MybatisPlus从入门到精通-基础篇

1.概述

MybatisPlus是一款Mybatis增强工具，用于简化开发，提高效率。 它在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

官网： https://mp.baomidou.com/

2.快速入门

2.0 准备工作

①准备数据

```
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'id',
  `user_name` varchar(20) NOT NULL COMMENT '用户名',
  `password` varchar(20) NOT NULL COMMENT '密码',
  `name` varchar(30) DEFAULT NULL COMMENT '姓名',
  `age` int(11) DEFAULT NULL COMMENT '年龄',
  `address` varchar(100) DEFAULT NULL COMMENT '地址',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

insert  into `user`(`id`,`user_name`,`password`,`name`,`age`,`address`) values
(1,'ruiwen','123','瑞文',12,'山东'),(2,'gailun','1332','盖伦',13,'平顶山'),
(3,'timu','123','提姆',22,'蘑菇石'),(4,'daji','1222','妲己',221,'狐山');
```

②创建SpringBoot工程

添加依赖

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.0</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
```

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

创建启动类

```
@SpringBootApplication
public class SGApplication {

    public static void main(String[] args) {
        SpringApplication.run(SGApplication.class);
    }
}
```

③准备实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Long id;
    private String userName;
    private String password;
    private String name;
    private Integer age;
    private String address;
}
```

2.1 使用MybatisPlus

①添加依赖

```
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.4.3</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
```

②配置数据库信息

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mp_db?characterEncoding=utf-
8&serverTimezone=UTC
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
```

③创建Mapper接口

创建Mapper接口继承BaseMapper接口

```java
public interface UserMapper extends BaseMapper<User> {
}
```

BaseMapper接口中已经提供了很多常用方法。所以我们只需要直接从容器中获取Mapper就可以进行操作了，不需要自己去编写Sql语句。

④配置Mapper扫描

在启动类上配置我们的Mapper在哪个包。

```java
@SpringBootApplication
@MapperScan("com.sangeng.mapper")
public class SGApplication {
    public static void main(String[] args) {
        SpringApplication.run(SGApplication.class,args);
    }
}
```

⑤获取Mapper进行测试

```java
@SpringBootTest
public class MPTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testQueryList(){
        System.out.println(userMapper.selectList(null));
    }

}
```

3.常用设置

3.1 设置表映射规则

默认情况下MP操作的表名就是实体类的类名，但是如果表名和类名不一致就需要我们自己设置映射规则。

3.1.1 单独设置

可以在实体类的类名上加上@TableName注解进行标识。

例如：

如果表名是tb_user，而实体类名是User则可以使用以下写法。

```
@TableName("tb_user")
public class User {
    //....
}
```

3.1.2 全局设置表名前缀

一般一个项目表名的前缀都是统一风格的，这个时候如果一个个设置就太麻烦了。我们可以通过配置来设置全局的表名前缀。

例如：

如果一个项目中所有的表名相比于类名都是多了个前缀： `tb_` 。这可以使用如下方式配置

```
mybatis-plus:
  global-config:
    db-config:
      #表名前缀
      table-prefix: tb_
```

3.2 设置主键生成策略

3.2.0 测试代码

```
    @Test
    public void testInsert(){
        User user = new User();
        user.setUserName("三更草堂222");
        user.setPassword("7777");
        int r = userMapper.insert(user);
        System.out.println(r);
    }
```

3.2.1 单独设置

默认情况下使用MP插入数据时，如果在我们没有设置主键生成策略的情况下默认的策略是基于雪花算法的自增id。

如果我们需要使用别的策略可以在定义实体类时，在代表主键的字段上加上`@TableId`注解，使用其`type`属性指定主键生成策略。

例如：

我们要设置主键自动增长则可以设置如下

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    //.....
}
```

全部主键策略定义在了枚举类`IdType`中，`IdType`有如下的取值

- `AUTO`

  数据库ID自增，**依赖于数据库**。该类型请确保数据库设置了 ID自增 否则无效

- `NONE`

  未设置主键类型。若在代码中没有手动设置主键，则会根据**主键的全局策略**自动生成（默认的主键全局策略是基于雪花算法的自增ID）

- `INPUT`

  需要手动设置主键，若不设置。插入操作生成SQL语句时，主键这一列的值会是`null`。

- `ASSIGN_ID`

  当没有手动设置主键，即实体类中的主键属性为空时，才会自动填充，使用雪花算法

- `ASSIGN_UUID`

当实体类的主键属性为空时，才会自动填充，使用UUID

3.2.2 全局设置

```
mybatis-plus:
  global-config:
    db-config:
      # id生成策略 auto为数据库自增
      id-type: auto
```

3.3 设置字段映射关系

默认情况下MP会根据实体类的属性名去映射表的列名。

如果数据库的列表和实体类的属性名不一致了我们可以使用`@TableField`注解的`value`属性去设置映射关系。

例如：

如果表中一个列名叫 address而 实体类中的属性名为addressStr则可以使用如下方式进行配置。

```
    @TableField("address")
    private String addressStr;
```

3.4 设置字段和列名的驼峰映射

默认情况下MP会开启字段名列名的驼峰映射， 即从经典数据库列名 A_COLUMN（下划线命名） 到经典 Java 属性名 aColumn（驼峰命名） 的类似映射 。

如果需要关闭我们可以使用如下配置进行关闭。

```
mybatis-plus:
  configuration:
    #是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A_COLUMN（下划线命名）到经典 Java 属性名 aColumn（驼峰命名） 的类似映射
    map-underscore-to-camel-case: false
```

3.5 日志

如果需要打印MP操作对应的SQL语句等，可以配置日志输出。

配置方式如下：

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

4.基本使用

4.1 插入数据

我们可以使用insert方法来实现数据的插入。

示例：

```
    @Test
    public void testInsert(){
        User user = new User();
        user.setUserName("三更草堂333");
        user.setPassword("7777888");
        int r = userMapper.insert(user);
        System.out.println(r);
    }
```

4.2 删除操作

我们可以使用**deleteXXX**方法来实现数据的删除。

示例：

```
    @Test
    public void testDelete(){
        List<Integer> ids = new ArrayList<>();
        ids.add(5);
        ids.add(6);
        ids.add(7);
        int i = userMapper.deleteBatchIds(ids);
        System.out.println(i);
    }
    @Test
    public void testDeleteById(){
        int i = userMapper.deleteById(8);
        System.out.println(i);
    }
    @Test
    public void testDeleteByMap(){
        Map<String, Object> map = new HashMap<>();
        map.put("name","提姆");
        map.put("age",22);
        int i = userMapper.deleteByMap(map);
        System.out.println(i);
    }
```

4.3 更新操作

我们可以使用**updateXXX**方法来实现数据的删除。

示例：

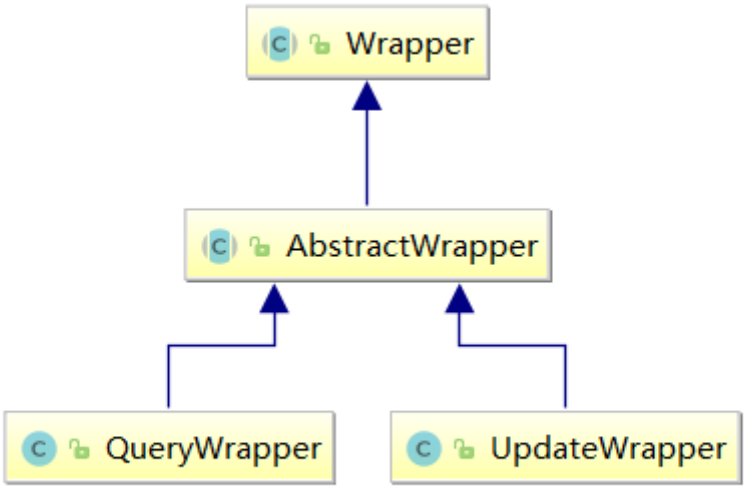
```
    @Test
    public void testUpdate(){
        //把id为2的用户的年龄改为14
        User user = new User();
        user.setId(2L);
        user.setAge(14);
        int i = userMapper.updateById(user);
        System.out.println(i);
    }
```

5.条件构造器Wrapper

5.1 概述

我们在实际操作数据库的时候会涉及到很多的条件。所以MP为我们提供了一个功能强大的条件构造器 `Wrapper` 。使用它可以让我们非常方便的构造条件。

其继承体系如下：



在其子类 `AbstractWrapper` 中提供了很多用于构造Where条件的方法。

`AbstractWrapper` 的子类 `QueryWrapper` 则额外提供了用于针对Select语法的 `select` 方法。可以用来设置查询哪些列。

`AbstractWrapper` 的子类 `UpdateWrapper` 则额外提供了用于针对SET语法的 `set` 方法。可以用来设置对哪些列进行更新。

完整的AbstractWrapper方法可以参照：https://baomidou.com/guide/wrapper.html#abstractwrapper

介绍是用来干什么的。它的实现类有哪些

QueryWrapper,UpdateWrapper，【LambdaQueryWrapper】

5.2 常用AbstractWrapper方法

> eq：equals，等于 gt：greater than ，大于 > ge：greater than or equals，大于等于≥ lt：less than，小于< le：less than or equals，小于等于≤ between：相当于SQL中的BETWEEN like：模糊匹配。like("name","黄")，相当于SQL的name like '%黄%' likeRight：模糊匹配右半边。likeRight("name","黄")，相当于SQL的name like '黄%' likeLeft：模糊匹配左半边。likeLeft("name","黄")，相当于SQL的name like '%黄' notLike：notLike("name","黄")，相当于SQL的name not like '%黄%' isNull isNotNull and：SQL连接符AND or：SQL连接符OR
>
> in: in(“age",{1,2,3})相当于 age in(1,2,3)
>
> groupBy: groupBy("id","name")相当于 group by id,name
>
> orderByAsc :orderByAsc("id","name")相当于 order by id ASC,name ASC
>
> orderByDesc :orderByDesc ("id","name")相当于 order by id DESC,name DESC

示例一

SQL语句如下：

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    age > 18 AND address = '狐山'
```

如果用Wrapper写法如下：

```
    @Test
    public void testWrapper01(){
        QueryWrapper wrapper = new QueryWrapper();
        wrapper.gt("age",18);
        wrapper.eq("address","狐山");
        List<User> users = userMapper.selectList(wrapper);
        System.out.println(users);
    }
```

示例二

SQL语句如下：

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    id IN(1,2,3) AND
    age BETWEEN 12 AND 29 AND
    address LIKE '%山%'
```

如果用Wrapper写法如下：

```
    @Test
    public void testWrapper02(){
        QueryWrapper<User> wrapper = new QueryWrapper<>();
        wrapper.in("id",1,2,3);
        wrapper.between("age",12,29);
        wrapper.like("address","山");
        List<User> users = userMapper.selectList(wrapper);
        System.out.println(users);
    }
```

示例三

SQL语句如下：

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    id IN(1,2,3) AND
    age > 10
ORDER BY
    age DESC
```

如果用Wrapper写法如下：

```
    @Test
    public void testWrapper03(){
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.in("id",1,2,3);
        queryWrapper.gt("age",10);
        queryWrapper.orderByDesc("age");
        List<User> users = userMapper.selectList(queryWrapper);
        System.out.println(users);
    }
```

5.3 常用QueryWrapper方法

QueryWrapper的 select 可以设置要查询的列。

示例一

> select(String... sqlSelect) 方法的测试为要查询的列名

SQL语句如下：

```
SELECT
    id,user_name
FROM
    USER
```

MP写法如下：

```
    @Test
    public void testSelect01(){
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.select("id","user_name");
        List<User> users = userMapper.selectList(queryWrapper);
        System.out.println(users);
    }
```

示例二

> select(Class entityClass, Predicate predicate)

方法的第一个参数为实体类的字节码对象，第二个参数为Predicate类型，可以使用lambda的写法，过滤要查询的字段 (主键除外) 。

SQL语句如下：

```
SELECT
    id,user_name
FROM
    USER
```

MP写法如下：

```
    @Test
    public void testSelect02(){
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.select(User.class, new Predicate<TableFieldInfo>() {
            @Override
            public boolean test(TableFieldInfo tableFieldInfo) {
                return "user_name".equals(tableFieldInfo.getColumn());
            }
        });
        List<User> users = userMapper.selectList(queryWrapper);
        System.out.println(users);
    }
```

示例三

> select(Predicate predicate)

方法第一个参数为Predicate类型，可以使用lambda的写法，过滤要查询的字段 (主键除外) 。

SQL语句如下：

```
SELECT
    id,user_name,PASSWORD,NAME,age
FROM
    USER
```

就是不想查询address这列，其他列都查询了

MP写法如下：

```
    @Test
    public void testSelect03(){
        QueryWrapper<User> queryWrapper = new QueryWrapper<>(new User());
        queryWrapper.select(new Predicate<TableFieldInfo>() {
            @Override
            public boolean test(TableFieldInfo tableFieldInfo) {
                return !"address".equals(tableFieldInfo.getColumn());
            }
        });
        List<User> users = userMapper.selectList(queryWrapper);
        System.out.println(users);
    }
```

5.4 常用UpdateWrapper方法

我们前面在使用update方法时需要创建一个实体类对象传入，用来指定要更新的列及对应的值。但是如果需要更新的列比较少时，创建这么一个对象显的有点麻烦和复杂。

我们可以使用UpdateWrapper的set方法来设置要更新的列及其值。同时这种方式也可以使用Wrapper去指定更复杂的更新条件。

示例

SQL语句如下：

```
UPDATE
    USER
SET
    age = 99
where
    id > 1
```

我们想把id大于1的用户的年龄修改为99，则可以使用如下写法：

```
    @Test
    public void testUpdateWrapper(){
        UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();
        updateWrapper.gt("id",1);
        updateWrapper.set("age",99);
        userMapper.update(null,updateWrapper);
    }
```

5.5 Lambda条件构造器

我们前面在使用条件构造器时列名都是用字符串的形式去指定。这种方式无法在编译期确定列名的合法性。

所以MP提供了一个Lambda条件构造器可以让我们直接以实体类的方法引用的形式来指定列名。这样就可以弥补上述缺陷。

示例

要执行的查询对应的SQL如下

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    age > 18 AND address = '狐山'
```

如果使用之前的条件构造器写法如下

```
    @Test
    public void testLambdaWrapper(){
        QueryWrapper<User> queryWrapper = new QueryWrapper();
        queryWrapper.gt("age",18);
        queryWrapper.eq("address","狐山");
        List<User> users = userMapper.selectList(queryWrapper);
    }
```

如果使用Lambda条件构造器写法如下

```
    @Test
    public void testLambdaWrapper2(){
        LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
        queryWrapper.gt(User::getAge,18);
        queryWrapper.eq(User::getAddress,"狐山");
        List<User> users = userMapper.selectList(queryWrapper);
    }
```

6.自定义SQL

虽然MP为我们提供了很多常用的方法，并且也提供了条件构造器。但是如果真的遇到了复制的SQL时，我们还是需要自己去定义方法，自己去写对应的SQL，这样SQL也更有利于后期维护。

因为MP是对mybatis做了增强，所以还是支持之前Mybatis的方式去自定义方法。

同时也支持在使用Mybatis的自定义方法时使用MP的条件构造器帮助我们进行条件构造。

接下去我们分别来讲讲。

6.0 准备工作

①准备数据

```
CREATE TABLE `orders` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `price` int(11) DEFAULT NULL COMMENT '价格',
  `remark` varchar(100) DEFAULT NULL COMMENT '备注',
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  `update_time` timestamp NULL DEFAULT NULL COMMENT '更新时间',
  `create_time` timestamp NULL DEFAULT NULL COMMENT '创建时间',
  `version` int(11) DEFAULT '1' COMMENT '版本',
  `del_flag` int(1) DEFAULT '0' COMMENT '逻辑删除标识,0-未删除,1-已删除',
  `create_by` varchar(100) DEFAULT NULL COMMENT '创建人',
  `update_by` varchar(100) DEFAULT NULL COMMENT '更新人',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;

/*Data for the table `orders` */
```

```
insert  into
`orders`(`id`,`price`,`remark`,`user_id`,`update_time`,`create_time`,`version`,`
del_flag`,`create_by`,`update_by`) values (1,2000,'无',2,'2021-08-24
21:02:43','2021-08-24 21:02:46',1,0,NULL,NULL),(2,3000,'无',3,'2021-08-24
21:03:32','2021-08-24 21:03:35',1,0,NULL,NULL),(3,4000,'无',2,'2021-08-24
21:03:39','2021-08-24 21:03:41',1,0,NULL,NULL);
```

②创建实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Orders  {

    private Long id;

    /**
     * 价格
     */
    private Integer price;

    /**
     * 备注
     */
    private String remark;

    /**
     * 用户id
     */
    private Integer userId;

    /**
     * 更新时间
     */
    private LocalDateTime updateTime;

    /**
     * 创建时间
     */
    private LocalDateTime createTime;

    /**
     * 版本
     */
    private Integer version;

    /**
     * 逻辑删除标识,0-未删除,1-已删除
     */
    private Integer delFlag;


}
```

6.1 Mybatis方式

①定义方法

在Mapper接口中定义方法

```
public interface UserMapper extends BaseMapper<User> {

    User findMyUser(Long id);
}
```

②创建xml

先配置xml文件的存放目录

```
mybatis-plus:
  mapper-locations: classpath*:/mapper/**/*.xml
```

创建对应的xml映射文件

③在xml映射文件中编写SQL

创建对应的标签，编写对应的SQL语句

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

    <select id="findMyUser" resultType="com.sangeng.domian.User">
       select * from user where id = #{id}
    </select>
</mapper>
```

6.2 Mybatis方式结合条件构造器

我们在使用上述方式自定义方法时。如果也希望我们的自定义方法能像MP自带方法一样使用条件构造器来进行条件构造的话只需要使用如下方式即可。

①方法定义中添加Warpper类型的参数

添加Warpper类型的参数，并且要注意给其指定参数名。

```
public interface UserMapper extends BaseMapper<User> {

    User findMyUserByWrapper(@Param(Constants.WRAPPER) Wrapper<User> wrapper);
}
```

②在SQL语句中获取Warpper拼接的SQL片段进行拼接。

```
    <select id="findMyUserByWrapper" resultType="com.sangeng.domian.User">
        select * from user ${ew.customSqlSegment}
    </select>
```

注意：不能使用#{}应该用${}

7.分页查询

7.1 基本分页查询

①配置分页查询拦截器

```
@Configuration
public class PageConfig {

    /**
     * 3.4.0之前的版本
     * @return
     */
   /* @Bean
    public PaginationInterceptor paginationInterceptor(){
        return  new PaginationInterceptor();
    }*/

    /**
     * 3.4.0之后版本
     * @return
     */
   @Bean
   public MybatisPlusInterceptor mybatisPlusInterceptor(){
       MybatisPlusInterceptor mybatisPlusInterceptor = new
MybatisPlusInterceptor();
       mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
       return mybatisPlusInterceptor;
   }
}
```

②进行分页查询

```
    @Test
    public void testPage(){
        IPage<User> page = new Page<>();
        //设置每页条数
        page.setSize(2);
        //设置查询第几页
        page.setCurrent(1);
        userMapper.selectPage(page, null);
        System.out.println(page.getRecords());//获取当前页的数据
        System.out.println(page.getTotal());//获取总记录数
        System.out.println(page.getCurrent());//当前页码
    }
```

7.2 多表分页查询

如果需要在多表查询时进行分页查询的话，就可以在mapper接口中自定义方法，然后让方法接收Page对象。

示例

需求

我们需要去查询Orders表，并且要求查询的时候除了要获取到Orders表中的字段，还要获取到每个订单的下单用户的用户名。

准备工作

SQL准备

```
SELECT
    o.*,u.`user_name`
FROM
    USER u,orders o
WHERE
    o.`user_id` = u.`id`
```

实体类修改

增加一个userName属性

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Orders  {
    //省略无关代码
    private String userName;
}
```

实现

①定义接口，定义方法

方法第一个测试定义成Page类型

```
public interface OrdersMapper extends BaseMapper<Orders> {

    IPage<Orders> findAllOrders(Page<Orders> page);
}
```

在xml中不需要关心分页操作，MP会帮我们完成。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.sangeng.mapper.OrdersMapper">
    <select id="findAllOrders" resultType="com.sangeng.domian.Orders">
        SELECT
            o.*,u.`user_name`
        FROM
            USER u,orders o
        WHERE
            o.`user_id` = u.`id`
    </select>
</mapper>
```

然后调用方法测试即可

```
    @Autowired
    private OrdersMapper ordersMapper;
    @Test
    public void testOrdersPage(){
        Page<Orders> page = new Page<>();
        //设置每页大小
        page.setSize(2);
        //设置当前页码
        page.setCurrent(2);
        ordersMapper.findAllOrders(page);
        System.out.println(page.getRecords());
        System.out.println(page.getTotal());
    }
```

8.Service 层接口

MP也为我们提供了Service层的实现。我们只需要编写一个接口，继承`IService`，并创建一个接口实现类继承`ServiceImpl`，即可使用。

相比于Mapper接口，Service层主要是支持了更多批量操作的方法。

8.1 基本使用

8.1.1 改造前

定义接口

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

定义实现类

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserMapper userMapper;

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

8.1.2 改造后

接口

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

}
```

实现类

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

}
```

测试

```
    @Autowired
    private UserService userService;


    @Test
    public void testSeervice(){
        List<User> list = userService.list();
        System.out.println(list);
    }
```

8.2自定义方法

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


    User test();
}
```

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

    @Autowired
    private OrdersMapper ordersMapper;

    @Override
    public User test() {
        UserMapper userMapper = getBaseMapper();
        List<Orders> orders = ordersMapper.selectList(null);
        User user = userMapper.selectById(3);
        //查询用户对于的订单
        QueryWrapper<Orders> wrapper = new QueryWrapper<>();
        wrapper.eq("user_id",3);
        List<Orders> ordersList = ordersMapper.selectList(wrapper);
        return user;
    }
}
```

9.代码生成器

MP提供了一个代码生成器，可以让我们一键生成实体类，Mapper接口，Service，Controller等全套代码 。使用方式如下

①添加依赖

```
        <!--mybatisplus代码生成器-->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-generator</artifactId>
            <version>3.4.1</version>
        </dependency>
        <!--模板引擎-->
        <dependency>
            <groupId>org.freemarker</groupId>
            <artifactId>freemarker</artifactId>
        </dependency>
```

②生成

修改相应配置后执行以下代码即可

```
public class GeneratorTest {
    @Test
    public void generate() {
        AutoGenerator generator = new AutoGenerator();

        // 全局配置
        GlobalConfig config = new GlobalConfig();
        String projectPath = System.getProperty("user.dir");
        // 设置输出到的目录
        config.setOutputDir(projectPath + "/src/main/java");
        config.setAuthor("sangeng");
        // 生成结束后是否打开文件夹
        config.setOpen(false);

        // 全局配置添加到 generator 上
        generator.setGlobalConfig(config);

        // 数据源配置
        DataSourceConfig dataSourceConfig = new DataSourceConfig();
        dataSourceConfig.setUrl("jdbc:mysql://localhost:3306/mp_db?
characterEncoding=utf-8&serverTimezone=UTC");
        dataSourceConfig.setDriverName("com.mysql.cj.jdbc.Driver");
        dataSourceConfig.setUsername("root");
        dataSourceConfig.setPassword("root");

        // 数据源配置添加到 generator
        generator.setDataSource(dataSourceConfig);

        // 包配置, 生成的代码放在哪个包下
        PackageConfig packageConfig = new PackageConfig();
        packageConfig.setParent("com.sangeng.mp.generator");

        // 包配置添加到 generator
        generator.setPackageInfo(packageConfig);
```

```
        // 策略配置
        StrategyConfig strategyConfig = new StrategyConfig();
        // 下划线驼峰命名转换
        strategyConfig.setNaming(NamingStrategy.underline_to_camel);
        strategyConfig.setColumnNaming(NamingStrategy.underline_to_camel);
        // 开启lombok
        strategyConfig.setEntityLombokModel(true);
        // 开启RestController
        strategyConfig.setRestControllerStyle(true);
        generator.setStrategy(strategyConfig);
        generator.setTemplateEngine(new FreemarkerTemplateEngine());

        // 开始生成
        generator.execute();
    }
}
```