

[🏠 / 开发指南 / 后端手册](#)[👤 芋道源码](#) [📅 2022-04-01](#)

# 🔗 MyBatis 数据库

[yudao-spring-boot-starter-mybatis](#) [🔗](#) 技术组件，基于 MyBatis Plus 实现数据库的操作。如果你没有学习过 MyBatis Plus，建议先阅读 [《芋道 Spring Boot MyBatis 入门》](#) [🔗](#) 文章。

## 友情提示

MyBatis 是最容易读懂的 Java 框架之一，感兴趣的话，可以看看芋芋写的 [《芋道 MyBatis 源码解析》](#) [🔗](#) 系列，已经有 18000 人学习过！

## # 1. 实体类

[BaseDO](#) [🔗](#) 是所有数据库实体的父类，代码如下：

```
@Data
public abstract class BaseDO implements Serializable {

    /**
     * 创建时间
     */
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;

    /**
     * 最后更新时间
     */
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;

    /**
     * 创建者，目前使用 AdminUserDO / MemberUserDO 的 id 编号
     *
     * 使用 String 类型的原因是，未来可能会存在非数值的情况，留好拓展性。
     */
    @TableField(fill = FieldFill.INSERT)
    private String creator;

    /**
     * 更新者，目前使用 AdminUserDO / MemberUserDO 的 id 编号
     *
     * 使用 String 类型的原因是，未来可能会存在非数值的情况，留好拓展性。
     */
    @TableField(fill = FieldFill.INSERT_UPDATE)
```

```
private String updater;
/**
 * 是否删除
 */
@TableLogic
private Boolean deleted;

}
```

- `createTime` + `creator` 字段，创建人相关信息。
- `updater` + `updateTime` 字段，创建人相关信息。
- `deleted` 字段，逻辑删除。

对应的 SQL 字段如下：

```
`creator` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci DEFAULT '
`create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
`updater` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci DEFAULT '
`update_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIME
`deleted` bit(1) NOT NULL DEFAULT b'0' COMMENT '是否删除',
```

## 1.1 主键编号

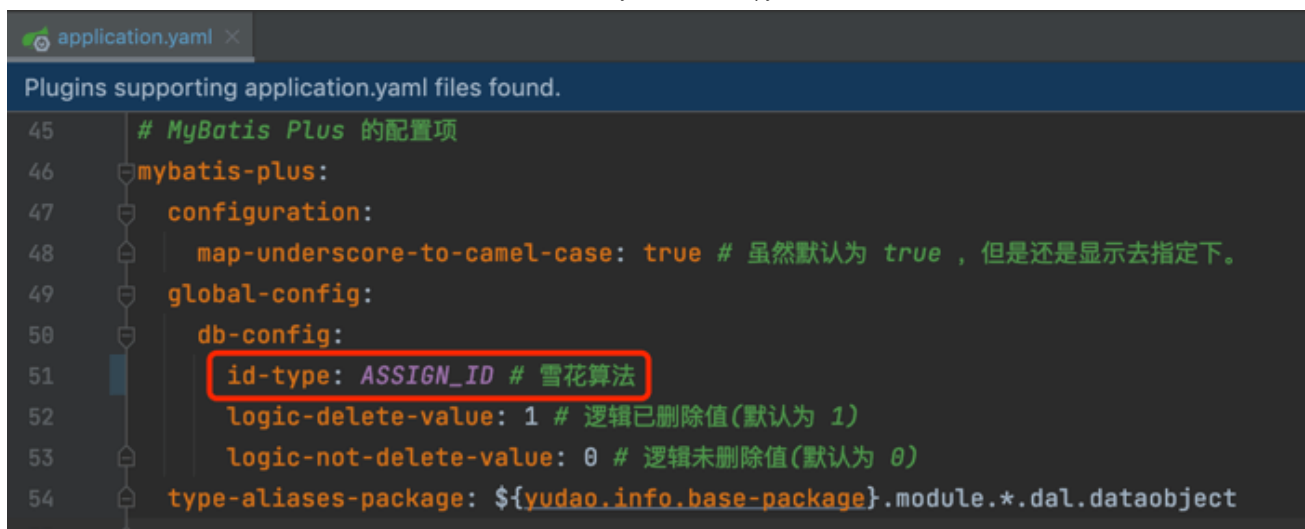
`id` 主键编号，推荐使用 Long 型自增，原因是：

- 自增，保证数据库是按顺序写入，性能更加优秀。
- Long 型，避免未来业务增长，超过 Int 范围。

对应的 SQL 字段如下：

```
`id` bigint NOT NULL AUTO_INCREMENT COMMENT '编号',
```

项目的 `id` 默认采用数据库自增的策略，如果希望使用 Snowflake 雪花算法，可以修改 `application.yaml` 配置文件，将配置项 `mybatis-plus.global-config.db-config.id-type` 修改为 `ASSIGN_ID`。如下图所示：

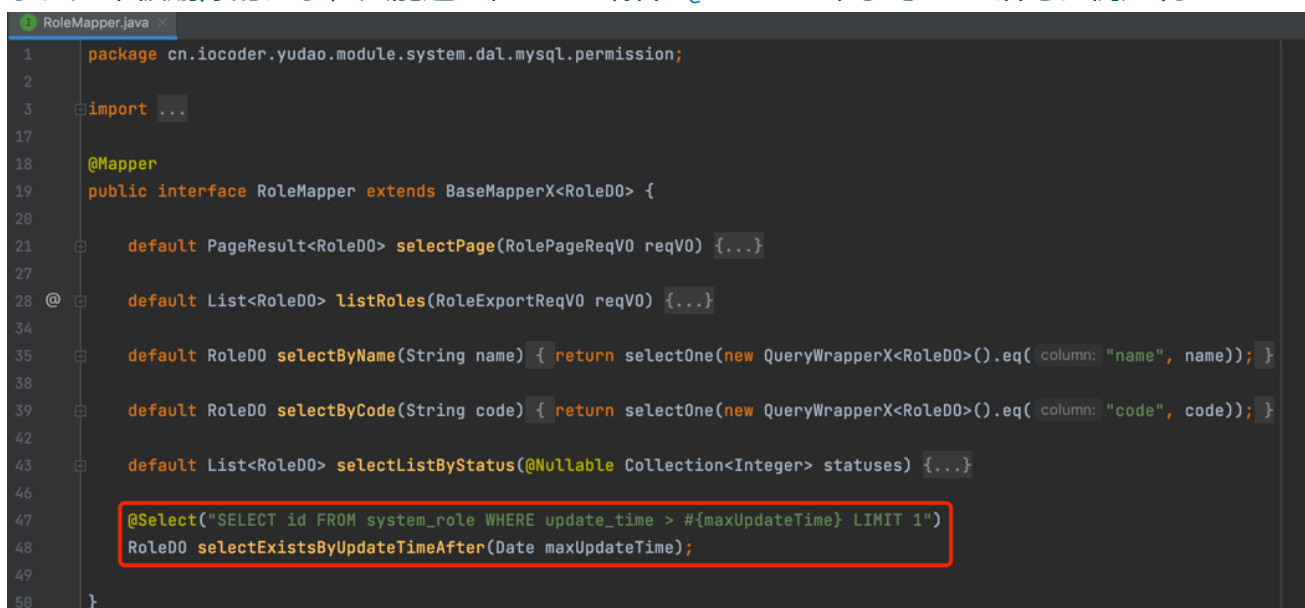


## 1.2 逻辑删除

所有表通过 `deleted` 字段来实现逻辑删除，值为 0 表示未删除，值为 1 表示已删除，可见 `application.yaml` 配置文件的 `logic-delete-value` 和 `logic-not-delete-value` 配置项。如下图所示：



① 所有 SELECT 查询，都会自动拼接 `WHERE deleted = 0` 查询条件，过滤已经删除的记录。如果被删除的记录，只能通过在 XML 或者 `@SELECT` 来手写 SQL 语句。例如说：



② 建立唯一索引时，需要额外增加 `delete_time` 字段，添加到唯一索引字段中，避免唯一索引冲突。例如说，`system_users` 使用 `username` 作为唯一索引：

- 未添加前：先逻辑删除了一条 `username = yudao` 的记录，然后又插入了一条 `username = yudao` 的记录时，会报索引冲突的异常。
- 已添加后：先逻辑删除了一条 `username = yudao` 的记录并更新 `delete_time` 为当前时间，然后又插入一条 `username = yudao` 并且 `delete_time` 为 0 的记录，不会导致唯一索引冲突。

## 1.3 自动填充

`DefaultDBFieldHandler` 基于 MyBatis 自动填充机制，实现 BaseDO 通用字段的自动设置。

代码如下如：

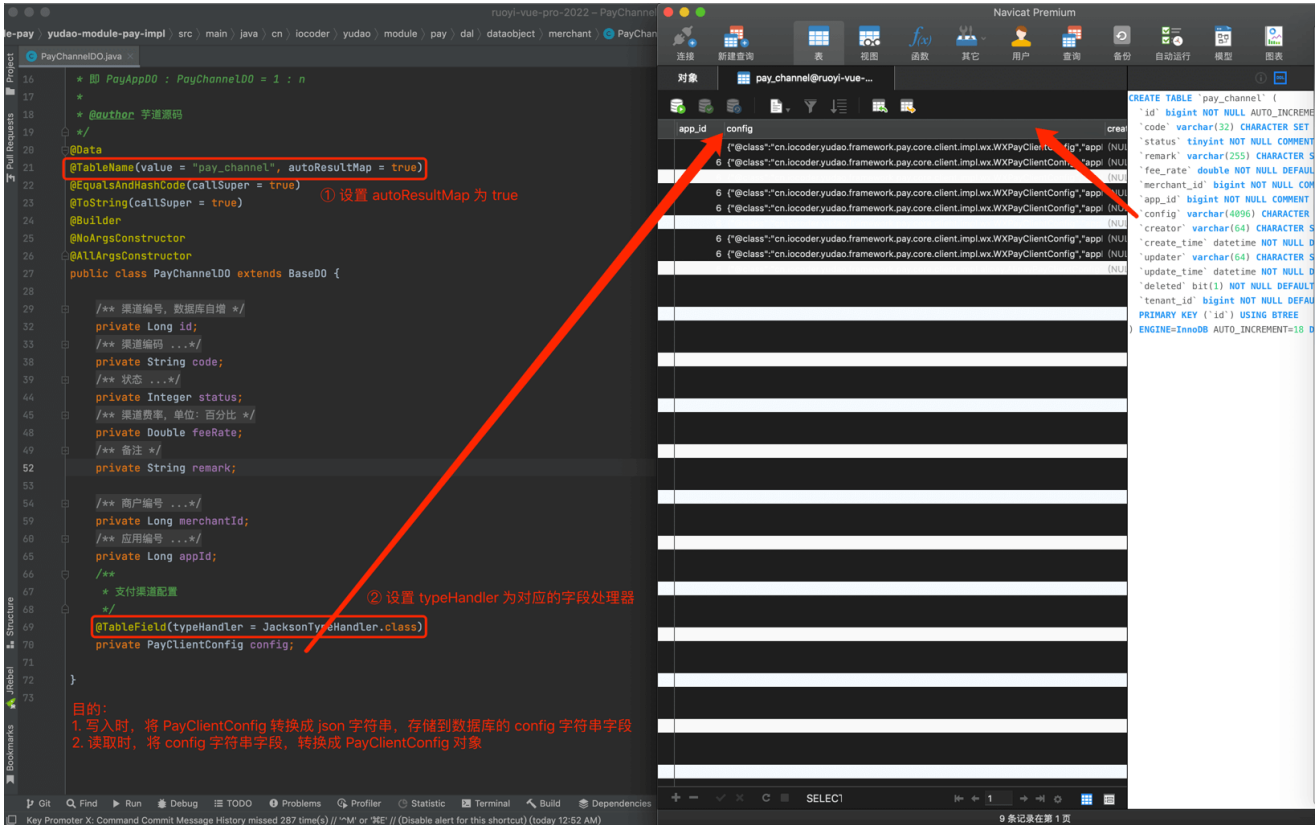
```
18 public class DefaultDBFieldHandler implements MetaObjectHandler {
19
20     @Override 插入时的填充 createTime、creator、updateTime、updater
21     public void insertFill(MetaObject metaObject) {
22         if (Objects.nonNull(metaObject) && metaObject.getOriginalObject() instanceof BaseDO) {
23             BaseDO baseDO = (BaseDO) metaObject.getOriginalObject();
24
25             Date current = new Date();
26             // 创建时间为空，则以当前时间为插入时间
27             if (Objects.isNull(baseDO.getCreateTime())) {
28                 baseDO.setCreateTime(current);
29             }
30             // 更新时间为空，则以当前时间为更新时间
31             if (Objects.isNull(baseDO.getUpdateTime())) {
32                 baseDO.setUpdateTime(current);
33             }
34
35             Long userId = WebFrameworkUtils.getLoginUserId();
36             // 当前登录用户不为空，创建人为空，则当前登录用户为创建人
37             if (Objects.nonNull(userId) && Objects.isNull(baseDO.getCreator())) {
38                 baseDO.setCreator(userId.toString());
39             }
40             // 当前登录用户不为空，更新人为空，则当前登录用户为更新人
41             if (Objects.nonNull(userId) && Objects.isNull(baseDO.getUpdater())) {
42                 baseDO.setUpdater(userId.toString());
43             }
44         }
45     }
46
47     @Override 更新时的填充 updateTime、updater
48     public void updateFill(MetaObject metaObject) {
49         // 更新时间为空，则以当前时间为更新时间
50         Object modifyTime = getFieldValByName("updateTime", metaObject);
51         if (Objects.isNull(modifyTime)) {
52             setFieldValByName("updateTime", new Date(), metaObject);
53         }
54
55         // 当前登录用户不为空，更新人为空，则当前登录用户为更新人
56         Object modifier = getFieldValByName("updater", metaObject);
57         Long userId = WebFrameworkUtils.getLoginUserId();
58         if (Objects.nonNull(userId) && Objects.isNull(modifier)) {
59             setFieldValByName("updater", userId.toString(), metaObject);
60         }
61     }
62 }

1 package cn.iocoder.yudao.framework.mybatis.core.dataobject;
2
3 import ...
4
5 /**
6  * 基础实体对象
7  *
8  * @author 芋道源码
9  */
10 @Data
11 public abstract class BaseDO implements Serializable {
12
13     /**
14      * 创建时间
15      */
16     @TableField(fill = FieldFill.INSERT) 插入时
17     private Date createTime;
18
19     /**
20      * 最后更新时间
21      */
22     @TableField(fill = FieldFill.INSERT_UPDATE) 插入 or 更新时
23     private Date updateTime;
24
25     /**
26      * 创建者，目前使用 AdminUserDO / MemberUserDO 的 id 编号
27      *
28      * 使用 String 类型的原因是，未来可能会存在非数值的情况，留好拓展性。
29      */
30     @TableField(fill = FieldFill.INSERT) 插入时
31     private String creator;
32
33     /**
34      * 更新者，目前使用 AdminUserDO / MemberUserDO 的 id 编号
35      *
36      * 使用 String 类型的原因是，未来可能会存在非数值的情况，留好拓展性。
37      */
38     @TableField(fill = FieldFill.INSERT_UPDATE) 插入 or 更新时
39     private String updater;
40
41     /**
42      * 是否删除
43      */
44     @TableLogic
45     private Boolean deleted;
46
47 }
48
49 }
```

## 1.4 “复杂”字段类型

MyBatis Plus 提供 `TypeHandler` 字段类型处理器，用于 `JavaType` 与 `JdbcType` 之间的转换。

示例如下：



常用的字段类型处理器有：

- [JacksonTypeHandler](#)：通用的 Jackson 实现 JSON 字段类型处理器。
- [JsonLongSetTypeHandler](#)：针对 `Set<Long>` 的 Jackson 实现 JSON 字段类型处理器。

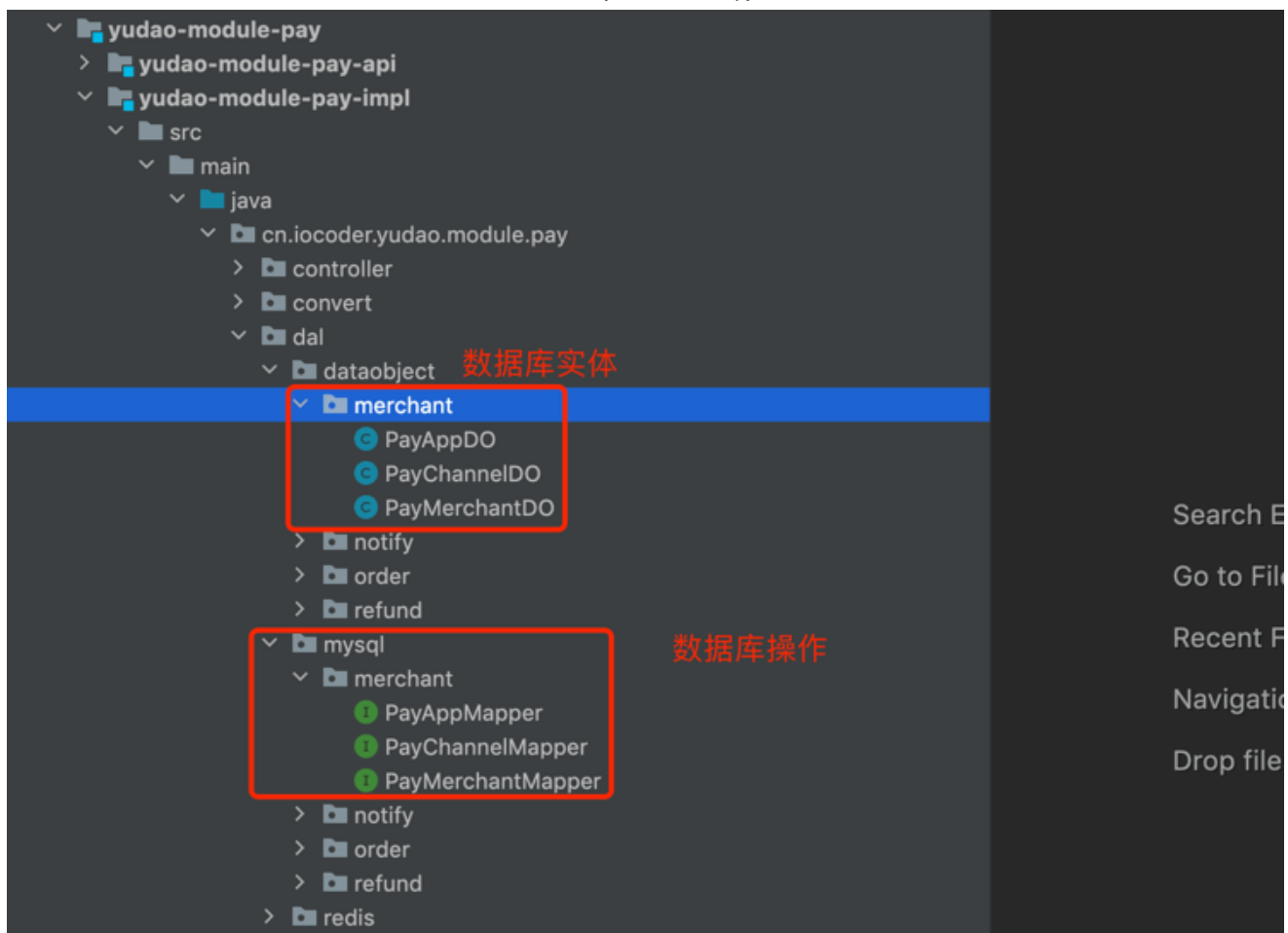
另外，如果你后续要拓展自定义的 TypeHandler 实现，可以添加到 `cn.iocoder.yudao.framework.mybatis.core.type` 包下。

注意事项：

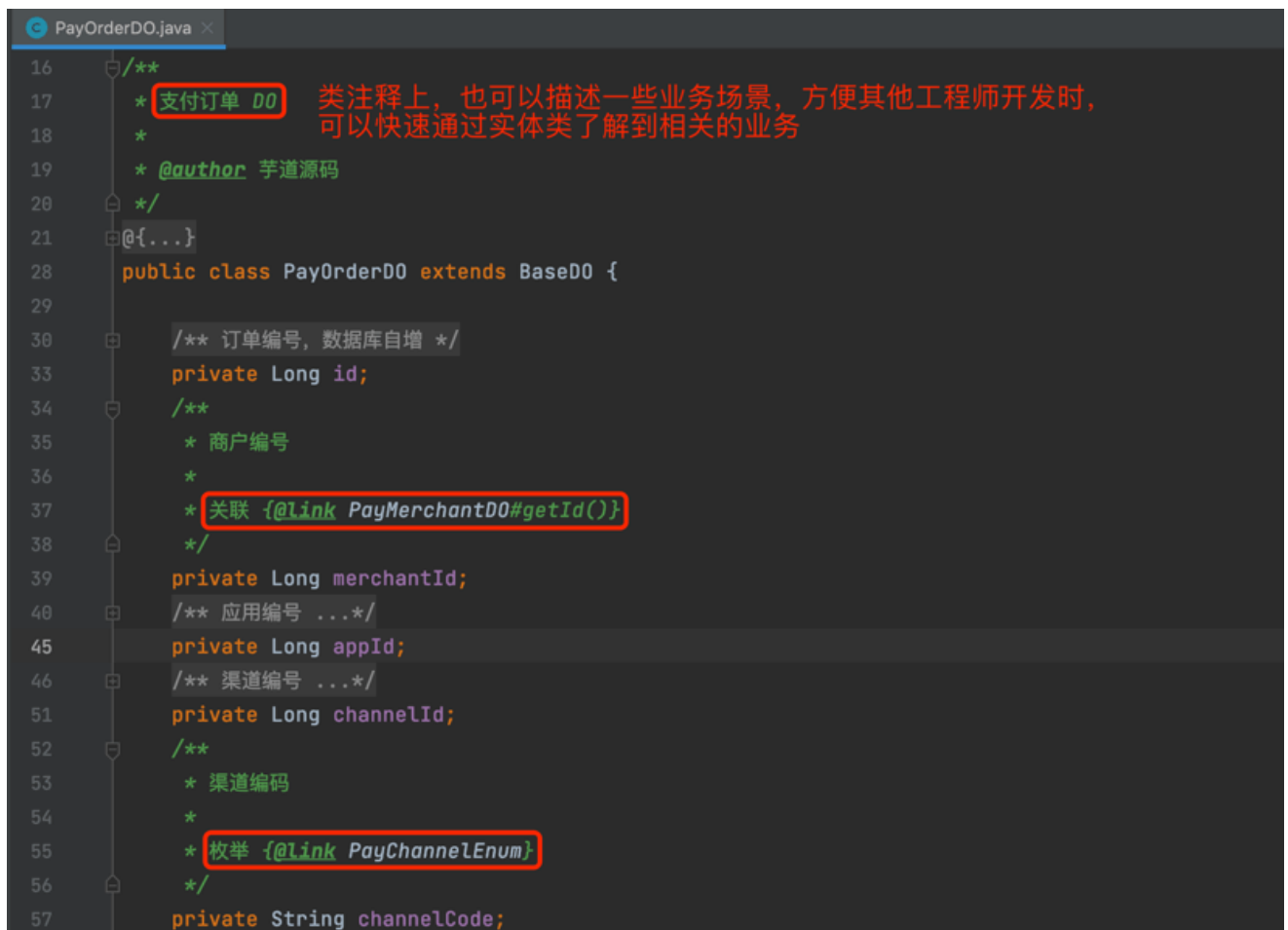
使用 TypeHandler 时，需要设置实体的 `@TableName` 注解的 `@autoResultMap = true`。

## 2. 编码规范

① 数据库实体类放在 `dal.dataobject` 包下，以 DO 结尾；数据库访问类放在 `dal.mysql` 包下，以 Mapper 结尾。如下图所示：

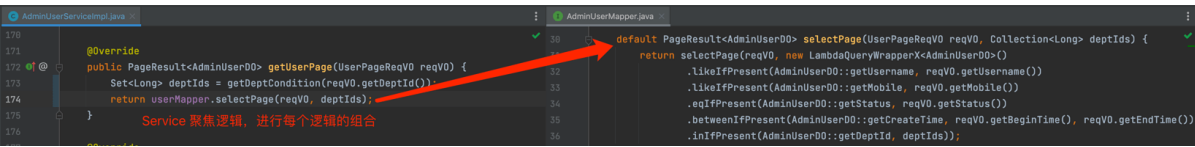


② 数据库实体类的注释要完整，特别是哪些字段是关联（外键）、枚举、冗余等等。例如说：





③ 禁止在 Controller、Service 中，直接进行 MyBatis Plus 操作。原因是：大量 MyBatis 操作散落在 Service 中，会导致 Service 的代码越来越乱，无法聚焦业务逻辑。

	示例
错误	
正确	

并且，通过只允许将 MyBatis Plus 操作编写 Mapper 层，更好的实现 SELECT 查询的复用，而不是 Service 会存在很多相同且重复的 SELECT 查询的逻辑。

④ Mapper 的 SELECT 查询方法的命名，采用 Spring Data 的 "Query methods" 策略，方法名使用 selectBy查询条件 规则。例如说：

```
@Mapper
public interface AdminUserMapper extends BaseMapperX<AdminUserDO> {

    default AdminUserDO selectByUsername(String username) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getUsername, username));
    }

    default AdminUserDO selectByEmail(String email) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getEmail, email));
    }

    default AdminUserDO selectByMobile(String mobile) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getMobile, mobile));
    }
}
```

⑤ 优先使用 LambdaQueryWrapper 条件构造器，使用方法获得字段名，避免手写 "字段" 可能写错的情况。例如说：

```
@Mapper
public interface AdminUserMapper extends BaseMapperX<AdminUserDO> {

    default AdminUserDO selectByUsername(String username) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getUsername, username));
    }

    default AdminUserDO selectByEmail(String email) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getEmail, email));
    }

    default AdminUserDO selectByMobile(String mobile) {
        return selectOne(new LambdaQueryWrapper<AdminUserDO>().eq(AdminUserDO::getMobile, mobile));
    }
}
```

⑥ 简单的单表查询，优先在 Mapper 中通过 default 方法实现。例如说：

```
@Mapper
public interface DeptMapper extends BaseMapperX<DeptD0> {

    default List<DeptD0> selectList(DeptListReqV0 reqV0) {
        return selectList(new LambdaQueryWrapperX<DeptD0>()
            .likeIfPresent(DeptD0::getName, reqV0.getName())
            .eqIfPresent(DeptD0::getStatus, reqV0.getStatus()));
    }

    default DeptD0 selectByParentIdAndName(Long parentId, String name) {
        return selectOne(new LambdaQueryWrapper<DeptD0>()
            .eq(DeptD0::getParentId, parentId)
            .eq(DeptD0::getName, name));
    }
}
```

### 3. CRUD 接口

[BaseMapperX](#) 接口，继承 MyBatis Plus 的 BaseMapper 接口，提供更强的 CRUD 操作能力。

#### 3.1 selectOne

[#selectOne\(...\)](#) 方法，使用指定条件，查询单条记录。示例如下：

```
@Mapper
public interface TenantMapper extends BaseMapperX<TenantD0> {

    default TenantD0 selectByName(String name) {
        return selectOne(TenantD0::getName, name);
    }
}
```

#### 3.2 selectCount

[#selectCount\(...\)](#) 方法，使用指定条件，查询记录的数量。示例如下：

```
@Mapper
public interface DeptMapper extends BaseMapperX<DeptD0> {

    default Long selectCountByParentId(Long parentId) {
        return selectCount(DeptD0::getParentId, parentId);
    }
}
```

#### 3.3 selectList

[#selectList\(...\)](#) 方法，使用指定条件，查询多条记录。示例如下：



```

@Mapper
public interface DeptMapper extends BaseMapperX<DeptDO> {

    default List<DeptDO> selectList(DeptListReqVO reqVO) {
        return selectList(new LambdaQueryWrapperX<DeptDO>()
            .likeIfPresent(DeptDO::getName, reqVO.getName())
            .eqIfPresent(DeptDO::getStatus, reqVO.getStatus()));
    }

    default DeptDO selectByParentIdAndName(Long parentId, String name) {
        return selectOne(new LambdaQueryWrapper<DeptDO>()
            .eq(DeptDO::getParentId, parentId)
            .eq(DeptDO::getName, name));
    }
}

```

### 3.4 selectPage

针对 MyBatis Plus 分页查询的二次封装，在 [BaseMapperX](#) 中实现，目的是使用项目自己的分页封装：

- 【入参】查询前，将项目的分页参数 [PageParam](#)，转换成 MyBatis Plus 的 [IPage](#) 对象。
- 【出参】查询后，将 MyBatis Plus 的分页结果 [IPage](#)，转换成项目的分页结果 [PageResult](#)。代码如下图：

```

public interface BaseMapperX<T> extends BaseMapper<T> {

    default PageResult<T> selectPage(PageParam pageParam, @Param("ew") Wrapper<T> queryWrapper) {
        // MyBatis Plus 查询
        IPage<T> mpPage = MyBatisUtils.buildPage(pageParam); ① 将 PageParam 中的 pageNo、pageSize 拼接为查询条件
        selectPage(mpPage, queryWrapper); ② 执行 select 分页查询，以及 select count(*) 数量查询
        // 将 MyBatis Plus 的 Page 结果，转换成自己的 PageResult 结果
        return new PageResult<>(mpPage.getRecords(), mpPage.getTotal()); ③ 转换分页的结果为项目的 PageResult
    }
}

```

具体的使用示例，可见 [TenantMapper](#) 类中，定义 selectPage 查询方法。代码如下：

```

@Mapper
public interface TenantMapper extends BaseMapperX<TenantDO> {

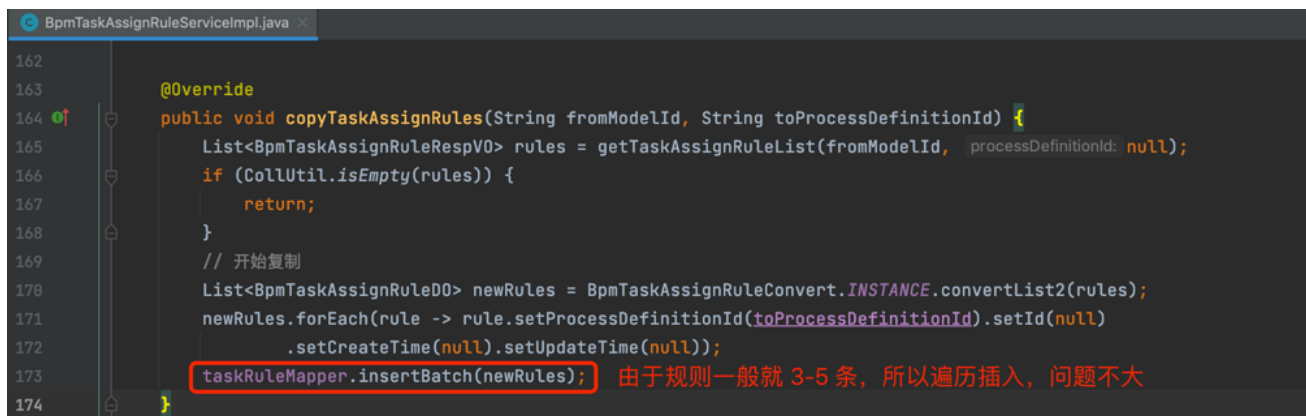
    default PageResult<TenantDO> selectPage(TenantPageReqVO reqVO) {
        return selectPage(reqVO, new LambdaQueryWrapperX<TenantDO>()
            .likeIfPresent(TenantDO::getName, reqVO.getName()) // 如果 name
            .likeIfPresent(TenantDO::getContactName, reqVO.getContactName())
            .likeIfPresent(TenantDO::getContactMobile, reqVO.getContactMobile)
            .eqIfPresent(TenantDO::getStatus, reqVO.getStatus()) // 如果 sta
            .betweenIfPresent(TenantDO::getCreateTime, reqVO.getBeginCreateTime(), reqVO.getEndCreateTime())
            .orderByDesc(TenantDO::getId)); // 按照 id 倒序
    }
}

```

完整实战，可见《[开发指南 —— 分页实现](#)》文档。

### 3.5 insertBatch

`#insertBatch(...)` 方法，遍历数组，逐条插入数据库中，适合少量数据插入，或者对性能要求不高的场景。示例如下：

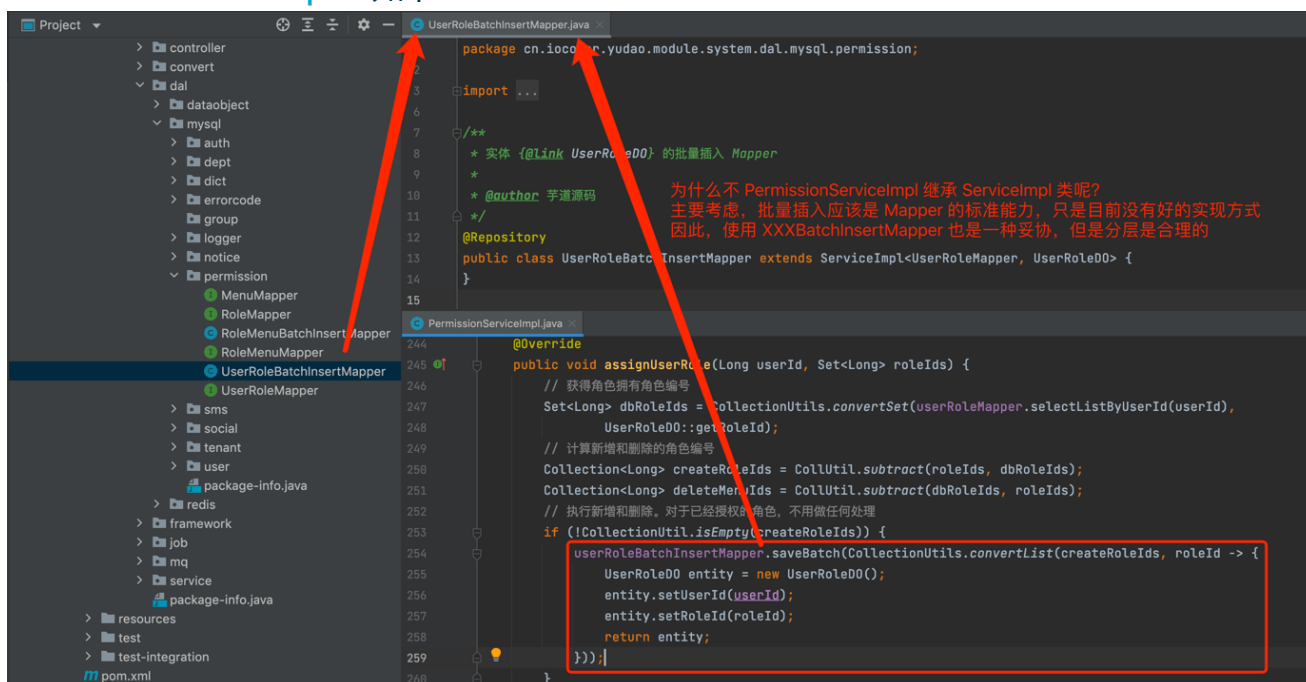


#### 为什么不使用 insertBatchSomeColumn 批量插入？

- 只支持 MySQL 数据库。其它 Oracle 等数据库使用会报错，可见 [InsertBatchSomeColumn](#) 说明。
- 未支持多租户。插入数据库时，多租户字段不会进行自动赋值。

## 4. 批量插入

绝大多数场景下，推荐使用 MyBatis Plus 提供的 IService 的 `#saveBatch()` 方法。示例 [PermissionServiceImpl](#) 如下：



## 5. 条件构造器

继承 MyBatis Plus 的条件构造器，拓展了 [LambdaQueryWrapperX](#) 和 [QueryWrapperX](#) 类，主要是增加 `xxxIfPresent` 方法，用于判断值不存在的时候，不要拼接到条件中。例如说：

```
public class LambdaQueryWrapperX<T> extends LambdaQueryWrapper<T> {  
  
    public LambdaQueryWrapperX<T> likeIfPresent(SFunction<T, ?> column, String val) {  
        if (StringUtils.hasText(val)) {  
            return (LambdaQueryWrapperX<T>) super.like(column, val);  
        }  
        return this;  
    }  
  
    public LambdaQueryWrapperX<T> inIfPresent(SFunction<T, ?> column, Collection<?> values) {  
        if (!CollectionUtils.isEmpty(values)) {  
            return (LambdaQueryWrapperX<T>) super.in(column, values);  
        }  
        return this;  
    }  
}
```

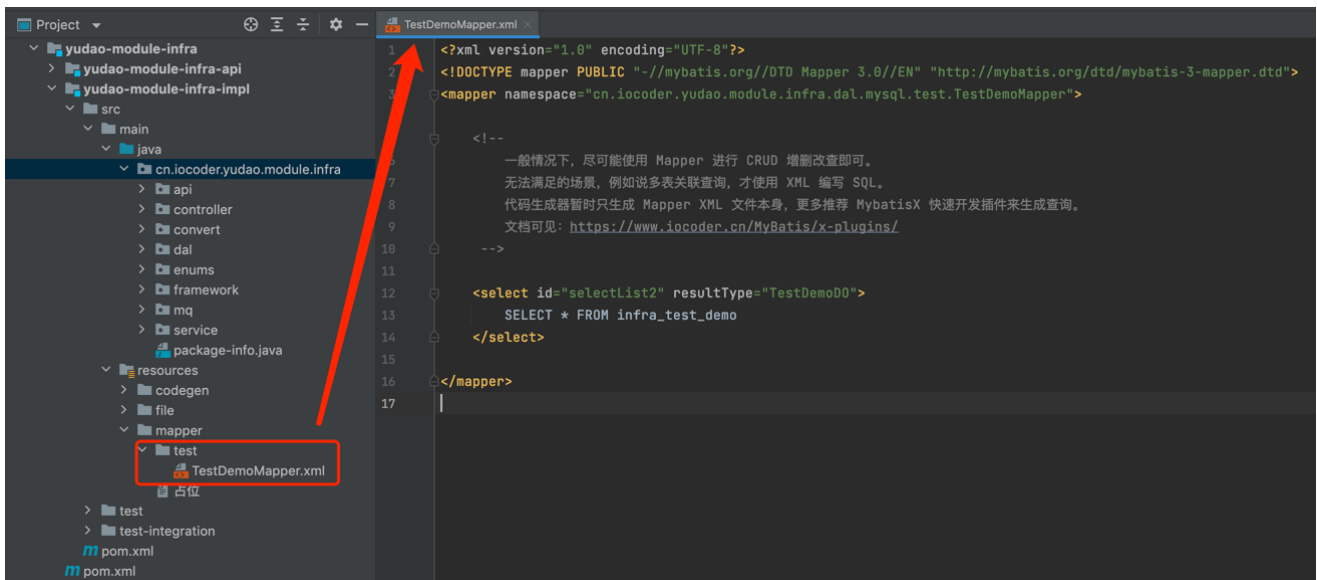
具体的使用示例如下：

```
@Mapper  
public interface AdminUserMapper extends BaseMapperX<AdminUserDO> {  
  
    default PageResult<AdminUserDO> selectPage(UserPageReqVO reqVO, Collection<Long> deptIds) {  
        return selectPage(reqVO, new LambdaQueryWrapperX<AdminUserDO>()  
            .likeIfPresent(AdminUserDO::getUsername, reqVO.getUsername())  
            .likeIfPresent(AdminUserDO::getMobile, reqVO.getMobile())  
            .eqIfPresent(AdminUserDO::getStatus, reqVO.getStatus())  
            .betweenIfPresent(AdminUserDO::getCreateTime, reqVO.getBeginTime(), reqVO.getEndTime())  
            .inIfPresent(AdminUserDO::getDeptId, deptIds));  
    }  
}
```

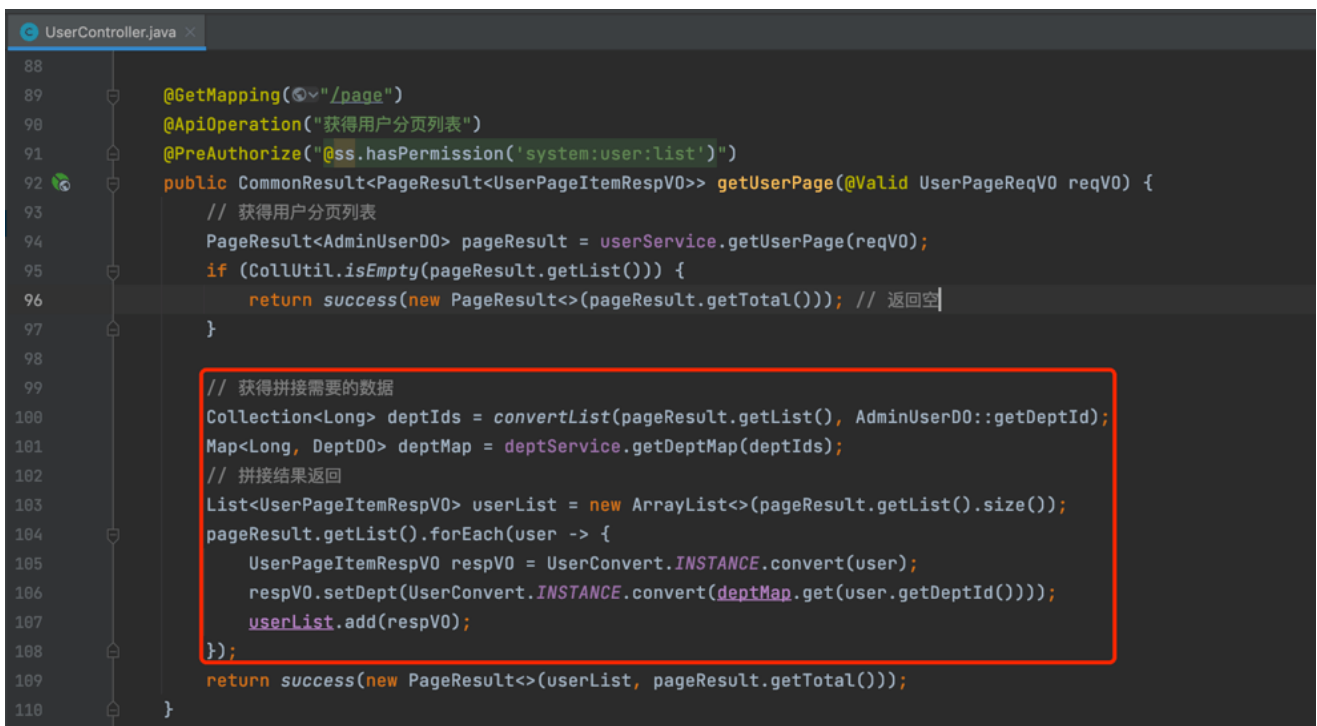
更加简洁

## 6. Mapper XML

默认配置下，MyBatis Mapper XML 需要写在各 `yudao-module-xxx-biz` 模块的 `resources/mapper` 目录下。示例 [TestDemoMapper.xml](#) 如下：



尽量避免数据库的连表（多表）查询，而是采用多次查询，Java 内存拼接的方式替代。例如说：



## 7. 字段加密

`EncryptTypeHandler` [🔗](#)，基于 `Hutool AES` [🔗](#) 实现字段的解密与解密。

例如说，`DataSourceConfig` [🔗](#) 的 `password` 密码需要实现加密存储，则只需要在该字段上添加 `EncryptTypeHandler` 处理器。示例代码如下：

```
@TableName(value = "infra_data_source_config", autoResultMap = true) // ① 添加 a
public class DataSourceConfigDO extends BaseDO {

    // ... 省略其它字段
    /**
     * 密码
     */
}
```

```
    */  
    @TableField(typeHandler = EncryptTypeHandler.class) // ② 添加 EncryptTypeHandler  
    private String password;  
  
}
```

另外，在 `application.yaml` 配置文件中，可使用 `mybatis-plus.encryptor.password` 设置加密密钥。

字段加密后，只允许使用**精准**匹配，无法使用模糊匹配。示例代码如下：

```
@Test // 测试使用 password 查询，可以查询到数据  
public void testSelectPassword() {  
    // mock 数据  
    DataSourceConfigDO dbDataSourceConfig = randomPojo(DataSourceConfigDO.class)  
    dataSourceConfigMapper.insert(dbDataSourceConfig); // @Sql: 先插入出一条存在的数  
  
    // 调用  
    DataSourceConfigDO result = dataSourceConfigMapper.selectOne(DataSourceConfigDO.class, new QueryWrapper<>()  
        .eq("password", EncryptTypeHandler.encrypt(dbDataSourceConfig.getPassword()))); // 重  
}
```

---

[← 系统日志](#)

[MyBatis 联表&分页查询→](#)



Theme by [Vdoing](#) | Copyright © 2019-2023 芋道源码 | MIT License