



MyBatis-Plus实战教程与开发建议

学习要求

本教程并非是从0开始的学习教程，需要一定的技术基础：

良好的Java基础：基本能驾驭面向对象编程

熟悉Spring/SpringBoot框架：spring/springboot/springmvc能随手甩出一套Helloworld

熟悉MyBatis框架：基本crud没问题

熟悉MySQL数据库：简单的crud SQL 信手拈来

教程目标

系统学习并掌握MyBatis-Plus框架实战使用，更好配合MyBatis框架使用，简化项目开发

教学环境

操作系统：Window7 / Window10

JDK：JDK11+

数据库：MySQL7+ 、Navicat12

IDE工具：idea2019

代码工具：Maven3.6+

引出MyBatis-Plus

先来回顾常规的MyBatis操作MySQL步骤：

1>创建数据库，创建表

2>创建项目，导入逆向工程相对应的依赖与配置

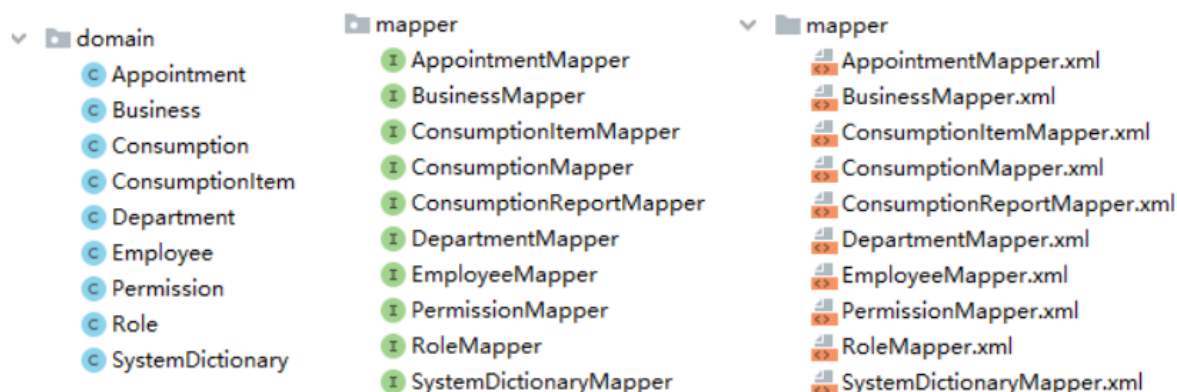
3>逆向工程创建实体/mapper接口/mapper.xml映射文件

4>导入mybatis依赖，mysql依赖

5>编写测试类，测试mapper接口实现CRUD操作

思考：当项目复杂之后MyBatis会出现什么问题？

看下图是之前接的外包项目使用MyBatis写的代码



分析：

项目中，每张表对应一个实体类，对应一个mapper.java接口，也对应一个mapper.xml配置文件。每个mapper.java接口都有重复的增、删、改、查4个基础方法，每一个Mapper.xml都有重复的增、删、改、查4个基础sql语句。使用逆向工程可以解决，但是存在很致命的问题，如果列发生变动，对应的sql改动那就是灾难啦。另外，如果一个项目很复杂，那这些文件就膨胀起来，对项目的加载与维护带来很大挑战。此时不禁想，有没有某种方法可以不写crud方法，不写crud sql语句也能完成crud操作呢？

答案是Yes：使用MyBatis-Plus！！

MyBatis-Plus简介

官网：<https://baomidou.com/>

GitHub：<https://github.com/baomidou/mybatis-plus>

Gitee：<https://gitee.com/baomidou/mybatis-plus>

简介

MyBatis-Plus（简称 **MP**）是一个 MyBatis 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

特性

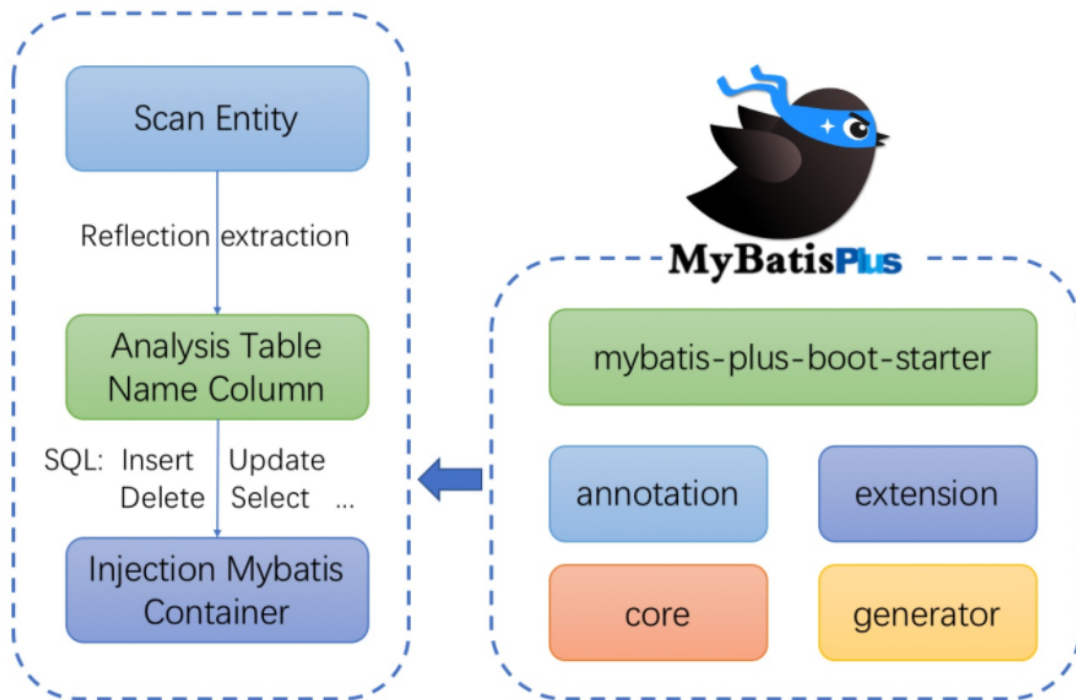
- **无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑
- **损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作
- **强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求
- **支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错
- **支持主键自动生成**：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题
- **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 `mybatis` 支持标准 SQL 的数据库，MyBatis-Plus 也支持

- 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

框架结构



MyBatis与MyBatis-Plus区别

MyBatis

优点:

- 1>SQL语句自由控制，较为灵活
- 2>SQL与业务代码分离，易于阅读与维护
- 3>提供动态SQL语句，可以根据需求灵活控制

缺点:

- 1>简单的crud操作也必须提供对应SQL语句
- 2>必须维护大量的xml文件
- 3>自身功能有限，要拓展只能依赖第三方插件

MyBatis-Plus 是在Mybatis的基础上进行二次开发的具有MyBatis所有功能，也添加了不少好用的功能

比如:

- 1>提供无sql的crud操作
 - 2>内置代码生成器，分页插件，性能分析插件等
 - 3>提供功能丰富的条件构造器快速进行无sql开发
 - 4>提供统一全局处理比如：乐观锁、逻辑删除等
-

MyBatis-Plus入门案例-Spring版

步骤1：创建数据库mybatis-plus，并创建employee表，并导入数据

```

1 CREATE TABLE `employee` (
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,
3   `name` varchar(255) DEFAULT NULL,
4   `password` varchar(255) DEFAULT NULL,
5   `email` varchar(255) DEFAULT NULL,
6   `age` int(11) DEFAULT NULL,
7   `admin` bit(1) DEFAULT NULL,
8   `dept_id` bigint(20) DEFAULT NULL,
9   PRIMARY KEY (`id`) USING BTREE
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW_FORMAT=DYNAMIC;

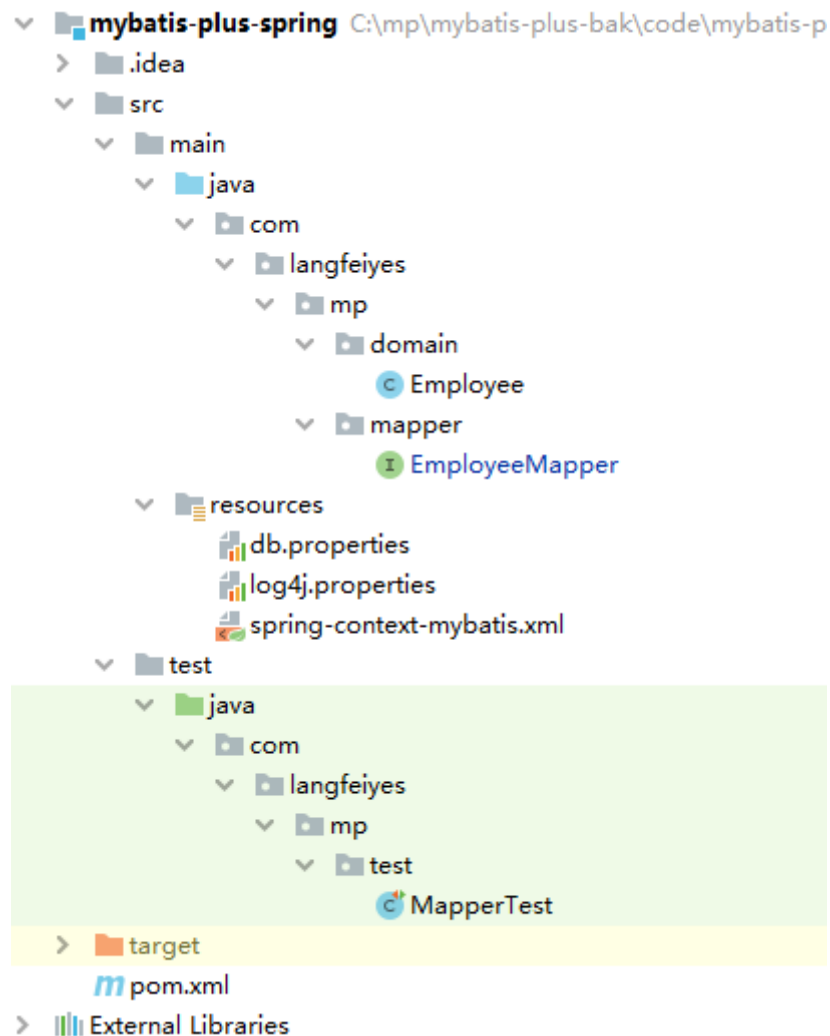
```

```

1 INSERT INTO `employee` VALUES (1, 'admin', '1', 'admin@abc.com', 40, b'1',
2   6);
3 INSERT INTO `employee` VALUES (2, '赵总', '1', 'zhaoz@langfeiyes.com', 35,
4   b'0', 1);
5 INSERT INTO `employee` VALUES (3, '赵一明', '1', 'zhaoy@langfeiyes.com', 25,
6   b'0', 1);
7 INSERT INTO `employee` VALUES (4, '钱总', '1', 'qianz@langfeiyes.com', 31,
8   b'0', 2);
9 INSERT INTO `employee` VALUES (5, '钱二明', '1', 'qianem@langfeiyes.com', 25,
10  b'0', 2);
11 INSERT INTO `employee` VALUES (6, '孙总', '1', 'sunz@langfeiyes.com', 35,
12  b'0', 3);
13 INSERT INTO `employee` VALUES (7, '孙三明', '1', 'sunsm@langfeiyes.com', 25,
14  b'0', 3);
15 INSERT INTO `employee` VALUES (8, '李总', '1', 'liz@langfeiyes.com', 35,
16  b'0', 4);
17 INSERT INTO `employee` VALUES (9, '李四明', '1', 'lism@langfeiyes.com', 25,
18  b'0', 4);
19 INSERT INTO `employee` VALUES (10, '周总', '1', 'zhouz@langfeiyes.com', 19,
20  b'0', 5);
21 INSERT INTO `employee` VALUES (11, '周五明', '1', 'zhouwm@langfeiyes.com',
22  25, b'0', 5);
23 INSERT INTO `employee` VALUES (12, '吴总', '1', 'wuz@langfeiyes.com', 41,
24  b'0', 5);
25 INSERT INTO `employee` VALUES (13, '吴六明', '1', 'wulm@langfeiyes.com', 33,
26  b'0', 5);
27 INSERT INTO `employee` VALUES (14, '郑总', '1', 'zhengz@langfeiyes.com', 35,
28  b'0', 3);
29 INSERT INTO `employee` VALUES (15, '郑七明', '1', 'zhengqm@langfeiyes.com',
30  25, b'0', 2);
31 INSERT INTO `employee` VALUES (16, '孙四明', '1', 'sunsim@langfeiyes.com',
32  25, b'0', 3);
33 INSERT INTO `employee` VALUES (17, '孙五明', '1', 'sunwm@langfeiyes.com', 25,
34  b'0', 3);
35 INSERT INTO `employee` VALUES (18, '李五明', '1', 'liwm@langfeiyes.com', 25,
36  b'0', 4);
37 INSERT INTO `employee` VALUES (19, '李六明', '1', 'lilm@langfeiyes.com', 25,
38  b'0', 4);
39 INSERT INTO `employee` VALUES (20, '叶子', '1', 'yezi@langfeiyes.com', 0,
40  b'0', 1);

```

2>创建maven项目-mybatis-plus-spring



3>pom.xml文件中导入相关依赖

```

1  <properties>
2      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3      <maven.compiler.source>11</maven.compiler.source>
4      <maven.compiler.target>11</maven.compiler.target>
5  </properties>
6
7
8  <dependencies>
9      <dependency>
10         <groupId>com.baomidou</groupId>
11         <artifactId>mybatis-plus</artifactId>
12         <version>3.4.3.4</version>
13     </dependency>
14
15     <!--数据连接池依赖-->
16     <dependency>
17         <groupId>com.alibaba</groupId>
18         <artifactId>druid</artifactId>
19         <version>1.2.8</version>
20     </dependency>
21
22     <!--mysql驱动依赖-->
23     <dependency>
24         <groupId>mysql</groupId>
25         <artifactId>mysql-connector-java</artifactId>

```

```

26         <version>8.0.22</version>
27     </dependency>
28     <!--自动生成getter/setter方法依赖-->
29     <dependency>
30         <groupId>org.projectlombok</groupId>
31         <artifactId>lombok</artifactId>
32         <version>1.18.16</version>
33         <scope>provided</scope>
34     </dependency>
35     <!--单元测试-->
36     <dependency>
37         <groupId>junit</groupId>
38         <artifactId>junit</artifactId>
39         <version>4.12</version>
40         <scope>test</scope>
41     </dependency>
42     <!--spring测试-->
43     <dependency>
44         <groupId>org.springframework</groupId>
45         <artifactId>spring-test</artifactId>
46         <version>5.2.16.RELEASE</version>
47     </dependency>
48
49     <!--spring环境依赖包-->
50     <dependency>
51         <groupId>org.springframework</groupId>
52         <artifactId>spring-context</artifactId>
53         <version>5.2.16.RELEASE</version>
54     </dependency>
55     <!--spring事务依赖包-->
56     <dependency>
57         <groupId>org.springframework</groupId>
58         <artifactId>spring-tx</artifactId>
59         <version>5.2.16.RELEASE</version>
60     </dependency>
61     <!--spring jdbc依赖包-->
62     <dependency>
63         <groupId>org.springframework</groupId>
64         <artifactId>spring-jdbc</artifactId>
65         <version>5.2.16.RELEASE</version>
66     </dependency>
67     <!--spring 日志依赖包-->
68     <dependency>
69         <groupId>org.slf4j</groupId>
70         <artifactId>slf4j-log4j12</artifactId>
71         <version>1.7.25</version>
72     </dependency>
73 </dependencies>

```

4>配置数据db.properties、日志文件log4j.properties、spring容器文件spring-context-mybatis.xml


```

1 jdbc.url=jdbc:mysql://localhost:3306/mybatis-plus?
  useUnicode=true&characterEncoding=utf8&serverTimezone=GMT%2B8
2 jdbc.username=root
3 jdbc.password=admin
4 jdbc.driverClassName=com.mysql.cj.jdbc.Driver

```

```

1 # Global logging configuration
2 log4j.rootLogger=ERROR, stdout
3
4 log4j.logger.com.langfeiyes.mp.mapper=TRACE
5 # Console output...
6 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
7 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
8 log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
9

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
4     xmlns:util="http://www.springframework.org/schema/util"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7       http://www.springframework.org/schema/context
8       http://www.springframework.org/schema/context/spring-context-3.0.xsd
9       http://www.springframework.org/schema/util
10      http://www.springframework.org/schema/util/spring-util-3.0.xsd">
11   <!-- 加载配置属性文件 -->
12   <context:property-placeholder location="classpath:db.properties" ignore-
  unresolvable="true" />
13   <!-- 配置数据源 -->
14   <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
  init-method="init" destroy-method="close">
15       <property name="driverClassName" value="${jdbc.driverClassName}">
</property>
16       <property name="username" value="${jdbc.username}"></property>
17       <property name="password" value="${jdbc.password}"></property>
18       <property name="url" value="${jdbc.url}"></property>
19   </bean>
20   <!-- 配置SessionFactory -->
21   <bean id="sqlSessionFactory"
  class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
22       <property name="dataSource" ref="dataSource"/>
23   </bean>
24   <!-- MyBatis 动态扫描 -->
25   <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
26       <property name="basePackage" value="com.langfeiyes.mp.mapper"/>
27   </bean>
28 </beans>

```



```

1  @Setter
2  @Getter
3  @ToString
4  public class Employee {
5      private Long id;
6      private String name;
7      private String password;
8      private String email;
9      private int age;
10     private int admin;
11     private Long deptId;
12 }

```

```

1  /**
2   * 持久层映射接口: mybatis-plus
3   * mybatis-plus mapper接口自定义
4   * 1: 自定义一个接口, 继承BaseMapper
5   * 2: 指定接口泛型: 当前接口操作实体对象: Employee
6   */
7  public interface EmployeeMapper extends BaseMapper<Employee> {
8  }

```

6>编写测试类

```

1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration({"classpath:spring-context-mybatis.xml"})
3  public class MapperTest {
4      @Autowired
5      private EmployeeMapper employeeMapper;
6
7      @Test
8      public void testSave(){
9          Employee employee = new Employee();
10         employee.setAdmin(1);
11         employee.setAge(18);
12         employee.setDeptId(1L);
13         employee.setEmail("yezi@langfeiyes.com");
14         employee.setName("yezi");
15         employee.setPassword("111");
16         employeeMapper.insert(employee);
17     }
18     @Test
19     public void testUpdate(){
20         Employee employee = new Employee();
21         employee.setId(1L);
22         employee.setAdmin(1);
23         employee.setAge(18);
24         employee.setDeptId(1L);
25         employee.setEmail("yezi@langfeiyes.com");
26         employee.setName("yezi");
27         employee.setPassword("111");
28         employeeMapper.updateById(employee);
29     }
30     @Test

```

```

31     public void testDelete(){
32         employeeMapper.deleteById(1L);
33     }
34     @Test
35     public void testGet(){
36         System.out.println(employeeMapper.selectById(1L));
37     }
38
39     @Test
40     public void testList(){
41         System.out.println(employeeMapper.selectList(null));
42     }
43 }

```



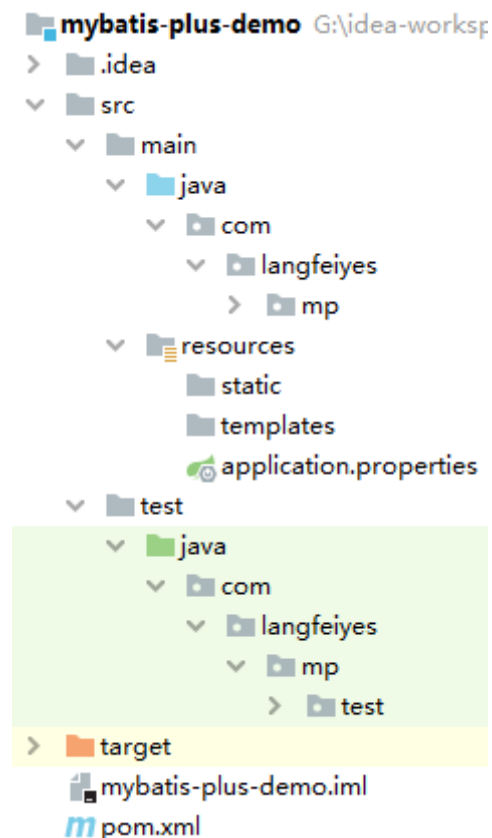
MyBatis-Plus入门案例-SpringBoot版

相对上面的spring版本的MyBatis-Plus，SpringBoot版本就简化多了。

1>创建数据库mybatis-plus，并创建employee表，并导入数据

此处跟上面的Spring版一样，不重复操作了

2>创建springboot项目-mybatis-plus-demo



3>pom.xml文件中导入springboot依赖、mybatis-plus依赖、其他依赖等

```
1      <!--springboot项目导入依赖-->
2      <parent>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-parent</artifactId>
5          <version>2.4.3</version>
6      </parent>
7
8
9      <dependencies>
10         <!--springboot项目基本组件依赖-->
11         <dependency>
12             <groupId>org.springframework.boot</groupId>
13             <artifactId>spring-boot-starter</artifactId>
14         </dependency>
15         <!--mybatis-plus依赖-->
16         <dependency>
17             <groupId>com.baomidou</groupId>
18             <artifactId>mybatis-plus-boot-starter</artifactId>
19             <version>3.4.0</version>
20         </dependency>
21         <!--数据库连接池依赖-->
22         <dependency>
23             <groupId>com.alibaba</groupId>
24             <artifactId>druid-spring-boot-starter</artifactId>
25             <version>1.1.17</version>
26         </dependency>
27         <!--mysql驱动依赖-->
28         <dependency>
29             <groupId>mysql</groupId>
30             <artifactId>mysql-connector-java</artifactId>
31             <version>8.0.22</version>
32         </dependency>
33         <!--测试依赖-->
34         <dependency>
35             <groupId>org.springframework.boot</groupId>
36             <artifactId>spring-boot-starter-test</artifactId>
37         </dependency>
38         <!--自动生成getter/setter方法依赖-->
39         <dependency>
40             <groupId>org.projectlombok</groupId>
41             <artifactId>lombok</artifactId>
42             <version>1.18.16</version>
43             <scope>provided</scope>
44         </dependency>
45     </dependencies>
```

4>在application.properties文件中配置数据库连接四要素与日志

```

1  #mysql
2  spring.datasource.url=jdbc:mysql://localhost:3306/mybatis-plus?
   useUnicode=true&characterEncoding=utf8&serverTimezone=GMT%2B8
3  spring.datasource.username=root
4  spring.datasource.password=admin
5  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
6
7  # 配置sql打印日志
8  mybatis-plus.configuration.log-
   impl=org.apache.ibatis.logging.stdout.StdoutImpl

```

5>编写Employee实体类与mapper接口

```

1  @Setter
2  @Getter
3  @ToString
4  public class Employee {
5      private Long id;
6      private String name;
7      private String password;
8      private String email;
9      private int age;
10     private int admin;
11     private Long deptId;
12 }

```

```

1  /**
2   * 持久层映射接口: mybatis-plus
3   * mybatis-plus mapper接口自定义
4   * 1: 自定义一个接口, 继承BaseMapper
5   * 2: 指定接口泛型: 当前接口操作实体对象: Employee
6   */
7  public interface EmployeeMapper extends BaseMapper<Employee> {
8  }

```

6>编写启动类, 指定mapper接口扫描路径

```

1  @SpringBootApplication
2  //mapper接口所在的包路径
3  @MapperScan(basePackages = "com.langfeiyes.mp.mapper")
4  public class App {
5      public static void main(String[] args) {
6          SpringApplication.run(App.class, args);
7      }
8  }

```

7>编写测试类

```

1  @SpringBootTest
2  public class MapperTest {
3      @Autowired
4      private EmployeeMapper employeeMapper;
5

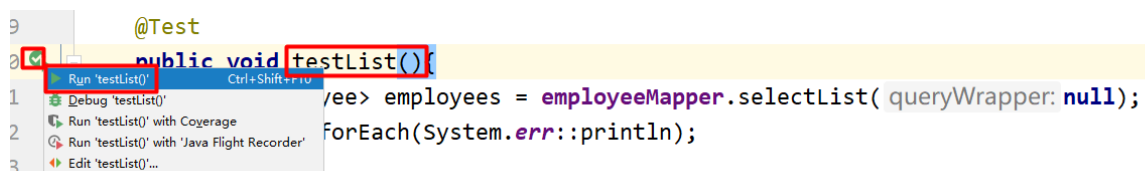
```

```

6      @Test
7      public void testSave(){
8          Employee employee = new Employee();
9          employee.setAdmin(1);
10         employee.setAge(18);
11         employee.setDeptId(1L);
12         employee.setEmail("yezi@langfeiyes.com");
13         employee.setName("yezi");
14         employee.setPassword("111");
15         employeeMapper.insert(employee);
16     }
17     @Test
18     public void testUpdate(){
19         Employee employee = new Employee();
20         employee.setId(1L);
21         employee.setAdmin(1);
22         employee.setAge(18);
23         employee.setDeptId(1L);
24         employee.setEmail("yezi@langfeiyes.com");
25         employee.setName("yezi");
26         employee.setPassword("111");
27         employeeMapper.updateById(employee);
28     }
29     @Test
30     public void testDelete(){
31         employeeMapper.deleteById(1L);
32     }
33     @Test
34     public void testGet(){
35         System.out.println(employeeMapper.selectById(1L));
36     }
37
38     @Test
39     public void testList(){
40         System.out.println(employeeMapper.selectList(null));
41     }
42 }

```

8>运行testList方法



<https://blog.csdn.net/langfeiyes>

观察SQL

image-20211211194046087

MyBatis-Plus入门案例解析

原理分析

提出问题

问题1: EmployeeMapper 接口没有定义crud方法, 为什么在CRUDTest测试类中可以调用crud方法?

答: EmployeeMapper接口继承了BaseMapper接口, BaseMapper接口自定义了很多crud方法, 所以即使EmployeeMapper接口没写, 它也可以从BaseMapper接口继承而来。

问题2: 整个项目没有编写SQL语句, 为什么可以实现CRUD操作, 比如上面查询可以打印出查询sql?

答: 项目中没有, 但是又能执行, 说明了啥? 说明了Mybatis-Plus框架帮我们实现了。

问题3: Mybatis-Plus框架如何实现SQL拼接的?

答:

以selectList为例子, 实现员工列表的SQL:

SELECT id,name,password,email,age,admin,dept_id FROM employee

这条SQL核心点是啥?

表: employee ,

列: id,name,password,email,age,admin,dept_id

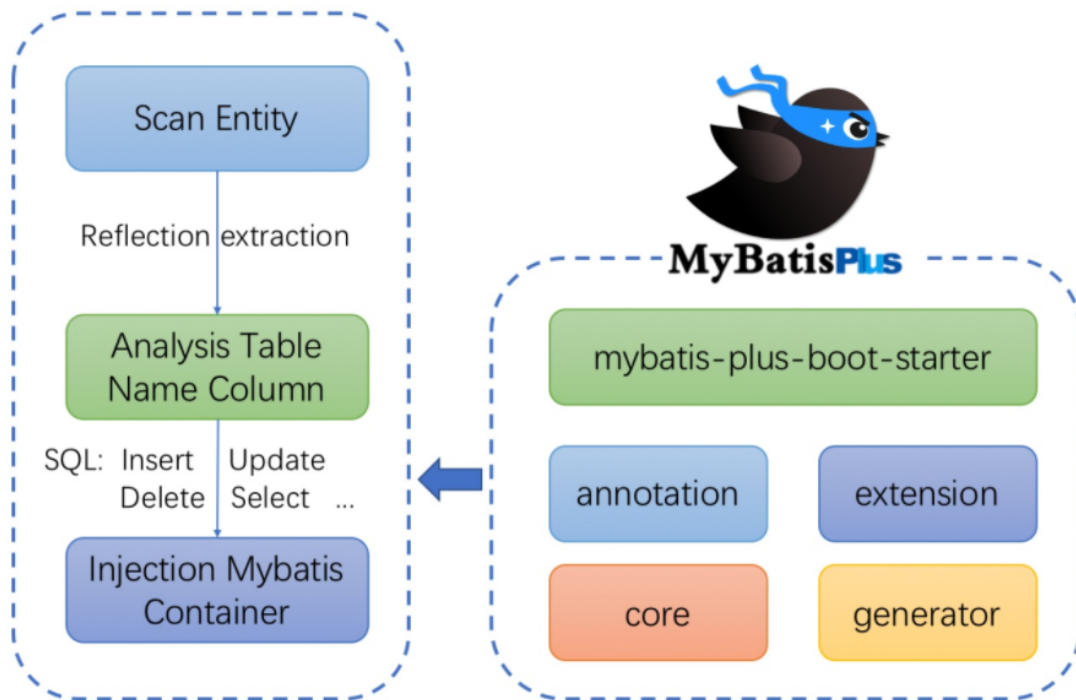
设想, 那如果mybatis-plus能通过某种手段拼接出表跟列, 那么执行sql不就有了么? 但这又引出一个问题: mybatis-plus是怎么找到要操作的表跟列呢?

观察下面接口:

```
1 public interface EmployeeMapper extends BaseMapper<Employee> {  
2 }
```

继承的BaseMapper接口中指定一个Employee泛型, 那么是否可以这么推测: mybatis-plus通过获取指定的泛型, 并解析该类型得到Employee类字节码对象, 然后通过反射操作获取类名与字段名。并将类名跟表名映射, 字段名跟列名映射, 那么上面说的表, 列不就齐了么?

对的, mybatis-plus底层就是这么实现的。我们再看回mybatis-plus架构图



单看左边，也能大体推测出mybatis-plus操作原理：

- 1>扫描实体对象，通过反射获取实体对象对应的表，对应的列
- 2>解析表，列名，做各种适配，然后根据调用的方法拼接出各种CRUD SQL语句
- 3>将拼接好的SQL语句交给mybatis容器执行。

到这Mybatis-Plus的helloworld就介绍啦。

源码分析

上面讲清楚原理，那么我们从源码角度去分析，MyBatis-Plus拼接SQL核心类

1>**ISqlInjector** SQL 自动注入器 完成sql拼接与注入

2>**AbstractMethod** 抽象的注入方法类，BaseMapper接口中所有方法的父类

子类有：SelectById, SelectList, DeleteById, UpdateById, Insert 等

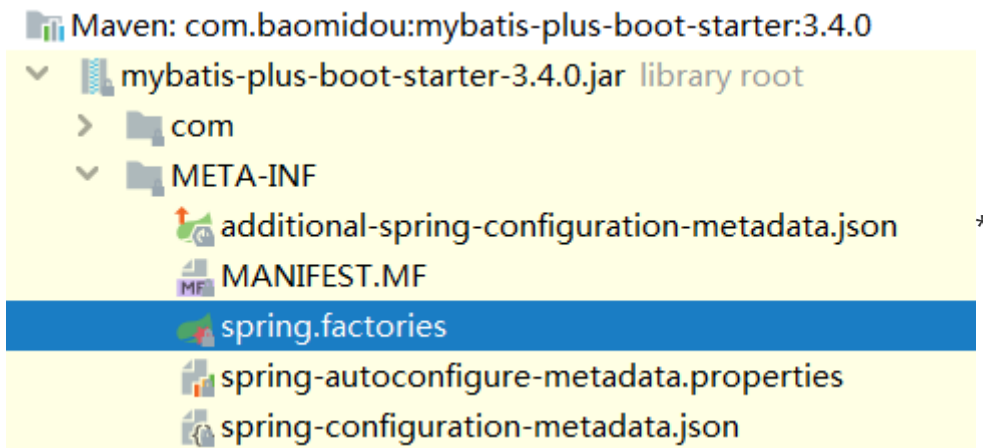
3>**MappedStatement** mapper接口方法与sql的映射对象，里面包含每个sql传入参数，返回值等。

4>**SqlMethod** MyBatis-Plus 定义枚举类，里面包含BaseMapper接口 主干sql语句

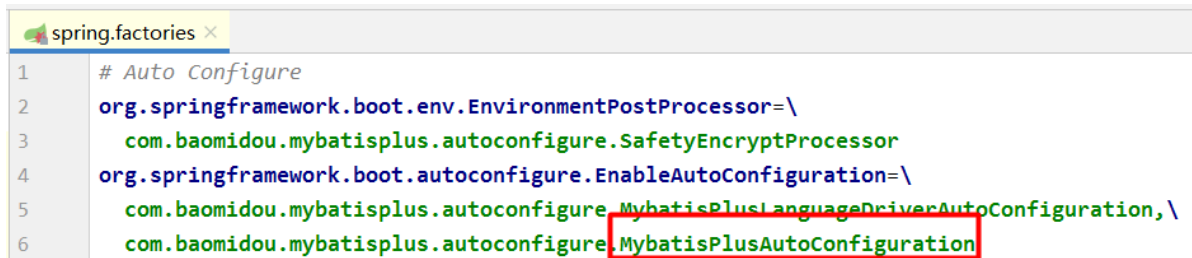
查看步骤：

1>查看mybatis-plus-boot-starter 启动器配置文件-Spring.factories

原因：自定义springboot启动器必须明确指定自动启动配置类，约定放置在spring.factories类中



2>查找mybatis-plus自动启动配置类然后解析



按着ctrl键，点击MybatisPlusAutoConfiguration，里面有个@Bean 实例方法，springboot容器加载时会自动创建对象，并交给容器管理



```

23     }
24     if (this.properties.getTypeAliasesSuperType() != null) {
25
26         factory.setTypeAliasesSuperType(this.properties.getTypeAliasesSuperType());
27     }
28     if (StringUtils.getLength(this.properties.getTypeHandlersPackage()))
29 {
30     factory.setTypeHandlersPackage(this.properties.getTypeHandlersPackage());
31     }
32     if (!ObjectUtils.isEmpty(this.typeHandlers)) {
33         factory.setTypeHandlers(this.typeHandlers);
34     }
35     Resource[] mapperLocations =
36 this.properties.resolveMapperLocations();
37     if (!ObjectUtils.isEmpty(mapperLocations)) {
38         factory.setMapperLocations(mapperLocations);
39     }
40     // TODO 修改源码支持定义 TransactionFactory
41     this.getBeanThen(TransactionFactory.class,
42 factory::setTransactionFactory);
43
44     // TODO 对源码做了一定的修改(因为源码适配了老旧的mybatis版本,但我们不需要适配)
45     Class<? extends LanguageDriver> defaultLanguageDriver =
46 this.properties.getDefaultScriptingLanguageDriver();
47     if (!ObjectUtils.isEmpty(this.languageDrivers)) {
48         factory.setScriptingLanguageDrivers(this.languageDrivers);
49     }
50
51     optional.ofNullable(defaultLanguageDriver).ifPresent(factory::setDefaultScriptingLanguageDriver);
52
53     // TODO 自定义枚举包
54     if (StringUtils.getLength(this.properties.getTypeEnumsPackage())) {
55
56         factory.setTypeEnumsPackage(this.properties.getTypeEnumsPackage());
57     }
58     // TODO 此处必为非 NULL
59     GlobalConfig globalConfig = this.properties.getGlobalConfig();
60     // TODO 注入填充器
61     this.getBeanThen(MetaObjectHandler.class,
62 globalConfig::setMetaObjectHandler);
63     // TODO 注入主键生成器
64     this.getBeanThen(IKeyGenerator.class, i ->
65 globalConfig.getDbConfig().setKeyGenerator(i));
66     // TODO 注入sql注入器
67     this.getBeanThen(ISqlInjector.class, globalConfig::setSqlInjector);
68     // TODO 注入ID生成器
69     this.getBeanThen(IdentifierGenerator.class,
70 globalConfig::setIdentifierGenerator);
71     // TODO 设置 GlobalConfig 到 MybatisSqlSessionFactoryBean
72     factory.setGlobalConfig(globalConfig);
73     return factory.getObject();
74 }

```

上面的diam中，下面代码是当前重点关注的了，

```
1 // TODO 注入sql注入器
2 this.getBeanThen(ISqlInjector.class, globalConfig::setSqlInjector);
```

按着ctrl键，点击进入ISqlInjector 类，它是一个借口，借口里面有唯一方法inspectInject

```
1 /**
2  * SQL 自动注入器接口
3  *
4  * @author hubin
5  * @since 2016-07-24
6  */
7 public interface ISqlInjector {
8
9     /**
10      * 检查SQL是否注入(已经注入过不再注入)
11      *
12      * @param builderAssistant mapper 信息
13      * @param mapperClass      mapper 接口的 class 对象
14      */
15     void inspectInject(MapperBuilderAssistant builderAssistant, Class<?>
mapperClass);
16 }
```

进入它的实现类，AbstractSqlInjector，核心方法

```
1 @Override
2 public void inspectInject(MapperBuilderAssistant builderAssistant, Class<?>
mapperClass) {
3     //通过Mapper接口获取泛型对象，比如：Employee
4     Class<?> modelClass = extractModelClass(mapperClass);
5     if (modelClass != null) {
6         String className = mapperClass.toString();
7         Set<String> mapperRegistryCache =
GlobalConfigUtils.getMapperRegistryCache(builderAssistant.getConfiguration()
);
8         if (!mapperRegistryCache.contains(className)) {
9             //获取Mapper接口中所有方法名
10             List<AbstractMethod> methodList =
this.getMethodList(mapperClass);
11             if (CollectionUtils.isNotEmpty(methodList)) {
12                 //解析Mapper接口获取方法对应的数据库，表，列等相关信息
13                 TableInfo tableInfo =
TableInfoHelper.initTableInfo(builderAssistant, modelClass);
14                 // 循环注入自定义方法(包括默认的)
15                 methodList.forEach(m -> m.inject(builderAssistant,
mapperClass, modelClass, tableInfo));
16             } else {
17                 logger.debug(mapperClass.toString() + ", No effective
injection method was found.");
18             }
19             mapperRegistryCache.add(className);
20         }
21     }
```

```
21 |     }  
22 | }
```

拼接sql最核心的方法

```
1 | methodList.forEach(m -> m.inject(builderAssistant, mapperClass, modelClass,  
    | tableInfo));
```

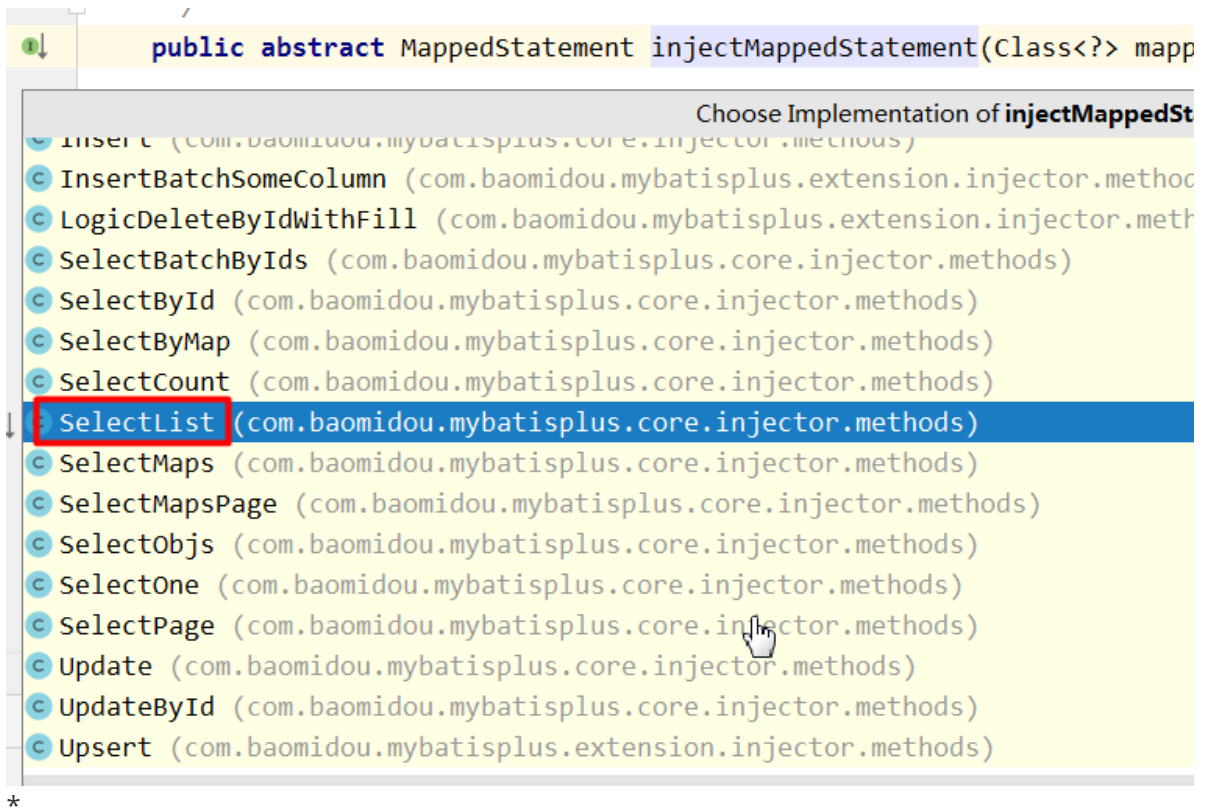
按着ctrl键，点击进入(AbstractMethod)类的inject 方法，

```
1 | /**  
2 |  * 注入自定义方法  
3 |  */  
4 | public void inject(MapperBuilderAssistant builderAssistant, Class<?>  
    | mapperClass, Class<?> modelClass, TableInfo tableInfo) {  
5 |     this.configuration = builderAssistant.getConfiguration();  
6 |     this.builderAssistant = builderAssistant;  
7 |     this.languageDriver =  
    | configuration.getDefaultScriptingLanguageInstance();  
8 |     /* 注入自定义方法 */  
9 |     injectMappedStatement(mapperClass, modelClass, tableInfo);  
10 | }
```

里面的injectMappedStatement 方法是核心，

```
1 | /**  
2 |  * 注入自定义 MappedStatement  
3 |  *  
4 |  * @param mapperClass mapper 接口  
5 |  * @param modelClass  mapper 泛型  
6 |  * @param tableInfo  数据库表反射信息  
7 |  * @return MappedStatement  
8 |  */  
9 | public abstract MappedStatement injectMappedStatement(Class<?> mapperClass,  
    | Class<?> modelClass, TableInfo tableInfo);  
10 |
```

injectMappedStatement 它是AbstractMethod的抽象方法，需要找对应的子类，这里以SelectList为例子展开讲



进入SelectList类，查看

```
1  /**
2   * 查询满足条件所有数据
3   *
4   * @author hubin
5   * @since 2018-04-06
6   */
7  public class SelectList extends AbstractMethod {
8
9      @Override
10     public MappedStatement injectMappedStatement(Class<?> mapperClass,
11         Class<?> modelClass, TableInfo tableInfo) {
12         //获取主干sql模本
13         SqlMethod sqlMethod = SqlMethod.SELECT_LIST;
14         //最终拼接出来的SQL
15         String sql = String.format(sqlMethod.getSql(), sqlFirst(),
16             sqlSelectColumns(tableInfo, true), tableInfo.getTableName(),
17             sqlWhereEntityWrapper(true, tableInfo), sqlComment());
18         SqlSource sqlSource = languageDriver.createSqlSource(configuration,
19             sql, modelClass);
20         return this.addSelectMappedStatementForTable(mapperClass,
21             getMethod(sqlMethod), sqlSource, tableInfo);
22     }
23 }
```

其中下面代码就是我们想要的

SqlMethod 枚举类的sql模本

```
1  SqlMethod sqlMethod = SqlMethod.SELECT_LIST;
```

```
1 SELECT_LIST("selectList", "查询满足条件所有数据", "<script>%s SELECT %s FROM %s\n%s\n</script>"),
```

通过模板拼接出的sql语句

```
1 //最终拼接出来的SQL
2 String.format(sqlMethod.getSql(), sqlFirst(), sqlSelectColumns(tableInfo, true), tableInfo.getTableName(),
```

```
1 <script><choose>
2 <when test="ew != null and ew.sqlFirst != null">
3   ${ew.sqlFirst}
4 </when>
5 <otherwise></otherwise>
6 </choose> SELECT <choose>
7 <when test="ew != null and ew.sqlSelect != null">
8   ${ew.sqlSelect}
9 </when>
10 <otherwise>id,name,password,email,age,admin,dept_id</otherwise>
11 </choose> FROM employee
12 <if test="ew != null">
13 <where>
14 <if test="ew.entity != null">
15 <if test="ew.entity.id != null">id=#{ew.entity.id}</if>
16 <if test="ew.entity['name'] != null"> AND name=#{ew.entity.name}</if>
17 <if test="ew.entity['password'] != null"> AND password=#{ew.entity.password}
18 </if>
19 <if test="ew.entity['email'] != null"> AND email=#{ew.entity.email}</if>
20 AND age=#{ew.entity.age}
21 AND admin=#{ew.entity.admin}
22 <if test="ew.entity['deptId'] != null"> AND dept_id=#{ew.entity.deptId}</if>
23 </if>
24 <if test="ew.sqlSegment != null and ew.sqlSegment != '' and
25 ew.nonEmptyOfWhere">
26 <if test="ew.nonEmptyOfEntity and ew.nonEmptyOfNormal"> AND</if>
27 ${ew.sqlSegment}
28 </if>
29 </where>
30 <if test="ew.sqlSegment != null and ew.sqlSegment != '' and
31 ew.emptyOfWhere">
32 ${ew.sqlSegment}
33 </if>
34 </if> <choose>
35 <when test="ew != null and ew.sqlComment != null">
36 ${ew.sqlComment}
37 </when>
38 <otherwise></otherwise>
39 </choose>
40 </script>
```

到这，sql拼接源码查看就结束啦

MyBatis-Plus常用注解

上面讲到MyBatis-Plus会解析实体类的类名跟属性名映射数据库的表名跟列名，如果类名与表名，属性名与列名不一致呢？这会怎样？

答案很明显，直接报错

```
public class Employee {  
    private Long id;  
    private String name;  
    private String password;  
    private String email;  
    private int age;  
    private int admin;  
    private Long deptId;  
}
```

对象 employee @mybatis-plus (l...	
新建	保存
另存为	
栏位	索引
名	类
id	b
ename	va
password	va
email	va
age	in
admin	b
dept_id	b

执行

```
@Test  
public void testList() {  
    System.out.println(employeeMapper.selectList(null));  
}
```

```
Caused by: java.sql.SQLException: Unknown column 'name' in 'field list'  
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:120)  
  
### SQL: SELECT id,name,password,email,age,admin,dept_id FROM employee  
### Cause: java.sql.SQLException: Unknown column 'name' in 'field list'
```

从图上标记错误信息看：name 列找不到。很正常，按照上一章节的分析，列名与属性名必须一一对应，这里不一致，报错就是必然的啦。

思考：该如果处理类名/属性名与表名/列名不一致的问题呢？

MyBatis-Plus的解决方案是：使用注解(文档：<https://baomidou.com/pages/223848/>)

MyBatis-Plus官方提供10个注解，这里我们先讲其他常用的3个，剩下的等用到后续用到对应功能时再详细展开讲。

@TableName

作用：默认情况下，实体名与表名一致，当不一致时，使用该注解显示指定表名

核心属性：value

案例：

```
1 @TableName("t_employee") //t_employee 是表名  
2 public class Employee {  
3     //.....  
4 }
```

@TableField

作用：默认情况下，实体属性名与列名一致，当不一致时，使用该注解显示指定列名

核心属性：value, exist

```
1 public class Employee {
2     @TableField(value="employename") //员工表列名: employename
3     private String name;
4 }
```

如果实体类中有多余的属性，且没有跟表中某一列映射，此时需要使用exist属性进行排除，这样mybatis-plus在拼接SQL时，放弃这个属性的解析

```
1 public class Employee {
2     @TableField(exist = false)
3     private Department dept;
4 }
```

@TableId

作用：默认情况下，没有明确指定表的主键，使用雪花算法生成一个唯一的long类型的id

核心属性：value, type

案例：

```
1 public class Employee {
2     @TableId(value="id", type=IdType.AUTO)
3     private Long id;
4 }
```

IdType的类型有：

IdType.AUTO ：数据库ID自增

IdType.NONE ：无状态,该类型为未设置主键类型(注解里等于跟随全局,全局里约等于 INPUT)

IdType.INPUT ：insert前自行set主键值

IdType.ASSIGN_ID ：分配ID(主键类型为Number(Long和Integer)或String)(since 3.3.0),使用接口IdentifierGenerator的方法nextId(默认实现类为DefaultIdentifierGenerator雪花算法)

IdType.ASSIGN_UUID :分配UUID,主键类型为String(since 3.3.0),使用接口IdentifierGenerator的方法nextUUID(默认default方法)

雪花算法计算出的long类型id---IdType.ASSIGN_ID

id	name	password	email	age	admin	dept_id
1471054587722940418	zhangsan	111	zhangsan@163.com	18	1	1

*

使用数据库自增----IdType.AUTO

id	name	password	email	age	admin	dept_id
1	zhangsan	111	zhangsan@163.com	18	1	1

*

@Version

拓展，后续讲乐观锁时候再讲

@TableLogic

拓展，后续讲逻辑删除时候再讲

MyBatis-Plus通用Mapper接口

BaseMapper介绍

回归到代码，MyBatis-Plus 编写实体对象mapper接口时，需要继承一个通用Mapper接口：
BaseMapper

还是以员工实体对象做为操作例子

```
1  @Setter
2  @Getter
3  @ToString
4  public class Employee {
5      private Long id;
6      private String name;
7      private String password;
8      private String email;
9      private int age;
10     private int admin;
11     private Long deptId;
12 }
```

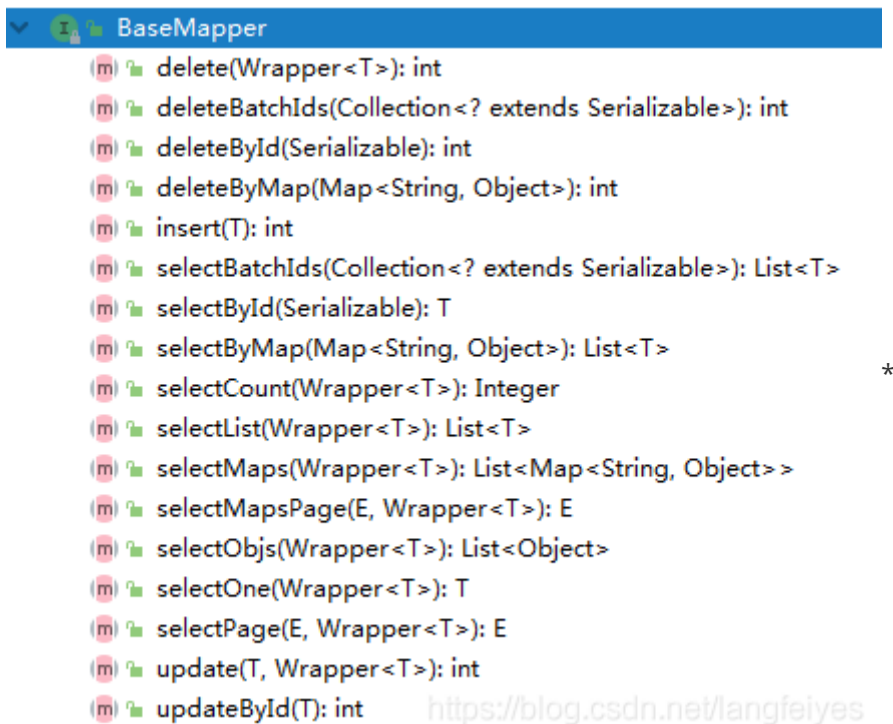
BaseMapper

```
1  public interface EmployeeMapper extends BaseMapper<Employee> {
2  }
```

MyBatis-Plus核心功能来自BaseMapper<实体类> 接口的实现

BaseMapper方法集

insert: 1个 update: 2个 delete: 4个 select: 10个



添加-insert

mybatis-plus中添加方法有1个:

```
1  /**
2   * 插入一条记录
3   *
4   * @param entity 实体对象
5   */
6  int insert(T entity);
```

用法: 传入要添加的实体对象

需求: 往employee表中添加一条记录

```
1  @Test
2  public void testSave(){
3      Employee employee = new Employee();
4      employee.setAdmin(1);
5      employee.setAge(18);
6      employee.setDeptId(1L);
7      employee.setEmail("zhangsan@163.com");
8      employee.setName("zhangsan");
9      employee.setPassword("111");
10     employeeMapper.insert(employee);
11 }
```

执行后SQL

```
1  INSERT INTO employee ( id, name, password, email, age, admin, dept_id )
   VALUES ( ?, ?, ?, ?, ?, ?, ? )
```

更新-update

mybatis-plus中更新方法有2个：

```
1  /**
2   * 根据 ID 修改
3   *
4   * @param entity 实体对象
5   */
6  int updateById(@Param(Constants.ENTITY) T entity);
7
8  /**
9   * 根据 whereEntity 条件，更新记录
10  *
11  * @param entity      实体对象 (set 条件值,可以为 null)
12  * @param updateWrapper 实体对象封装操作类 (可以为 null,里面的 entity 用于生成 where
    语句)
13  */
14  int update(T entity, Wrapper<T> updateWrapper);
```

updateById

用法：以id作为更新条件更新指定实体对象

需求：更新id为1员工姓名为：zhangdasan

```
1  @Test
2  public void testUpdateById2(){
3      Employee employee = new Employee();
4      employee.setId(1L);
5      employee.setName("zhangdasan");
6      employeeMapper.updateById(employee);
7  }
```

执行后SQL

```
1  UPDATE employee SET name=?, age=?, admin=? WHERE id=?
```

改前：

id	name	password	email	age	admin	dept_id
1	zhangsan	111	zhangsan@163.com	18	1	1

*

改后：

id	name	password	email	age	admin	dept_id
1	zhangdasan	111	zhangsan@163.com	0	0	1

*

执行之后就发现一个问题，employee表中id为1的员工数据age跟 admin这列数据丢失了，再看打印的SQL，本来只改name字段按理SQL中会拼接set name = ? 但是SQL也拼接了age 跟admin列的更新，怎么回事呢？

这里就涉及到MyBatis-Plus拼接SQL的规则啦：

MyBatis-Plus拼接SQL规则：

1: 实体对象作为方法参数, 如果属性值为null, 该属性不参与方法SQL拼接, 反之, 则参与拼接。

2: 实体对象作为方法参数, 如果属性为基本类型, 有默认值, 该属性参与方法SQL拼接

上面的updateById方法操作中, name, password, email属性值都是null, 所以update sql语句set中并没有拼接name, password, email 列的更新。而 age, admin 2个属性属于基本类型, 不显示设置属性值, 那默认就是0, 0。MyBatis-Plus 认为该属性有值, 便在update sql语句set 中拼接 age = ?, name = ?。这操作最终导致了表中数据丢失。

那怎么解决上述问题呢

方案1: 使用包装类型

```
1 private Integer age;
2 private Integer admin;
```

包装类型默认值为null, 不参与set拼接

方案2: 使用先查询, 再替换, 后更新

```
1 @Test
2 public void testUpdate(){
3     //先查询
4     Employee employee = employeeMapper.selectById(1L);
5     //再替换
6     employee.setName("zhangxiaosan");
7     //后更新
8     employeeMapper.updateById(employee);
9 }
```

先查询, 可将表的所有字段查询出来, 那么employee所有属性便有值, 替换要更新属性, 最后update时可以保证set 的列都要值, 最终表数据能还原。

方案3: 使用update(null, wrapper)方法。

需求: 更新id为1员工姓名为: zhangdasan

```
1 @Test
2 public void testUpdate2(){
3     //wrapper :暂时认为是sql 语句中where
4     UpdateWrapper<Employee> wrapper = new UpdateWrapper<>();
5     wrapper.eq("id", 21L); //等价于: 拼接 where id = 21
6     wrapper.set("name", "zhangzhongsan");
7     employeeMapper.update(null, wrapper);
8 }
```

执行后SQL

```
1 UPDATE employee SET name=? WHERE (id = ?)
```

这里有一个注意要点, update的第一参数为null, 如果传了employee, sql 拼接跟上面的updateById一样, 起不到想要的效果。

这种方案, 准确的讲应该叫: 使用wrapper方式。

思考：updateById 跟update 2个方法该如何选用

```
1  /**
2   * update跟updateById方法的选用
3   *
4   * 使用updateById 场景
5   * 1>where条件是id时候update场景
6   * 2>进行全量(所有字段)更新时候
7   *
8   * 使用update场景
9   * 1>where条件是不确定,或者多条件的update场景
10  * 2>进行部分字段更新时候
11  */
```

删除-delete

mybatis-plus中删除方法有4个

```
1  /**
2   * 根据 ID 删除
3   *
4   * @param id 主键ID
5   */
6  int deleteById(Serializable id);
7
8  /**
9   * 根据 columnMap 条件，删除记录
10  *
11  * @param columnMap 表字段 map 对象
12  */
13  int deleteByMap(Map<String, Object> columnMap);
14
15  /**
16  * 根据 entity 条件，删除记录
17  *
18  * @param wrapper 实体对象封装操作类（可以为 null）
19  */
20  int delete(wrapper<T> wrapper);
21
22  /**
23  * 删除（根据ID 批量删除）
24  *
25  * @param idList 主键ID列表(不能为 null 以及 empty)
26  */
27  int deleteBatchIds(Collection<? extends Serializable> idList);
```

deleteById

用法：删除指定id的实体信息

需求：删除id为1的员工信息

```

1 | @Test
2 | public void testDeleteById(){
3 |     employeeMapper.deleteById(1L);
4 | }

```

执行后SQL

```

1 | DELETE FROM employee WHERE id=?

```

deleteBatchIds

用法：批量删除指定多个id对象信息

需求：删除id为1,2,3的员工信息

```

1 | @Test
2 | public void testDeleteBatchIds(){
3 |     employeeMapper.deleteBatchIds(Arrays.asList(1L, 2L, 3L));
4 | }

```

执行后SQL

```

1 | DELETE FROM employee WHERE id IN ( ? , ? , ? )

```

deleteByMap

用法：按条件删除，具体条件放置在map集合中

需求：删除name=zhangsan并且age=18的员工信息

```

1 | @Test
2 | public void testDeleteByMap(){
3 |     //key: 列, value: 要匹配的条件值
4 |     Map<String, Object> map = new HashMap<>();
5 |     map.put("name", "zhangsan");
6 |     map.put("age", 18);
7 |     //多条件删除, map里面装的都是where 条件
8 |     employeeMapper.deleteByMap(map);
9 | }

```

执行后SQL

```

1 | DELETE FROM employee WHERE name = ? AND age = ?

```

delete

用法：按条件删除，具体条件通过条件构造器拼接

需求：删除name=zhangsan并且age=18的员工信息


```

1  @Test
2  public void testDelete(){
3      //如果更新条件操作使用:  UpdateWrapper
4      //其他条件操作使用:  QueryWrapper
5      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
6      wrapper.eq("age", 18);
7      wrapper.eq("name", "zhangsan"); //条件是and拼接
8      employeeMapper.delete(wrapper);
9  }

```

执行完的SQL

```

1  DELETE FROM employee WHERE (age = ? AND name = ?)

```

查询-select

mybatis-plus中查询方法有10个

```

1  /**
2   * 根据 ID 查询
3   *
4   * @param id 主键ID
5   */
6  T selectById(Serializable id);
7
8  /**
9   * 查询（根据ID 批量查询）
10  *
11  * @param idList 主键ID列表(不能为 null 以及 empty)
12  */
13  List<T> selectBatchIds(Collection<? extends Serializable> idList);
14
15  /**
16   * 查询（根据 columnMap 条件）
17   *
18   * @param columnMap 表字段 map 对象
19   */
20  List<T> selectByMap( Map<String, Object> columnMap);
21
22  /**
23   * 根据 entity 条件，查询一条记录
24   *
25   * @param queryWrapper 实体对象封装操作类（可以为 null）
26   */
27  T selectOne(wrapper<T> queryWrapper);
28
29  /**
30   * 根据 wrapper 条件，查询总记录数
31   *
32   * @param queryWrapper 实体对象封装操作类（可以为 null）
33   */
34  Integer selectCount(wrapper<T> queryWrapper);
35
36  /**

```

```

37  * 根据 entity 条件，查询全部记录
38  *
39  * @param queryWrapper 实体对象封装操作类（可以为 null）
40  */
41  List<T> selectList( wrapper<T> queryWrapper);
42
43  /**
44  * 根据 wrapper 条件，查询全部记录
45  *
46  * @param queryWrapper 实体对象封装操作类（可以为 null）
47  */
48  List<Map<String, Object>> selectMaps(wrapper<T> queryWrapper);
49
50  /**
51  * 根据 wrapper 条件，查询全部记录
52  * <p>注意： 只返回第一个字段的值</p>
53  *
54  * @param queryWrapper 实体对象封装操作类（可以为 null）
55  */
56  List<Object> selectObjs(wrapper<T> queryWrapper);
57
58  /**
59  * 根据 entity 条件，查询全部记录（并翻页）
60  *
61  * @param page          分页查询条件（可以为 RowBounds.DEFAULT）
62  * @param queryWrapper  实体对象封装操作类（可以为 null）
63  */
64  <E extends IPage<T>> E selectPage(E page, wrapper<T> queryWrapper);
65
66  /**
67  * 根据 wrapper 条件，查询全部记录（并翻页）
68  *
69  * @param page          分页查询条件
70  * @param queryWrapper  实体对象封装操作类
71  */
72  <E extends IPage<Map<String, Object>>> E selectMapsPage(E page, wrapper<T>
    queryWrapper);

```

selectById

用法：查询指定id 对象信息

需求：查询id=1的员工信息

```

1  @Test
2  public void testselectById(){
3      Employee employee = employeeMapper.selectById(1L);
4  }

```

执行完SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE id=?

```

selectBatchIds

用法：批量查询指定的id对象信息

需求：查询id=1,2,3的员工信息

```
1  @Test
2  public void testselectBatchIds(){
3      employeeMapper.selectBatchIds(Arrays.asList(1L, 2L, 3L));
4  }
```

执行完SQL

```
1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE id IN ( ?
    , ? , ? )
```

selectByMap

用法：查询满足条件实体信息，条件封装到map集合中

需求：查询age=18并且name=zhangsan的员工信息

```
1  @Test
2  public void testselectByMap(){
3      Map<String, Object> map = new HashMap<>();
4      map.put("age", 18);
5      map.put("name", "zhangsan");
6      List<Employee> employees = employeeMapper.selectByMap(map);
7  }
```

执行完SQL

```
1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE name = ?
    AND age = ?
```

selectCount

用法：查询满足条件实体信息记录总条数

需求：查询员工表记录条数

```
1  @Test
2  public void testselectCount(){
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      //如果统计所有数据: selectCount(null)
5      //Integer count = employeeMapper.selectCount(wrapper);
6      Integer count = employeeMapper.selectCount(wrapper);
7  }
```

执行完SQL

```
1  SELECT COUNT( 1 ) FROM employee
```

selectList

用法：查询满足条件实体信息记录， 返回List

需求：查询满足条件的所有的员工信息， 返回List

```
1  @Test
2  public void testselectList(){
3      Querywrapper<Employee> wrapper = new Querywrapper<>();
4      List<Employee> list = employeeMapper.selectList(wrapper);
5      for (Employee employee : list) {
6          System.out.println(employee);
7      }
8  }
```

执行完SQL

```
1  SELECT id,name,password,email,age,admin,dept_id FROM employee
```

selectMaps

用法：查询满足条件实体信息记录， 返回List<Map<String, Object>>

需求：查询满足条件的所有的员工信息， 返回List

```
1  @Test
2  public void testselectMaps(){
3      Querywrapper<Employee> wrapper = new Querywrapper<>();
4      List<Map<String, Object>> mapList = employeeMapper.selectMaps(wrapper);
5      for (Map<String, Object> map : mapList) {
6          System.out.println(map);
7      }
8  }
```

执行完SQL

```
1  SELECT id,name,password,email,age,admin,dept_id FROM employee
```

比较selectList 跟 selectMap 2个方法，一个返回值：List<实体> 一个是List

思考：开发中什么时候使用selectList，什么时候使用selectMaps呢？

```
1  /**
2   * 如果sql语句执行完之后，得到的数据能封装成实体对象时使用：selectList
3   * 如果sql语句执行完之后，得到的数据不能封装成实体对象时使用：selectMaps
4   *
5   * 比如：按部门id分组，统计每个部门员工的个数。
6   *
7   * select dept_id, count(id) count from employee group by dept_id
8   * 此时返回的2列无法封装成employee对象(原因：没有对应的属性与列对应)，此时要么封装成vo值
   对象，要么就是map对象。
9  */
```

selectPage

用法：分页查询满足条件实体信息记录

需求：查询第二页员工数据，每页显示3条，（分页返回的数据是员工对象）

mybatis-plus分页查询步骤分2步：

1>在配置类中添加分页插件（springboot项目）

```
1 //分页
2 @Bean
3 public MybatisPlusInterceptor mybatisPlusInterceptor() {
4     MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
5     PaginationInnerInterceptor paginationInnerInterceptor = new
6     PaginationInnerInterceptor(DbType.MYSQL);
7     paginationInnerInterceptor.setOverflow(true); //合理化
8     interceptor.addInnerInterceptor(paginationInnerInterceptor);
9     return interceptor;
10 }
```

2>编写分页代码

```
1 @Test
2 public void testselectPage(){
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4     //等价于: PageResult PageInfo
5     //参数1: 当前页, 参数2: 每页显示条数
6     Page<Employee> page = new Page<>(2,3);
7     Page<Employee> p = employeeMapper.selectPage(page, wrapper);
8     System.out.println(p == page); //true
9     System.out.println("当前页: " + page.getCurrent());
10    System.out.println("每页显示条数: " + page.getSize());
11    System.out.println("总页数: " + page.getPages());
12    System.out.println("总数: " + page.getTotal());
13    System.out.println("当前页数据: " + page.getRecords());
14 }
```

执行完SQL

```
1 SELECT COUNT(1) FROM employee
2
3 SELECT id,name,password,email,age,admin,dept_id FROM employee LIMIT ?,?
```

selectMapsPage

用法：跟selectPage一样，区别在与selectMapsPage返回分页数据集合泛型是Map, selectPage是实体对象。

selectOne

用法：查询满足条件实体对象，如果有多条数据返回，抛异常。

需求：查询name=zhangsan, password=1的员工数据

```

1  @Test
2  public void testselectOne(){
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.eq("name", "zhangsan");
5      wrapper.eq("password", "1");
6      Employee employee = employeeMapper.selectOne(wrapper);
7      System.out.println(employee);
8  }

```

执行完SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name = ?
    AND password = ?)

```

selectObjs

用法：查询满足条件实体对象，返回指定列的集合，如果没有指定列，默认返回第一列

需求：查询员工表所有员工名称

```

1  @Test
2  public void testselectObjs(){
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.select("name");
5      List<Object> list = employeeMapper.selectObjs(wrapper);
6      list.forEach(System.out::println);
7  }

```

执行完SQL

```

1  SELECT name FROM employee

```

到这，通用的Mapper接口就介绍完了。

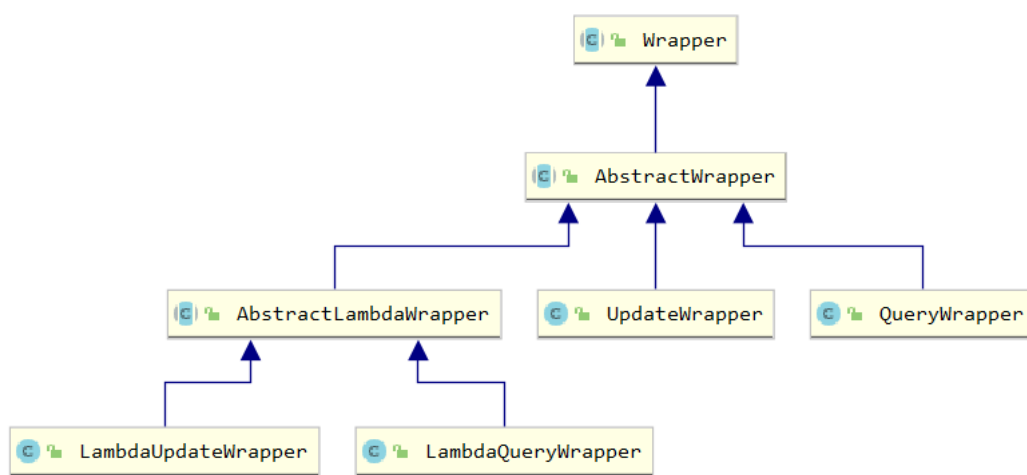
MyBatis-Plus条件构造器-Wrapper

Wrapper介绍

<https://baomidou.com/pages/10c804/>

条件构造器-Wrapper按照官方的给出的解释是用于生成 sql 的 where 条件，但在使用的过程中，发现 Wrapper 不仅仅是用于拼接 where 条件，可以用于拼接 set 语法，select 语法。所以，wrapper 准确的来说，更像 mybatis 中的动态标签。在 sql 任意位置完成 sql 片段组装。

类体系结构



<https://blog.csdn.net/langfelyes>

按功能分

修改型: UpdateWrapper LambdaUpdateWrapper

查询型: QueryWrapper LambdaQueryWrapper

按操作分

传统型: QueryWrapper UpdateWrapper

Lambda: LambdaQueryWrapper LambdaUpdateWrapper

不管怎么分，它们都有一个共同的父类：**AbstractWrapper**，父类里面定义wrapper核心的条件方法比如：

allEq eq ne gt ge lt le between notBetween like notLike likeLeft likeRight isNull isNotNull in notIn
inSql notInSql groupBy orderByAsc orderByDesc orderBy having func or and nested apply last
exists notExists 等

Update类型的Wrapper独有操作方法：**set setSql**

Query类型的Wrapper独有操作方法：**select**

更新-Wrapper

更新类型的Wrapper有2个

UpdateWrapper 与 **LambdaUpdateWrapper** 这2个Wrapper都是用于对数据修改的条件构造器。

需求：将id=1的员工name修改为zhangsanfeng

UpdateWrapper


```

1  @Test
2  public void testUpdate(){
3      UpdateWrapper<Employee> wrapper = new UpdateWrapper<>();
4      wrapper.eq("id", 1L);
5      wrapper.set("name", "zhangsanfeng");
6      employeeMapper.update(null, wrapper);
7  }

```

独有方法操作

需求1其实还是一种写法，使用到：setSql

```

1  @Test
2  public void testUpdate(){
3      UpdateWrapper<Employee> wrapper = new UpdateWrapper<>();
4      wrapper.eq("id", 1L);
5      //wrapper.set("name", "zhangsanfeng");
6      wrapper.setSql("name='zhangsanfeng'");
7      employeeMapper.update(null, wrapper);
8  }

```

setSql表示使用拼接SQL片段方式进行操作，区别从打印的SQL可以看出

```

1  -- set方式:
2  UPDATE employee SET name=? WHERE (id = ?)
3  -- 参数:
4  zhangsanfeng(String), 1(Long)
5
6  -- setSql方式:
7  UPDATE employee SET name='zhangsanfeng' WHERE (id = ?)
8  -- 参数:
9  1(Long)

```

2个对比，可以很明显看出set方式使用预编译操作方法，而setSql 直接进行拼接，可能存sql注入风险，不建议使用，操作上更推荐set方式。

问题

从上面更新代码上看，开发存在一定隐患，设置条件/更新的列都是直接使用字符串，如果手误写错，编译期是无法检查的，需要等执行之后才会出现异常。存在一定瑕疵。怎办，改进解决方案就是：

LambdaUpdateWrapper

LambdaUpdateWrapper

```

1  @Test
2  public void testUpdateLambda(){
3      LambdaUpdateWrapper<Employee> wrapper = new LambdaUpdateWrapper<>();
4      wrapper.eq(Employee::getId, 1L);
5      wrapper.set(Employee::getName, "zhangsanfeng");
6      employeeMapper.update(null, wrapper);
7  }

```

LambdaUpdateWrapper 的update 跟 UpdateWrapper 的update操作一样，唯一区别点在于 UpdateWrapper的操作列都是以字符串的形式存在，LambdaUpdateWrapper 的操作列使用lambda表达式。

乍一看，感觉高大上，点开查看源码，你发现：Employee::getName其实最终还是会解析出name属性来。简单的理解就是通过Employee类中的getName方法，将name属性解析出来。这么折腾好处是**比直接写字符串方式多了一种操作前检查(避免出现手误)**

```
1  @Override
2  public LambdaUpdateWrapper<T> set(boolean condition, SFunction<T, ?> column,
3  Object val) {
4      if (condition) {
5          sqlSet.add(String.format("%s=%s", columnToString(column),
6          formatSql("{0}", val)));
7      }
8      return typedThis;
9  }
10
11 protected String columnToString(SFunction<T, ?> column, boolean onlyColumn)
12 {
13     return getColumn(LambdaUtils.resolve(column), onlyColumn);
14 }
15
16 private String getColumn(SerializedLambda lambda, boolean onlyColumn) {
17     Class<?> aClass = lambda.getInstantiatedType();
18     tryInitCache(aClass);
19     String fieldName =
20     PropertyNamer.methodToProperty(lambda.getImplMethodName());
21     ColumnCache columnCache = getColumnCache(fieldName, aClass);
22     return onlyColumn ? columnCache.getColumn() :
23     columnCache.getColumnSelect();
24 }
```

查询-Wrapper

查询类型的Wrapper有2个

QueryWrapper 与 **LambdaQueryWrapper** 这2个Wrapper都是用于对数据修改的条件构造器。

需求：查询name=zhangsan, age=18 的员工信息

QueryWrapper

```
1  @Test
2  public void testQuery(){
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.eq("name", "zhangsan");
5      wrapper.eq("age", 18);
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }
```

独有方法操作

query类型wrapper也有自己的独有的方法：**select**

需求：查询name=zhangsan, age=18 的员工信息，只需要查询id, name 2列

```
1  @Test
2  public void testQuery(){
3      Querywrapper<Employee> wrapper = new Querywrapper<>();
4      wrapper.select("id", "name");
5      wrapper.eq("name", "zhangsan");
6      wrapper.eq("age", 18);
7      List<Employee> list = employeeMapper.selectList(wrapper);
8  }
```

里面的wrapper.select("id", "name"); 方法表示查询结果返回id, name 2列，等价于：

```
1  SELECT id,name FROM employee WHERE (name = ? AND age = ?)
2
3  -- 如果不加select 方法，默认查询所有列，等价于：
4
5  SELECT * FROM employee WHERE (name = ? AND age = ?)
```

问题

同UpdateWrapper

LambdaQueryWrapper

```
1  @Test
2  public void testQueryLambda(){
3      LambdaQuerywrapper<Employee> wrapper = new LambdaQuerywrapper<>();
4      wrapper.eq(Employee::getName, "zhangsan");
5      wrapper.eq(Employee::getAge, 18);
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }
```

2个Wrapper操作跟上面update操作一个样，都是将字符串的操作列转换成lambda方式，其实没有本质上的区别。

构建Wrapper实例

```
1  @Test
2  public void testWrapper(){
3      //wrapper对象的创建
4      //query
5      Querywrapper<Employee> querywrapper1 = new Querywrapper<>();
6      Querywrapper<Employee> querywrapper2 = wrappers.<Employee>query();
7
8      LambdaQuerywrapper<Employee> querywrapper3 = new LambdaQuerywrapper<>();
9      LambdaQuerywrapper<Employee> querywrapper4 = wrappers.
10     <Employee>lambdaQuery();
11     LambdaQuerywrapper<Employee> querywrapper5 = querywrapper1.lambda();
```

```

12      //update
13      UpdateWrapper<Employee> updateWrapper1 = new UpdateWrapper<>();
14      UpdateWrapper<Employee> updateWrapper2 = wrappers.<Employee>update();
15
16      LambdaUpdateWrapper<Employee> updateWrapper3 = new LambdaUpdateWrapper<>
17      ();
18      LambdaUpdateWrapper<Employee> updateWrapper4 = wrappers.
19      <Employee>lambdaUpdate();
20      LambdaUpdateWrapper<Employee> updateWrapper5 = updateWrapper1.lambda();
21  }

```

其中的Wrappers 是mybatis-plus官方提供的构建Wrapper实例的工具类。

Wrapper选择

单从可读性来看，建议使用传统的Wrapper，从预防手误来看，建议使用LambadWrapper

实际开发以公司规定为准则。

Wrapper练习

将之前写的CRUD操作转换成LambadWrapper方法

MyBatis-Plus条件查询

概要

1> 前一章节重点介绍了MyBatis-Plus中的wrapper体系与操作方式，本章节重点讲的wrapper中的query操作，以QueryWrapper 实例为切入点，讲解常用的条件查询，UpdateWrapper涉及到的条件同理可得即可。

2>一样沿用前几篇使用的employee 表/实体/mapper等代码

```

1  @Setter
2  @Getter
3  @ToString
4  @TableName("employee")
5  public class Employee {
6      @TableId(value = "id", type= IdType.AUTO)
7      private Long id;
8      private String name;
9      private String password;
10     private String email;
11     private int age;
12     private int admin;
13     private Long deptId;
14 }

```

列投影-select

select重载的方法有3个，其实就2个，一个功能重复了

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();  
wrapper.select  
    m select(String... columns) QueryWrapper<Employee>  
    m select(Class<Employee> entityClass, Predicate<TableFieldInfo> predicate) QueryWrapper<Employee>  
    m select(Predicate<TableFieldInfo> predicate) QueryWrapper<Employee>
```

用法：从查询结果集中挑选指定列

需求：查询所有员工信息，返回员工的age跟name属性

select(String....)

```
1 @Test  
2 public void testQuery1() {  
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();  
4     wrapper.select("name", "age"); //列的投影， 挑选哪一些列， 参数是列名  
5     List<Employee> list = employeeMapper.selectList(wrapper);  
6     System.out.println(list);  
7 }
```

执行后SQL

```
1 SELECT name,age FROM employee
```

使用注意，如果列使用别名，那就按照sql语法编写别名，换句话说讲select里面参数其实就是sql中的select语句，语法一样

```
1 wrapper.select("name as ename", "age as eage");
```

sql效果

```
1 SELECT name as ename,age as eage FROM employee
```

还有简便的写法：sql片段方式

```
1 @Test  
2 public void testQuery1() {  
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();  
4     //wrapper.select("name", "age"); //列的投影， 挑选哪一些列， 参数是列名  
5     wrapper.select("name, age"); //参数是sql 片段  
6     List<Employee> list = employeeMapper.selectList(wrapper);  
7     System.out.println(list);  
8 }
```

注意，这种写法参数为sql片接，容易出现sql注入风险

select(Class entityClass, Predicate predicate)

这个方法理解起来相对麻烦

entityClass：表示指定查询实体对象，比如：当前操作员工表，那么指定Employee.class

predicate：判断型函数接口，接口有个test方法，参数是TableFieldInfo

TableFieldInfo: 表字段信息对象, 将表的列抽象成java对象

方法意思: 指定查询对象, 使用predicate定义条件列的规则, 满足条件的列, 挑选出来。

需求: 查询列名以 字母 "e" 结尾的列

```
1  @Test
2  public void testQuery1_1() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.select(Employee.class, tableFieldInfo ->
5          tableFieldInfo.getColumn().endsWith("e"));
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  SELECT id,name,age FROM employee
```

查询出来的列有id, name, age。id是默认查询, name 跟 age 列都有e字母, 所以能查询出来

需求2: 查询列名长度大于 5 的列

```
1  @Test
2  public void testQuery1_1() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.select(Employee.class, tableFieldInfo ->
5          tableFieldInfo.getColumn().length() > 5);
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  SELECT id,password,dept_id FROM employee
```

排序

MyBatis-Plus的排序有8个, 具体分为3种: orderByAsc / orderByDesc / orderBy

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.orderby~
```

m	orderBy(boolean condition, boolean isAsc, String... columns)	QueryWrapper<Employee>
m	orderByAsc(String... columns)	QueryWrapper<Employee>
m	orderByAsc(boolean condition, String... columns)	QueryWrapper<Employee>
m	orderByDesc(String... columns)	QueryWrapper<Employee>
m	orderByDesc(boolean condition, String... columns)	QueryWrapper<Employee>
m	orderByAsc(String column)	QueryWrapper<Employee>
m	orderByDesc(String column)	QueryWrapper<Employee>

*

orderByAsc : 正序排 3个

orderByDesc : 倒序排 3个

orderBy: 1个

用法: 对查询结果集排序, 可以单列排, 可以多列排

orderByAsc(String column)/orderByDesc(String column) : 单列正排序/倒排序

需求: 查询所有员工信息, 按age正序排/倒序排

```
1  @Test
2  public void testQuery2() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.orderByAsc("age"); //正序
5      //wrapper.orderByDesc("age"); //倒序
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  -- 正序
2  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC
3  -- 倒序
4  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   DESC
```

需求: 查询所有员工信息, 按age正序排, 如果age一样, 按id正序排

需求: 查询所有员工信息, 按age正序排, 如果age一样, 按id倒序排

```
1  @Test
2  public void testQuery2() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.orderByAsc("age");
5      wrapper.orderByAsc("id"); //正序
6      //wrapper.orderByDesc("id"); //倒序
7      List<Employee> list = employeeMapper.selectList(wrapper);
8      System.out.println(list);
9  }
```

执行后SQL

```
1  -- 先正序后正序
2  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC,id ASC
3  -- 先正序后倒序
4  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC,id DESC
```

这里注意, 排序列谁先谁后是根据orderBy方法调用顺序决定的, 操作时务必小心

orderByAsc(String... column)/orderByDesc(String... column) : 多列正排序/倒排序

需求：查询所有员工信息，按age正序排，如果age一样，按id正序排

需求：查询所有员工信息，按age倒序排，如果age一样，按id倒序排

```
1  @Test
2  public void testQuery2() {
3      Querywrapper<Employee> wrapper = new Querywrapper<>();
4      wrapper.orderByAsc("age", "id"); //都正序
5      //wrapper.orderByDesc("age", "id"); //都倒序
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  -- 都是正序
2  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC,id ASC
3
4  -- 都是倒序
5  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   DESC,id DESC
```

从上面也可以看出多列排序其实跟单列排序本质一样没啥区别

orderByAsc(boolean condition, String... column)/orderByDesc(boolean condition, String... column)：带条件判断多列正排序/倒排序

condition：排序控制开关，当condition这个参数为true时，才会拼接sql排序语句

需求：查询所有员工信息，按age正序排

```
1  @Test
2  public void testQuery2_1() {
3      Querywrapper<Employee> wrapper = new Querywrapper<>();
4      wrapper.orderByAsc(true, "age"); //true
5      //wrapper.orderByAsc(false, "age"); //false
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行完SQL

```
1  -- true
2  SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC
3  -- false
4  SELECT id,name,password,email,age,admin,dept_id FROM employee
```

orderBy(boolean condition, boolean isAsc,String... columns)：带条件判断多列排序

condition：当condition这个参数为true时，才对sql语句进行排序操作

isAsc: 是否为正序排, true: 表示正序, false: 表示倒序

columns: 排序的列, 可以多列, 可单列

需求: 查询所有员工信息, 按age正序排

需求: 查询所有员工信息, 按age倒序排

```
1 @Test
2 public void testQuery2_2() {
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4     wrapper.orderBy(true, true, orderBy); //正序
5     wrapper.orderBy(true, false, orderBy); //倒序
6     List<Employee> list = employeeMapper.selectList(wrapper);
7     System.out.println(list);
8 }
```

执行完SQL

```
1 -- 正序
2 SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   ASC
3 -- 倒序
4 SELECT id,name,password,email,age,admin,dept_id FROM employee ORDER BY age
   DESC
```

比较运算

allEq

全等比较符, 它重载方法有6个, 功能由传入的参数决定

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.allEq(
    m allEq(Map<String, V> params) QueryWrapper<Employee>
    m allEq(Map<String, V> params, boolean null2IsNull) QueryWrapper<Employee>
    m allEq(BiPredicate<String, V> filter, Map<String, V> params) QueryWrapper<Employee>
    m allEq(boolean condition, Map<String, V> params, boolean null2IsNull) QueryWrapper<Employee>
    m allEq(BiPredicate<String, V> filter, Map<String, V> params, boolean null2IsNull) QueryWrapper<Employee>
    m allEq(boolean condition, BiPredicate<String, V> filter, Map<String, V> params, boolean null2IsNull) QueryWrapper<Employee>
```

用法: 全等, 所有条件必须相等, 条件使用Map<String, Object>封装, key:条件列, value: 条件值

需求: 查询name=dafei, age=18的员工信息

allEq(Map<String, V> params)

```

1  @Test
2  public void testQuery3() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      Map<String, Object> map = new HashMap<>();
5      map.put("name", "dafei");
6      map.put("age", 18);
7      wrapper.allEq(map);
8      List<Employee> list = employeeMapper.selectList(wrapper);
9      System.out.println(list);
10 }

```

执行后SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name = ?
    AND age = ?)

```

allEq(Map<String, V> params, boolean null2IsNull)

null2IsNull: 如果params参数中, 如果某个key对应的value值为null时,

true: 表示使用is null 替换, false: 表示忽略该条件

```

1  @Test
2  public void testQuery3_1() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      Map<String, Object> map = new HashMap<>();
5      map.put("name", null);
6      map.put("age", 18); //value值为null
7      //wrapper.allEq(map, true); //拼接
8      wrapper.allEq(map, false); //不拼接
9      List<Employee> list = employeeMapper.selectList(wrapper);
10     System.out.println(list);
11 }

```

```

1  -- true 拼接
2  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name IS
    NULL AND age = ?)
3
4  -- false 不拼接
5  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age = ?)

```

allEq(BiPredicate<String, V> filter, Map<String, V> params)

filter: 判断型函数接口, 用于过滤params中的条件, 接口有个test方法

需求: 查询满足指定params条件的员工数据, 附加条件, 如果params的key长度小于4, 不参与sql条件查询

```

1  @Test
2  public void testQuery3_2() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      Map<String, Object> map = new HashMap<>();
5      map.put("name", "dafei");
6      map.put("age", 18);
7      wrapper.allEq((key, value)->{
8          return key.length() > 4;
9      }, map);
10     List<Employee> list = employeeMapper.selectList(wrapper);
11     System.out.println(list);
12 }

```

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name =
   ?)

```

params参数key值有age，跟name，函数接口做了限制，key长度必须大于4，所以age被排除，符合要求只有name这个条件。sql值拼接name条件

allEq(boolean condition, Map<String, V> params, boolean null2IsNull)

condition: allEq控制开关，当condition为true，执行allEq语法，拼接查询条件，为false，不执行。

allEq(boolean condition, BiPredicate<String, V> filter, Map<String, V> params, boolean null2IsNull);

跟上面介绍重复了，同理可得即可。

eq

重载方法有2个

```

QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.eq

```

m	eq(String column, Object val)	QueryWrapper<Employee>
m	eq(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

用法：等值条件过滤，sql: where 列 = 值

eq(String column, Object value): 等值匹配

需求：查询name=dafei的员工信息

```

1  @Test
2  public void testQuery4_1() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.eq("name", "dafei");
5      List<Employee> list = employeeMapper.selectList(wrapper);
6      System.out.println(list);
7  }

```

执行后SQL

```
1 | SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name = ?)
```

eq(boolean condition, String column, Object value): 带开关的等值匹配

带开关的eq 操作， 使用跟上面操作一样

ne

重载方法有2个

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.ne
```

m	ne(String column, Object val)	QueryWrapper<Employee>
m	ne(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

用法：不等条件过滤， sql： where 列 <> 值

ne(String column, Object value): 不等值匹配

需求： 查询name != dafei的用户信息

```
1 | @Test
2 | public void testQuery4_2() {
3 |     QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4 |     wrapper.ne("name", "dafei");
5 |     List<Employee> list = employeeMapper.selectList(wrapper);
6 |     System.out.println(list);
7 | }
```

执行后SQL

```
1 | SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name <> ?)
```

ne(boolean condition, String column, Object value): 带开关的不等值匹配

带开关的ne 操作， 使用跟上面操作一样

gt/ge

gt有2个重载方法， ge也有2个重载方法

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.gt
```

m	gt(String column, Object val)	QueryWrapper<Employee>
m	gt(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.ge
```

m	ge(String column, Object val)	QueryWrapper<Employee>
m	ge(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

用法gt: great than 大于, sql: where 列 > 值

用法ge: greates than and equals, 大于等于, sql: where 列 >= 值

gt(String column, Obejct value) / ge(String column, Obejct value): 大于/大于等于比较

需求: 查询 age 大于 18 的员工信息

需求: 查询age 大于等于 18 的员工信息

```
1  @Test
2  public void testQuery4_3() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.gt("age", 18); //大于 18
5      //wrapper.ge("age", 18); //大于等于18
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  -- gt
2  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age > ?)
3
4  -- ge
5  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age >=
  ?)
```

gt(boolean condition, String column, Obejct value) / ge(boolean condition, String column, Obejct value): 带开关的大于/大于等于比较

带开关的gt/ ge 操作, 使用跟上面操作一样

lt/le

lt有2个重载方法, le也有2个重载方法

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.lt
```

m	lt(String column, Object val)	QueryWrapper<Employee>
m	lt(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.le
```

m	le(String column, Object val)	QueryWrapper<Employee>
m	le(boolean condition, String column, Object val)	QueryWrapper<Employee>

*

用法lt: less than 小于, sql: where 列 < 值

用法le: less than and equals, 小于等于, sql: where 列 <= 值

lt(String column, Object value) / le(String column, Object value): 小于/小于等于比较

需求: 查询 age 小于 18 的员工信息

需求: 查询age 小于等于 18 的员工信息

```
1  @Test
2  public void testQuery4_4() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.lt("age", 18); //小于 18
5      //wrapper.le("age", 18); //小于等于18
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }
```

执行后SQL

```
1  -- lt
2  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age < ?)
3
4  -- le
5  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age <=
  ?)
```

lt(boolean condition, String column, Object value) / le(boolean condition, String column, Object value): 带开关的小于/小于等于比较

带开关的lt/ le 操作, 使用跟上面操作一样

isNull/NotNull

isNull 重载2个方法, isNotNull重载2个方法

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.isNull
```

m isNull(String column)	QueryWrapper<Employee>
m isNull(boolean condition, String column)	QueryWrapper<Employee>

*

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.isno
```

m isNotNull(String column)	QueryWrapper<Employee>	*
m isNotNull(boolean condition, String column)	QueryWrapper<Employee>	

用法isNull: 列判null条件, sql: where 列 is null

用法isNotNull: 列判定不为null条件, sql: where 列 is not null

isNull(String column, Object value) / isNotNull(String column, Object value): 列是否为null/不为null判断

需求: 查询 dept_id 为null 的员工信息

需求: 查询 dept_id 不为null 的员工信息

```

1  @Test
2  public void testQuery4() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.isNull("dept_id"); // 为null
5      //wrapper.isNotNull("dept_id"); // 不为null
6      List<Employee> list = employeeMapper.selectList(wrapper);
7      System.out.println(list);
8  }

```

执行后SQL

```

1  -- is null
2  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (dept_id
   is null)
3
4  -- is not null
5  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (dept_id
   is not null)

```

isNull(boolean condition, String column, Object value) / isNotNull(boolean condition, String column, Object value): 带开关的列是否为null/不为null判断

带开关的null/ isNotNull 操作，使用跟上面操作一样

in/notIn

in重载4个方法，notIn重载4个方法

```

QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.in

```

m in(String column, Object... values)	QueryWrapper<Employee>
m in(String column, Collection<?> coll)	QueryWrapper<Employee>
m in(boolean condition, String column, Object... values)	QueryWrapper<Employee>
m in(boolean condition, String column, Collection<?> coll)	QueryWrapper<Employee>

*

```

QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.notIn

```

m notIn(String column, Object... value)	QueryWrapper<Employee>
m notIn(String column, Collection<?> coll)	QueryWrapper<Employee>
m notIn(boolean condition, String column, Object... values)	QueryWrapper<Employee>
m notIn(boolean condition, String column, Collection<?> coll)	QueryWrapper<Employee>

*

用法in：列在指定列表数据中，sql：where 列 in (值1，值2，值3.....)

用法notIn：列不在指定列表数据中，sql：where 列 not in (值1，值2，值3.....)

in(String column, Object...value) / notIn(String column, Object... value): 可变参数方式

需求：查询 id = 1 或者 id = 2 或者 id = 3 的员工信息

需求：查询 id != 1 或者 id != 2 或者 id != 3 的员工信息

```

1  @Test
2  public void testQuery5() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.in("id", 1L, 2L, 3L);    // in
5      //wrapper.notIn("id", 1L, 2L, 3L); //not in
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

执行后SQL

```

1  -- in
2  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (id IN
   (?, ?, ?))
3
4  -- not in
5  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (id NOT
   IN (?, ?, ?))

```

in(String column, Collection<?> coll) / notIn(String column, Collection<!> coll): 集合方式

```

1  @Test
2  public void testQuery5() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.in("id", Arrays.asList(1L, 2L, 3L));    //in
5      //wrapper.notIn("id", Arrays.asList(1L, 2L, 3L)); //not in
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

执行后SQL

跟上面操作一模一样

in(boolean condition, String column, Object...value) / notIn(boolean condition, String column, Object... value): 可变参数方式

in(boolean condition, String column, Collection<!> coll) / notIn(boolean condition, String column, Collection<!> coll): 数组的方式

带开关的in/ notIn 操作， 使用跟上面操作一样

inSql/notInSql

inSql重载2个方法， notInSql重载2个方法

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.inSql~
```

m	inSql(String column, String inValue)	QueryWrapper<Employee>
m	inSql(boolean condition, String column, String inValue)	QueryWrapper<Employee>

*

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
```

```
wrapper.notInSql~
```

m	notInSql(String column, String inValue)	QueryWrapper<Employee>
m	notInSql(boolean condition, String column, String inValue)	QueryWrapper<Employee>

*

用法inSql: 列在指定列表数据中, sql: where 列 in (值1, 值2, 值3.....)

用法notInSql: 列不在指定列表数据中, sql: where 列 not in (值1, 值2, 值3.....)

inSql(String column,String value) / notInSql(String column, String value)

跟之前in /notIn 方法的区别是, 方法参数value是一个sql片段

需求: 查询 id = 1 或者 id = 2 或者 id = 3 的员工信息

需求: 查询 id != 1 或者 id != 2 或者 id != 3 的员工信息

```
1 @Test
2 public void testQuery5() {
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4     wrapper.inSql("id","1, 2, 3");    // in,value是sql片段
5     //wrapper.notInSql("id","1, 2, 3"); //not in
6     List<Employee> list = employeeMapper.selectList(wrapper);
7 }
```

执行后SQL

```
1 -- in sql
2 SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (id IN
  (1,2,3))
3
4 -- not in sql
5 SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (id NOT
  IN (1,2,3))
```

inSql(boolean condition, String column, String value) / notInSql(boolean condition, String column, String value)

带开关的inSql/ notInSql 操作, 使用跟上面操作一样

模糊查询

like/notLike/likeLeft/likeRight

like 相关的方法有8个

```
QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.like~
```

```
m like(String column, Object val)
m like(boolean condition, String column, Object val)
m likeRight(String column, Object val)
m likeRight(boolean condition, String column, Object val) *
m likeLeft(String column, Object val)
m likeLeft(boolean condition, String column, Object val)
m notLike(String column, Object val)
m notLike(boolean condition, String column, Object val)
```

用法：模糊查询，sql：

like: where 列 like "%值%"

notLike: where 列 not like "%值%"

likeLeft: where 列 like "%值"

likeRight: where 列 like "值%"

需求：查询名字中含有“ye”字样的员工信息 like

需求：查询名字中不含有“ye”字样的员工信息 notLike

需求：查询名字中以“ye”字样结尾员工信息 likeLeft

需求：查询名字中以“ye”字样开头员工信息 likeRight

```
1 @Test
2 public void testQuery6() {
3     QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4     wrapper.like("name", "ye"); // like "%ye%"
5     //wrapper.notLike("name", "ye"); // not like "%ye%"
6     //wrapper.likeLeft("name", "ye"); // like "%ye"
7     //wrapper.likeRight("name", "ye"); // like "ye%"
8     List<Employee> list = employeeMapper.selectList(wrapper);
9 }
```

执行后SQL

```
1 SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name
   LIKE ?)
2
3 -- "%ye%" -- like
4 -- "%ye" -- likeLeft
5 -- "ye%" -- likeRight
```

%%

where name like _a; //代表name列满足2个字符串，第二字符是a字样

逻辑运算

and/or/nested

and

```
wrapper.and(
    m and(Consumer<UpdateWrapper<Employee>> consumer)
    m and(boolean condition, Consumer<UpdateWrapper<Employee>> consumer)
```

用法：逻辑与，sql：where 条件1 and 条件2

需求：查询年龄介于18~30岁的员工信息

```

1  @Test
2  public void testQuery8() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      //多个条件默认使用and
5      wrapper.ge("age", 18).le("age", 30);
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

执行后SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age >= ?
    AND age <= ?)

```

条件的链式连接，默认是使用and进行连接

or

```

wrapper.or
  m or()
  m or(boolean condition)
  m or(Consumer<UpdateWrapper<Employee>> consumer)
  m or(boolean condition, Consumer<UpdateWrapper<Employee>> consumer)

```

用法：逻辑或，sql：where 条件1 or 条件2

需求：查询年龄小于18或大于30岁的员工信息

```

1  @Test
2  public void testQuery8() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      //多个条件使用or
5      wrapper.lt("age", 18)
6          .or()
7          .gt("age", 30);
8      List<Employee> list = employeeMapper.selectList(wrapper);
9  }

```

执行后SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (age < ?
    OR age > ?)

```

多个条件连接默认使用and拼接，如果是or操作需要明确调用or方法。

逻辑条件嵌套

需求：查询名字带有"ye"字样，或者年龄介于18~30岁的员工信息

```

1  @Test
2  public void testQuery10() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.like("name", "ye")
5          .or(wr -> wr.le("age", 30).ge("age", 18));
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

执行后SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name
   LIKE ? OR (age <= ? AND age >= ?))

```

需求：查询名字带有"ye"字样，并且年龄小于18或大于30岁的员工信息

```

1  @Test
2  public void testQuery11() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.like("name", "ye")
5          .and(wr -> wr.lt("age", 18).or().gt("age", 30));
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

执行后SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE (name
   LIKE ? AND (age < ? OR age > ?))

```

nested

```

wrapper.nested(
    m nested(Consumer<QueryWrapper<Employee>> consumer)
    m nested(boolean condition, Consumer<QueryWrapper<Employee>> consumer)

```

逻辑条件嵌套，等价于sqlwhere添加中的小括号，可以改变条件优先级。

需求：查询名字带有"ye"字样，或者年龄介于18~30岁的员工信息

```

1  @Test
2  public void testQuery10() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.like("name", "ye").or()
5          .nested(wr -> wr.le("age", 30).ge("age", 18));
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

需求：查询名字带有"ye"字样，并且年龄小于18或大于30岁的员工信息

```

1  @Test
2  public void testQuery11() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.like("name", "ye")
5          .nested(wr -> wr.lt("age", 18).or().gt("age", 30));
6      List<Employee> list = employeeMapper.selectList(wrapper);
7  }

```

分组查询

group by/having

group by

重载的方法有3个

分组函数，指定列进行封装，可以多个也可以1个

```

QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.groupBy
    m groupBy(String... columns) *
    m groupBy(boolean condition, String... columns)
    m groupBy(String column)

```

having

重置方法2个

用于筛选分组之后条件数据

```

QueryWrapper<Employee> wrapper = new QueryWrapper<>();
wrapper.having
    m having(String sqlHaving, Object... params) *
    m having(boolean condition, String sqlHaving, Object... params)

```

用法：分组查询，sql：group by 列 having 条件

需求：查询每个部门员工个数

```

1  @Test
2  public void testQuery12() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.select("dept_id", "count(id) count");
5      wrapper.groupBy("dept_id");
6      List<Map<String, Object>> mapList = employeeMapper.selectMaps(wrapper);
7      mapList.forEach(System.out::println);
8  }

```

执行后SQL

```

1  SELECT dept_id,count(id) count FROM employee GROUP BY dept_id

```

需求：查询每个部门员工个数，筛选出个数大于3的部门

```

1  @Test
2  public void testQuery13() {
3      QueryWrapper<Employee> wrapper = new QueryWrapper<>();
4      wrapper.select("dept_id", "count(id) count");
5      wrapper.groupBy("dept_id");
6      //wrapper.having("count > 3");
7      wrapper.having("count > {0}", 3);
8      List<Map<String, Object>> mapList = employeeMapper.selectMaps(wrapper);
9      mapList.forEach(System.out::println);
10 }

```

执行后SQL

```

1  -- count > 3
2  SELECT dept_id,count(id) count FROM employee GROUP BY dept_id HAVING count >
3  3
4  -- count > {0}
5  SELECT dept_id,count(id) count FROM employee GROUP BY dept_id HAVING count >
6  ?

```

自定义SQL

上面的wrapper查询，针对简单sql场景非常简便，但是如果业务相对复杂，需要要求更灵活的SQL时(比如多表关联查询，动态sql等)，wrapper有点力不从心了，此时可以使用自定义sql方式。这种方式不是啥新鲜货，其实就是还原之前的Mybatis的XxxxMapper.xml写法。

Mapper接口

```

1  public interface EmployeeMapper extends BaseMapper<Employee> {
2      List<Employee> listByXmlSingle(); //自定义方法
3  }

```

Mapper.xml

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3  "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5  <mapper namespace="com.langfeiyes.mp.mapper.EmployeeMapper" >
6
7      <resultMap id="BaseResultMap" type="com.langfeiyes.mp.domain.Employee" >
8          <id column="id" jdbcType="BIGINT" property="id" />
9          <result column="name" jdbcType="VARCHAR" property="name" />
10         <result column="password" jdbcType="VARCHAR" property="password" />
11         <result column="email" jdbcType="VARCHAR" property="email" />
12         <result column="age" jdbcType="INTEGER" property="age" />
13         <result column="admin" jdbcType="BIT" property="admin" />
14         <result column="dept_id" property="deptId" />
15     </resultMap>
16
17     <select id="listByXmlSingle" resultMap="BaseResultMap">
18         select id, name, password, email, age, admin, dept_id
19     </select>
20 </mapper>

```

```

18         from employee
19     </select>
20
21 </mapper>

```

测试

```

1 @Test
2 public void testQuery14() {
3     List<Employee> list = employeeMapper.listByXmlSingle();
4 }

```

到这常用的条件方法就介绍到这，剩下的条件方法同理即可，技巧就是sql的关键语法就是Wrapper对象的方法。

MyBatis-Plus关联关系

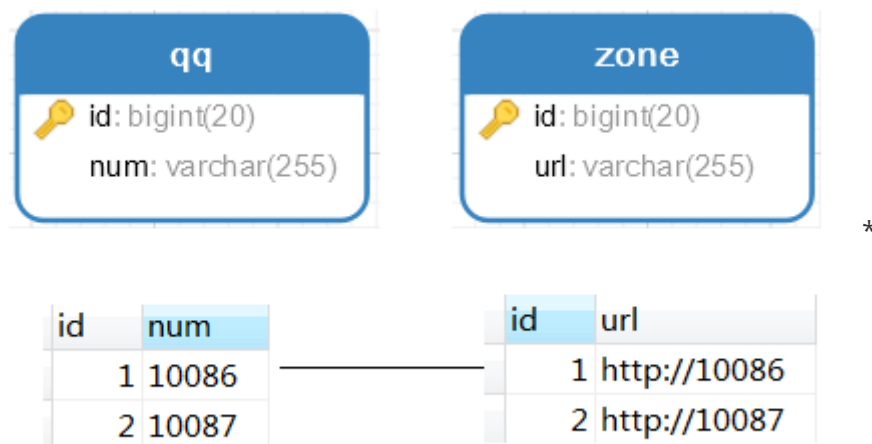
关系型数据对象与对象间以下几种关联关系

一对一关系

A表一条数据只能映射到B表唯一——条数据

以QQ号与QQ空间为例子：一个QQ号对应一个QQ空间

表设计：



关系维护：同id原则，公用id

对象设计：

```

1  @Data
2  public class QQ {
3      private Long id;
4      private String num;
5      private Zone zone;    //维护关系由主表维护
6  }
7
8  @Data
9  class Zone {
10     private Long id;
11     private String url;
12 }

```

MyBatis

在做查询时，mybatis的写法

```

1  <resultMap id="BaseResultMap" type="com.langfeiyes.mp.domain.QQ" >
2      <id column="id" jdbcType="BIGINT" property="id" />
3      <result column="num" jdbcType="VARCHAR" property="num" />
4      <result column="z_id" property="zone.id" />
5      <result column="z_url" property="zone.url" />
6  </resultMap>
7
8  <select id="selectSingle" resultMap="BaseResultMap">
9      select q.id, q.num, z.id z_id, z.url z_url from qq q left join zone z on
10     q.id = z.id
11 </select>

```

MyBatis-Plus

MyBatis-Plus使用额外sql方式进行查询

```

1  @Test
2  public void testQuery10() {
3      QueryWrapper<QQ> wrapper = new QueryWrapper<>();
4      List<QQ> list = QQMapper.selectList(wrapper);    //先查素有QQ对象
5      for(QQ q : list){
6          q.setZone(zoneMapper.selectById(q.getId()));    //通过QQ对象带有的zoneid查
7          // 询zone对象(id一样)
8      }
9  }

```

一对多关系

A表一条数据可以映射B表多条数据

以部门与员工为例子：一个部门可以对应多个员工，此时主角是部门

表设计：

一方



id	dname
1	小卖部
2	大卖部

多方



id	ename	deptId
1	dafei	1
2	xiaofei	1

*

关系维护：在多方(员工)设置外键，引致部门主键

对象设计：

```
1  @Data
2  class Department {
3      private Long id;
4      private String dname;
5      //1对多关系，一方为主导，对象中关系维护交给一方
6      private List<Employee> es = new ArrayList<>();
7  }
8
9  @Data
10 class Employee {
11     private Long id;
12     private String ename;
13 }
```

MyBatis

一堆多关系查询，需要使用Collection进行额外配置

```
1  <resultMap id="BaseResultMap" type="com.langfeiyes.mp.domain.Department" >
2      <id column="id" jdbcType="BIGINT" property="id" />
3      <result column="dname" jdbcType="VARCHAR" property="dname" />
4      <collection property="es" ofType="com.langfeiyes.mp.domain.Employee">
5          <id column="e_id" property="id"/>
6          <result column="e_name" property="ename"/>
7      </collection>
8
9  </resultMap>
10
11 <select id="selectList" resultMap="BaseResultMap">
12     select d.id, d.dname, e.id e_id, e.ename e_name
```

```

13     from department d left join employee e on d.id = e.deptid
14 </select>

```

MyBatis-Plus

MyBatis-Plus使用额外sql方式进行查询

```

1  @Test
2  public void testQuery10() {
3      QueryWrapper<Department> wrapper = new QueryWrapper<>();
4      List<Department> list = departmentMapper.selectList(wrapper);    //先查部门
    对象集合
5      for(Department d : list){
6          d.setEs(employeeMapper.selectByDeptId(d.getId()));    //通过部门id查询
    员工集合
7      }
8  }

```

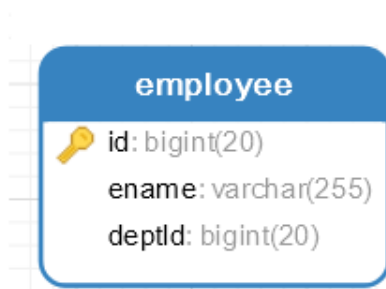
多对一关系

跟一对多关系一样，区分在所站的角度。

以部门与员工为例子：多个员工属于一个部门，此时主角是员工

表设计：

多方



id	ename	deptId
1	dafei	1
2	xiaofei	1

一方



id	dname
1	小卖部
2	大卖部

*

关系维护：在多方(员工)设置外键，引致部门主键

对象设计：

```

1  @Data
2  class Department {
3      private Long id;
4      private String dname;

```

```

5
6 }
7
8 @Data
9 class Employee {
10     private Long id;
11     private String ename;
12     //多对1关系，多方为主导，对象中关系维护交给多方
13     private Department dept;
14 }
15

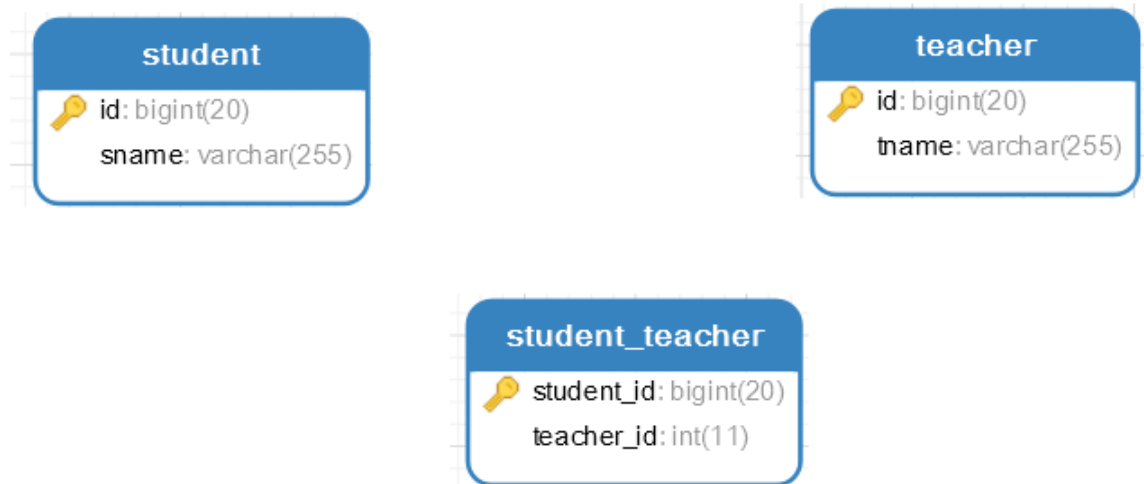
```

多对多关系

A表的一条数据可以映射B表多条数据，B表的一条数据也可以映射A表多条数据

以老师与学生为例子：一个老师可以教多个学生，一个学生可以给多个老师教

表设计：



id	sname
1	张三
2	张四

id	tname
1	dafei
2	xiaofei

student_id	teacher_id
1	1
1	2
2	1
2	2

*

关系维护：设计中间表，使用额外的表存放多对多关系映射id

对象设计：

```

1  @Data
2  class Student {
3      private Long id;
4      private String sname;
5      //多对多关系，核心对象为主导，对象中关系维护交它维护
6      private List<Teacher> ts = new ArrayList<>();
7  }
8  @Data
9  class Teacher {
10     private Long id;
11     private String tname;
12 }

```

上面面多对一，多对多操作跟之前的一对一，一堆多操作，大同小异，这里不展开讲了。

总之，MyBatis-Plus 对多表操作并不是很友好，只支持额外SQL查询方式，开发中如果复杂的SQL建议使用xml的方式。MyBatis-Plus无缝兼容Mybatis，因为：MyBatis-Plus只做增强，不做修改。

MyBatis-PlusAR模式

概念

百度百科

Active Record(活动记录)，是一种领域模型模式，特点是一个实体类对应关系型数据库中的一张表，而实体类实例对应表中的数据。ActiveRecord 严格遵循标准的 ORM 模型：表映射到实体对象，列映射到实体对象属性。

Active Record规定

一张表对应一个实体类，类的每一个对象实例对应于表中一行记录；

ActiveRecord 本身是一个实体类，同时实现持久化api，可自己实现CRUD逻辑

实现

MyBatis-Plus支持**Active Record**操作模式，这里以部门对象为了例子讲解。

步骤1：创建部门数据库表

```

1  CREATE TABLE `department` (
2      `id` bigint NOT NULL AUTO_INCREMENT,
3      `name` varchar(255) DEFAULT NULL,
4      `sn` varchar(255) DEFAULT NULL,
5      PRIMARY KEY (`id`)
6  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

步骤2：编写部门实体对象

```

1  @Setter
2  @Getter
3  @TableName("department")
4  public class Department extends Model<Department> {
5      @TableId(value = "id", type= IdType.AUTO)
6      private Long id;
7      private String name;
8      private String sn;
9  }

```

这里必须强调，实现AR模式需要2个前提之一：

实体对象必须继承核心类：Model，并且明确指定操作的实体类泛型

步骤3：编写部门实体对象对应的mapper接口

```

1  public interface DepartmentMapper extends BaseMapper<Department> {
2  }

```

实现AR模式需要2个前提之二：

必须提供实体对象对应的Mapper接口

步骤4：编写测试类

```

1  @SpringBootTest
2  public class ARTest {
3      @Test
4      public void testInsert(){
5          Department department = new Department();
6          department.setName("小卖部");
7          department.setSn("xmb");
8          department.insert();
9      }
10     @Test
11     public void testDelete(){
12         Department department = new Department();
13         department.deleteById(1L);
14     }
15     @Test
16     public void testUpdate(){
17         Department department = new Department();
18         department.setName("小卖部2");
19         department.setSn("xmb2");
20         department.setId(1L);
21         department.updateById();
22     }
23     @Test
24     public void testGet(){
25         Department department = new Department();
26         System.out.println(department.selectById(1L));
27     }
28     @Test
29     public void testList(){
30         Department department = new Department();

```

```

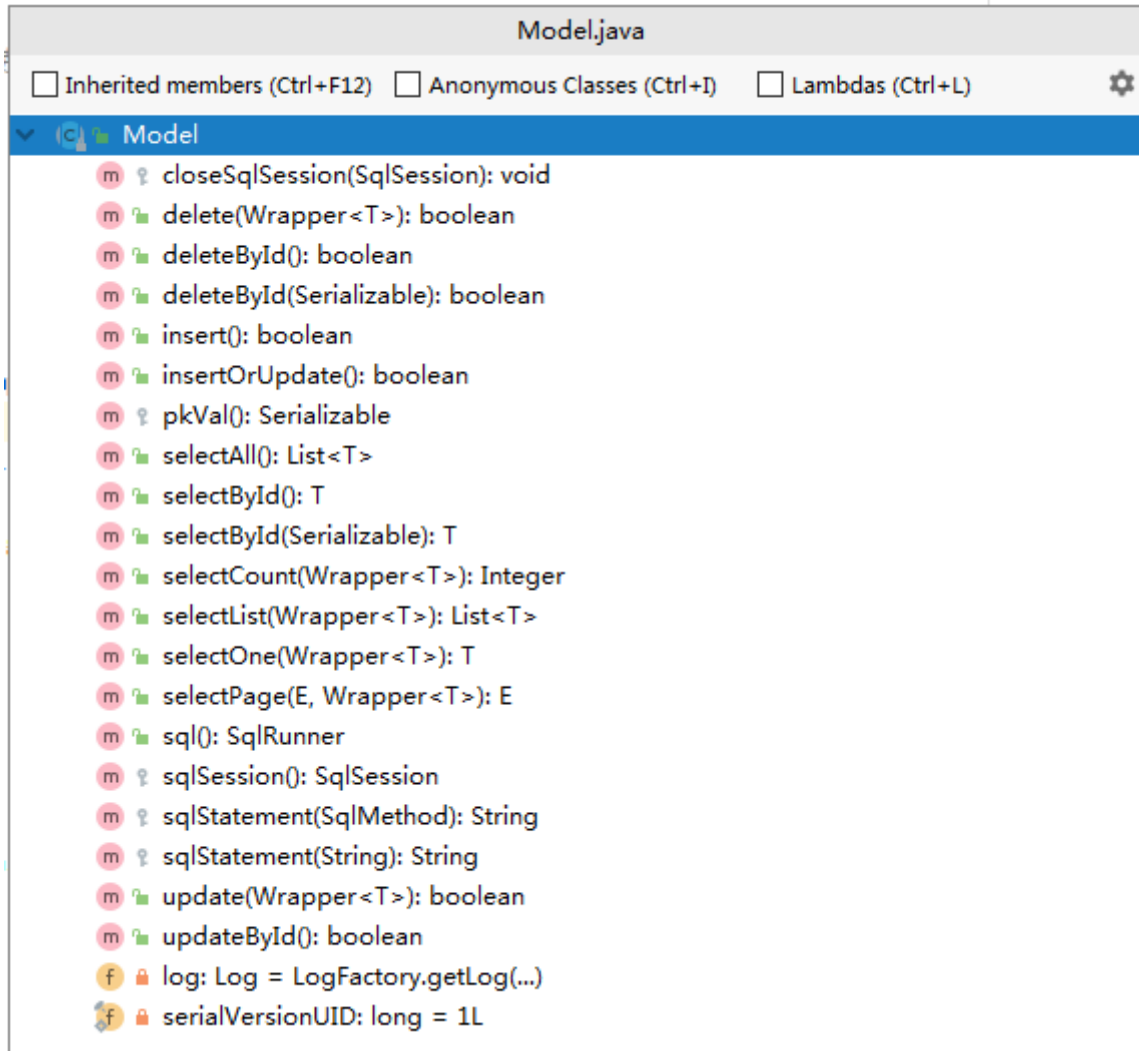
31     department.selectAll();
32     }
33 }

```

常用API

打开Model类，查看源码，发现AR执行以下CRUD方法

添加：2个，删除：3个，更新：2个，查询：7个



操作上跟BaseMapper接口的api一样，这里就不展开讲了。

原理

debug 下面代码

```

1  @Test
2  public void testGet(){
3      Department department = new Department();
4      System.out.println(department.selectById(1L));
5  }

```

点进去selectById方法

```

1  /**
2  * 根据 ID 查询

```

```

3  *
4  * @param id 主键ID
5  */
6  public T selectById(Serializable id) {
7      //获取操作sql的SqlSession对象
8      SqlSession sqlSession = sqlSession();
9      try {
10         //使用sqlSession执行对应sql，此处的sqlSession是Mybatis的sqlSession
11         //只需要拼接sql即可执行sql语句，这里的sqlStatement就是拼接sql用的内部方法
12         return sqlSession.selectOne(sqlStatement(SqlMethod.SELECT_BY_ID),
13             id);
14     } finally {
15         closeSqlSession(sqlSession);
16     }
17 }

```

```

1  /**
2  * 获取SqlStatement
3  *
4  * @param sqlMethod sqlMethod
5  */
6  protected String sqlStatement(SqlMethod sqlMethod) {
7      return sqlStatement(sqlMethod.getMethod());
8  }
9
10 /**
11 * 获取SqlStatement
12 *
13 * @param sqlMethod sqlMethod
14 */
15 protected String sqlStatement(String sqlMethod) {
16     //无法确定对应的mapper，只能用注入时候绑定的了。
17     return SqlHelper.table(getClass()).getSqlStatement(sqlMethod);
18 }

```

结合上面的源码，AR模式其实很简单，理由调用的方法取拼接对应的sql，然后通过sql直接调用的Mybatis中sqlSession执行即可。

MyBatis-Plus逻辑删除

概念

实际开发中，数据删除一般有2种选择：

1: 物理删除

物理删除，也称为硬删除，指的是数据直接从数据库中移除，对应的SQL语句：DELETE FROM 表 where 条件，这种删除成功后，数据就无法再恢复啦。

2: 逻辑删除

逻辑删除，也称为软删除，数据并没有真正删除，而是通过设置数据状态是否可显示，后续查询进行状态过滤，从而隐藏数据显示以达到删除对应的效果。比如：设置is_delete 数据状态，0表示正常，1表示删除。后续的查询sql 加上 where is_delete = 0 就可以过滤删除的数据。

一般开发选用的是逻辑删除，原因有2方面，一是项目数据非常重要不能随意删除，一是项目运行产生数据一般不会是独立，它可能会通过外键形式与其他数据产生关联，如果贸然删除该数据会引起系统不稳定，甚至异常。

举个例子：比如一个用户注册之后，进行发帖，留言，点评，收藏点赞等一些操作，会在系统相关表留下该用户相关信息，如果用户销户，使用物理删除方式删除该用户数据，那么之前与用户产生关联数据表都出现脏数据啦，这很可能导致系统bug。

局部逻辑删除

MyBatis-Plus也支持逻辑删除，分别是：局部逻辑删除，全局逻辑删除

具体实现步骤如下：

步骤1：在employee表添加一个del列，类型是int

步骤2：修改Employee实体类

```
1 public class Employee {
2     //省略其他字段
3
4     @TableLogic(value = "1", delval = "0")
5     private int del;
6 }
```

@TableLogic

作用：逻辑删除注解，一般开发不建议硬删除数据(从数据库删除)，建议使用软删除(数据不删除仅仅改数据状态，列表时做数据过滤)

核心属性：value, delval

value：表示未删除的数据状态

delval：表示删除之后的数据状态

步骤3：测试

```
1 @Test
2 public void testDelete(){
3     employeeMapper.deleteById(1L);
4 }
```

原先删除语法被转行成更新语法：

```
1 UPDATE employee SET del=0 WHERE id=? AND del=1
```

普通列表查询


```

1  @Test
2  public void testselectById(){
3      Employee employee = employeeMapper.selectById(1L);
4  }

```

执行SQL

```

1  SELECT id,name,password,email,age,admin,dept_id FROM employee WHERE id=? and
    del = 1

```

上面操作属于局部逻辑删除，针对是Employee对象，如果其他表也需要逻辑删除怎办？可以全局逻辑删除

全局逻辑删除

步骤1：在employee表添加一个del列，类型是int

步骤2：修改Employee实体类

```

1  public class Employee {
2      //省略其他字段
3
4      //@TableLogic(value = "1", delval = "0")
5      //此处不需要配置注解，在全局配置文件直接指定
6      private int del;
7  }

```

步骤3：配置主配置文件：application.yml

```

1  mybatis-plus:
2      global-config:
3          db-config:
4              logic-delete-field: del # 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略
              不配置步骤2)
5              logic-delete-value: 1 # 逻辑已删除值(默认为 1)
6              logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
7

```

如果是application.properties文件

```

1  # 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略不配置步骤2)
2  mybatis-plus.global-config.db-config.logic-delete-field=del
3  # 逻辑已删除值(默认为 1)
4  mybatis-plus.global-config.db-config.logic-delete-value=0
5  # 逻辑未删除值(默认为 0)
6  mybatis-plus.global-config.db-config.logic-not-delete-value=1

```

步骤4：测试

跟之前操作一样。

全局逻辑删除需要注意，必须在所有表上统一加上del字段，要不无法实现。

MyBatis-Plus乐观锁

概念

多事务环境下如何保证数据库操作安全呢？常用的一种解决方案就是对操作数据表进行加锁处理。根据实现思路不同分：悲观锁与乐观锁2种。

悲观锁：悲观的认为多事务操作同一数据是及其不安全的，所以A事务在操作数据时，其他任何事务不允许对该数据进行修改，只能等待A事务操作结束后才可以执行。

乐观锁：乐观的认为A事务在操作数据时，期间不会有其他事务进行干扰，能顺利完成事务操作。

实现

悲观锁

悲观锁解决方案非常简单，直接在操作sql中加上**for update** 语句即可

```
1 | select * from 表 where 条件列=值 for update
```

使用for update 操作，可以认为是给每次操作都加上表级别悲观锁，在事务没结束前，其他事务必须等待。

事务1

步骤1：启动事务

步骤2：使用悲观锁方式查询部门表(此时锁表)

步骤3：提交事务

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from department for update;
+-----+-----+
| id | name | sn |
+-----+-----+
| 2 | 开发部 | kf |
+-----+-----+
1 row in set (0.04 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> |
```

事务2

在事务执行到步骤2时，执行下面逻辑，发现事务停滞，等待事务1commit之后才继续往下执行。

```
mysql> update department set sn = "11" ;
Query OK, 1 row affected (6.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> |
```

注意：并发环境下都不建议使用悲观锁，因为悲观锁容易锁表，导致事务等待，性能低下。

乐观锁

乐观锁操作跟悲观锁区别在于不对操作的数据表进行加锁，而是使用version字段去规避。

以部门表为例子

步骤1：给部门表添加version列,默认值为0

对象	department @mybatis-plus ...		
开始事务	文本	筛选	排序
id	name	sn	version
1	开发部	dev	0
2	测试部	test	0

步骤2：实现乐观锁操作

需求：修改id=1数据sn字段改为：kfb

1>先查询

```
1 | select id, name, sn, version from department where id = 1;
```

得到该数据列version = 0

2>再修改，注意version版本更新

```
1 | update department set sn = "kfb", version = version + 1 where id = 1 and  
   | version = 0
```

这里有2个注意要点：

- 1：更新 sn 字段的同时，version 列数据也跟随变动
- 2：更新sql 除了基本的id=1条件外，多了version = 0 条件。

为什么要这么设计？原因：乐观锁操作没有加锁，任意事务都可以同步操作，多事务同时操作，总有一个先成功，一成功则修改了version字段值，其他事务version 条件自然就不上啦，update失败。

id	name	sn	version
1	开发部	dev	0
2	测试部	test	0

事务1：

需求：修改id=1数据sn列为：kfb1

1>先查询

select id, name, sn, version from department where id = 1
得到version = 0

2>更新

update department set sn = 'kfb1', version = version + 1
where id = 1 and version = 0

事务2：

需求：修改id=1数据sn列为：kfb2

1>先查询

select id, name, sn, version from department where id = 1
得到version = 0

2>更新

update department set sn = 'kfb2', version = version + 1
where id = 1 and version = 0

此时，事务1，事务2都执行更新步骤，因为有先后修改的说法。假设事务1的更新语句先执行的，department表的数据发生改动：id = 1 那条数据发生变化：version = 2，事务2再执行update时，where version = 0 就无法找到能修改的数据了，那修改失败。

*

步骤3：判断更新是否成功

乐观锁判断是否更新成功，就看执行update语句之后，返回的影响数据行数，如果行数大于0表示修改成功，如果行数等于0表示修改失败，放弃这次修改，一切重来。直到修改成功为止。

MyBatis-Plus版乐观锁

MyBatis-Plus 使用@Version注解实现乐观锁

还是以部门表为例

步骤1：在部门表添加version字段

对象	department @mybatis-plus ...		
开始事务	文本 筛选 排序		
id	name	sn	version
1	开发部	dev	0
2	测试部	test	0

步骤2：部门实体对象添加version字段

```

1 public class Department {
2     //省略其他字段
3     @Version
4     private int version;
5
6 }
```

步骤3：配置类中配置支持拦截器乐观锁

```

1 @Bean
2 public MybatisPlusInterceptor optimisticLockerInterceptor() {
3     MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
4     interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
5     return interceptor;
6 }
```

步骤4: 正常执行更新方法

```
1  @Test
2  public void testUpdate(){
3      //先查询
4      Department dept = departmentMapper.selectById(1L);
5      //再替换
6      dept.setId(1L);
7      dept.setName("小卖部");
8      dept.setSn("sell");
9      //最后更新
10     departmentMapper.updateById(dept);
11 }
```

执行后SQL

```
1  UPDATE department SET name=?, sn=?, version=? WHERE id=? AND version=?
```

MyBatis-Plus会自动讲乐观锁逻辑加载到sql中

使用Mybatis-Plus注意:

乐观锁支持的数据类型只有:int,Integer,long,Long,Date,Timestamp,LocalDateTime

仅支持 **updateById(id)** 与 **update(entity, wrapper)** 方法

另外, 每次操作前都是**先查询, 替换, 再更新**, 否则乐观锁无效

MyBatis-Plus通用枚举

平常开发中, 某些实体对象涉及到状态的变化, 一般使用方式有以下2种:

以员工实体举例子

1>使用常量的方式

```
1  public class Employee {
2      public static final int STATUS_NORMAL = 0; //在职
3      public static final int STATUS_LEAVE = 1; //离职
4      //员工状态
5      private int status;
6      //省略其他属性
7  }
```

2>使用枚举方式

```

1 //员工状态枚举
2 @Getter
3 public enum EmployeeStatus {
4     NORMAL(0, "在职"), LEAVE(1, "离职");
5     private int status;
6     private String label;
7     private EmployeeStatus(int status, String label){
8         this.status = status;
9         this.label = label;
10    }
11 }

```

```

1 public class Employee {
2     //员工状态
3     private EmployeeStatus status = EmployeeStatus.NORMAL;
4     //省略其他属性
5 }

```

使用常理操作方式与枚举操作方式其实大同小异，枚举赋予状态更多的操作空间，比如对应的方法，对应的属性，灵活性更高。

MyBatis-Plus 也支持枚举字段操作

还是以员工操作为例子

步骤1：修改员工表，添加status列，类型为int

步骤2：创建EmployeeStatus 枚举类与添加status字段

方案1

```

1 @Getter
2 public enum EmployeeStatus implements IEnum<Integer> {
3     NORMAL(0, "在职"), LEAVE(1, "离职");
4     private int status;
5     private String label;
6     private EmployeeStatus(int status, String label){
7         this.status = status;
8         this.label = label;
9     }
10    @Override
11    public Integer getValue() {
12        //在表status列中存什么值
13        return this.status;
14    }
15 }

```

方案2

```

1  @Getter
2  public enum EmployeeStatus{
3      NORMAL(0, "在职"), LEAVE(1, "离职");
4
5      @EnumValue
6      private int status;
7      private String label;
8      private EmployeeStatus(int status, String label){
9          this.status = status;
10         this.label = label;
11     }
12 }

```

使用注解方式替换接口的实现。

步骤3: 在Employee对象添加status字段

```

1  public class Employee {
2      //员工状态
3      private EmployeeStatus status = EmployeeStatus.NORMAL;
4      //省略其他属性
5  }

```

步骤4: 在application.properties 文件配置操作类

明确指定枚举类所有包路径

```

1  mybatis-plus.type-enums-package=com.langfeiyes.mp.enums

```

步骤5: 测试

```

1  @Test
2  public void testSave(){
3      Employee employee = new Employee();
4      employee.setAdmin(1);
5      employee.setAge(18);
6      employee.setDeptId(1L);
7      employee.setEmail("zhangsan@163.com");
8      employee.setName("zhangsan");
9      employee.setPassword("111");
10     //执行状态
11     employee.setStatus(EmployeeStatus.LEAVE);
12     employeeMapper.insert(employee);
13 }

```

执行SQL

```

1  ==> Preparing: INSERT INTO employee ( id, status, name, password, email,
2  age, admin, dept_id, del, version ) VALUES ( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )
3  ==> Parameters: 1(Long), 1(Integer), zhangsan(String), 111(String),
4  zhangsan@163.com(String), 18(Integer), 1(Integer), 1(Long), 0(Integer),
5  0(Integer)

```

MyBatis-Plus类型处理器

概念

类型处理器，用于 JavaType 与 JdbcType 之间的转换，简单的理解就是对象属性与列间的映射处理。

举个例子：员工对象在Mapper.xml中对象属性与列间映射

```
1 <resultMap id="BaseResultMap" type="com.langfeiyes.mp.Employee" >
2   <id column="id" property="id" jdbcType="BIGINT" />
3   <result column="name" property="name" jdbcType="VARCHAR"
typeHandler="xxxTypeHanler"/>
4   <result column="phone" property="phone" jdbcType="VARCHAR" />
5   <result column="email" property="email" jdbcType="VARCHAR" />
6   <result column="status" property="status" jdbcType="CHAR" />
7   <result column="del" property="del" jdbcType="CHAR" />
8 </resultMap>
```

在没有显示指定具体类型处理器，那就是使用默认处理器，默认处理器可以处理简单类型映射：比如8大基本类型，时间，字符串等。但是，如果是相对复杂类型映射呢？

要实现复杂的类型映射，那么就需要使用自定义类型处理器，自定义映射规则。

实现

需求：员工表中有手机号码列，格式如下

contact
13700000000,13700000001,13700000002

转成员工对象contact集合

```
1 public class Employee {
2     private List<String> contact = new ArrayList<>();
3 }
```

很明显，varchar类型的contact 列是无法直接转换list集合，此时需要使用自定义类型处理器啦。

步骤1：自定义类型转换处理器

```
1 /**
2  * 将字符串转换成字符串数组
3  * eg: 13700000000,13700000001,13700000002,    ---> List<String>
4  * 将字符串数组元素拼接字符串
5  * eg: List<String>    ---> 13700000000,13700000001,13700000002,
6  */
7 public class ListTypeHandler extends BaseTypeHandler<List<String>> {
8
9     //将list集合字符串元素使用，方式拼接，然后存到列中
10    @Override
11    public void setNonNullParameter(PreparedStatement preparedStatement, int
i, List<String> list, JdbcType jdbcType) throws SQLException {
```



```

12         if(list != null && list.size() > 0){
13             StringBuilder sb = new StringBuilder(100);
14             for (String s : list) {
15                 sb.append(s).append(",");
16             }
17             preparedStatement.setString(i, sb.toString());
18         }
19     }
20
21     //从数据库查询出列值，然后拆分，转换成字符串集合
22     @Override
23     public List<String> getNullableResult(ResultSet resultSet, String
column) throws SQLException {
24         String value = resultSet.getString(column);
25         return this.parseToList(value);
26     }
27     //从数据库查询出列值，然后拆分，转换成字符串集合
28     @Override
29     public List<String> getNullableResult(ResultSet resultSet, int i) throws
SQLException {
30         String value = resultSet.getString(i);
31         return this.parseToList(value);
32     }
33     //从数据库查询出列值，然后拆分，转换成字符串集合
34     @Override
35     public List<String> getNullableResult(CallableStatement
callableStatement, int i) throws SQLException {
36         String value = callableStatement.getString(i);
37         return this.parseToList(value);
38     }
39
40     //字符串截取转换成list集合
41     private List<String> parseToList(String value){
42         List<String> list = new ArrayList<>();
43         if(StringUtils.hasText(value)){
44             String[] sArr = value.split(",");
45             if(sArr.length > 0){
46                 for (String s : sArr) {
47                     if(StringUtils.hasText(s)){
48                         list.add(s.trim());
49                     }
50                 }
51             }
52         }
53         return list;
54     }
55 }

```

步骤2: 给员工表添加contact列，类型为varchar

步骤3: 给Employee对象添加contact属性，类型是List

```

1 public class Employee {
2     //指定类型转换处理器
3     @TableField(typeHandler = ListTypeHandler.class)
4     private List<String> contact = new ArrayList<>();
5 }

```

步骤4: 在application.properties配置文件中配置处理器所在包

```

1 mybatis-plus.type-handlers-package=com.langfeiyes.mp.handler

```

步骤5: 添加与查询测试

```

1 @Test
2 public void testSave(){
3     Employee employee = new Employee();
4     employee.setAdmin(1);
5     employee.setAge(18);
6     employee.setDeptId(1L);
7     employee.setEmail("zhangsan@163.com");
8     employee.setName("zhangsan");
9     employee.setPassword("111");
10    //执行状态
11    employee.setStatus(EmployeeStatus.LEAVE);
12    //联系电话
13    employee.setContact(Arrays.asList("13700000000",
14    "13700000001", "13700000002"));
15    employeeMapper.insert(employee);
16 }

```

```

contact *
13700000000,13700000001,13700000002,

```

```

1 @Test
2 public void testGet(){
3     Employee employee = employeeMapper.selectById(1L);
4     System.out.println(employee.getContact());
5 }

```

```
[13700000000, 13700000001, 13700000002] *
```

MyBatis-Plus通用Service接口

概念

Java项目一般使用三层结构开发:

表现层: 接收请求, 调用业务方法处理请求, 响应请求

业务层: 也叫服务层, 实现业务逻辑, 调用持久层实现数据组合操作

持久层: 完成数据的CRUD操作

前面讲的Mapper接口操作属于持久层，如果项目加入服务层，那代码该如何构建呢？

传统的业务层

使用MyBatis-Plus之前，传统业务层构建方式：以员工操作为例子

步骤1：构建员工业务层服务接口

```
1 public interface IEmployeeService {
2     void save(Employee employee);
3     void update(Employee employee);
4     void delete(Long id);
5     Employee get(Long id);
6     List<Employee> list();
7 }
```

步骤2：实现员工业务层接口

```
1 @Service
2 public class EmployeeServiceImpl implements IEmployeeService {
3     @Autowired
4     private EmployeeMapper mapper;
5
6     @Override
7     public void save(Employee employee) {
8         mapper.insert(employee);
9     }
10
11     @Override
12     public void update(Employee employee) {
13         mapper.updateById(employee); //必须全量更新
14     }
15
16     @Override
17     public void delete(Long id) {
18         mapper.deleteById(id);
19     }
20
21     @Override
22     public Employee get(Long id) {
23         return mapper.selectById(id);
24     }
25     @Override
26     public List<Employee> list() {
27         return mapper.selectList(null);
28     }
29 }
30
```

步骤3：实现服务方法测试

传统业务层服务方法需要自己调用Mapper接口方法去实现，相对麻烦。再看MyBatis-Plus提供简化操作

MyBatis-Plus业务层

步骤1：构建员工业务层服务接口

```
1  /**
2  *1>自定义一个业务服务接口：IEmployeeService,继承父接口：IService
3  *2>明确指定父接口泛型：当前接口操作实体对象：Employee
4  */
5  public interface IEmployeeService extends IService<Employee> {
6  }
```

步骤2：实现员工业务层接口

```
1  /**
2  *1>定义服务接口实现类，实现IEmployeeService接口
3  *2>继承通用的父接口实现类：ServiceImpl
4  *3>明确指定通用的父接口实现类2个泛型：
5  *    1: 当前服务类操作的实体对象对应的Mapper接口：EmployeeMapper
6  *    2: 当前服务类操作的实体对象：Employee
7  */
8  @Service
9  public class EmployeeServiceImpl extends ServiceImpl<EmployeeMapper,
10 Employee> implements IEmployeeService {
11 }
12
```

步骤3：实现服务方法测试

```
1  @SpringBootTest
2  public class ServiceTest {
3      @Autowired
4      private IEmployeeService employeeService;
5      @Test
6      public void testSave(){
7          Employee employee = new Employee();
8          employee.setAdmin(1);
9          employee.setAge(18);
10         employee.setDeptId(1L);
11         employee.setEmail("zhangsan@163.com");
12         employee.setName("zhangsan");
13         employee.setPassword("111");
14         employeeService.save(employee);
15     }
16
17     @Test
18     public void testUpdate(){
19         Employee employee = new Employee();
20         employee.setId(1327139013313564673L);
21         employee.setAdmin(1);
22         employee.setAge(18);
23         employee.setDeptId(1L);
24         employee.setEmail("zhangsan@163.com");
25         employee.setName("zhangxiaosan");
```

```

26         employee.setPassword("111");
27         employeeService.updateById(employee);
28     }
29     @Test
30     public void testDelete(){
31         employeeService.removeById(11L);
32     }
33     @Test
34     public void testGet(){
35         System.out.println(employeeService.getById(11L));
36     }
37
38     @Test
39     public void testList(){
40         List<Employee> employees = employeeService.list();
41         employees.forEach(System.err::println);
42     }
43
44 }

```

常用服务层api

IService.java

☒ Inherited members (Ctrl+F12)
 ☐ Anonymous Classes (Ctrl+I)
 ☐ Lambdas (Ctrl+L)
 ⚙️

▼
I
Service

- m f count(): int
- m f count(Wrapper<T>): int
- m f getBaseMapper(): BaseMapper<T>
- m f getById(Serializable): T
- m f getMap(Wrapper<T>): Map<String, Object>
- m f getObj(Wrapper<T>, Function<? super Object, V>): V
- m f getOne(Wrapper<T>): T
- m f getOne(Wrapper<T>, boolean): T
- m f lambdaQuery(): LambdaQueryChainWrapper<T>
- m f lambdaUpdate(): LambdaUpdateChainWrapper<T>
- m f list(): List<T>
- m f list(Wrapper<T>): List<T>
- m f listByIds(Collection<? extends Serializable>): List<T>
- m f listByMap(Map<String, Object>): List<T>
- m f listMaps(): List<Map<String, Object>>
- m f listMaps(Wrapper<T>): List<Map<String, Object>>
- m f listObjs(): List<Object>
- m f listObjs(Function<? super Object, V>): List<V>
- m f listObjs(Wrapper<T>): List<Object>
- m f listObjs(Wrapper<T>, Function<? super Object, V>): List<V>

*

```

m  📄 page(E): E
m  📄 page(E, Wrapper<T>): E
m  📄 pageMaps(E): E
m  📄 pageMaps(E, Wrapper<T>): E
m  📄 query(): QueryChainWrapper<T>
m  📄 remove(Wrapper<T>): boolean
m  📄 removeById(Serializable): boolean
m  📄 removeByIds(Collection<? extends Serializable>): boolean
m  📄 removeByMap(Map<String, Object>): boolean
m  📄 save(T): boolean
m  📄 saveBatch(Collection<T>): boolean
(m) 📄 saveBatch(Collection<T>, int): boolean
(m) 📄 saveOrUpdate(T): boolean
m  📄 saveOrUpdate(T, Wrapper<T>): boolean
m  📄 saveOrUpdateBatch(Collection<T>): boolean
(m) 📄 saveOrUpdateBatch(Collection<T>, int): boolean
m  📄 update(): UpdateChainWrapper<T>
m  📄 update(T, Wrapper<T>): boolean
m  📄 update(Wrapper<T>): boolean

```

*

上面就是IService接口提供的实现方法，几乎涵盖了数据库常规操作。

方法解析

思考一个问题，IService接口是怎么实现的？

以IService接口中的getById()方法为例子

```

1  /**
2   * 根据 ID 查询
3   *
4   * @param id 主键ID
5   */
6  default T getById(Serializable id) {
7      return getBaseMapper().selectById(id);
8  }

```

源码上，getById是一个接口默认方法，默认实现是调用getBaseMapper()对象去执行selectById方法

```

1  /**
2   * 获取对应 entity 的 BaseMapper
3   *
4   * @return BaseMapper
5   */
6  BaseMapper<T> getBaseMapper();

```

getBaseMapper方法也是IService接口定义的方法，继续追踪其接口实现：

ServiceImpl类的getBaseMapper方法

```

1 public class ServiceImpl<M extends BaseMapper<T>, T> implements IService<T>
2 {
3     @Autowired
4     protected M baseMapper;
5
6     @Override
7     public M getBaseMapper() {
8         return baseMapper;
9     }
10 }

```

从上面代码可以看到baseMapper是使用@Autowired 方式从spring容器中注入的，具体类型是泛型对象M, 而在定义EmployeeServiceImpl类时，具体代码：

```

1 @Service
2 public class EmployeeServiceImpl extends ServiceImpl<EmployeeMapper,
3     Employee> implements IEmployeeService {
4 }

```

可以确认，以EmployeeServiceImpl为例子，那么ServiceImpl中M的泛型就是EmployeeMapper类型，那么ServiceImpl可以等价

```

1 public class ServiceImpl<EmployeeMapper extends BaseMapper<T>, T> implements
2     IService<T> {
3
4     @Autowired
5     protected EmployeeMapper baseMapper;
6
7     @Override
8     public EmployeeMapper getBaseMapper() {
9         return baseMapper;
10    }
11 }

```

最后IService方法中getById

```

1 /**
2  * 根据 ID 查询
3  *
4  * @param id 主键ID
5  */
6 default T getById(Serializable id) {
7     return getBaseMapper().selectById(id);
8 }

```

等价于：

```

1  /**
2   * 根据 ID 查询
3   *
4   * @param id 主键ID
5   */
6  default T getById(Serializable id) {
7      //其中的baseMapper就是employeeMapper
8      return baseMapper.selectById(id);
9  }

```

到这，可以确定IService接口中方法底层都是使用Mapper接口方法实现，那么IService 接口方法操作就可以同理可得啦。

MyBatis-Plus代码生成器

概念

代码生成器，顾名思义就是通过程序生成想要的代码。其原理非常简单，可以简单理解为：模板 + 数据 = 输出。

比如：

模板--Mapper.ftl

```

1  package ${basepackage}.mapper;
2
3  import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4  import ${basepackage}.${domain};
5
6  public interface ${domain}Mapper extends BaseMapper<${domain}> {
7  }

```

数据

```

1  {
2      basepackage:"com.langfeiyes.mp",
3      domain:"Department"
4  }

```

输入：数据 + 模板

将模板中`${占位符}` 替换成数据，并将渲染好的结果输出到DepartmentMapper.java文件中

```

1  package com.langfeiyes.mp.mapper;
2
3  import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4  import com.langfeiyes.mp.domain.Department;
5
6  public interface DepartmentMapper extends BaseMapper<Department> {
7  }

```


那么此时的DepartmentMapper.java类就编写好了，后续如果将Department替换成Employee，那么EmployeeMapper.xml也就能快速编写成功。

```
1 package com.langfeiyes.mp.mapper;
2
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4 import com.langfeiyes.mp.domainEmployee;
5
6 public interface EmployeeMapper extends BaseMapper<Employee> {
7 }
```

MyBatis-Plus 也提供一套代码生成器-AutoGenerator，通过 AutoGenerator 可以快速生成 Entity、Mapper、Mapper XML、Service、Controller 等各个模块的代码，极大的提升了开发效率。

旧版

旧版代码生成器指的是mybatis-plus-generator 3.5.1 以下版本

以User用户为例子

步骤1：创建User数据表

```
1 CREATE TABLE `mybatis-plus`.`user` (
2   `id` bigint(0) NOT NULL AUTO_INCREMENT COMMENT '用户id',
3   `username` varchar(255) NULL COMMENT '用户名',
4   `password` varchar(255) NULL COMMENT '密码',
5   PRIMARY KEY (`id`)
6 );
```

步骤2：导入相关依赖

```
1 <!--代码生成器-->
2 <dependency>
3   <groupId>com.baomidou</groupId>
4   <artifactId>mybatis-plus-generator</artifactId>
5   <version>3.5.0</version>
6 </dependency>
7
8 <!--模板依赖-freemarker-->
9 <dependency>
10   <groupId>org.freemarker</groupId>
11   <artifactId>freemarker</artifactId>
12   <version>2.3.31</version>
13 </dependency>
14
15 <!--模板依赖-beetl-->
16 <dependency>
17   <groupId>com.iibeetl</groupId>
18   <artifactId>beetl</artifactId>
19   <version>3.9.3.RELEASE</version>
20 </dependency>
21
22 <!--模板依赖-velocity-->
```

```

23 <dependency>
24     <groupId>org.apache.velocity</groupId>
25     <artifactId>velocity-engine-core</artifactId>
26     <version>2.3</version>
27 </dependency>

```

注意上面3个模板只需要导入一个即可，默认是Velocity

步骤3: 编写代码生成器

```

1 package com.langfeiyes.mp.generator;
2
3
4 import com.baomidou.mybatisplus.generator.AutoGenerator;
5 import com.baomidou.mybatisplus.generator.config.*;
6 import com.baomidou.mybatisplus.generator.engine.BeetlTemplateEngine;
7 import com.baomidou.mybatisplus.generator.engine.FreemarkerTemplateEngine;
8 import com.baomidou.mybatisplus.generator.engine.VelocityTemplateEngine;
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.stereotype.Component;
11
12 import javax.sql.DataSource;
13
14
15 /**
16  * 代码生成器执行逻辑:
17  * 1>配置数据源-直接数据库加载表，通过解析表自动创建 domain, mapper, service
18  * 2>配置全局参数-核心: 创建文件的文件摆放的位置(路径)
19  * 3>配置包参数-明确指定包路径，模块名，各种类包名称
20  * 4>配置模板参数-明确指出模板所在位置，模板路径
21  * 5>配置策略参数-用过配置可以定制哪些表需要进行代码生成，哪些不需要，表一些通用前缀配置，列前缀配置等。
22  * 6>配置注入参数-配置与模板交互的自定义参数，作为拓展
23  */
24 @Component
25 public class MyCodeGenerator {
26
27     @Autowired
28     private DataSource dataSource;    //数据源
29
30
31     //全局配置
32     private GlobalConfig getGlobalConfig(){
33         GlobalConfig config = new GlobalConfig.Builder()
34             //如果之前已经执行过代码生成，后续再执行是覆盖之前代码
35             //fileOverride()
36             //执行代码生成之后，要不开文件，true是打开，false: 不开
37             .openDir(true)
38             //执行代码生成之后，代码文件输出到哪个位置
39             .outputDir(System.getProperty("user.dir") +
40                 "/src/main/java")
41             //代码上面注释: 作者
42             .author("dafei")
43             //是否启动swagger2配置模式，该方法表示启动

```

```

43         //enableSwagger()
44         .build();
45     return config;
46 }
47
48
49 //包配置
50 private PackageConfig getPackageConfig(){
51     PackageConfig config = new PackageConfig.Builder()
52         //指定包路径(倒写域名)
53         .parent("com.langfeiyes")
54         //模板名(项目名)
55         .moduleName("mp")
56         //实体类所在包名
57         .entity("domain")
58         //服务层接口所在包名
59         .service("service")
60         //服务层接口实现类所在包名
61         .serviceImpl("service.impl")
62         //mapper接口所在包名
63         .mapper("mapper")
64         //mapper.xml文件所在包名
65         .xml("mapper")
66         //表现层类所在包名
67         .controller("web.controller")
68         .build();
69     return config;
70 }
71
72
73 //模板配置
74 private TemplateConfig getTemplateConfig(){
75     TemplateConfig config = new TemplateConfig.Builder()
76         //指定实体模板路径
77         //.entity("不是自定义模板, 不建议修改, 使用即可")
78         //.service("不是自定义模板, 不建议修改, 使用即可")
79         //.mapper("不是自定义模板, 不建议修改, 使用即可")
80         //.mapperXml("不是自定义模板, 不建议修改, 使用即可")
81         //.controller("不是自定义模板, 不建议修改, 使用即可")
82         .build();
83
84     return config;
85 }
86
87
88 //策略配置-配置是表相关
89 private StrategyConfig getStrategyConfig(){
90     StrategyConfig config = new StrategyConfig.Builder()
91         //需要进行代码生成的表(不写, 默认是数据库所有表)
92         .addInclude("user")
93         //排除哪些不进行代码生成的表
94         //.addExclude()
95         //统一指定表的前缀
96         .addTablePrefix("t_")
97         .build();

```

```

98
99         return config;
100     }
101
102     //注入配置-
103     private InjectionConfig getInjectionConfig(){
104         InjectionConfig config = new InjectionConfig.Builder()
105             .build();
106         return config;
107     }
108
109     //执行代码构造
110     public void excute(){
111         //构造代码生成器实例对象
112         AutoGenerator autoGenerator = new AutoGenerator( new
DataSourceConfig.Builder(dataSource).build())
113             //全局配置
114             .global(getGlobalConfig())
115             //包配置
116             .packageInfo(getPackageConfig())
117             //模板配置
118             .template(getTemplateConfig())
119             //策略配置
120             .strategy(getStrategyConfig())
121             //注入配置-自定义模板使用
122             .injection(getInjectionConfig());
123
124         //autoGenerator.execute(); //默认使用velocity模板引擎
125         //autoGenerator.execute(new VelocityTemplateEngine());
126         autoGenerator.execute(new FreemarkerTemplateEngine());
127         //autoGenerator.execute(new BeetlTemplateEngine());
128     }
129
130 }
131

```

步骤4: 测试

```

1  @SpringBootTest
2  public class CodeBuilderTest {
3      @Autowired
4      MyCodeGenerator myCodeGenerator;
5      @Test
6      public void testCode() throws SQLException {
7          myCodeGenerator.excute();
8      }
9  }

```

注意: 如果是低版本的druid, 会包错误: SQLFeatureNotSupportedException

主要是druid数据源中dataSource.getConnection().getSchema() 没有实现, 直接丢异常, 换新版本即可

```

1 <dependency>
2   <groupId>com.alibaba</groupId>
3   <artifactId>druid-spring-boot-starter</artifactId>
4   <version>1.2.8</version>
5 </dependency>

```

新版

新版代码生成器指的是mybatis-plus-generator 3.5.1 以上版本，操作最大的变化是使用了lambda语法
以User用户为例子

前面操作操作步骤都一样

步骤1：创建user用户表

步骤2：导入相关依赖

```

1 <dependency>
2   <groupId>com.baomidou</groupId>
3   <artifactId>mybatis-plus-generator</artifactId>
4   <version>3.5.1</version>
5 </dependency>
6

```

步骤3：编写代码生成器代码

```

1 package com.langfeiyes.mp.generator;
2
3 import com.alibaba.druid.pool.DruidDataSource;
4 import com.baomidou.mybatisplus.generator.AutoGenerator;
5 import com.baomidou.mybatisplus.generator.FastAutoGenerator;
6 import com.baomidou.mybatisplus.generator.config.*;
7 import com.baomidou.mybatisplus.generator.engine.FreemarkerTemplateEngine;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.stereotype.Component;
10
11 import javax.sql.DataSource;
12
13 @Component
14 public class MyCodeGenerator {
15
16     @Autowired
17     private DataSource dataSource;
18
19     public void excute() {
20         FastAutoGenerator.create(new DataSourceConfig.Builder(dataSource))
21             .globalConfig(builder -> builder
22                 .outputDir(System.getProperty("user.dir") +
23                     "/src/main/java") //输出目录
24                 .author("dafei") //作者
25                 .enableSwagger() //是否开启swagger模式
26                 .fileOverride() //如果文件已经存在则覆盖
27             )
28             .packageConfig(builder -> builder

```

```

28         .parent("com.langfeiyes")           //父包名，根包
29         .moduleName("mp")                   //模块名
30         .entity("domain")                   //指定实体对象包名，
默认: entity
31         .service("service")                 //指定服务接口包名，
默认: service
32         .serviceImpl("service.impl")        //指定服务接口实现
类包名,默认: service.impl
33         .mapper("mapper")                  //指定映射接口包名，
默认: mapper
34         .xml("xml")                        //指定映射xml文件
包名,默认: mapper.xml
35         .controller("controller")          //指定controller
包名,默认: controller
36     )
37     /*.templateConfig(builder -> builder
38         //entity("entity")                 //指定实体对象模
板路径
39         //service("service", //指定服务接口模板路径
40         //    "serviceImpl") //指定服务接口模板路径
41         //mapper("mapper")                //指定映射接口模
板路径
42         //mapperXml("xml")                //指定映
射xml文件模板路径
43         //controller("controller")        //指定
controller模板路径
44     )*/
45     .strategyConfig(builder -> builder
46         //addFieldPrefix("字段前缀")
47         //addExclude("排除哪些表")
48         //addInclude("指定哪些表")
49         .addInclude("user")
50         //addTablePrefix("添加表前缀")
51     )
52     .templateEngine(new FreemarkerTemplateEngine())
53     .execute();
54 }
55
56 }

```

步骤4: 测试

```

1  @SpringBootTest
2  public class CodeBuilderTest {
3      @Autowired
4      MyCodeGenerator myCodeGenerator;
5      @Test
6      public void testCode() throws SQLException {
7          myCodeGenerator.excute();
8      }
9  }
10

```

MyBatis-Plus最佳实践【项目实战】

本章节目的是讲前面所学结合业务需求搭建出一个完整的项目，讲解过程中尽量让大伙明白项目开发种种细节。

这里注意，后续的课程就当大伙已经熟悉使用：springboot，springmvc，mybatis-plus相关框架啦

项目演示

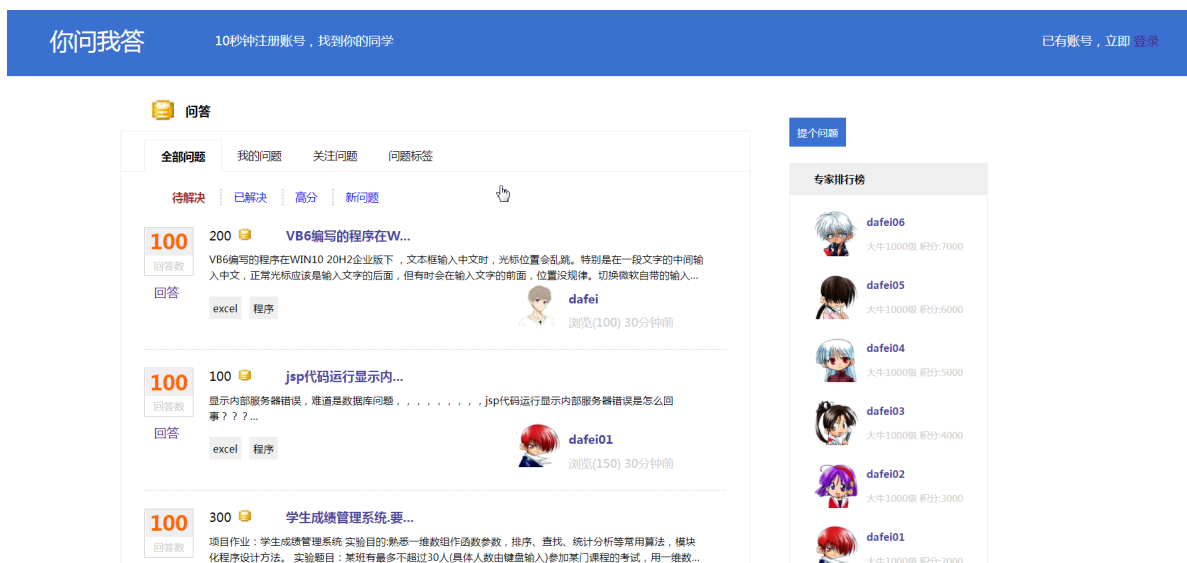
1>在mysql创建askme数据库

2>将askme.sql脚本 导入数据库中

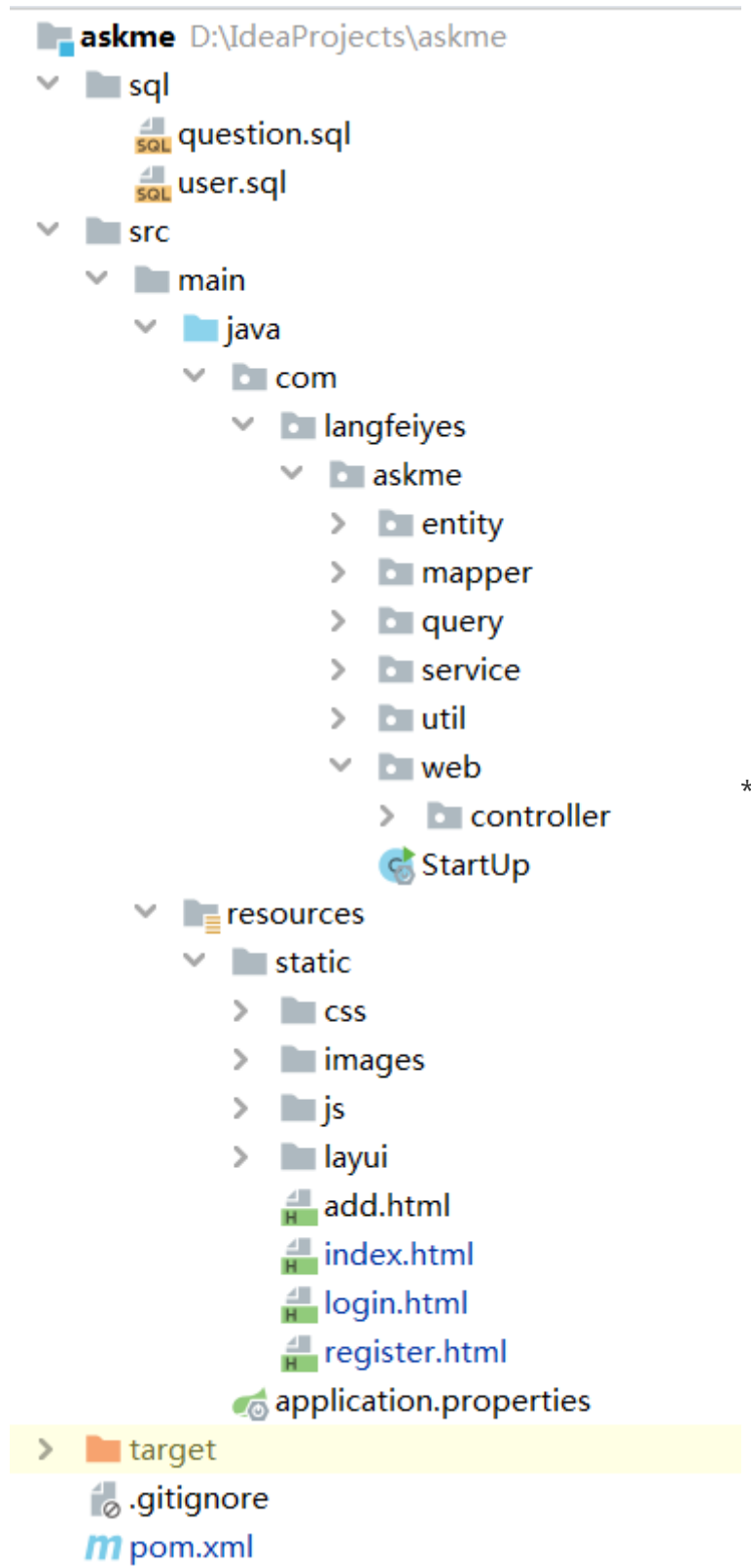
3>将项目import到idea中

4>右键执行StartUp类

5>启动项目，输入：<http://localhost:8080/> 进入首页



项目搭建



技术栈：springboot + mybatis-plus + layui

标准的三层结构

集成Mybatis-Plus






1>配置Mybatis-Plus相关依赖

2>配置代码生成器

3>构建代码

其中涉及到Restful风格接口可以查看：<https://edu.csdn.net/course/detail/36544>

用户表设计

专家排行榜	
	dafei06 大牛1000级 积分:7000
	dafei05 大牛1000级 积分:6000
	dafei04 大牛1000级 积分:5000
	dafei03 大牛1000级 积分:4000
	dafei02 大牛1000级 积分:3000

*

表设计套路:

1>所见所得，页面展示出来的数据，表中肯定有一一对应

2>隐藏字段，结合个人经历与需求，推测一些隐藏字段(状态，时间，序号)

3>确认与现有表或将要设计的表之间的关联关系。

结合排行，登录，注册页面，大体推断用户对象基本属性

id	username	password	avatar	level	score	status
1	dafei	1	/images/0.gif	1000	1000	0
2	dafei01	1	/images/1.gif	1000	2000	0
3	dafei02	1	/images/2.gif	1000	3000	0
4	dafei03	1	/images/3.gif	1000	4000	0
5	dafei04	1	/images/4.gif	1000	5000	0
6	dafei05	1	/images/5.gif	1000	6000	0
7	dafei06	1	/images/6.gif	1000	7000	0

*

用户数据模型准备

1>用户表创建

2>用户实体类

3>用户Mapper

4>UserService

5>UserController

用户注册

用 户 名 您的用户名和登录名

设 置 密 码 请输入密码

确 认 密 码 请再次输入密码

验 证 码 请输入验证码

74ce

☐ 阅读并同意 [《错题用户注册协议》](#) [已有账号,立即登录](#)

立 即 注 册

*

需求：注册验证码获取

1>UUID创建随机4位验证码

2>将验证码保存2份，一份保存在session，一份传回页面

3>用户注册时，将输入的验证码跟session验证进行比较，如果一样，执行后续注册逻辑

```
1 package com.langfeiyes.askme.web.controller;
2
3
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpSession;
11 import java.util.UUID;
12
13 @RestController
14 @RequestMapping("verifyCodes")
15 public class CodeController {
16     //登录验证码
17     @GetMapping("/login")
18     public String login(HttpSession session){
19         String loginCode = UUID.randomUUID().toString().replaceAll("-",
20         "").substring(0, 4);
21         session.setAttribute("login_code", loginCode);
22         return loginCode;
23     }
24     //注册验证码
25     @GetMapping("/regist")
26     public String regist(HttpSession session){
27         String registCode = UUID.randomUUID().toString().replaceAll("-",
28         "").substring(0, 4);
29         session.setAttribute("regist_code", registCode);
30         return registCode;
31     }
32 }
```

需求：用户注册接口设计

统一返回对象

```
1 package com.langfeiyes.askme.util;
2
3 import lombok.Getter;
4 import lombok.NoArgsConstructor;
5 import lombok.Setter;
6
7 @Getter
8 @Setter
9 @NoArgsConstructor
10 public class JsonResult {
11
12     private int code;
```

```

13     private String msg;
14     private Object data;
15
16
17     public JsonResult(int code, String msg, Object data) {
18         this.code = code;
19         this.msg = msg;
20         this.data = data;
21     }
22
23     public static JsonResult success(){
24         return new JsonResult(200,null, null);
25     }
26
27     public static JsonResult success(Object data){
28         return new JsonResult(200,null, data);
29     }
30     public static JsonResult error(String msg){
31         return new JsonResult(500,msg, null);
32     }
33 }

```

```

1  package com.langfeiyes.askme.web.controller;
2
3
4  import com.langfeiyes.askme.util.JsonResult;
5  import com.langfeiyes.askme.entity.User;
6  import com.langfeiyes.askme.service.IUserService;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RestController;
10
11  import javax.servlet.http.HttpSession;
12  import java.util.List;
13
14  @RestController
15  @RequestMapping("users")
16  public class UserController {
17
18      @Autowired
19      private IUserService userService;
20
21
22      /**
23       * restful风格接口（请求映射方法）设计考量点
24       * 1>请求路径---/users/regist
25       * 2>请求方法---post
26       * 3>请求参数--username password code
27       * 4>请求响应--json格式数据里面2个属性： code msg
28       */
29      @PostMapping("/regist")
30      public JsonResult regist(String username, String password, String code,
31                              HttpSession session){
32
33          //1: 校验验证码是否正确

```

```

33         String regist_code = session.getAttribute("regist_code").toString();
34         if(code == null || !code.equalsIgnoreCase(regist_code)){
35             return JsonResult.error("验证码错误~");
36         }
37         //2:执行用户注册逻辑
38         try {
39             userService.regist(username, password);
40         } catch (Exception e) {
41             e.printStackTrace();
42             return JsonResult.error(e.getMessage());
43         }
44
45         return JsonResult.success();
46     }
47 }
48

```

UserServiceImpl

```

1  @Override
2  public void regist(String username, String password) {
3      //检查用户名是否唯一
4      QueryWrapper<User> wrapper = new QueryWrapper<>();
5      wrapper.eq("username", username);
6      User one = super.getOne(wrapper);
7      if(one != null){
8          throw new RuntimeException("当前账号已经存在");
9      }
10     //完成注册： 封装用户对象并保存
11     User user = new User();
12     user.setUsername(username);
13     user.setPassword(password); //真实开发需要安装公司要求进行加密，此处简单写，假
    装已经加密了
14     user.setAvatar("/images/0.gif"); //默认头像
15     user.setLevel(1);
16     user.setScore(0);
17     user.setStatus(User.STATUS_NORMAL);
18     super.save(user);
19 }

```

涉及到知识点

1>restful 接口请求设计

2>统一响应对象设计

用户登录

用户名

您的用户名和登录名

密 码

请输入密码

验证码

请输入验证码

7126

☐ 阅读并同意 《你问我答用户注册协议》 [没有账号,立即注册](#)

立即登录

需求：获取登录验证码

跟之前用户注册验证操作一样

需求：用户登录

1>页面输入账号密码与验证码

2>判断传入验证码与session验证码是否一致，一致执行后续逻辑，不一致提示

3>以账号与密码作为条件查询用户对象，如果用户对象为null，提示账号密码错误，如果不为null，将用户对象返回页面

4>将登陆成功用户信息缓存到session中，保证后续请求的时候能识别用户是否已经登录

5>页面将返回的user对象缓存到cookie中，做用户信息回显。

UserController

```
1  @PostMapping("/login")
2  public JsonResult login(String username, String password, String code,
    HttpSession session){
3
4      //1: 校验验证码是否正确
5      String login_code = session.getAttribute("login_code").toString();
6      if(code == null || !code.equalsIgnoreCase(login_code)){
7          return JsonResult.error("验证码错误~");
8      }
9      //2: 执行用户登录逻辑
10     try {
11         User user = userService.login(username, password);
12     }
```

```

13         //将登陆成功用户添加到session对象中
14         session.setAttribute("user_in_session", user);
15         return JsonResult.success(user);
16     } catch (Exception e) {
17         e.printStackTrace();
18         return JsonResult.error(e.getMessage());
19     }
20
21
22 }

```

UserServiceImpl

```

1  @Override
2  public User login(String username, String password) {
3      QueryWrapper<User> wrapper = new QueryWrapper<>();
4      wrapper.eq("username", username);
5      wrapper.eq("password", password);
6      User one = super.getOne(wrapper);
7      if(one == null){
8          throw new RuntimeException("账号或密码错误");
9      }
10     return one;
11 }

```

专家排行

需求：按照用户分数大小倒序排（没被冻结的用户，倒序排，前10个）

专家排行榜



dafei06

大牛1000级 积分:7000



dafei05

大牛1000级 积分:6000



dafei04

大牛1000级 积分:5000



dafei03

大牛1000级 积分:4000



dafei02

大牛1000级 积分:3000



dafei01

大牛1000级 积分:2000



dafei

大牛1000级 积分:1000

*

UserController

```
1 @RequestMapping("/rank")
2 public JsonResult rank(){
3     List<User> users = userService.rank();
4     return JsonResult.success(users);
5 }
```

UserServiceImpl


```
1 @Override
2 public List<User> rank() {
3     QueryWrapper<User> wrapper = new QueryWrapper<>();
4     wrapper.eq("status", User.STATUS_NORMAL).orderByDesc("score").last("
5         limit 10");
6     return super.list(wrapper);
7 }
```

问题表设计

100

回答数

200 VB6编写的程序在W...

VB6编写的程序在WIN10 20H2企业版下，文本框输入中文时，光标位置会乱跳。特别是在一段文字的中间输入中文，正常光标应该是输入文字的后面，但有时会在输入文字的前面，位置没规律。切换微软自带的输入...

回答

excel 程序

dafei

浏览(100) 30分钟前

100

回答数

100 jsp代码运行显示内...

显示内部服务器错误，难道是数据库问题，，，，，，，，，jsp代码运行显示内部服务器错误是怎么回事？？？...

回答

excel 程序

dafei01

浏览(150) 30分钟前

分析效果推，设计出问题表字段

id	title	score	ask_id	status	replynum	viewnum	content	create_time
1	VB6编写的程序在WIN10	200	1	0	100	100	VB6编写的程序在WIN10 20H2企业版下，文本框输入中文时，光标位置会乱跳。特别是在一段文字的中间输入中文，正常光标应该是输入文字的后面，但有时会在输入文字的前面，位置没规律。切换微软自带的输入...	2021-12-24 00:00:00
2	jsp代码运行显示内部服	100	2	0	100	150	显示内部服务器错误，难道是数据库问题，，，，，，，，，jsp代码运行显示内部服务器错误是怎么回事？？？...	2021-12-24 00:00:00
3	学生成绩管理系统.要怎	300	3	0	100	2000	项目作业：学生成绩管理系统实验目的:熟悉一维数组作函数参数	2021-12-24 00:00:00
4	django将上传的excel表	500	4	0	100	1000	求一套很详细很详细的代码讲解或者很详细的思路也可以，刚入行	2021-12-24 00:00:00
5	大家看看围棋打谱吧整	1000	5	0	100	1111	大家了解了解，希望大家花时间帮忙整下，急需帮助	2021-12-24 00:00:00
6	C# postmessage 模拟	502	6	1	100	587	最近遇到个问题，使用 postmessage 每秒向【A窗口】发送一	2021-12-24 00:00:00

问题数据模型准备

- 1>问题表创建
- 2>问题实体类
- 3>问题Mapper
- 4>问题Service
- 5>问题Controller

查询参数设计



问答

全部问题

我的问题

关注问题

问题标签

待解决

已解决

高分

新问题

*

分析查询各种条件，设计查询条件处理对象

```
1  /**
2   * 分页信息
3   */
4  @Setter
5  @Getter
6  public class QueryObject {
7      private int currentPage = 1;
8      private int pageSize = 3;
9  }
10
11
12  /**
13   * 问题查询对象
14   */
15  @Setter
16  @Getter
17  public class QuestionQuery extends QueryObject {
18  }
19
```

问题分页查询

自定义分页查询方法

启动类

```
1  //分页
2  @Bean
3  public MybatisPlusInterceptor mybatisPlusInterceptor() {
4      MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
5      PaginationInnerInterceptor paginationInnerInterceptor = new
6      PaginationInnerInterceptor(DbType.MYSQL);
7      paginationInnerInterceptor.setOverflow(true); //合理化
8      interceptor.addInnerInterceptor(paginationInnerInterceptor);
9      return interceptor;
10 }
```

QuestionServiceImpl

```

1  @Service
2  @Transactional
3  public class QuestionServiceImpl extends ServiceImpl<QuestionMapper,
    Question> implements IQuestionService {
4
5      @Autowired
6      private IUserService userService;
7
8
9      //分页查询步骤: 1>配置分页插件 2>编写分页逻辑
10     @Override
11     public Page<Question> queryPage(QuestionQuery qo) {
12         Page<Question> page = new Page<>(qo.getCurrentPage(),
13         qo.getPageSize());
14         QueryWrapper<Question> wrapper = new QueryWrapper<>();
15         super.page(page, wrapper);
16         //处理用户
17         for (Question record : page.getRecords()) {
18             record.setAsk(userService.getById(record.getAskId()));
19         }
20         return page;
21     }
22 }

```

问题条件查询

全部问题

我的问题

关注问题

问题标签

待解决

已解决

高分

新问题

查询对象

```

1  /**
2   * 问题查询对象
3   */
4  @Setter
5  @Getter
6  public class QuestionQuery extends QueryObject {
7      private int status = -1;
8      private String orderBy;
9  }

```

QuestServiceImpl

```

1  @Service
2  @Transactional
3  public class QuestionServiceImpl extends ServiceImpl<QuestionMapper,
    Question> implements IQuestionService {
4
5      @Autowired

```

问题添加

发表问题

标题

描述

是莫

项目总结

一个完整的项目结构搭建起来啦，MyBatis-Plus在项目中使用时就明显了，后续自己找一些开源项目或者改造之前项目都行，巩固MyBatis-plus的学习。

总结与个人使用MyBatis-Plus建议

- MyBatis-Plus 简介
- MyBatis-Plus 入门案例与解析
- 常用的MyBatis-Plus注解
- 通用的Mapper接口与API详解
- 实现零SQL利器-条件构造器
- MyBatis-Plus条件查询
- MyBatis-Plus关联关系
- MyBatis-Plus-AR模式开发
- MyBatis-Plus通用的Service接口与API详解
- MyBatis-Plus逻辑删除实现与原理分析
- MyBatis-Plus乐观锁实现与原理分析
- MyBatis-Plus通用枚举实现
- MyBatis-Plus类型处理实现与原理分析
- MyBatis-Plus AutoGenerator 代码生成器实现
- MyBatis-Plus 项目实践
- 课程总结与开发建议

MyBatis-Plus使用小建议

1>简单，单表操作项目首选MyBatis-Plus

2>复杂，多表操项目选择MyBatis-Plus +MyBatis

3>追求代码结构清爽，追求极致性能，代码有重构要求项目不建议使用MyBatis-Plus

