = userMapper.selectList(null);  
        System.out.println(userList);  
    }  
}
```

1.2.10 精简 springboot 的相关日志

去除 mybatisplus 的 logo

```
mybatis-plus:  
  global-config:  
    banner: false
```

去除 springboot 的 logo

```
spring:  
  main:  
    banner-mode: off
```

1.2.11 MybatisPlus 的执行日志

```
mybatis-plus:  
  configuration:  
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

1.2.12 小结

本章节我们通过编写了一个 MybatisPlus 的入门案例，体验到了 MybatisPlus 的便捷性，在后面的学习中，各种各样的 MybatisPlus 的操作都非常容易上手，可以说 MybatisPlus 是一款学习成本非常低的增强工具。

2. 【基础篇】

2.1 通用 Mapper 接口介绍

有关于通用 Mapper 接口，之前我们已经看到了，我们自己编写的 Mapper 接口继承自 BaseMapper 接口，由 BaseMapper 接口提供了很多单表的增删改查相关的操作方法，在入门案例中，我们测试了查询所有的操作。在这一章节，我们介绍一些简单的 Mapper 接口中的方法，主要是感受一下，Mapper 接口中对于单表的增删改查的操作都有涉及。更加高级的一些操作，随后的章节会讲到。

2.1.1 Mapper 接口-简单插入数据

插入一条数据

```
@Test
void insert(){
    User user = new User();
    user.setId(6L);
    user.setName("Mike");
    user.setAge(33);
    user.setEmail("test6@powernode.com");
    userMapper.insert(user);
}
```

2.1.2 Mapper 接口-简单删除数据

根据 id 删除数据

```
@Test
void deleteOne(){
    userMapper.deleteById(6L);
}
```

2.1.3 Mapper 接口-简单修改数据

测试修改全部数据

```
@Test
void updateById(){
    User user = new User();
    user.setId(6L);
    user.setName("迈克");
    user.setAge(35);
    user.setEmail("maike6@powernode.com");
    userMapper.updateById(user);
}
```

测试修改部分数据

```
@Test
void updateById(){
    User user = new User();
    user.setId(6L);
    user.setName("Mike");
}
```

```
user.setAge(35);
user.setEmail("test6@powernode.com");
userMapper.updateById(user);
}
```

2.1.4 Mapper 接口-简单查询数据

根据 Id 查询

```
@Test
void updateById(){
    User user = new User();
    user.setId(6L);
    user.setName("Mike");
    user.setAge(35);
    user.setEmail("test6@powernode.com");
    userMapper.updateById(user);
}
```

查询所有

```
@Test
void selectList() {
    List<User> userList = userMapper.selectList(null);
    System.out.println(userList);
}
```

2.1.5 小结

本章我们测试了通过接口提供的基本增删改查的实现，可以感受到，将来我们有这些增删改查需求的时候，直接找到对应的方法调用，由 Mapper 接口的代理对象就会直接给我们拼接好指定的 SQL 语句，完成查询。

本章节我们测试了一些基本的增删改查操作，有关于条件查询、分页查询等高级的查询操作，在随后章节会统一讲解。

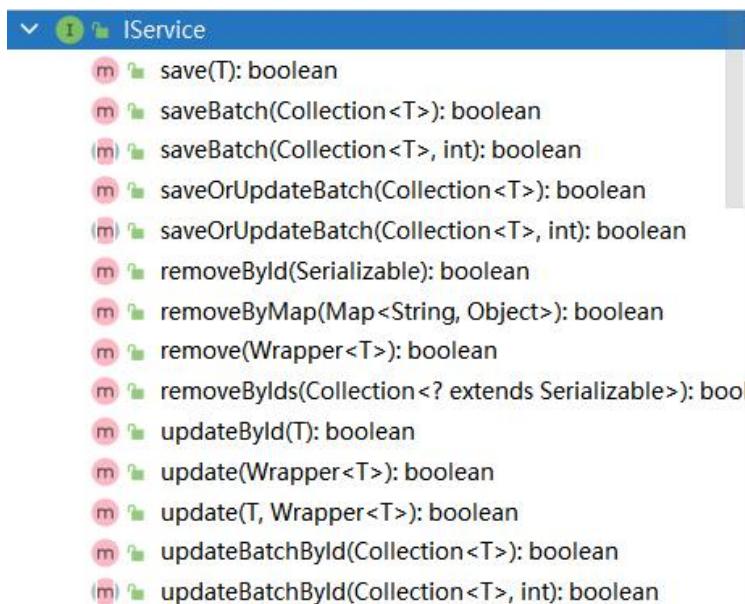
2.2 通用 service 接口介绍

除了 Mapper 接口, MybatisPlus 还提供了 IService 接口和对应的实现类 ServiceImpl, 该实现类已经提供好了一些对应的 Service 相关的方法, 在某些场景下, 我们可以直接使用 ServiceImpl 提供的方法, 实现对应的功能。

IService 接口

```
public interface IService<T>
```

IService 接口中包含了 service 相关的一些增删改查方法



ServiceImpl 实现类

```
public class ServiceImpl<M extends BaseMapper<T>, T> implements IService<T>
```

ServiceImpl 实现类提供了 service 相关的增删改查方法的实现

```

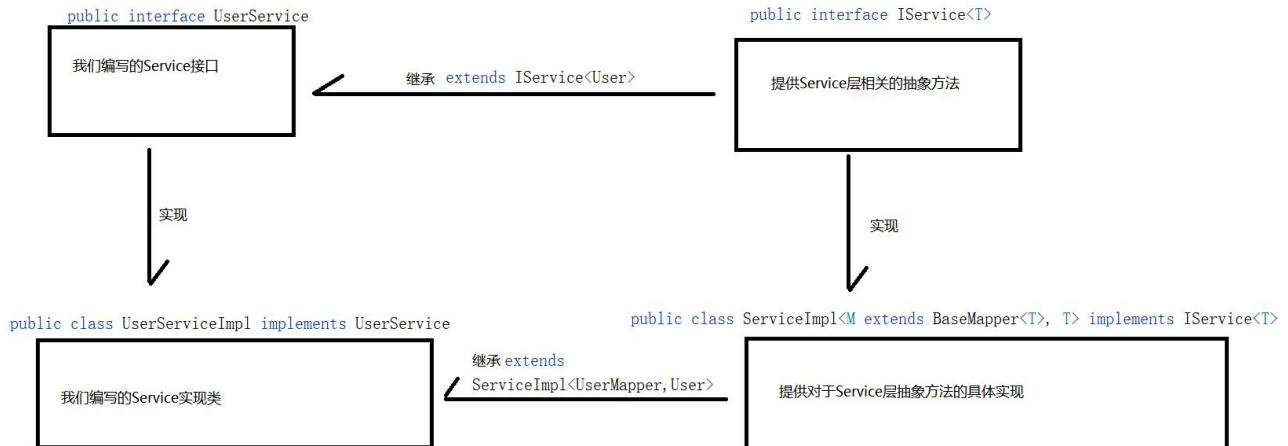
    \_ C ServiceImpl
        m ServiceImpl()
        m getBaseMapper(): M ↑IService
        m getClass(): Class<T> ↑IService
        m retBool(Integer): boolean
        m currentMapperClass(): Class<T>
        m currentModelClass(): Class<T>
        m sqlSessionBatch(): SqlSession
        m closeSqlSession(SqlSession): void
        m sqlStatement(SqlMethod): String
        m saveBatch(Collection<T>, int): boolean ↑IService
        m getSqlStatement(SqlMethod): String
        m saveOrUpdate(T): boolean ↑IService
        m saveOrUpdateBatch(Collection<T>, int): boolean ↑IService
        m updateBatchById(Collection<T>, int): boolean ↑IService
    
```

UserService 接口继承自 IService 接口

```
public interface UserService extends IService<User>
```

UserServiceImpl 类继承 ServiceImpl<UserMapper, User>

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements UserService
```



注入 UserService 对象，测试相关方法

```
@Autowired  
private UserService userService;
```

2.2.1 Service 接口-简单插入数据

```
@Test  
void insertService(){  
    User user = new User();  
    user.setId(7L);  
    user.setName("zhangsan");  
    user.setAge(35);  
    user.setEmail("zhangsan@powernode.com");  
  
    userService.save(user);  
}
```

2.2.2 Service 接口-简单删除数据

```
@Test  
void deleteService(){  
    userService.removeById(7L);  
}
```

2.2.3 Service 接口-简单修改数据

```
@Test  
void updateService(){  
    User user = new User();  
    user.setId(6L);  
    user.setAge(40);  
    userService.updateById(user);  
}
```

2.2.4 Service 接口-简单查询数据

```
@Test  
void selectService(){  
    List<User> userList = userService.selectList();  
    System.out.println(userList);  
}
```

2.2.5 小结

通过继承 MybatisPlus 提供的 Service 接口，我们既可以拓展自己的 service 方法，也可以使用现有的一些 service 方法

2.3 自定义接口方法

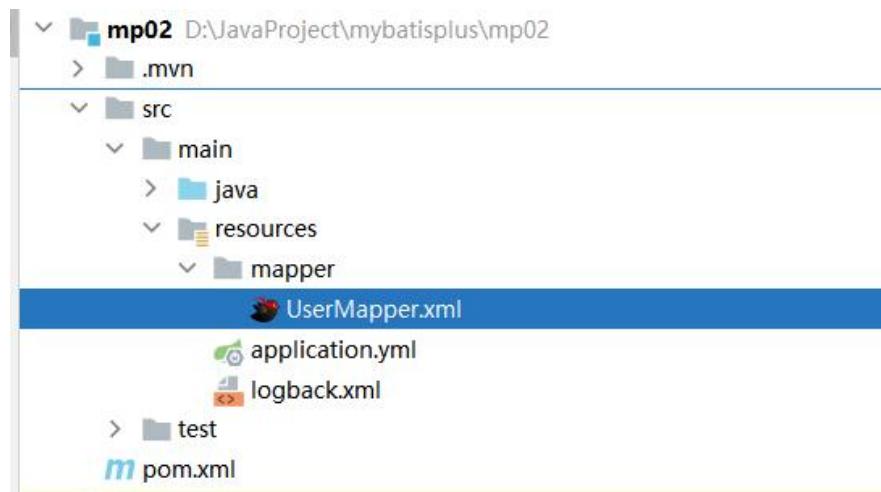
MybatisPlus 除了给我们提供了这些丰富的接口方法以外，对于我们自己的需求，也可以编写自定义的接口方法，我们通过自己编写 SQL 语句的形式，实现想要的 SQL 需求

2.3.1 自定义 Mapper 接口方法

Mapper 接口中提供抽象方法

```
@Mapper  
public interface UserMapper extends BaseMapper<User> {  
    User selectByName(String name);  
}
```

提供映射配置文件，提供对应的 SQL 语句



```
<?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.powernode.mapper.UserMapper">

    <select id="selectByName" resultType="com.powernode.domain.User">
        select * from user where name = #{value}
    </select>

</mapper>
```

测试自定义的 Mapper 接口方法

```
@Test
void myMethod(){
    User user = userMapper.selectByName("Jone");
    System.out.println(user);
}
```

2.3.2 小结

通过本章节的学习，我们学会了自定义接口的方法，自定义接口的语法规规范和之前编写 Mybatis 的语法规规范一样，所以可以看出，MybatisPlus 是无侵入式的，也就是引入了 MybatisPlus 并不会对于原先的语法造成任何改变。

3. 【进阶篇】

3.1 映射

学习过 Mybatis 的同学应该知道，Mybatis 框架之所以能够简化数据库操作，是因为他内部的映射机制，通过自动映射，进行数据的封装，我们只要符合映射规则，就可以快速高效的完成 SQL 操作的实现。

既然 MybatisPlus 是基于 Mybatis 的增强工具，所以也具有这样的映射规则。

我们先来了解一下自动映射规则。

3.1.1 自动映射规则

【1】表名和实体类名映射 -> 表名 user 实体类名 User

【2】字段名和实体类属性名映射 -> 字段名 name 实体类属性名 name

【3】字段名下划线命名方式和实体类属性小驼峰命名方式映射 ->

字段名 user_email 实体类属性名 userEmail

MybatisPlus 支持这种映射规则，可以通过配置来设置

`map-underscore-to-camel-case: true` 表示支持下划线到驼峰的映射
`map-underscore-to-camel-case: false` 表示不支持下划线到驼峰的映射

```
mybatis-plus:  
  configuration:  
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl  
    map-underscore-to-camel-case: true
```

3.1.2 表映射

通过`@TableName()`注解指定映射的数据库表名，就会按照指定的表名进行映射

如：此时将数据库的表名改为 powershop_user，要完成表名和实体类名的映射，需要将实体类名也要指定为 powershop_user

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@TableName("powershop_user")
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

如果有很多实体类，对应到数据库中的很多表，我们不需要每个依次配置，只需要配置一个全局的设置，他都会给每个实体类名前面添加指定的前缀，这里我们演示一下全局配置的效果

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: powershop_
```

3.1.3 字段映射

什么场景下会改变字段映射呢？

【1】当数据库字段和表实体类的属性不一致时，我们可以使用`@TableField()`注解改变字段和属性的映射，让注解中的名称和表字段保持一致

如：此时将数据库字段的名称我们改为 username，在根据实体类的属性拼接 SQL 的使用，就会使用`@TableField()`中指定的名称 username 进行拼接，完成查询

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User {  
    @TableField("username")  
    private String name;  
}
```

此时的 SQL 语句是这样的

```
SELECT id,username AS name,email FROM powershop_user
```

【2】数据库字段和表实体类的属性一致，框架在拼接 SQL 语句的时候，会使用属性名称直接拼接 sql 语句，例如：

```
SELECT id,username AS name,age,email,desc FROM powershop_user
```

这条语句直接进行查询的时候，会出现错误

Error querying database. Cause: java.sql.SQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'desc FROM powershop_user' at line 1

原因是，desc 属于关键字，不能直接用于 sql 查询，要解决这个问题，就需要将 desc 字段加上``符号，将他变为不是关键字，才能完成查询，那这个问题的根本也是改变生成的 SQL 语句的字段名称，也就是我们需要通过@TableField()改变实体类的属性名称，将 desc 变为`desc`，就可以解决这个问题

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User {  
    @TableField("`desc`")
```

```
private String desc;  
}
```

```
Creating a new SqlSession  
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@147e0734] was not registered for synchronization because synchronization is disabled  
JDBC Connection [HikariProxyConnection@1608392036 wrapping com.mysql.cj.jdbc.ConnectionImpl@5350ab17] will not be managed by Spring  
==> Preparing: SELECT id,username AS name,age,email,`desc` FROM powershop_user  
==> Parameters:  
<== Columns: id, name, age, email, desc  
<== Row: 1, Jone, 18, test1@baomidou.com, null  
<== Row: 2, Jack, 20, test2@baomidou.com, null  
<== Row: 3, Tom, 28, test3@baomidou.com, null  
<== Row: 4, Sandy, 21, test4@baomidou.com, null  
<== Row: 5, Billie, 24, test5@baomidou.com, null  
<== Row: 6, Mike, 40, test6@powernode.com, null  
<== Total: 6
```

此时可以观察到，框架拼接生成的 SQL 语句的字段名称变为了 `desc`，这样是可以正常完成查询的

3.1.4 字段失效

当数据库中有字段不希望被查询，我们可以通过 `@TableField(select = false)` 来隐藏这个字段，那在拼接 SQL 语句的时候，就不会拼接这个字段

如：如果不显示年龄信息，那么可以在 `age` 属性上添加这个注解，来隐藏这个字段

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User {  
    @TableField(select = false)  
    private Integer age;  
}
```

生成的 SQL 语句如下，通过查询生成的 SQL 语句，发现并没有拼接 `age` 字段

```
JDBC Connection [HikariProxyConnection@1608392036 wrapping com.mysql.cj.jdbc.ConnectionImpl@5350ab17] will not be managed by Spring
==> Preparing: SELECT id,username AS name,email,'desc' FROM powershop_user
==> Parameters:
<==   Columns: id, name, email, desc
<==     Row: 1, Jone, test1@baomidou.com, null
<==     Row: 2, Jack, test2@baomidou.com, null
<==     Row: 3, Tom, test3@baomidou.com, null
<==     Row: 4, Sandy, test4@baomidou.com, null
<==     Row: 5, Billie, test5@baomidou.com, null
<==     Row: 6, Mike, test6@powernode.com, null
<==   Total: 6
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@7336fd8f]
[User(id=1, name=Jone, age=null, email=test1@baomidou.com, desc=null), User(id=2, name=Jack, age=null, email=test2@baomidou.com, desc=null),
```

3.1.5 视图属性

在实际开发中，有些字段不需要数据库存储，但是却需要展示，需要展示也就是意味着实体类中需要存在这个字段，我们称这些实体类中存在但是数据库中不存在的字段，叫做视图字段。

根据之前的经验，框架会默认将实体类中的属性作为查询字段进行拼接，那我们来思考，像这种视图字段，能够作为查询条件么，显示是不能的。因为数据库中没有这个字段，所以查询字段如果包含这个字段，SQL语句会出现问题。我们通过`@TableField(exist = false)`来去掉这个字段，不让他作为查询字段。

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableField(exist = false)
    private Integer online;
}
```

可以看到查询结果中不包含该字段

```
--> Preparing: SELECT id,username AS name,email,`desc` FROM powershop_user
--> Parameters:
<==    Columns: id, name, email, desc
<==      Row: 1, Jone, test1@baomidou.com, null
<==      Row: 2, Jack, test2@baomidou.com, null
<==      Row: 3, Tom, test3@baomidou.com, null
<==      Row: 4, Sandy, test4@baomidou.com, null
<==      Row: 5, Billie, test5@baomidou.com, null
<==      Row: 6, Mike, test6@powernode.com, null
<==  Total: 6
```

3.1.6 小结

这一章节，我们讲解了 MybatisPlus 的映射规则，以及如果通过注解配置来改变这种映射规则。

3.2 条件构造器

3.2.1 条件构造器介绍

之前我们进行的 MybatisPlus 的操作，没有涉及到条件查询，实际上在开发需求中条件查询是非常普遍的。接下来我们就来讲解如何使用 MybatisPlus 完成条件查询。

首先，想要使用 MybatisPlus 完成条件查询，基于面向对象的思想，万物皆对象，那么查询条件也需要使用对象来完成封装。我们先看一下，在 MybatisPlus 中，和条件有关的类有哪些，他们之间有什么关系，理清楚了这个，我们在传递条件对象的时候，就很清晰了。

3.2.2 Wrapper

抽象类，条件类的顶层，提供了一些获取和判断相关的方法

```
public abstract class Wrapper<T> implements ISqlSegment
```

3.2.3 AbstractWrapper

抽象类，Wrapper 的子类，提供了所有的条件相关方法

```
public abstract class AbstractWrapper<T, R, Children extends AbstractWrapper<T, R, Children>> extends Wrapper<T>  
implements Compare<Children, R>, Nested<Children, Children>, Join<Children>, Func<Children, R> {
```

3.2.4 AbstractLambdaWrapper

抽象类，AbstractWrapper 的子类，确定字段参数为方法引用类型

```
public abstract class AbstractLambdaWrapper<T, Children extends AbstractLambdaWrapper<T, Children>>  
extends AbstractWrapper<T, SFunction<T, ?>, Children> {
```

3.2.5 QueryWrapper

类，AbstractWrapper 的子类，如果我们需要传递 String 类型的字段信息，创建该对象

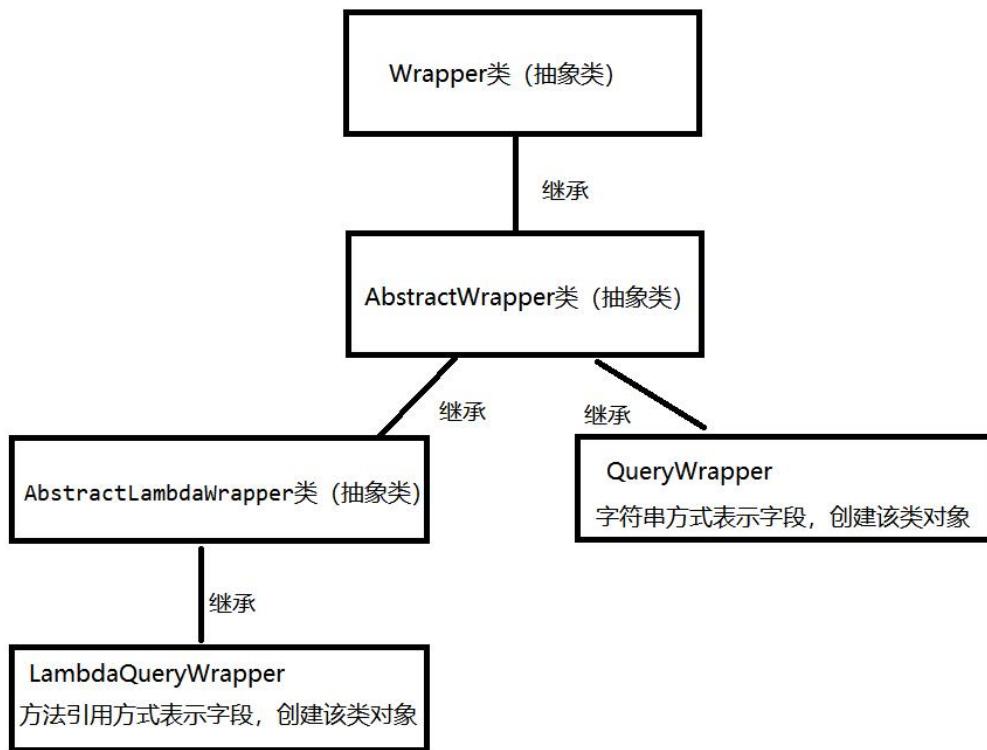
```
public class QueryWrapper<T> extends AbstractWrapper<T, String, QueryWrapper<T>>  
implements Query<QueryWrapper<T>, T, String> {
```

3.2.6 LambdaQueryWrapper

类，AbstractLambdaWrapper 的子类，如果我们需要传递方法引用方式的字段信息，创建该对象

```
public class LambdaQueryWrapper<T> extends AbstractLambdaWrapper<T, LambdaQueryWrapper<T>>  
implements Query<LambdaQueryWrapper<T>, T, SFunction<T, ?>> {
```

该图为以上各类的关系，我们在编写代码的时候，只需要关注 QueryWrapper 和 LambdaQueryWrapper



3.2.7 小结

通过学习类的继承体系，我们知道，我们需要重点掌握 **QueryWrapper** 和 **LambdaQueryWrapper** 这两个类，在一般情况下，我们大多选择 **LambdaQueryWrapper**，因为选择这种方式传递参数，不用担心拼写错误问题。

3.3 等值查询

3.3.1 eq

使用 **QueryWrapper** 对象，构建查询条件

```
@Test  
void eq(){  
    //1. 创建 QueryWrapper 对象
```

```
QueryWrapper<User> queryWrapper = new QueryWrapper<>();
//2.设置条件，指定 String 字段名称和值
queryWrapper.eq("name", "Jack");
//3.使用条件完成查询
User user = userMapper.selectOne(queryWrapper);
System.out.println(user);
}
```

测试效果

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name = ?)
==> Parameters: Jack(String)
<==    Columns: id, name, age, email
<==          Row: 2, Jack, 20, test2@baomidou.com
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@660b1a9d]
User(id=2, name=Jack, age=20, email=test2@baomidou.com)
```

我们思考如果每次都是自己进行字段名称的编写，有可能会出现名称写错的情况，怎么避免这种情况呢，我们可以使用 LambdaQueryWrapper 对象，在构建字段时，使用方法引用的方式来选择字段，这样做可以避免字段拼写错误出现问题。

代码如下：

```
@Test
void eq2(){
    //1.创建QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定 String 字段名称和值
    lambdaQueryWrapper.eq(User::getName, "Jack");
    //3.使用条件完成查询
    User user = userMapper.selectOne(lambdaQueryWrapper);
    System.out.println(user);
}
```

还要考虑一种情况，我们构建的条件是从哪里来的？应该是从客户端通过请求发送过来的，

由服务端接收的。在网站中一般都会有多个条件入口，用户可以选择一个或多个条件进行查询，那这个时候在请求时，我们不能确定所有的条件都是有值的，部分条件可能用户没有传值，那该条件就为 null。

比如在电商网站中，可以选择多个查询条件。



那为 null 的条件，我们是不需要进行查询条件拼接的，否则就会出现如下情况，将为 null 的条件进行拼接，筛选后无法查询出结果

```
@Test
voidisNull(){
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    lambdaQueryWrapper.eq(User::getName, null);
    User user = userMapper.selectOne(lambdaQueryWrapper);
    System.out.println(user);
}
```

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name = ?)
--> Parameters: null
<--      Total: 0
```

当然我们要解决这个问题，可以先判断是否为空，根据判断结果选择是否拼接该字段，这个功能其实不需要我们写，由 MybatisPlus 的方法已经提供好了。

```
@Test
```

```
voidisNull2(){
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    String name = null;
    lambdaQueryWrapper.eq(name != null, User::getName, name);
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

3.3.2 allEq

先演示一下如何通过多个 eq, 构建多条件查询

```
@Test
void allEq1(){
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    lambdaQueryWrapper.eq(User::getName, "Jone");
    lambdaQueryWrapper.eq(User::getAge, 18);

    User user = userMapper.selectOne(lambdaQueryWrapper);
    System.out.println(user);
}
```

如果此时有多个条件需要同时判断，我们可以将这多个条件放入到 Map 集合中，更加的方便

```
@Test
void allEq2(){
    //1. 创建 QueryWrapper 对象
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();

    //2. 构建条件 Map
    HashMap<String, Object> hashMap = new HashMap<>();
    hashMap.put("name", "Jone");
    hashMap.put("age", null);

    //3. 使用条件完成查询
    queryWrapper.allEq(hashMap, false);
    User user = userMapper.selectOne(queryWrapper);
    System.out.println(user);
```

{}

```
allEq(Map<R, V> params, boolean null2IsNull)
```

参数 params:表示传递的 Map 集合

参数 null2IsNull:表示对于为 null 的条件是否判断 isNull

3.3.3 ne

```
@Test
void ne(){
    //1.创建QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定 String 字段名称和值
    String name = "Jone";
    lambdaQueryWrapper.ne(User::getName, name);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name <> ?)
==> Parameters: Jone(String)
```

3.4 范围查询

3.4.1 gt

```
@Test
void gt(){
    //1.创建QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    Integer age = 18;
```

```
lambdaQueryWrapper.gt(User::getAge,age);
//3.使用条件完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age > ?)
--> Parameters: 18(Integer)
```

3.4.2 ge

```
@Test
void gt(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    Integer age = 18;
    lambdaQueryWrapper.ge(User::getAge,age);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age >= ?)
--> Parameters: 18(Integer)
```

3.4.3 lt

```
@Test
void lt(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    Integer age = 18;
```

```
lambdaQueryWrapper.lt(User::getAge,age);
//3.使用条件完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age < ?)
==> Parameters: 18(Integer)
```

3.4.4 le

```
@Test
void le(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    Integer age = 18;
    lambdaQueryWrapper.le(User::getAge,age);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age <= ?)
==> Parameters: 18(Integer)
```

3.4.5 between

```
@Test
void between(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
```

```
//2.设置条件，指定字段名称和值
lambdaQueryWrapper.between(User::getAge,18,30);
//3.使用条件完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

3.4.6 notBetween

```
@Test
void notBetween(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.notBetween(User::getAge,18,30);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

3.5 模糊查询

3.5.1 like

```
@Test
void like(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.like(User::getName, "J");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name LIKE ?)
--> Parameters: %J%(String)
```

3.5.2 notLike

```
@Test
void notLike(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.notLike(User::getName, "J");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name NOT LIKE ?)
--> Parameters: %J%(String)
```

3.5.3 likeLeft

```
@Test
void likeLeft(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.likeLeft(User::getName, "e");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name LIKE ?)
--> Parameters: %e(String)
```

3.5.4 likeRight

```
@Test
void likeLeft(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.likeLeft(User::getName, "J");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name LIKE ?)
--> Parameters: J%(String)
```

3.6 判空查询

3.6.1 isNull

```
@Test
void isNull(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称
    lambdaQueryWrapper.isNull(User::getName);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name IS NULL)
--> Parameters:
```

3.6.2 isNotNull

```
@Test
void isNotNull(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称
    lambdaQueryWrapper.isNotNull(User::getName);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name IS NOT NULL)
--> Parameters:
```

3.7 包含查询

3.7.1 in

```
@Test
void in(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    ArrayList<Integer> arrayList = new ArrayList<>();
    Collections.addAll(arrayList,18,20,21);
    lambdaQueryWrapper.in(User::getAge,arrayList);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

```
@Test
void in2(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.in(User::getAge,18,20,21);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age IN (?, ?, ?))
--> Parameters: 18(Integer), 20(Integer), 21(Integer)
```

3.7.2 notIn

```
@Test
void notIn(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    ArrayList<Integer> arrayList = new ArrayList<>();
    Collections.addAll(arrayList,18,20,21);
    lambdaQueryWrapper.notIn(User::getAge,arrayList);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

```
@Test
void notIn2(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.notIn(User::getAge,18,20,21);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age NOT IN (?, ?, ?))
--> Parameters: 18(Integer), 20(Integer), 21(Integer)
```

3.7.3 inSql

```
@Test
void inSql(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.inSql(User::getAge,"18,20,22");
    //3.使用条件完成查询
}
```

```
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age IN (18,20,21))
--> Parameters:
```

```
@Test
void inSql2(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.inSql(User::getAge,"select age from powershop_user where age > 20");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age IN (select age from powershop_user where age > 20))
--> Parameters:
```

3.7.4 notInSql

```
@Test
void notInSql(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定字段名称和值
    lambdaQueryWrapper.notInSql(User::getAge, "18,20,21");
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age NOT IN (18,20,21))
--> Parameters:
```

```
@Test
void notInSql2(){
    //1. 创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2. 设置条件，指定字段名称和值
    lambdaQueryWrapper.notInSql(User::getAge,"select age from powershop_user where
    age > 20");
    //3. 使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age NOT IN (18,20,21))
--> Parameters:
```

3.8 分组查询

3.8.1 groupBy

```
@Test
void groupBy(){
    //1. 创建 QueryWrapper 对象
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    //2. 设置条件，指定字段名称和值
    queryWrapper.groupBy("age");
    queryWrapper.select("age,count(*) as field_count");
    //3. 使用条件完成查询
    List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);
    System.out.println(maps);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT age,count(*) as field_count FROM powershop_user GROUP BY age
--> Parameters:
<==   Columns: age, field_count
<==       Row: 18, 1
<==       Row: 20, 2
<==       Row: 24, 1
<==       Row: 28, 1
<==   Total: 4
```

实际查询结果

age	field_count
18	1
20	2
24	1
28	1

封装结果

```
[{field_count=1, age=18}, {field_count=2, age=20}, {field_count=1, age=24}, {field_count=1, age=28}]
```

3.9 聚合查询

3.9.1 having

```
@Test
void having(){
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    //分组字段
    queryWrapper.groupBy("age");
    //查询字段
    queryWrapper.select("age,count(*) as field_count");
    //聚合条件筛选
    queryWrapper.having("field_count = 1");
    List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);
    System.out.println(maps);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT age,count(*) as field_count FROM powershop_user GROUP BY age HAVING field_count = 1
--> Parameters:
<==   Columns: age, field_count
<==       Row: 18, 1
<==       Row: 24, 1
<==       Row: 28, 1
<==   Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@675ec28b]
[{field_count=1, age=18}, {field_count=1, age=24}, {field_count=1, age=28}]
```

3.10 排序查询

3.10.1 orderByAsc

```
@Test
void orderByAsc(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定升序排序字段
    lambdaQueryWrapper.orderByAsc(User::getAge,User::getId);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user ORDER BY age ASC,id ASC
--> Parameters:
```

3.10.2 orderByDesc

```
@Test
void orderByDesc(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定降序排序字段
    lambdaQueryWrapper.orderByDesc(User::getAge,User::getId);
```

```
//3.使用条件完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user ORDER BY age DESC,id DESC
--> Parameters:
```

3.10.3 orderBy

```
@Test
void orderBy(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.设置条件，指定降序排序字段
    lambdaQueryWrapper.orderBy(true,true,User::getId);
    lambdaQueryWrapper.orderBy(true,false,User::getAge);
    //3.使用条件完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user ORDER BY id ASC,age DESC
--> Parameters:
```

3.11 func 查询

3.11.1 func

```
@Test
void func(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建逻辑判断语句
    lambdaQueryWrapper.func(i -> {
        if(true) {
```

```
i.eq(User::getId, 1);
}else {
    i.ne(User::getId, 1);
}
});
//3.完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (id = ?)
--> Parameters: 1(Integer)
```

3.12 逻辑查询

3.12.1 and

正常拼接默认就是 and, 例如

```
@Test
void and(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建条件查询语句
    lambdaQueryWrapper.gt(User::getAge, 22).lt(User::getAge, 30);
    //3.完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age > ? AND age < ?)
--> Parameters: 22(Integer), 30(Integer)
```

and 也可以进行嵌套

```
@Test
void and2(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建条件查询语句
    lambdaQueryWrapper.eq(User::getName, "wang").and(i ->
        i.gt(User::getAge, 26).or().lt(User::getAge, 22));
    //3.完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (name = ? AND (age > ? OR age < ?))
--> Parameters: wang(String), 22(Integer), 30(Integer)
```

3.12.2 or

```
@Test
void or(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建条件查询语句
    lambdaQueryWrapper.lt(User::getAge, 20).or().gt(User::getAge, 23);
    //3.完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age < ? OR age > ?)
--> Parameters: 20(Integer), 23(Integer)
```

OR 嵌套

```
@Test
void or2(){
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    lambdaQueryWrapper.eq(User::getName, "wang").or(i ->
        i.gt(User::getAge, 22).lt(User::getAge, 26));
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (age > ? OR (name = ? AND age = ?))
--> Parameters: 30(Integer), Jone(String), 18(Integer)
```

3.12.3 nested

```
@Test
void nested(){
    //1. 创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2. 构建条件查询语句
    lambdaQueryWrapper.nested(i -> i.eq(User::getName, "Billie").ne(User::getAge,
22));
    //3. 完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE ((name = ? AND age <> ?))
--> Parameters: Billie(String), 22(Integer)
```

3.13 自定义条件查询

3.13.1 apply

```
@Test
void apply(){
    //1. 创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2. 构建条件查询语句
```

```
lambdaQueryWrapper.apply("id = 1");
//3.完成查询
List<User> users = userMapper.selectList(lambdaQueryWrapper);
System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (id = 1)
==> Parameters:
```

3.14 last 查询

3.14.1 last

```
@Test
void last(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建条件查询语句
    lambdaQueryWrapper.last("limit 0,2");
    //3.完成查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name,age,email FROM powershop_user limit 0,2
==> Parameters:
```

3.15 exists 查询

3.15.1 exists

```
@Test
void exists(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建查询条件
    lambdaQueryWrapper.exists("select id from powershop_user where age = 18");
    //3.查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (EXISTS (select id from powershop_user where age = 18))
--> Parameters:
```

3.15.2 notExists

```
@Test
void notExists(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建查询条件
    lambdaQueryWrapper.notExists("select id from powershop_user where age = 33");
    //3.查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
--> Preparing: SELECT id,name,age,email FROM powershop_user WHERE (NOT EXISTS (select id from powershop_user where age = 33))
--> Parameters:
```

3.16 字段查询

3.16.1 select

```
@Test
void select(){
    //1.创建QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.构建查询条件
    lambdaQueryWrapper.select(User::getId,User::getName);
    //3.查询
    List<User> users = userMapper.selectList(lambdaQueryWrapper);
    System.out.println(users);
}
```

拼接的 SQL 如下

```
==> Preparing: SELECT id,name FROM powershop_user
==> Parameters:
<==     Columns: id, name
```

3.17 小结

本章节讲解了条件查询中的各种查询操作，我们可以通过面向对象的形式，直接调用方法来完成 mysql 中的条件查询

4. 【高级篇】

4.1 主键策略

4.1.1 主键生成策略介绍

首先大家先要知道什么是主键，主键的作用就是唯一标识，我们可以通过这个唯一标识来

定位到这条数据。

当然对于表数据中的主键，我们可以自己设计生成规则，生成主键。但是在更多的场景中，没有特殊要求的话，我们每次自己手动生成的比较麻烦，我们可以借助框架提供好的主键生成策略，来生成主键。这样比较方便快捷

在 MybatisPlus 中提供了一个注解，是@TableId，该注解提供了各种的主键生成策略，我们可以通过使用该注解来对于新增的数据指定主键生成策略。那么在以后新增数据的时候，数据就会按照我们指定的主键生成策略来生成对应的主键。

4.1.2 AUTO 策略

该策略为跟随数据库表的主键递增策略，前提是数据库表的主键要设置为自增

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	🔑 1	主键ID
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱

默认:

自动递增

无符号

填充零

此处要设置好下次递增的数字

保存

字段	索引	外键	触发器	选项	注释	SQL 预览
引擎:	InnoDB					
表空间:						
存储:						
字符集:	utf8mb4					
排序规则:	utf8mb4_general_ci					
自动递增:	7					
行格式:	DYNAMIC					
平均行长度:	0					
最大行数:	0					
最小行数:	0					
键块大小:	0					
数据目录:						
索引目录:						
统计数据自动重计:						
统计数据持久:						
统计样本页面:	0					
压缩:						
<input type="checkbox"/> 加密						
分区						

实体类添加注解，指定主键生成策略

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
    
```

```
@Test
void primaryKey(){
    User user = new User();
    user.setName("Mary");
    user.setAge(35);
    user.setEmail("test7@powernode.com");
    userMapper.insert(user);
}
```

拼接的 SQL 语句如下

```
--> Preparing: INSERT INTO powershop_user ( name, age, email ) VALUES ( ?, ?, ? )
--> Parameters: Tom(String), 35(Integer), test7@powernode.com(String)
<==    Updates: 1
```

4.1.3 INPUT 策略

该策略表示，必须由我们手动的插入 id，否则无法添加数据

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableId(type = IdType.INPUT)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

由于我们不使用 AUTO 了，所以把自动递增去掉

字段	索引	外键	触发器	选项	注释	SQL 预览		
名								
id							 1	主键ID
name								姓名
age								年龄
email								邮箱

默认:

自动递增

无符号

填充零

这里如果我们省略不写 id,会发现，无法插入数据

```
@Test
void primaryKey(){
    User user = new User();
    user.setName("Jerry");
    user.setAge(38);
    user.setEmail("test8@powernode.com");
    userMapper.insert(user);
}
```

```
org.springframework.dao.DataIntegrityViolationException:
### Error updating database. Cause: java.sql.SQLIntegrityConstraintViolationException: Column 'id' cannot be null
### The error may exist in com/powernode/mapper/UserMapper.java (best guess)
### The error may involve com.powernode.mapper.UserMapper.insert-Inline
### The error occurred while setting parameters
### SQL: INSERT INTO powershop_user ( id, name, age, email ) VALUES ( ?, ?, ?, ? )
### Cause: java.sql.SQLIntegrityConstraintViolationException: Column 'id' cannot be null
; Column 'id' cannot be null; nested exception is java.sql.SQLIntegrityConstraintViolationException: Column 'id' cannot be null
```

但是我们自己指定了 id,发现可以添加成功

```
@Test
void primaryKey(){
    User user = new User();
    user.setId(8L);
    user.setName("Jerry");
    user.setAge(38);
    user.setEmail("test8@powernode.com");
    userMapper.insert(user);
}
```

4.1.4 ASSIGN_ID 策略

我们来思考一下，像之前这种自动递增的方式，有什么问题？

如果我们将来一张表的数据量很大，我们需要进行分表。

常见的分表策略有两种：

【1】水平拆分

id	name	age	email
1	Jone	18	test1@baomidou.co
2	Jack	20	test2@baomidou.co
3	Tom	22	test3@baomidou.co
4	Sandy	22	test4@baomidou.co
5	Billie	25	test5@baomidou.co
6	Mike	40	test6@powernode.co
7	Tom	35	test7@powernode.co

水平拆分就是将一个大的表按照数据量进行拆分

【2】垂直拆分

id	name	age	email
1	Jone	18	test1@baomidou.co
2	Jack	20	test2@baomidou.co
3	Tom	22	test3@baomidou.co
4	Sandy	22	test4@baomidou.co
5	Billie	25	test5@baomidou.co
6	Mike	40	test6@powernode.c
7	Tom	35	test7@powernode.c

垂直拆分就是将一个大的表按照字段进行拆分

其实我们对于拆分后的数据，有三点需求，就拿水平拆分来说：

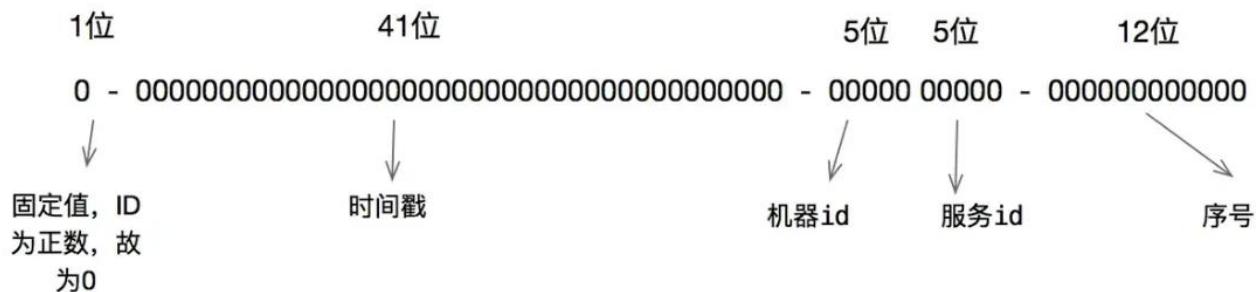
- 【1】之前的表的主键是有序的，拆分后还是有序的
- 【2】虽然做了表的拆分，但是每条数据还需要保证主键的唯一性
- 【3】主键最好不要直接暴露数据的数量，这样容易被外界知道关键信息

那就需要有一种算法，能够实现这三个需求，这个算法就是雪花算法

雪花算法是由一个 64 位的二进制组成的，最终就是一个 Long 类型的数值。

主要分为四部分存储

- 【1】1 位的符号位，固定值为 0
- 【2】41 位的时间戳
- 【3】10 位的机器码，包含 5 位机器 id 和 5 位服务 id
- 【4】12 位的序列号



使用雪花算法可以实现有序、唯一、且不直接暴露排序的数字。

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableId(type = IdType.ASSIGN_ID)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

```
@Test
void primaryKey(){
    User user = new User();
    user.setName("Jerry");
    user.setAge(38);
    user.setEmail("test8@powernode.com");
    userMapper.insert(user);
}
```

我们可以在插入后发现一个19位长度的id,该id就是雪花算法生成的id,这是二级制的十进制表示形式

id	name	age	email
1	Jone	18	test1@baomidou.co
2	Jack	20	test2@baomidou.co
3	Tom	22	test3@baomidou.co
4	Sandy	22	test4@baomidou.co
5	Billie	25	test5@baomidou.co
6	Mike	40	test6@powernode.co
7	Tom	35	test7@powernode.co
1612617261598654465	Jerry	38	test8@powernode.co

4.1.5 NONE 策略

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableId(type = IdType.NONE)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}

```

NONE 策略表示不指定主键生成策略，当我们没有指定主键生成策略或者主键策略为 NONE 的时候，他跟随的是全局策略，那我们来看一下他的全局策略默认是什么

全局配置中 id-type 是用于配置主键生成策略的，我们可以看一下 id-type 的默认值

```

global-config:
  banner: false
db-config:
  table-prefix: powershop_
  id-type:

```

```
@Data
```

```
public static class DbConfig {  
    // 主键类型  
    private IdType idType = IdType.ASSIGN_ID;
```

通过查看源码发现，id-type 的默认值就是雪花算法

4.1.6 ASSIGN_UUID 策略

UUID (Universally Unique Identifier) 全局唯一标识符，定义为一个字符串主键，采用 32 位数字组成，编码采用 16 进制，定义了在时间和空间都完全唯一的系统信息。

UUID 的编码规则：

【1】1~8 位采用系统时间，在系统时间上精确到毫秒级保证时间上的唯一性；

【2】9~16 位采用底层的 IP 地址，在服务器集群中的唯一性；

【3】17~24 位采用当前对象的HashCode 值，在一个内部对象上的唯一性；

【4】25~32 位采用调用方法的一个随机数，在一个对象内的毫秒级的唯一性。

通过以上 4 种策略可以保证唯一性。在系统中需要用到随机数的地方都可以考虑采用 UUID 算法。

我们想要演示 UUID 的效果，需要改变一下表的字段类型和实体类的属性类型

将数据库表的字段类型改为 varchar(50)

名	类型	长度	小数点	不是 null	虚拟	键
id	varchar	50	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>	
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>	
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>	

将实体类的属性类型改为 String，并指定主键生成策略为 IdType.ASSIGN_UUID

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    @TableId(type = IdType.ASSIGN_UUID)
    private String id;
    private String name;
    private Integer age;
    private String email;
}

```

完成数据的添加

```

@Test
void primaryKey(){
    User user = new User();
    user.setName("Jerry");
    user.setAge(38);
    user.setEmail("test8@powernode.com");
    userMapper.insert(user);
}

```

我们会发现，成功添加了一条数据，id 为 uuid 类型

id	name	age	email
056d1c968f8f48bc3b9618aedbed7556	Jerry	38	test8@powernode.c
1	Jone	18	test1@baomidou.co
2	Jack	20	test2@baomidou.co
3	Tom	22	test3@baomidou.co
4	Sandy	22	test4@baomidou.co
5	Billie	25	test5@baomidou.co
6	Mike	40	test6@powernode.c
7	Tom	35	test7@powernode.c

4.1.7 小结

本章节讲解了主键生成策略，我们可以通过指定主键生成策略来生成不同的主键 id，从而达到对于数据进行唯一标识的作用。

4.2 分页

分页操作在实际开发中非常的常见，我们在各种平台和网站中都可以看到分页的效果。

例如：京东商城的分页效果



例如：百度的分页效果



在 MybatisPlus 中我们如何配置分页呢？这里我们思考一下

在 MybatisPlus 中的查询语句是怎么实现的，我们可以通过两种方式实现查询语句

【1】通过 MybatisPlus 提供的方法来实现条件查询

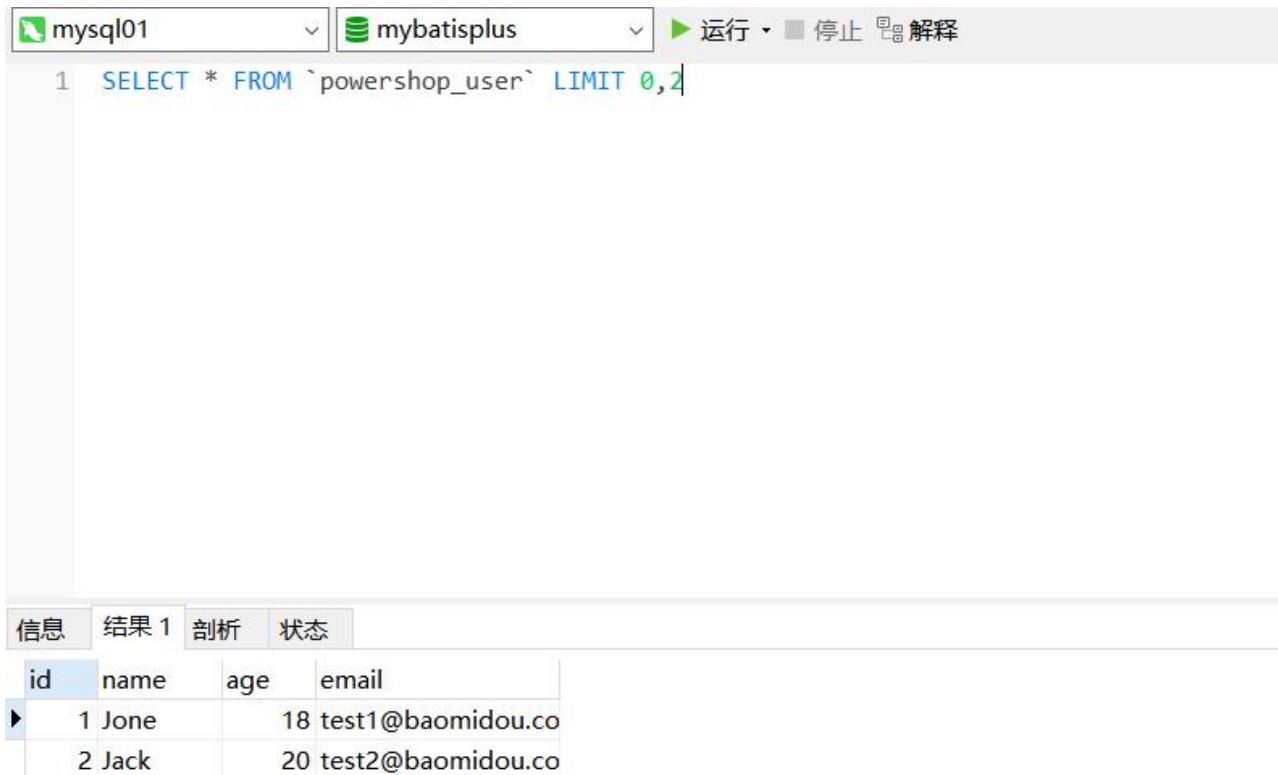
【2】通过自定义 SQL 语句的方式来实现查询

接下来我们就来演示这两种分页方式如何实现

4.2.1 分页插件

在大部分场景下，如果我们的 SQL 没有这么复杂，是可以直接通过 MybatisPlus 提供的方法来实现查询的，在这种情况下，我们可以通过配置分页插件来实现分页效果

分页的本质就是需要设置一个拦截器，通过拦截器拦截了 SQL，通过在 SQL 语句的结尾添加 limit 关键字，来实现分页的效果



The screenshot shows the MySQL Workbench interface. At the top, there are two dropdown menus: 'mysql01' and 'mybatisplus'. To the right of these are buttons for '运行' (Run), '停止' (Stop), and '解释' (Explain). Below the menu bar, a query is entered in the editor:

```
1 SELECT * FROM `powershop_user` LIMIT 0,2
```

Below the editor, the results are displayed in a table. The table has four columns: id, name, age, and email. There are two rows of data:

id	name	age	email
1	Jone	18	test1@baomidou.co
2	Jack	20	test2@baomidou.co

接下来看一下配置的步骤

【1】通过配置类来指定一个具体数据库的分页插件，因为不同的数据库的方言不同，具

体生成的分页语句也会不同，这里我们指定数据库为 Mysql 数据库

```
@Configuration
public class MybatisPlusConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new
PaginationInnerInterceptor(DbType.MYSQL));
        return interceptor;
    }
}
```

【2】实现分页查询效果

```
@Test
void selectPage(){
    //1.创建 QueryWrapper 对象
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //2.创建分页查询对象,指定当前页和每页显示条数
    IPage<User> page = new Page<>(1,3);
    //3.执行分页查询
    userMapper.selectPage(page, lambdaQueryWrapper);
    //4.查看分页查询的结果
    System.out.println("当前页码值: "+page.getCurrent());
    System.out.println("每页显示数: "+page.getSize());
    System.out.println("总页数: "+page.getPages());
    System.out.println("总条数: "+page.getTotal());
    System.out.println("当前页数据: "+page.getRecords());
}
```

```
==> Preparing: SELECT COUNT(*) FROM powershop_user
==> Parameters:
<==   Columns: COUNT(*)
<==   Row: 8
<==   Total: 1
==> Preparing: SELECT id,name,age,email FROM powershop_user LIMIT ?
==> Parameters: 3(Long)
<==   Columns: id, name, age, email
<==   Row: 1, Jone, 18, test1@baomidou.com
<==   Row: 2, Jack, 20, test2@baomidou.com
<==   Row: 3, Tom, 22, test3@baomidou.com
<==   Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@20ffb8d5]
当前页码值: 1
每页显示数: 3
总页数: 3
总条数: 8
当前页数据: [User(id=1, name=Jone, age=18, email=test1@baomidou.com), User(id=2, name=Jack, age=20, email=test2@baomidou.com), User(id=3, name=Tom, age=22, email=test3@baomidou.com)]
```

4.2.2 自定义分页插件

在某些场景下，我们需要自定义 SQL 语句来进行查询。接下来我们来演示一下自定义 SQL 的分页操作

【1】在 UserMapper.xml 映射配置文件中提供查询语句

```
<?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.powernode.mapper.UserMapper">

  <select id="selectByName" resultType="com.powernode.domain.User">
    select * from powershop_user where name = #{name}
  </select>

</mapper>
```

【2】在 Mapper 接口中提供对应的方法，方法中将 IPage 对象作为参数传入

```
@Mapper
public interface UserMapper extends BaseMapper<User> {
    IPage<User> selectByName(IPage<User> page, String name);
}
```

【3】表数据为

id	name	age	email
1	wang	18	wang@powernode.com
2	Jack	20	test2@baomidou.com
3	Tom	28	test3@baomidou.com
4	Sandy	20	test4@baomidou.com
5	Billie	24	test5@baomidou.com
6	Mary	35	mary@powernode.com
7	Mary	35	mary@powernode.com

【4】实现分页查询效果

```
@Test
void selectPage2(){
    //1. 创建分页查询对象, 指定当前页和每页显示条数
    IPage<User> page = new Page<>(1,2);
    //2. 执行分页查询
    userMapper.selectByName(page, "Mary");
    //3. 查看分页查询的结果
    System.out.println("当前页码值: "+page.getCurrent());
    System.out.println("每页显示数: "+page.getSize());
    System.out.println("总页数: "+page.getPages());
    System.out.println("总条数: "+page.getTotal());
    System.out.println("当前页数据: "+page.getRecords());
}
```

4.2.3 小结

这里我们学习了两种分页的配置方法，将来以后我们在进行条件查询的时候，可以使用分页的配置进行配置。

4.3 ActiveRecord 模式

4.3.1 ActiveRecord 介绍

ActiveRecord(活动记录，简称 AR)，是一种领域模型模式，特点是一个模型类对应关系

型数据库中的一个表，而模型类的一个实例对应表中的一行记录。ActiveRecord 一直广受解释型动态语言（PHP、Ruby 等）的喜爱，通过围绕一个数据对象进行 CRUD 操作。而 Java 作为为准静态（编译型）语言，对于 ActiveRecord 往往只能感叹其优雅，所以 MP 也在 AR 道路上进行了一定的探索，仅仅需要让实体类继承 Model 类且实现主键指定方法，即可开启 AR 之旅。

4.3.2 ActiveRecord 实现

接下来我们来看一下 ActiveRecord 的实现步骤

【1】让实体类继承 Model<User>类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User extends Model<User> {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

我们可以看到，Model 类中提供了一些增删改查方法，这样的话我们就可以直接使用实体类对象调用这些增删改查方法了，简化了操作的语法，但是他的底层依然是需要 UserMapper 的，所以持久层接口并不能省略

【2】测试 ActiveRecord 模式的增删改查

添加数据

```
@Test
void activeRecordAdd(){
    User user = new User();
    user.setName("wang");
    user.setAge(35);
    user.setEmail("wang@powernode.com");
    user.insert();
}
```

删除数据

```
@Test
void activeRecordDelete(){
    User user = new User();
    user.setId(8L);
    user.deleteById();
}
```

修改数据

```
@Test
void activeRecordUpdate(){
    User user = new User();
    user.setId(6L);
    user.setAge(50);
    user.updateById();
}
```

查询数据

```
@Test
void activeRecordSelect(){
    User user = new User();
    user.setId(6L);
    User result = user.selectById();
    System.out.println(result);
}
```

4.4 SimpleQuery 工具类

4.4.1 SimpleQuery 介绍

SimpleQuery 可以对 selectList 查询后的结果用 Stream 流进行了一些封装，使其可以返回一些指定结果，简洁了 api 的调用

4.4.2 list

演示基于字段封装集合

```
@Test
void testList(){
    List<Long> ids = SimpleQuery.list(new
LambdaQueryWrapper<User>().eq(User::getName, "Mary"), User::getId);
    System.out.println(ids);
}
```

演示对于封装后的字段进行 lambda 操作

```
@Test
void testList2(){
    List<String> names = SimpleQuery.list(new
LambdaQueryWrapper<User>().eq(User::getName, "Mary"), User::getName, e ->
Optional.of(e.getName()).map(String::toLowerCase).ifPresent(e::setName));
    System.out.println(names);
}
```

4.4.3 map

演示将所有的对象以 id, 实体的方式封装为 Map 集合

```
@Test
void testMap(){
    //将所有元素封装为 Map 形式
    Map<Long, User> idEntityMap = SimpleQuery.keyMap(
        new LambdaQueryWrapper<>(), User::getId);
    System.out.println(idEntityMap);
}
```

演示将单个对象以 id, 实体的方式封装为 Map 集合

```
@Test
void testMap2(){
    //将单个元素封装为 Map 形式
    Map<Long, User> idEntityMap = SimpleQuery.keyMap(
        new LambdaQueryWrapper<User>().eq(User::getId, 1L), User::getId);
    System.out.println(idEntityMap);
}
```

演示只想要 id 和 name 组成的 map

```
@Test
void testMap3(){
    //只想要只想要 id 和 name 组成的 map
    Map<Long, String> idNameMap = SimpleQuery.map(new LambdaQueryWrapper<>(),
User::getId, User::getName);
    System.out.println(idNameMap);
}
```

4.4.4 Group

演示分组效果

```
@Test
void testGroup(){
    Map<String, List<User>> nameUsersMap = SimpleQuery.group(new
LambdaQueryWrapper<>(), User::getName);
    System.out.println(nameUsersMap);
}
```

4.4.5 小结

在这一小节，我们演示了 SimpleQuery 提供的 Stream 流式操作的方法，通过这些操作继续为我们提供了查询方面简化的功能

5. 【拓展篇】

5.1 逻辑删除

前面我们完成了基本的增删改查操作，但是对于删除操作来说，我们思考一个问题，在实际开发中我们真的会将数据完成从数据库中删除掉么？

当然是不会的，这里我们举个例子：

在电商网站中，我们会上架很多商品，这些商品下架以后，我们如果将这些商品从数据库中删除，那么在年底统计商品数据信息的时候，这个商品要统计的，所以这个商品信息我们是不能删除的。

编号	销售数量	总金额	商品编号
1	5	10000	1
2	10	2000	2
3	20	5000	3
4	50	20000	4

编号	商品	单价
1	手机	2000
2	水果	200
3	火锅	250
4	化妆品	400

如果商城中的商品下架了，这时候我们将商品从数据库删掉

编号	销售数量	总金额	商品编号
1	5	10000	1
2	10	2000	2
3	20	5000	3
4	50	20000	4

编号	商品	单价
1	手机	2000
2	水果	200
3	火锅	250
4	化妆品	400

那到了年终总结的时候，我们要总结一下这一年的销售额，发现少了 20000，这肯定不合理。所以我们是不能将数据真实删除的。

这里我们就采取逻辑删除的方案，逻辑删除的操作就是增加一个字段表示这个数据的状态，如果一条数据需要删除，我们通过改变这条数据的状态来实现，这样既可以表示这条数据是删除的状态，又保留了数据以便以后统计，我们来实现一下这个效果。

【1】先在表中增加一列字段，表示是否删除的状态，这里我们使用的字段类型为 int 类型，通过 1 表示该条数据可用，0 表示该条数据不可用

名	类型	长度	小数点	不是 null	虚拟
id	bigint	64	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>
status	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>

【2】实体类添加一个字段为 Integer，用于对应表中的字段

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User extends Model<User> {
    private Long id;
    private String name;
    private Integer age;
}
    
```

```
private String email;
@TableLogic(value = "1", delval = "0")
private Integer status;
}
```

【3】测试逻辑删除效果

```
@Test
void logicDelete(){
    userMapper.deleteById(7L);
}
```

查看拼接的 SQL 语句，我们发现在执行删除操作的时候，语句变成了修改，是将这条数据的状态由 1 变为的 0，表示这条数据为删除状态

```
==> Preparing: UPDATE powershop_user SET status=0 WHERE id=? AND status=1
==> Parameters: 7(Long)
<==     Updates: 1
```

id	name	age	email	status
1	Jone	18	test1@baomidou.com	1
2	Jack	20	test2@baomidou.com	1
3	Tom	22	test3@baomidou.com	1
4	Sandy	22	test4@baomidou.com	1
5	Billie	25	test5@baomidou.com	1
6	Mike	50	test6@powernode.com	1
7	Tom	23	test7@powernode.com	0

我们还可以通过全局配置来实现逻辑删除的效果

```

spring:
  datasource:
    password: root
    username: root
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false
  main:
    banner-mode: off
  mybatis-plus:
    configuration:
      log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
      map-underscore-to-camel-case: false
    global-config:
      banner: false
      db-config:
        table-prefix: powershop_
        logic-delete-field: status
        logic-not-delete-value: 1
        logic-delete-value: 0
  
```

5.2 通用枚举

首先我们先来回顾一下枚举，什么是枚举呢？

当我们想要表示一组信息，这组信息只能从一些固定的值中进行选择，不能随意写，在这种场景下，枚举就非常的合适。

例如我们想要表示性别，性别只有两个值，要么是男性，要么是女性，那我们就可以使用枚举来描述性别。

【1】我们先在表中添加一个字段，表示性别，这里我们一般使用 int 来描述，因为 int 类型可以通过 0 和 1 这两个值来表示两个不同的性别

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	64	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	主键ID
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
gender	int	255	0	<input type="checkbox"/>	<input type="checkbox"/>		性别
status	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		状态

【2】编写枚举类

```
public enum GenderEnum {  
  
    MAN(0, "男"),  
    WOMAN(1, "女");  
  
    private Integer gender;  
    private String genderName;  
  
    GenderEnum(Integer gender, String genderName) {  
        this.gender = gender;  
        this.genderName = genderName;  
    }  
}
```

【3】实体类添加相关字段

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User extends Model<User> {  
    private Long id;  
    private String name;  
    private Integer age;  
    private String email;  
    private GenderEnum gender;  
    private Integer status;  
}
```

【4】添加数据

```
@Test  
void enumTest(){  
    User user = new User();  
    user.setName("liu");  
    user.setAge(29);  
    user.setEmail("liu@powernode.com");
```

```
user.setGenderEnum(GenderEnum.MAN);
user.setStatus(1);
userMapper.insert(user);
}
```

此时我们查看控制台，会发现添加失败了

```
org.springframework.jdbc.BadSqlGrammarException:
### Error updating database. Cause: java.sql.SQLSyntaxErrorException: Unknown column 'genderEnum' in 'field list'
### The error may exist in com/powernode/mapper/UserMapper.java (best guess)
### The error may involve com.powernode.mapper.UserMapper.insert-Inline
### The error occurred while setting parameters
### SQL: INSERT INTO powershop_user ( id, name, age, email, genderEnum, status ) VALUES ( ?, ?, ?, ?, ?, ? )
### Cause: java.sql.SQLSyntaxErrorException: Unknown column 'genderEnum' in 'field list'
; bad SQL grammar []; nested exception is java.sql.SQLSyntaxErrorException: Unknown column 'genderEnum' in 'field list'
```

原因是无法将一个枚举类型作为 int 数字插入到数据库中。不过我们对于枚举类型都给了对应的 int 的值，所以这里我们只需要进行一个配置，就可以将枚举类型作为数字插入到数据库中，为属性 gender，添加上@EnumValue 注解

```
public enum GenderEnum {
    MAN(0, "男"),
    WOMAN(1, "女");

    @EnumValue
    private Integer gender;
    private String genderName;

    GenderEnum(Integer gender, String genderName) {
        this.gender = gender;
        this.genderName = genderName;
    }
}
```

此时我们再次执行添加操作，发现可以成功添加数据，而枚举类型的值也作为数据被插入到数据库中

```

==> Preparing: INSERT INTO powershop_user ( id, name, age, email, gender, status ) VALUES ( ?, ?, ?, ?, ?, ? )
==> Parameters: 1615948685680631810(Long), liu(String), 29(Integer), liu@powernode.com(String), 0(Integer), 1(Integer)
<==    Updates: 1
    
```

id	name	age	email	gender	status
1	Jone	18	test1@baomidou.com	(Null)	1
2	Jack	20	test2@baomidou.com	(Null)	1
3	Tom	22	test3@baomidou.com	(Null)	1
4	Sandy	22	test4@baomidou.com	(Null)	1
5	Billie	25	test5@baomidou.com	(Null)	1
6	Mike	50	test6@powernode.com	(Null)	1
7	Tom	23	test7@powernode.com	(Null)	0
1615948685680631810	liu	29	liu@powernode.com	0	1

5.3 字段类型处理器

在某些场景下，我们在实体类中是使用 Map 集合作为属性接收前端传递过来的数据的，但是这些数据存储在数据库时，我们使用的是 json 格式的数据进行存储，json 本质是一个字符串，就是 varchar 类型。那怎么做到实体类的 Map 类型和数据库的 varchar 类型的互相转换，这里就需要使用到字段类型处理器来完成。

【1】我们先在实体类中添加一个字段，Map 类型

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User extends Model<User> {
    private Long id;
    private String name;
    private Integer age;
    private String email;
    private GenderEnum gender;
    private Integer status;
    private Map<String, String> contact;//联系方式
}
    
```

【2】在数据库中我们添加一个字段，为 varchar 类型

字段	索引	外键	触发器	选项	注释	SQL 预览
名						
id						
name						
age						
email						
gender						
status						
contact						

【3】为实体类添加上对应的注解，实现使用字段类型处理器进行不同类型数据转换

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@TableName(autoResultMap = true)//查询时将 json 字符串封装为 Map 集合
public class User extends Model<User> {
    private Long id;
    private String name;
    private Integer age;
    private String email;
    private GenderEnum gender;
    private Integer status;
    @TableField(typeHandler = FastjsonTypeHandler.class)//指定字段类型处理器
    private Map<String, String> contact;//联系方式
}
    
```

【4】字段类型处理器依赖 Fastjson 这个 Json 处理器，所以我们需要引入对应的依赖

```

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.76</version>
</dependency>
    
```

【5】测试添加操作

```
@Test
void typeHandler(){
    User user = new User();
    user.setName("zhang");
    user.setAge(28);
    user.setEmail("zhang@powernode.com");
    user.setGender(GenderEnum.MAN);
    user.setStatus(1);
    HashMap<String, String> contact = new HashMap<>();
    contact.put("phone", "010-1234567");
    contact.put("tel", "13388889999");
    user.setContact(contact);

    userMapper.insert(user);
}
```

执行的 SQL 语句如下

```
--> Preparing: INSERT INTO powershop_user ( id, name, age, email, gender, status, contact ) VALUES ( ?, ?, ?, ?, ?, ?, ? )
--> Parameters: 1617915941843202049(Long), zhang(String), 28(Integer), zhang@powernode.com(String), 0(Integer),
1(Integer), {"phone":"010-1234567","tel":"13388889999"}(String)
<==    Updates: 1
```

通过观察 SQL 语句，我们发现当插入一个 Map 类型的字段的时候，该字段会转换为 String 类型

查看数据库中的信息，发现添加成功

id	name	age	email	gender	status	contact
1	Jone	18	test1@baomidou.com	(Null)	1 (Null)	
2	Jack	20	test2@baomidou.com	(Null)	1 (Null)	
3	Tom	22	test3@baomidou.com	(Null)	1 (Null)	
4	Sandy	22	test4@baomidou.com	(Null)	1 (Null)	
5	Billie	25	test5@baomidou.com	(Null)	1 (Null)	
6	Mike	50	test6@powernode.com	(Null)	1 (Null)	
7	Tom	23	test7@powernode.com	(Null)	0 (Null)	
1615948685680631810	liu	29	liu@powernode.com	0	1 (Null)	
1617915941843202049	zhang	28	zhang@powernode.com	0	1 {"phone":"010-1234567","tel":"13388889999"}	

【6】测试查询操作，通过结果发现，从数据库中查询出来的数据，已经被转到 Map 集合

```

    @Test
    void typeHandlerSelect(){
        List<User> users = userMapper.selectList(null);
        System.out.println(users);
    }

```

```

==> Preparing: SELECT id,name,age,email,gender,status,contact FROM powershop_user WHERE status=1
==> Parameters:
<==   Columns: id, name, age, email, gender, status, contact
<==   Row: 1, Jone, 18, test1@baomidou.com, null, 1, null
<==   Row: 2, Jack, 20, test2@baomidou.com, null, 1, null
<==   Row: 3, Tom, 22, test3@baomidou.com, null, 1, null
<==   Row: 4, Sandy, 22, test4@baomidou.com, null, 1, null
<==   Row: 5, Billie, 25, test5@baomidou.com, null, 1, null
<==   Row: 6, Mike, 50, test6@powernode.com, null, 1, null
<==   Row: 1615948685680631810, liu, 29, liu@powernode.com, 0, 1, null
<==   Row: 1617915941843202049, zhang, 28, zhang@powernode.com, 0, 1, {"phone":"010-1234567","tel":"13388889999"}
<==   Total: 8

```

5.4 自动填充功能

在项目中有一些属性，如果我们不希望每次都填充的话，我们可以设置为自动填充，比如常见的`时间`，`创建时间和更新时间`可以设置为自动填充。

【1】在数据库的表中添加两个字段

名	类型	长度	小数位	个是否 null	虚拟	键	注释
<code>id</code>	<code>bigint</code>	64	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	主键ID
<code>name</code>	<code>varchar</code>	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
<code>age</code>	<code>int</code>	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
<code>email</code>	<code>varchar</code>	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
<code>gender</code>	<code>int</code>	255	0	<input type="checkbox"/>	<input type="checkbox"/>		性别
<code>status</code>	<code>int</code>	11	0	<input type="checkbox"/>	<input type="checkbox"/>		状态
<code>contact</code>	<code>varchar</code>	255	0	<input type="checkbox"/>	<input type="checkbox"/>		联系方式
<code>create_time</code>	<code>datetime</code>	0	0	<input type="checkbox"/>	<input type="checkbox"/>		创建时间
<code>update_time</code>	<code>datetime</code>	0	0	<input type="checkbox"/>	<input type="checkbox"/>		更新时间

注意只有设置了下划线和小驼峰映射，这种 mysql 的写法才能和实体类完成映射

`mybatis-plus:`

`configuration:`

`log-impl: org.apache.ibatis.logging.stdout.StdoutImpl`

`map-underscore-to-camel-case: true`

【2】在实体类中，添加对应字段，并为需要自动填充的属性指定填充时机

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName(autoResultMap = true)
public class User extends Model<User> {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    private Integer status;
    private GenderEnum gender;
    @TableField(typeHandler = FastjsonTypeHandler.class)
    private Map<String, String> contact;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;
}
```

【3】编写自动填充处理器，指定填充策略

```
@Component
public class MyMetaHandler implements MetaObjectHandler {
    @Override
    public void insertFill(MetaObject metaObject) {
        setFieldValByName("createTime", new Date(), metaObject);
        setFieldValByName("updateTime", new Date(), metaObject);
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        setFieldValByName("updateTime", new Date(), metaObject);
    }
}
```

【4】这里在插入前先设置一下 mysql 时区

```
set GLOBAL time_zone = '+8:00'
select NOW()
```

通过查看发现，目前时区时间正常

NOW()

▶ 2023-01-28 00:00:18

【5】再将配置文件的时区修改为 serverTimezone=Asia/Shanghai

```
spring:
  datasource:
    password: root
    username: root
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=false
```

【6】测试插入操作

```
@Test
void testFill(){
    User user = new User();
    user.setName("wang");
    user.setAge(35);
    user.setEmail("wang@powernode.com");
    user.setGender(GenderEnum.MAN);
    user.setStatus(1);
    HashMap<String, String> contact = new HashMap<>();
    contact.put("phone", "010-1234567");
    contact.put("tel", "13388889999");
    userMapper.insert(user);
}
```

name	age	email	gender	status	contact	create_time	update_time
Jone	18	test1@baomidou.com	(Null)	1 (Null)		(Null)	(Null)
Jack	20	test2@baomidou.com	(Null)	1 (Null)		(Null)	(Null)
Tom	22	test3@baomidou.com	(Null)	1 (Null)		(Null)	(Null)
Sandy	22	test4@baomidou.com	(Null)	1 (Null)		(Null)	(Null)
Billie	25	test5@baomidou.com	(Null)	1 (Null)		(Null)	(Null)
Mike	50	test6@powernode.com	(Null)	1 (Null)		(Null)	(Null)
Tom	23	test7@powernode.com	(Null)	0 (Null)		(Null)	(Null)
liu	29	liu@powernode.com	0	1 (Null)		(Null)	(Null)
zhang	28	zhang@powernode.com	0	1 {"phone":"010-1234567","tel":"13388889999"}		(Null)	(Null)
wang	50	wang@powernode.com	0	1 (Null)		2023-01-28 00:14:31	2023-01-28 00:14:31

【7】测试更新操作

```
@Test
void testFill2(){
    User user = new User();
    user.setId(6L);
    user.setName("wang");
    user.setAge(35);
    user.setEmail("wang@powernode.com");
    user.setGender(GenderEnum.MAN);
    user.setStatus(1);
    HashMap<String, String> contact = new HashMap<>();
    contact.put("phone", "010-1234567");
    contact.put("tel", "13388889999");

    userMapper.updateById(user);
}
```

Jone	18	test1@baomidou.com	(Null)	1	(Null)	(Null)	(Null)
Jack	20	test2@baomidou.com	(Null)	1	(Null)	(Null)	(Null)
Tom	22	test3@baomidou.com	(Null)	1	(Null)	(Null)	(Null)
Sandy	22	test4@baomidou.com	(Null)	1	(Null)	(Null)	(Null)
Billie	25	test5@baomidou.com	(Null)	1	(Null)	(Null)	(Null)
Mike	50	test6@powernode.com	(Null)	1	(Null)	(Null)	(Null)
Tom	23	test7@powernode.com	(Null)	0	(Null)	(Null)	(Null)
liu	29	liu@powernode.com	0	1	(Null)	(Null)	(Null)
zhang	28	zhang@powernode.com	0	1	{"phone":"010-1234567","tel":"13388889999"}	(Null)	(Null)
wang	50	wang@powernode.com	0	1	(Null)	2023-01-28 00:14:31	2023-01-28 00:22:37

5.5 防全表更新与删除插件

在实际开发中，全表更新和删除是非常危险的操作，在 MybatisPlus 中，提供了插件和防止这种危险操作的发生

先演示一下全表更新的场景

```
@Test
public void testUpdateAll(){
    User user = new User();
    user.setGender(GenderEnum.MAN);
    userService.saveOrUpdate(user, null);
}
```

id	name	age	email	status	gender	contact
1	wang	18	wang@powernode.com	1	0	(Null)
2	Jack	20	test2@baomidou.com	1	0	(Null)
3	Tom	28	test3@baomidou.com	1	0	(Null)
4	Mary	20	test4@baomidou.com	1	0	(Null)
5	Billie	24	test5@baomidou.com	1	0	(Null)
6	Mary	35	mary@powernode.com	1	0	(Null)
1626897530392526849	wangwu	39	wang@powernode.com	1	0	(Null)

这是很危险的

如何解决呢？

注入 MybatisPlusInterceptor 类，并配置 BlockAttackInnerInterceptor 拦截器

```
@Configuration
public class MybatisPlusConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));
        interceptor.addInnerInterceptor(new BlockAttackInnerInterceptor());
        return interceptor;
    }
}
```

测试全表更新，会出现抛出异常，防止了全表更新

```
@SpringBootTest
public class QueryTest {

    @Autowired
    private UserService userService;

    @Test
    void allUpdate(){
        User user = new User();
        user.setId(999L);
        user.setName("wang");
        user.setEmail("wang@powernode.com");
    }
}
```

```
userService.saveOrUpdate(user,null);
}
}
```

```
JDBC Connection [HikariProxyConnection@392226196 wrapping com.mysql.cj.jdbc.ConnectionImpl@4af70b83] will not be managed by Spring
original SQL: UPDATE powershop_user SET name=?,
email=?
SQL to parse, SQL: UPDATE powershop_user SET name=?,
email=?
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3407aa4f]

org.mybatis.spring.MyBatisSystemException: nested exception is org.apache.ibatis.exceptions.PersistenceException:
### Error updating database. Cause: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException: Prohibition of table update operation
### The error may exist in com/powernode/mapper/UserMapper.java (best guess)
### The error may involve com.powernode.mapper.UserMapper.update
### The error occurred while executing an update
### Cause: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException: Prohibition of table update operation
```

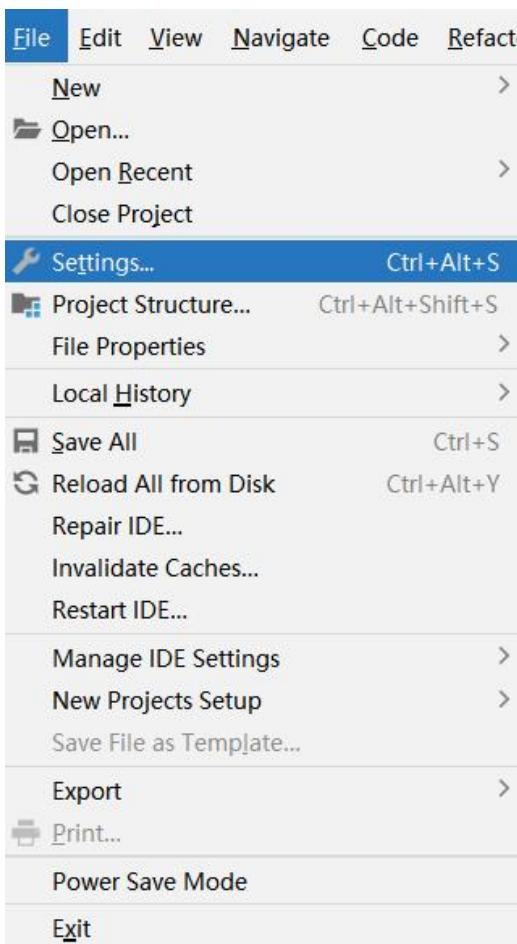
5.6 MybatisX 快速开发插件

5.6.1 安装

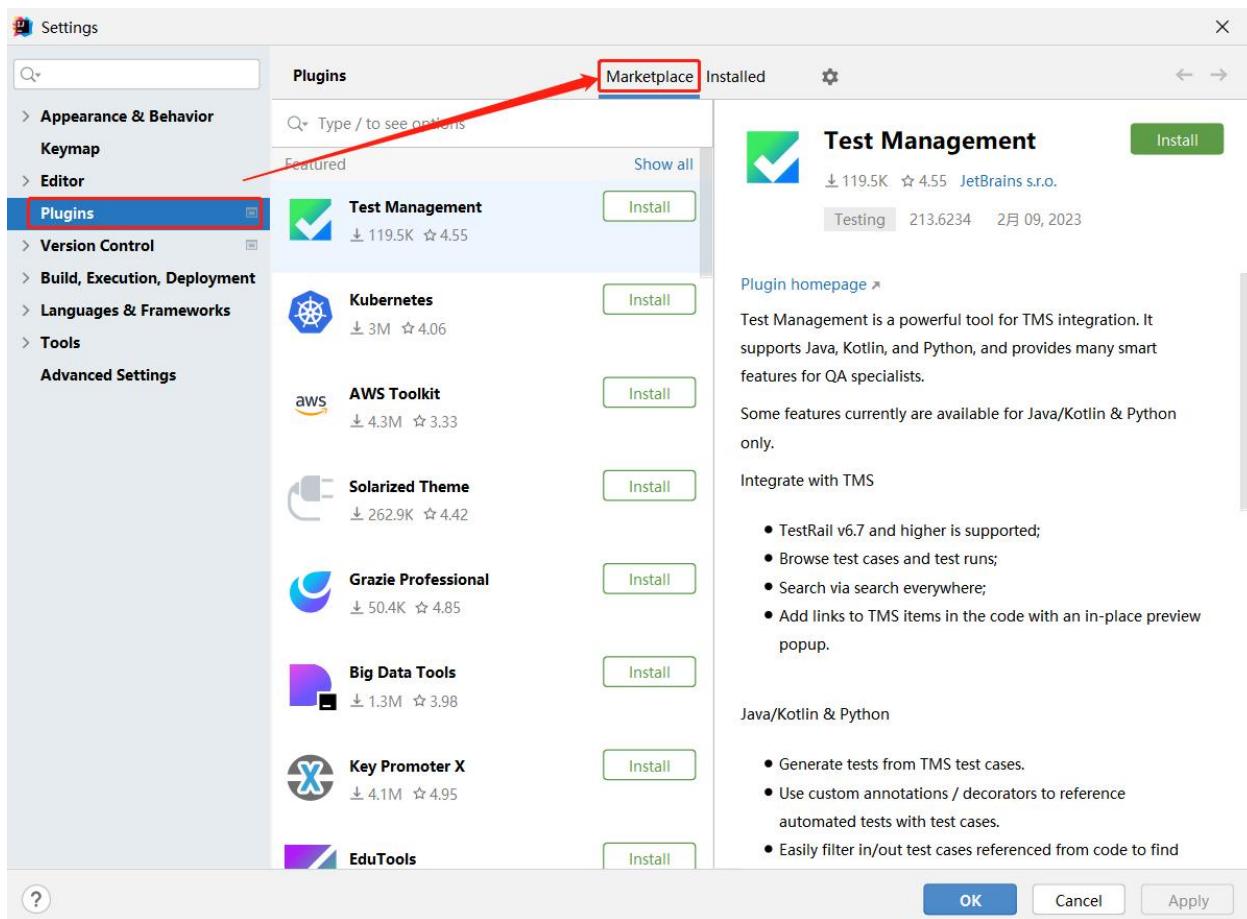
MybatisX 是一款 IDEA 提供的插件, 目的是为了我们简化 Mybatis 以及 MybatisPlus 框架而生。

我们来看一下, 如何在 IDEA 中安装插件

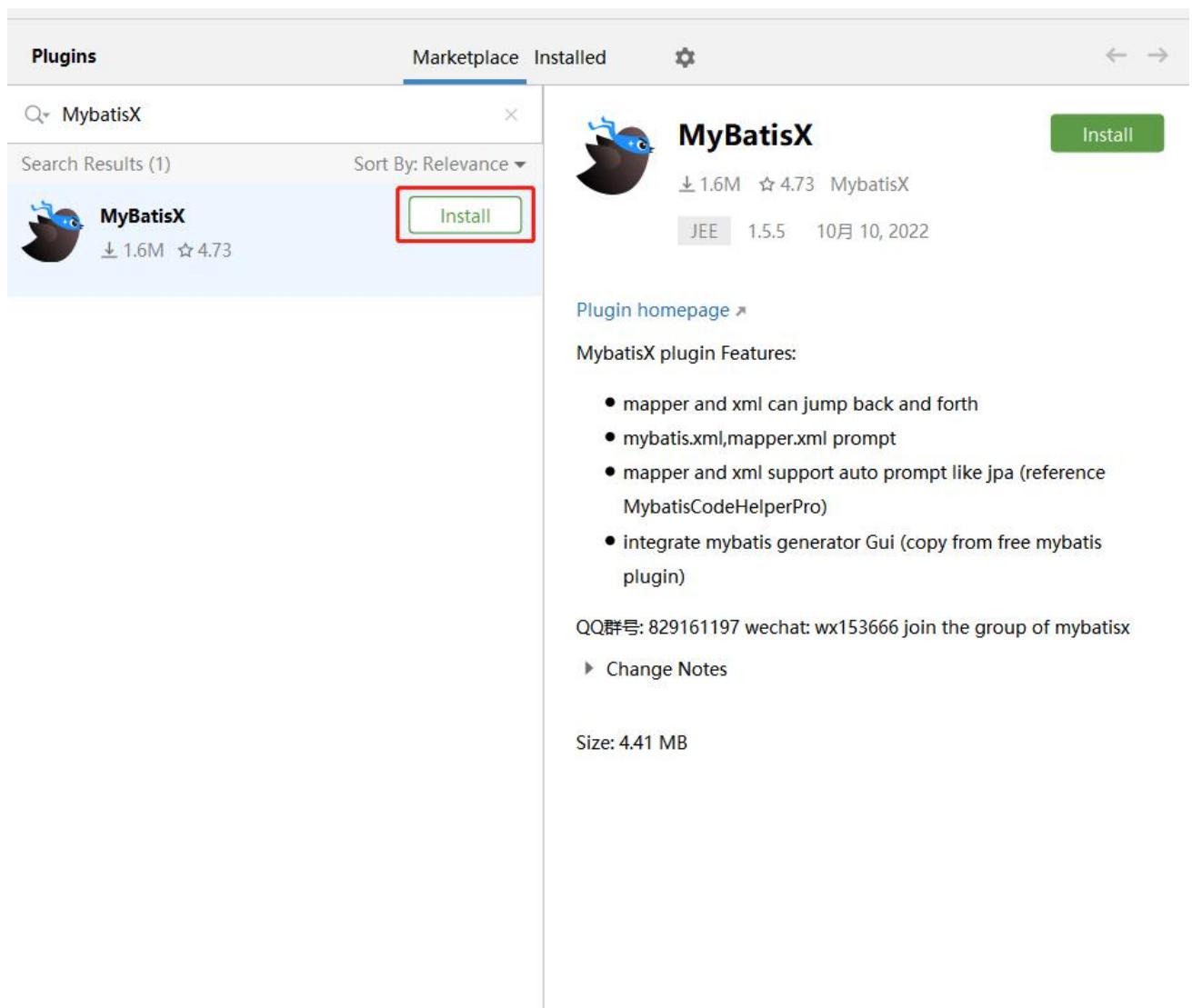
【1】首先选择 File -> Settings



【2】选择 Plugins



【3】搜索 MybatisX，点击安装



The screenshot shows the Power Node plugin marketplace interface. On the left, a search results panel displays one result for 'MybatisX'. On the right, the detailed view for the 'MyBatisX' plugin is shown. The plugin icon is a black bird with blue wings. It has a download count of 1.6M and a rating of 4.73. The version is 1.5.5, released on October 10, 2022, for JEE. A green 'Install' button is highlighted with a red box. Below the plugin details, there's a 'Plugin homepage' link, a section for 'MybatisX plugin Features' listing several bullet points, and a note about a QQ group and WeChat. The size of the plugin is 4.41 MB.

Plugins Marketplace Installed ↶ ↷

Q MybatisX ×

Search Results (1) Sort By: Relevance ▾

 MyBatisX
↓ 1.6M ☆ 4.73

MyBatisX

↓ 1.6M ☆ 4.73 MybatisX

JEE 1.5.5 10月 10, 2022

Plugin homepage ×

MybatisX plugin Features:

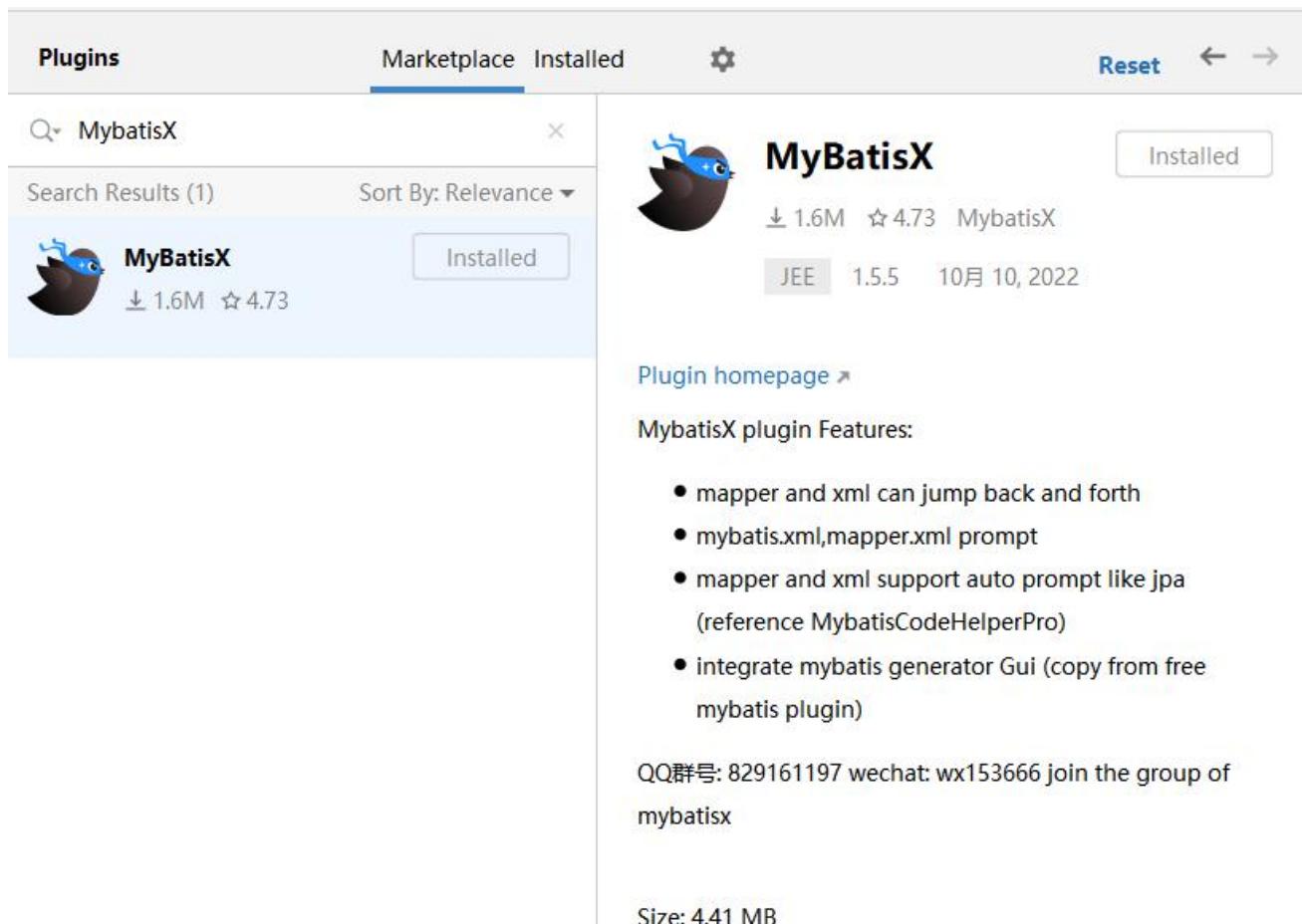
- mapper and xml can jump back and forth
- mybatis.xml,mapper.xml prompt
- mapper and xml support auto prompt like jpa (reference MybatisCodeHelperPro)
- Integrate mybatis generator Gui (copy from free mybatis plugin)

QQ群号: 829161197 wechat: wx153666 join the group of mybatisx

▶ Change Notes

Size: 4.41 MB

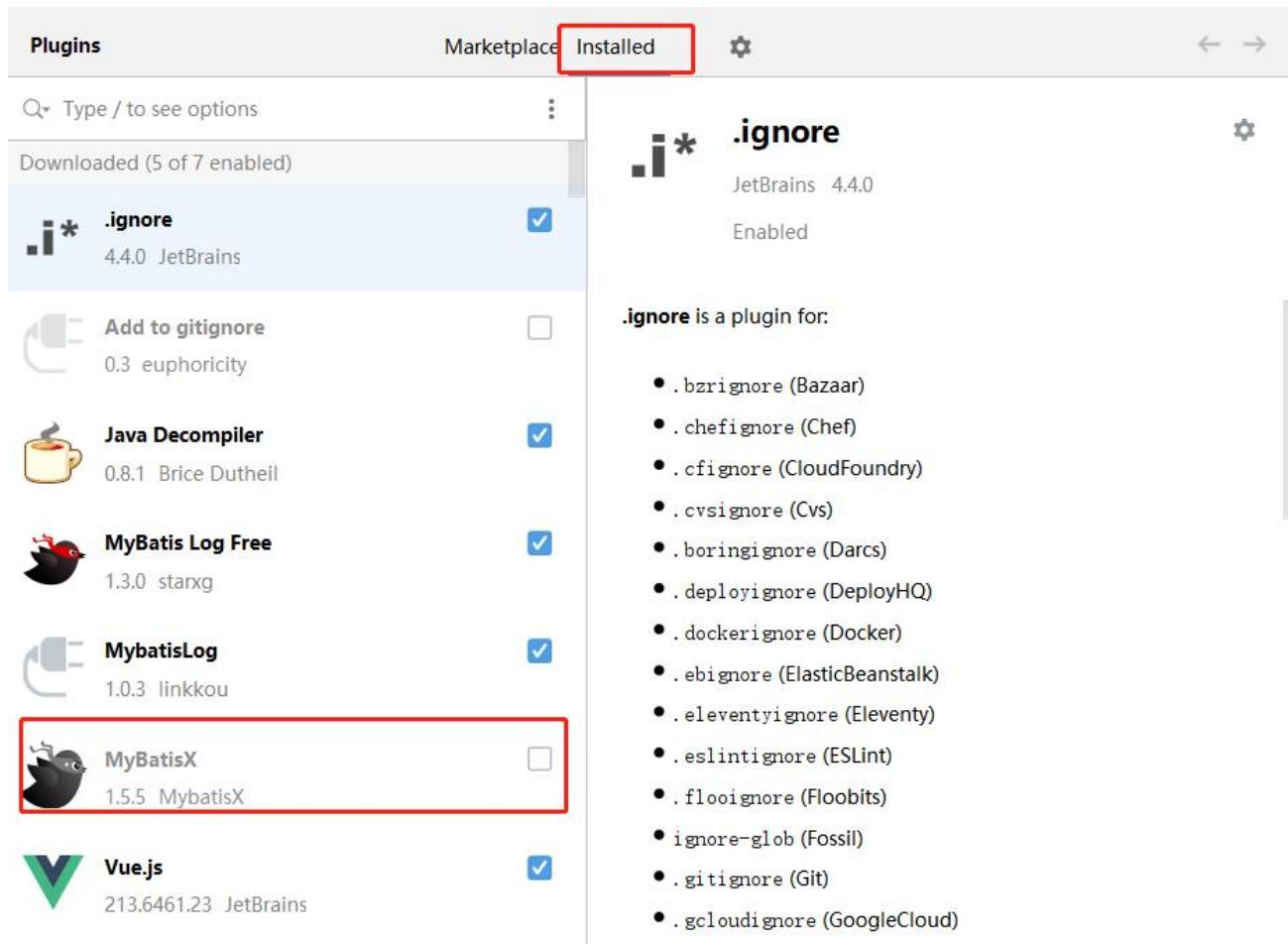
【4】这种效果就是安装完毕了



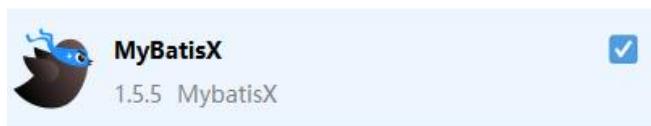
The screenshot shows the IDEA Marketplace interface. At the top, there are tabs for 'Plugins', 'Marketplace' (which is selected), and 'Installed'. Below the tabs, a search bar contains the text 'MybatisX'. The results list shows one item: 'MyBatisX' by 'MybatisX'. The item details page on the right shows the following information:

- Icon: A black bird icon with blue highlights.
- Name: MyBatisX
- Downloads: 1.6M
- Rating: 4.73
- Category: MybatisX
- Version: 1.5.5
- Release Date: 10月 10, 2022
- Plugin homepage: [Plugin homepage ↗](#)
- Features:
 - mapper and xml can jump back and forth
 - mybatis.xml,mapper.xml prompt
 - mapper and xml support auto prompt like jpa (reference MybatisCodeHelperPro)
 - integrate mybatis generator Gui (copy from free mybatis plugin)
- QQ群号: 829161197 wechat: wx153666 join the group of mybatisx
- Size: 4.41 MB

【5】在已安装的目录下，我们可以看到有了这个IDEA插件



【6】此时我们勾选他，表示启用

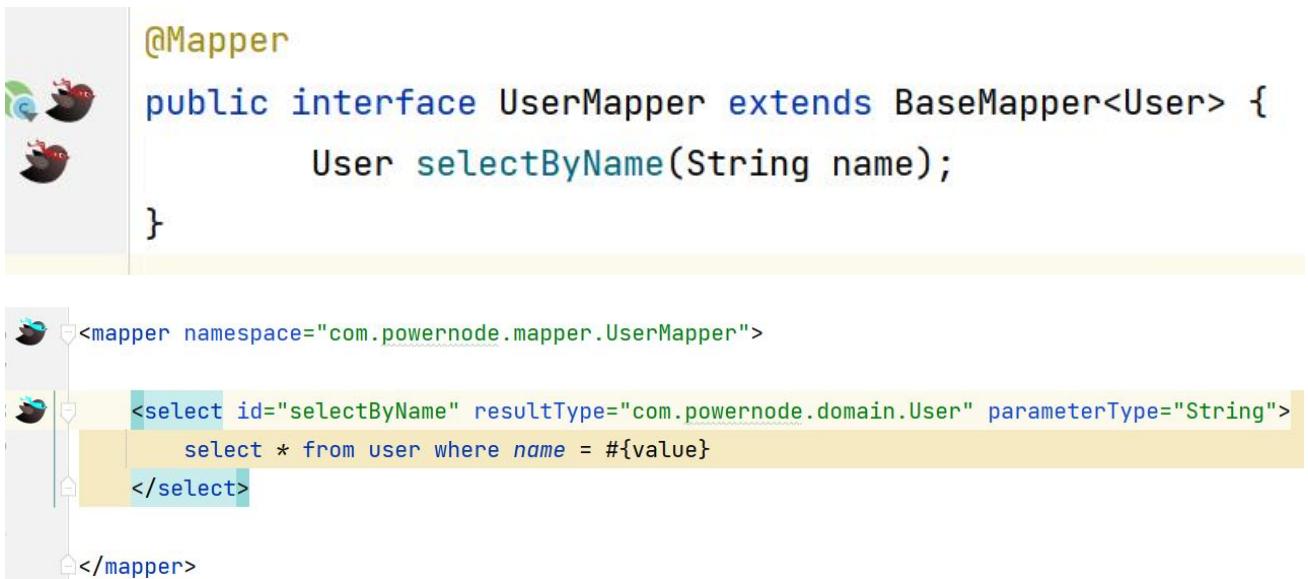


【7】重启 IDEA，让该插件生效，至此 MybatisX 插件就安装完毕了

5.6.2 功能

插件安装好以后，我们来看一下插件的功能

1、Mapper 接口和映射文件的跳转功能



The screenshot shows a code editor with two files. On the left is a Java interface file:

```
@Mapper
public interface UserMapper extends BaseMapper<User> {
    User selectByName(String name);
}
```

On the right is an XML configuration file:

```
<mapper namespace="com.powernode.mapper.UserMapper">
    <select id="selectByName" resultType="com.powernode.domain.User" parameterType="String">
        select * from user where name = #{value}
    </select>
</mapper>
```

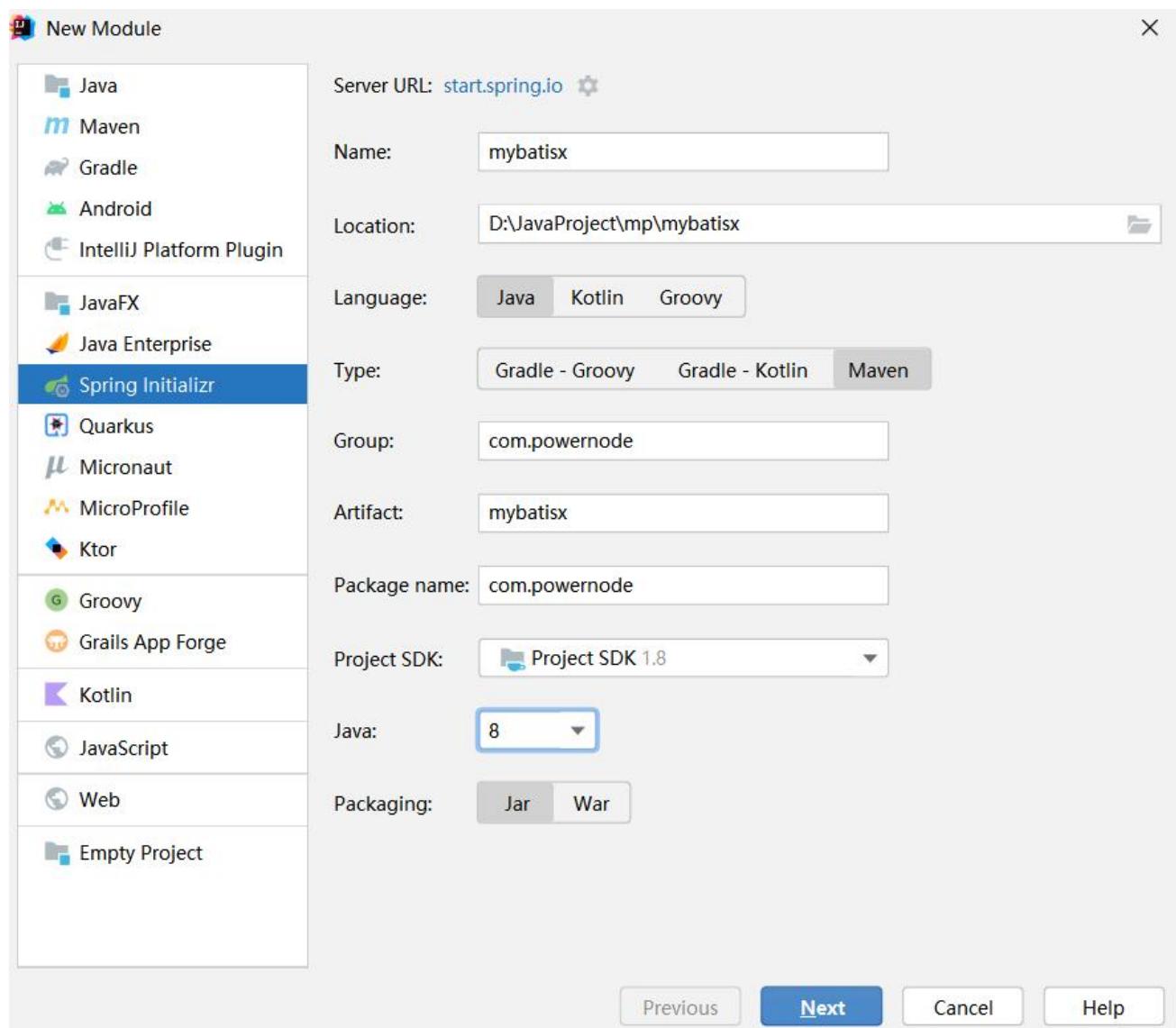
2、逆向工程

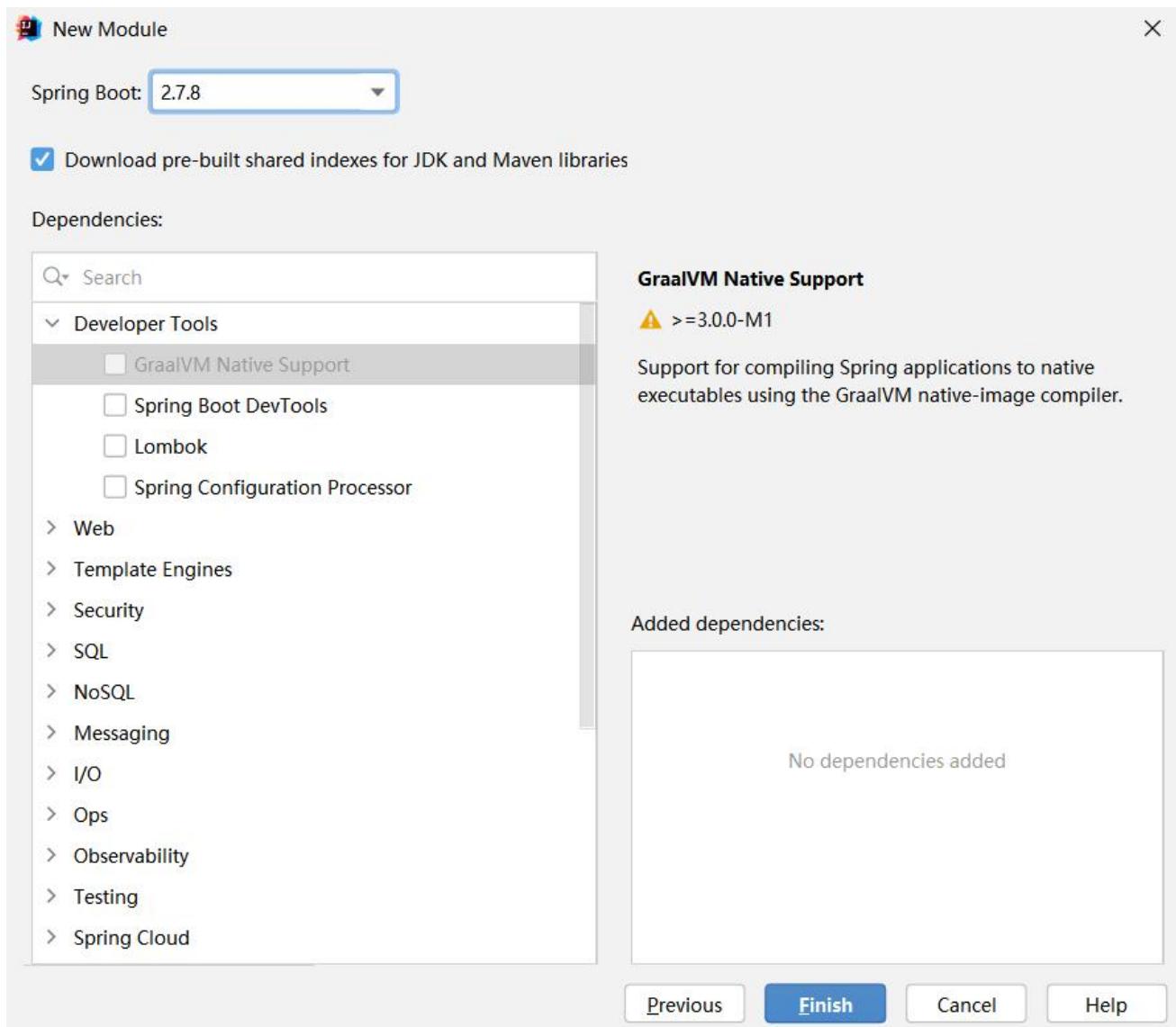
逆向工程就是通过数据库表结构，逆向产生 Java 工程的结构

包括以下几点：

- (1) 实体类
- (2) Mapper 接口
- (3) Mapper 映射文件
- (4) Service 接口
- (5) Service 实现类

在这里我们创建一个模块，用于测试逆向工程功能





引入依赖，和编写对应的配置文件信息

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
```

```
<scope>test</scope>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.31</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>

<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3</version>
</dependency>

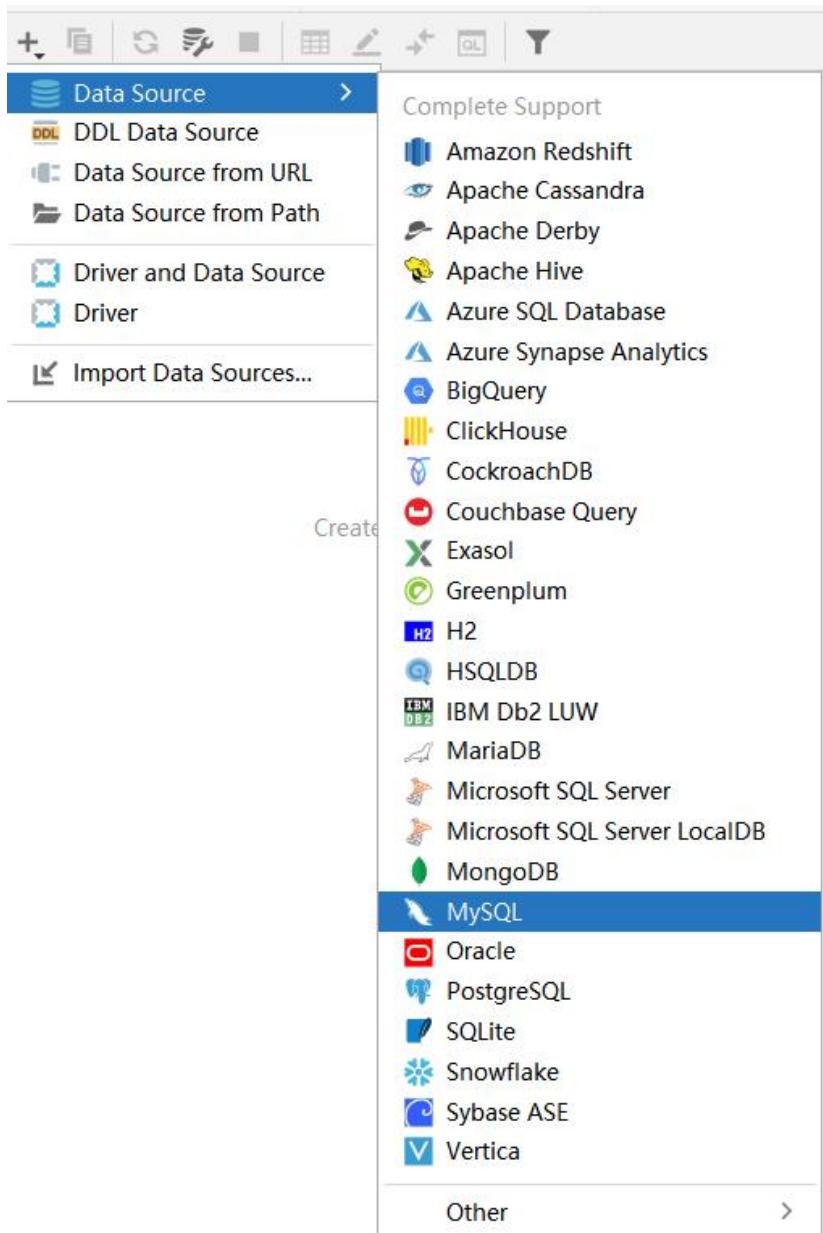
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>
```

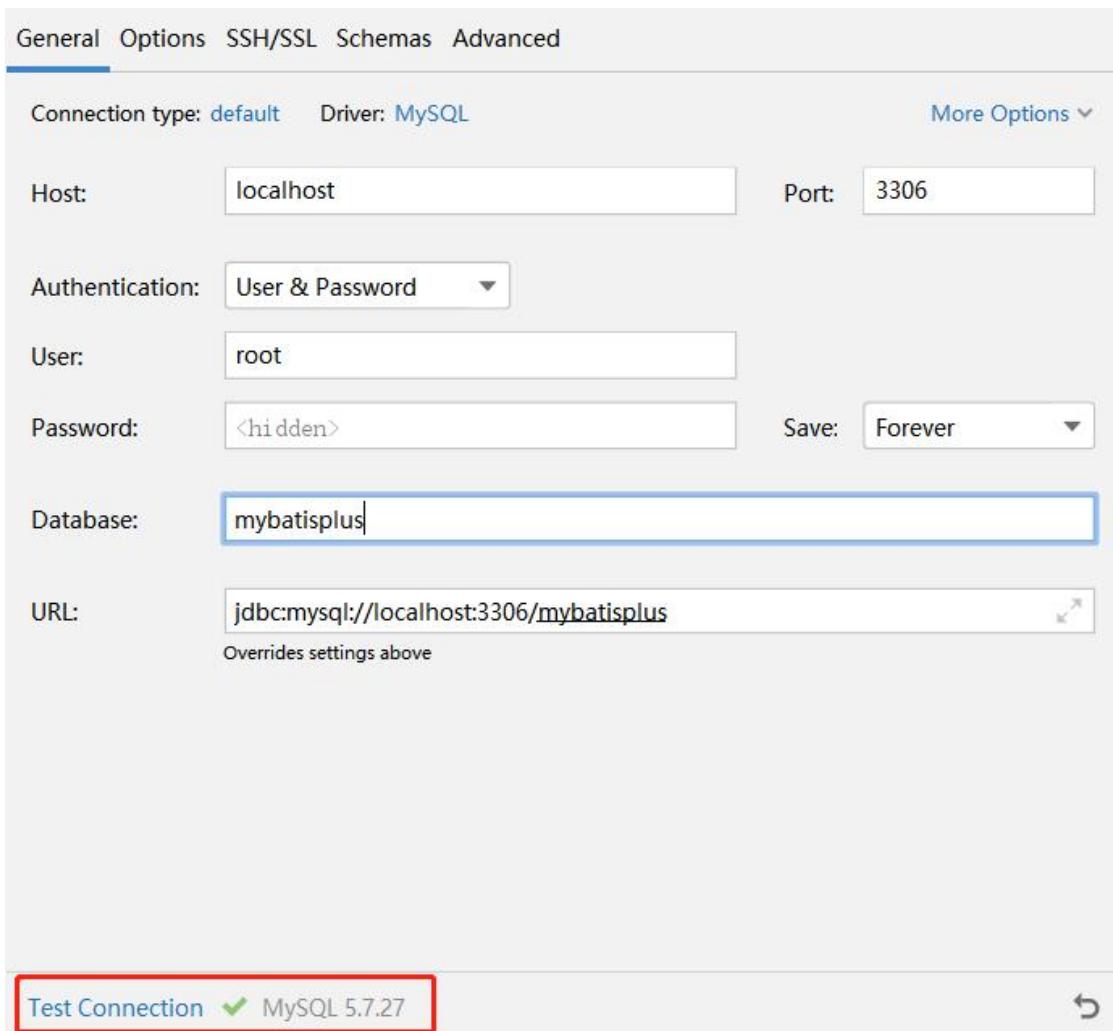
```
spring:
  datasource:
    username: root
    password: root
    url:
      jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUni
```

```
code=true&useSSL=false  
driver-class-name: com.mysql.cj.jdbc.Driver
```

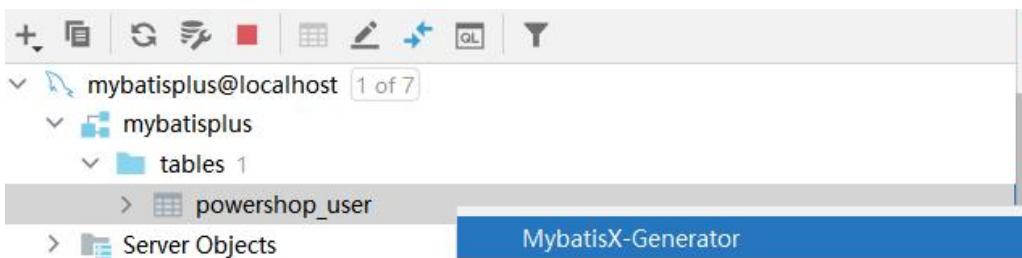
【1】首先使用 IDEA 连接 mysql



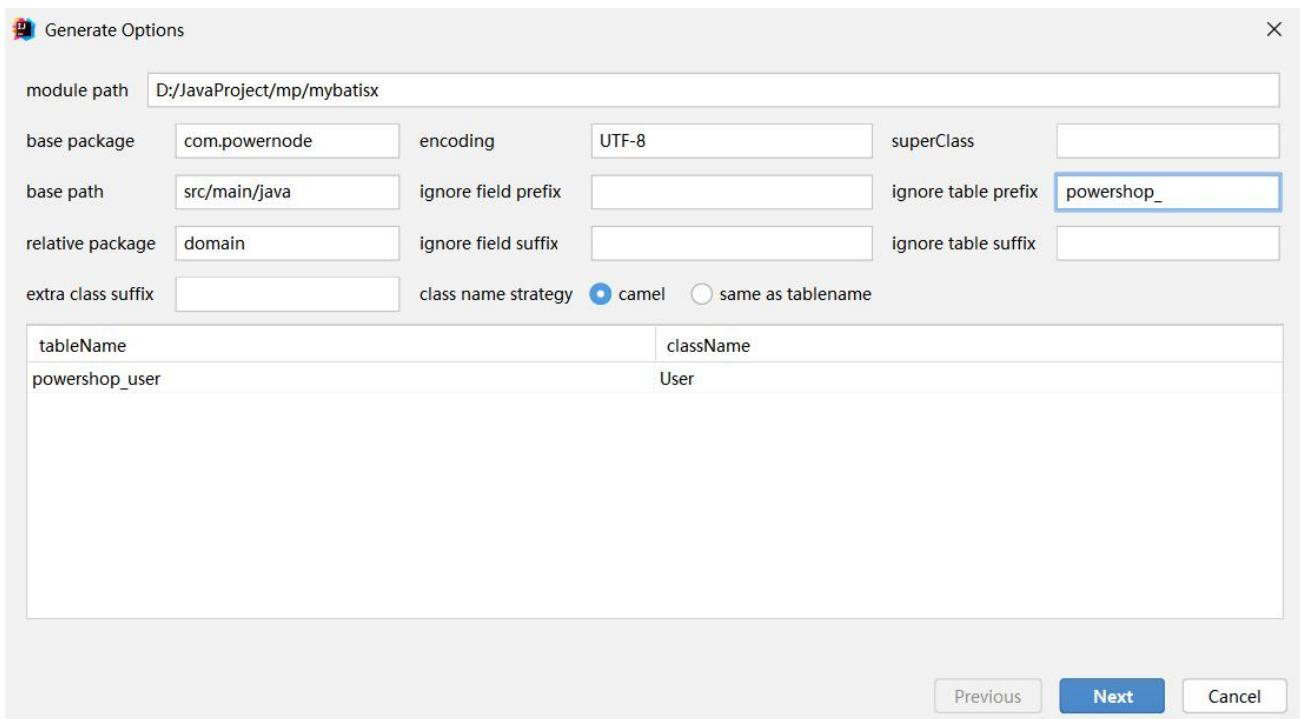
【2】填写连接信息，测试连接通过



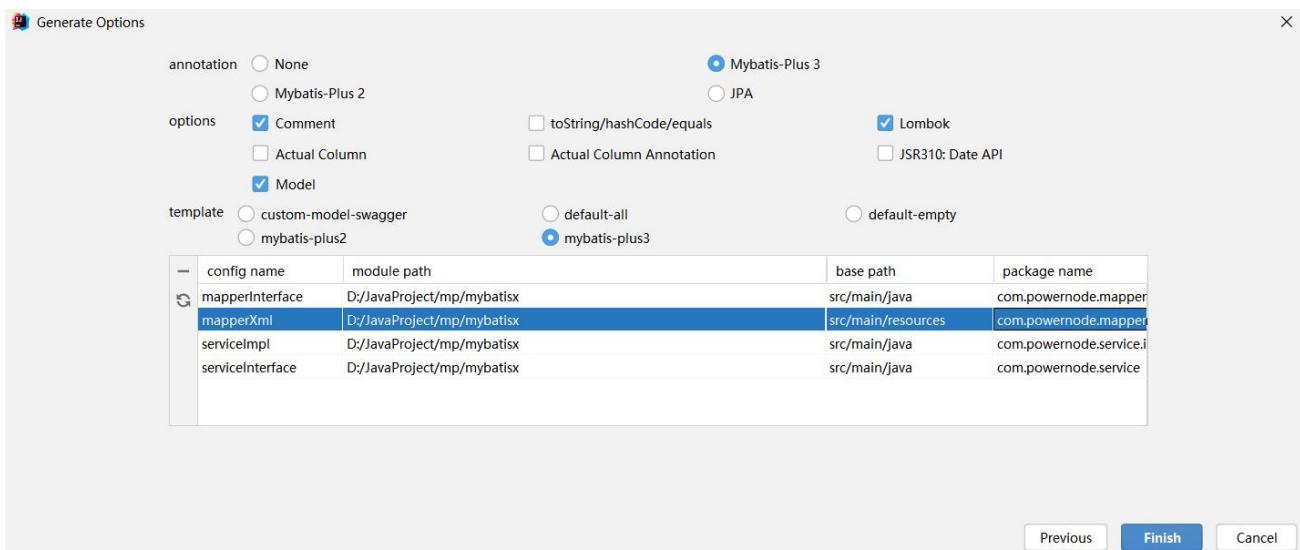
【3】找到表右键，选择插件的逆向工程选项



【4】编写逆向工程配置信息



【5】编写生成信息



【6】观察生成结构，发现下面这些内容都已产生

- (1) 实体类
- (2) Mapper 接口
- (3) Mapper 映射文件

(4) Service 接口

(5) Service 映射文件

【7】接下来我们在 Mapper 接口上添加@Mapper 注解

```
@Mapper
public interface UserMapper extends BaseMapper<User> {
}
```

【8】测试代码环境

```
3.5.3
2023-02-10 11:37:44.025 INFO 68488 --- [           main] com.powernode.MybatisxApplicationTests : Started MybatisxApplicationTests in 3.
2023-02-10 11:37:44.324 INFO 68488 --- [           main] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Starting...
2023-02-10 11:37:44.509 INFO 68488 --- [           main] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Start completed.
[User(id=1, name=wang, age=18, email=wang@powernode.com), User(id=2, name=Jack, age=20, email=test2@baomidou.com), User(id=3, name=Tom, age=22)]
2023-02-10 11:37:44.583 INFO 68488 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown initiated...
2023-02-10 11:37:44.590 INFO 68488 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown completed.
Disconnected from the target VM, address: '127.0.0.1:52152', transport: 'socket'
```

3、常见需求代码生成

虽然 Mapper 接口中提供了一些常见方法，我们可以直接使用这些常见的方法来完成 sql 操作，但是对于实际场景中各种复杂的操作需求来说，依然是不够用的，所以 MybatisX 提供了更多的方法，以及可以根据这些方法直接生成对应的 sql 语句，这样使得开发变得更加的简单。

可以根据名称联想常见的操作

```
@Mapper
public interface UserMapper extends BaseMapper<User> {
    //添加操作
    int insertSelective(User user);

    //删除操作
    int deleteByNameAndAge(@Param("name") String name, @Param("age") Integer age);
```

```
//修改操作
int updateNameByAge(@Param("name") String name, @Param("age") Integer age);

//查询操作
List<User> selectAllByAgeBetween(@Param("beginAge") Integer beginAge,
@Param("endAge") Integer endAge);
}
```

在映射配置文件中，会生成对应的sql，并不需要我们编写

```
<insert id="insertSelective">
    insert into powershop_user
    <trim prefix "(" suffix ")" suffixOverrides=",">
        <if test="id != null">id,</if>
        <if test="name != null">name,</if>
        <if test="age != null">age,</if>
        <if test="email != null">email,</if>
    </trim>
    values
    <trim prefix "(" suffix ")" suffixOverrides=",">
        <if test="id != null">#{id,jdbcType=BIGINT},</if>
        <if test="name != null">#{name,jdbcType=VARCHAR},</if>
        <if test="age != null">#{age,jdbcType=INTEGER},</if>
        <if test="email != null">#{email,jdbcType=VARCHAR},</if>
    </trim>
</insert>

<delete id="deleteByNameAndAge">
    delete
    from powershop_user
    where name = #{name,jdbcType=VARCHAR}
        AND age = #{age,jdbcType=NUMERIC}
</delete>

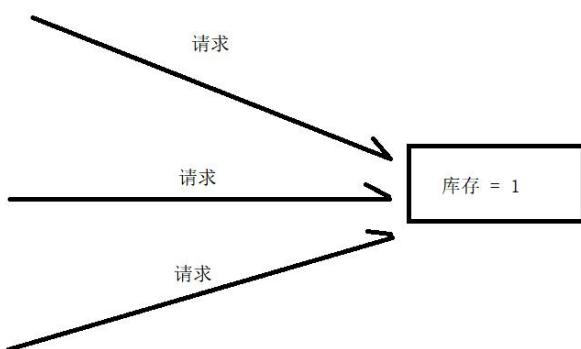
<update id="updateNameByAge">
    update powershop_user
    set name = #{name,jdbcType=VARCHAR}
```

```
where age = #{age,jdbcType=NUMERIC}  
</update>  
  
<select id="selectAllByAgeBetween" resultMap="BaseResultMap">  
    select  
    <include refid="Base_Column_List"/>  
    from powershop_user  
    where  
        age between #{beginAge,jdbcType=INTEGER} and #{endAge,jdbcType=INTEGER}  
</select>
```

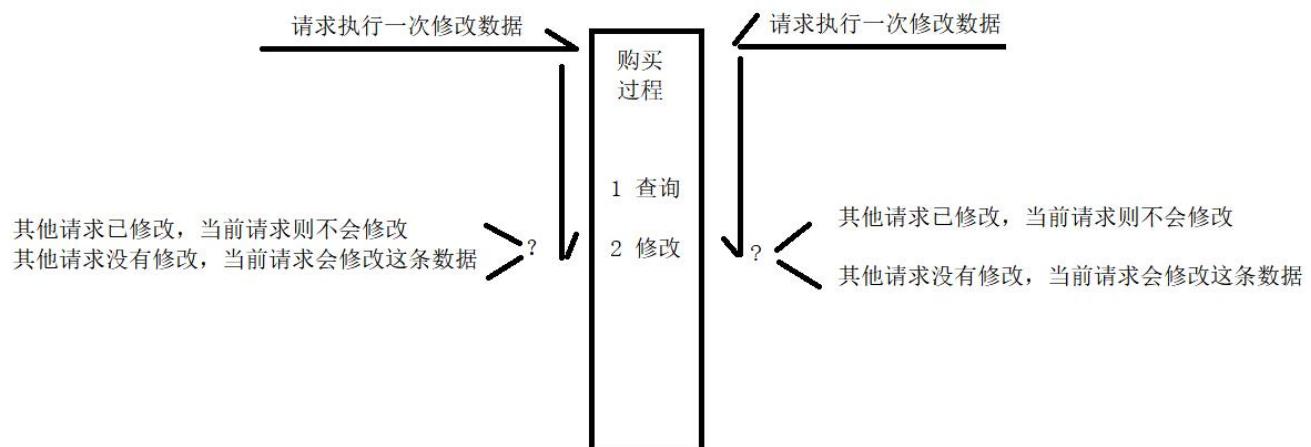
5.7 乐观锁

首先我们先要了解开发中的一个常见场景，叫做并发请求。并发请求就是在同一时刻有多少个请求同时请求服务器资源，如果是获取信息，没什么问题，但是如果对于信息做修改操作呢，那就会出现问题。

这里举一个具体的例子，比如目前商品的库存只剩余 1 件了，这个时候有多个用户都想要购买这件商品，都发起了购买商品的请求，那么能让这多个用户都购买到么，肯定是不行的，因为多个用户都买到了这件商品，那么就会出现超卖问题，库存不够是没法发货的。所以在开发中就要解决这种超卖的问题



抛开超卖这一种场景，诸如此类并发访问的场景非常多，这类场景的核心问题就是，一个请求在执行的过程中，其他请求不能改变数据，如果是一次完整的请求，在该请求的过程中其他请求没有对于这个数据产生修改操作，那么这个请求是能够正常修改数据的。如果该请求在改变数据的过程中，已经有其他请求改变了数据，那该请求就不去改变这条数据了。



想要解决这类问题，最常见的就是加锁的思想，锁可以用验证在请求的执行过程中，是否有数据发生改变。

常见的数据库锁类型有两种，悲观锁和乐观锁。

一次完成的修改操作是，先查询数据，然后修改数据。

悲观锁：悲观锁是在查询的时候就锁定数据，在这次请求未完成之前，不会释放锁。等到这次请求完毕以后，再释放锁，释放了锁以后，其他请求才可以对于这条数据完成读写。

这样做的操作能够保证读取到的信息就是当前的信息，保证了信息的正确性，但是并发效率很低，在实际开发中使用悲观锁的场景很少，因为在并发时我们是要保证效率的。

乐观锁：乐观锁是通过表字段完成设计的，他的核心思想是，在读取的时候不加锁，其他请求依然可以读取到这个数据，在修改的时候判断一个数据是否有被修改过，如果有被修改过，那本次请求的修改操作失效。

具体的通过 sql 是这样实现的

```
Update 表 set 字段 = 新值, version = version + 1 where version = 1
```

这样做的操作是不会对于数据读取产生影响，并发的效率较高。但是可能目前看到的数据并不是真实信息数据，是被修改之前的，但是在很多场景下是可以容忍的，并不是产生很大影响，例如很多时候我们看到的是有库存，或者都加入到购物车了，但是点进去以后库存没有了。

接下来我们来看一下乐观锁的使用

【1】在数据库表中添加一个字段 `version`, 表示版本，默认值是 1

名	类型	长度	小数点	不是 null	虚拟	键	注释
<code>id</code>	<code>bigint</code>	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	主键ID
<code>name</code>	<code>varchar</code>	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
<code>age</code>	<code>int</code>	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
<code>email</code>	<code>varchar</code>	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
<code>version</code>	<code>int</code>	11		<input type="checkbox"/>	<input type="checkbox"/>		版本

生成后的效果

<code>id</code>	<code>name</code>	<code>age</code>	<code>email</code>	<code>version</code>
1	wang	18	wang@powernode.com	1
2	Jack	20	test2@baomidou.com	1
3	Tom	28	test3@baomidou.com	1
4	Sandy	21	test4@baomidou.com	1
5	Billie	24	test5@baomidou.com	1
6	乔丹	40	test6@powernode.com	1

【2】找到实体类，添加对应的属性，并使用`@Version` 标注为这是一个乐观锁字段信息

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class User {  
    private Long id;  
    private String name;  
    private Integer age;  
    private String email;  
    @Version  
    private Integer version;  
}
```

【3】因为要对每条修改语句完成语句的增强，这里我们通过拦截器的配置，让每条修改的sql语句在执行的时候，都加上版本控制的功能

```
@Configuration  
public class MybatisPlusConfig {  
  
    @Bean  
    public MybatisPlusInterceptor mybatisPlusInterceptor(){  
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();  
        interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());  
        return interceptor;  
    }  
}
```

【4】测试效果，这里我们模拟先查询，再修改

```
@Test  
void updateTest(){  
    User user = userMapper.selectById(6L);  
    user.setName("li");  
    userMapper.updateById(user);  
}
```

我们通过查看拼接好的SQL语句发现，查询时将User的数据查询出来，是包含version

版本信息的

```
=> Preparing: SELECT id,name,age,email,version FROM powershop_user WHERE id=?  
=> Parameters: 6(Long)  
<== Columns: id, name, age, email, version  
<== Row: 6, li, 40, test6@powernode.com, 1  
<== Total: 1
```

当我们完成修改时，他会将版本号 + 1

```
=> Preparing: UPDATE powershop_user SET name=?, age=?, email=?, version=? WHERE id=? AND version=?  
=> Parameters: li(String), 40(Integer), test6@powernode.com(String), 2(Integer), 6(Long), 1(Integer)  
<== Updates: 1
```

此时查看数据发现，更改姓名后，version 已经为 2 了

id	name	age	email	version
1	wang	18	wang@powernode.com	1
2	Jack	20	test2@baomidou.com	1
3	Tom	28	test3@baomidou.com	1
4	Sandy	21	test4@baomidou.com	1
5	Billie	24	test5@baomidou.com	1
6	li	40	test6@powernode.com	2

接下来我们模拟一下，当出现多个修改请求的时候，是否能够做到乐观锁的效果。

乐观锁的效果是，一个请求在修改的过程中，是允许另一个请求查询的，但是修改时会通过版本号是否改变来决定是否修改，如果版本号变了，证明已经有请求修改过数据了，那这次修改不生效，如果版本号没有发生变化，那就完成修改。

```
@Test
void updateTest2(){
    //模拟操作 1 的查询操作
    User user1 = userMapper.selectById(6L);

    //模拟操作 2 的查询操作
    User user2 = userMapper.selectById(6L);

    //模拟操作 2 的修改操作
    user2.setName("lisi");
    userMapper.updateById(user2);

    //模拟操作 1 的修改操作
    user1.setName("zhangsan");
    userMapper.updateById(user1);
}
```

我们来看下这段代码的执行过程，这段代码其实是两次操作，只不过操作 1 在执行的过程中，有操作 2 完成了对于数据的修改，这时操作 1 就无法再次进行修改了

操作 1 的查询：此时版本为 2

```
>>> Preparing: SELECT id,name,age,email,version FROM powershop_user WHERE id=?
>>> Parameters: 6(Long)
<==> Columns: id, name, age, email, version
<==> Row: 6, li, 40, test6@powernode.com, 2
<==> Total: 1
```

操作 2 的查询：此时版本为 2

```
<==> Columns: id, name, age, email, version
<==> Row: 6, li, 40, test6@powernode.com, 2
<==> Total: 1
```

操作 2 的修改：此时检查版本，版本没有变化，所以完成修改，并将版本改为 3

```

==> Preparing: UPDATE powershop_user SET name=?, age=?, email=?, version=? WHERE id=? AND version=?
==> Parameters: lisi(String), 40(Integer), test6@powernode.com(String), 3(Integer), 6(Long), 2(Integer)
<==    Updates: 1
    
```

id	name	age	email	version
1	wang	18	wang@powernode.com	1
2	Jack	20	test2@baomidou.com	1
3	Tom	28	test3@baomidou.com	1
4	Sandy	21	test4@baomidou.com	1
5	Billie	24	test5@baomidou.com	1
6	lisi	40	test6@powernode.com	3

操作 1 的修改：此时检查版本，版本已经有最初获取的版本信息发生了变化，所以杜绝修

改

```

==> Preparing: UPDATE powershop_user SET name=?, age=?, email=?, version=? WHERE id=? AND version=?
==> Parameters: zhangsan(String), 40(Integer), test6@powernode.com(String), 3(Integer), 6(Long), 2(Integer)
<==    Updates: 0
    
```

5.8 代码生成器

代码生成器和逆向工程的区别在于，代码生成器可以生成更多的结构，更多的内容，允许我们能够配置生成的选项更多。在这里我们演示一下代码生成器的用法。

【1】参考官网，使用代码生成器需要引入两个依赖

```

<!--代码生成器依赖-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.5.3</version>
</dependency>

<!--freemarker 模板依赖-->
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.31</version>

```

```
</dependency>
```

【2】编写代码生成器代码

```
@SpringBootTest
class GeneratorApplicationTests {
    public static void main(String[] args) {

        FastAutoGenerator.create("jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false", "root", "root")
            .globalConfig(builder -> {
                builder.author("powernode") // 设置作者
                    //.enableSwagger() // 开启 swagger 模式
                    .fileOverride() // 覆盖已生成文件
                    .outputDir("D://");
            })
            .packageConfig(builder -> {
                builder.parent("com.powernode") // 设置父包名
                    .moduleName("mybatisplus") // 设置父包模块名
                    .pathInfo(Collections.singletonMap(OutputFile.xml, "D://"));
            })
            .strategyConfig(builder -> {
                builder.addInclude("powershop_user") // 设置需要生成的表名
                    .addTablePrefix("powershop"); // 设置过滤表前缀
            })
            .templateEngine(new FreemarkerTemplateEngine()) // 使用 Freemarker 引擎模板,
        默认的是 Velocity 引擎模板
            .execute();
    }
}
```

【3】执行，查看生成效果

5.9 执行 SQL 分析打印

在我们日常开发工作当中，避免不了查看当前程序所执行的 SQL 语句，以及了解它的执行时间，方便分析是否出现了慢 SQL 问题。我们可以使用 MybatisPlus 提供的 SQL 分析打印的功能，来获取 SQL 语句执行的时间。

【1】由于该功能依赖于 p6spy 组件，所以需要在 pom.xml 中先引入该组件

```
<dependency>
    <groupId>p6spy</groupId>
    <artifactId>p6spy</artifactId>
    <version>3.9.1</version>
</dependency>
```

【2】在 application.yml 中进行配置

将驱动和 url 修改

```
spring:
  datasource:
    driver-class-name: com.p6spy.engine.spy.P6SpyDriver
    url: jdbc:p6spy:mysql
```

【3】在 resources 下，创建 spy.properties 配置文件

```
#3.2.1 以上使用
moduleList=com.baomidou.mybatisplus.extension.p6spy.MybatisPlusLogFactory,com.p6spy.engine.outage.P6OutageFactory

# 自定义日志打印
logMessageFormat=com.baomidou.mybatisplus.extension.p6spy.P6SpyLogger

# 日志输出到控制台
appender=com.baomidou.mybatisplus.extension.p6spy.StdoutLogger

# 使用日志系统记录 sql
#appender=com.p6spy.engine.spy.appender.Slf4JLogger
```

```
# 设置 p6spy driver 代理
deregisterdrivers=true

# 取消 JDBC URL 前缀
useprefix=true

# 配置记录 Log 例外,可去掉的结果集 error,info,batch,debug,statement,commit,rollback,result,resultset.
excludecategories=info,debug,result,commit,resultset

# 日期格式
dateformat=yyyy-MM-dd HH:mm:ss

# 实际驱动可多个
#driverlist=org.h2.Driver

# 是否开启慢 SQL 记录
outagedetection=true

# 慢 SQL 记录标准 2 秒
outagedetectioninterval=2
```

【4】测试

执行查询所有的操作，可以看到 sql 语句的执行时间

```
==> Preparing: SELECT id,name,age,email FROM powershop_user
==> Parameters:
    Consume Time: 9 ms 2023-02-07 14:44:38
    Execute SQL: SELECT id,name,age,email FROM powershop_user
```

5.10 多数据源

在学习多数据源之前，我们先来了解一下分库分表

当一个项目的数据库的数据十分庞大时，在完成 SQL 操作的时候，需要检索的数据就会更

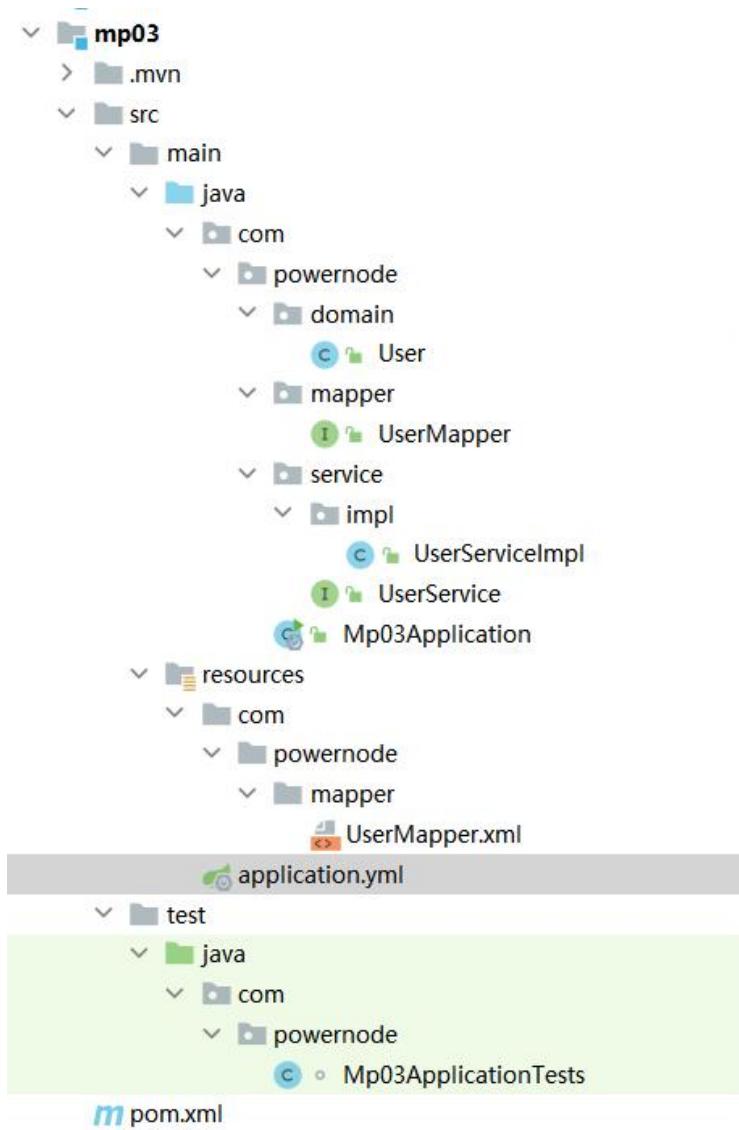
多，我们会遇到性能问题，会出现 SQL 执行效率低的问题。

针对这个问题，我们的解决方案是，将一个数据库中的数据，拆分到多个数据库中，从而减少单个数据库的数据量，从分摊访问请求的压力和减少单个数据库数据量这两个方面，都提升了效率。

我们来演示一下，在 MybatisPlus 中，如何演示数据源切换的效果

【1】先创建一个新的模块，将之前模块中的内容复制过来

结构如下



【2】引入依赖

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>dynamic-datasource-spring-boot-starter</artifactId>
    <version>3.1.0</version>
</dependency>
```

【3】创建新的数据库，提供多数据源环境

数据库



表数据

id	name	age	email
1	李寻欢	18	test11@baomidou.com
2	张三丰	20	test22@baomidou.com
3	张无忌	28	test33@baomidou.com
4	赵敏	21	test44@baomidou.com
5	周芷若	24	test55@baomidou.com
6	杨逍	40	test66@powernode.com

【4】编写配置文件，指定多数据源信息

```
spring:
  datasource:
    dynamic:
      primary: master
      strict: false
    datasource:
      master:
        username: root
        password: root
        url:
          jdbc:mysql://localhost:3306/mybatisplus?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false
          driver-class-name: com.mysql.cj.jdbc.Driver
      slave_1:
        username: root
        password: root
        url:
          jdbc:mysql://localhost:3306/mybatisplus2?serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false
          driver-class-name: com.mysql.cj.jdbc.Driver
```

【5】创建多个 Service，分别使用@DS 注解描述不同的数据源信息

```
@Service
@DS("master")
public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements UserService { }
```

```
@Service
@DS("slave_1")
public class UserServiceImpl2 extends ServiceImpl<UserMapper, User> implements UserService{ }
```

【6】测试 service 多数据源环境执行结果

```
@SpringBootTest
class Mp03ApplicationTests {

    @Autowired
    private UserServiceImpl userServiceImpl;

    @Autowired
    private UserServiceImpl2 userServiceImpl2;

    @Test
    public void select(){
        User user = userServiceImpl.getById(1L);
        System.out.println(user);
    }

    @Test
    public void select2(){
        User user = userServiceImpl2.getById(1L);
        System.out.println(user);
    }
}
```

【7】观察测试结果，发现结果可以从两个数据源中获取

```
==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE id=?
==> Parameters: 1(Long)
<==    Columns: id, name, age, email
<==      Row: 1, Jone, 18, test1@baomidou.com
<==    Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4ecd8ab1]
User(id=1, name=Jone, age=18, email=test1@baomidou.com)

==> Preparing: SELECT id,name,age,email FROM powershop_user WHERE id=?
==> Parameters: 1(Long)
<==    Columns: id, name, age, email
<==      Row: 1, 李寻欢, 18, test11@baomidou.com
<==    Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@300a38e7]
User(id=1, name=李寻欢, age=18, email=test11@baomidou.com)
```