# **【MyBatis-Plus 核心接口】BaseMapper 和 IService 深度解析**

在使用 MyBatis-Plus（简称 MP）进行开发时，BaseMapper 和 IService 接口是我们老朋友了，不知道你会不会跟我一样好奇：为什么实现了 BaseMapper 或 IService 接口，我们就能轻松操作数据库？这背后有哪些工作机制？本文将带你一步步探究，并结合 CRUD 操作分类讲解两者的常用方法

# 一、BaseMapper 核心功能

BaseMapper 是 MyBatis-Plus 提供的一个基础接口，用于封装最常见的 CRUD 操作。只要你的实体类对应了这个 Mapper 接口，就能使用其提供的方法与数据库交互。

**核心原理**

MP 内置 SQL 解析器：BaseMapper 的方法背后是 MP 内置的动态 SQL 解析器，它根据泛型参数（实体类）自动生成 SQL 语句。

无需重复编写：常见的增删改查方法已经内置，实现了开发效率的提升。

**常用方法分类**

BaseMapper 提供的方法可以分为以下几类：

**新增操作**

插入一条记录，忽略非空字段。

int insert(T entity)

**删除操作**

根据主键删除

int deleteById(Serializable id)

根据主键批量删除

int deleteBatchIds(Collection<? extends Serializable> idList)

根据字段条件删除

int deleteByMap(Map<String, Object> columnMap)

**更新操作**

根据主键更新

int updateById(@Param("et") T entity)

根据条件更新

int update(@Param("et") T entity, @Param("ew") Wrapper<T> updateWrapper)

**查询操作**

根据主键查询

T selectById(Serializable id)

根据主键批量查询

List<T> selectBatchIds(Collection<? extends Serializable> idList)

根据 Map 条件查询

List<T> selectByMap(Map<String, Object> columnMap)

根据条件查询列表

List<T> selectList(@Param("ew") Wrapper<T> queryWrapper)

分页查询

IPage<T> selectPage(IPage<T> page, @Param("ew") Wrapper<T> queryWrapper)

**【说明】**

T 是实体类泛型，代表数据库表的映射对象。

Serializable id 是实体类的主键类型。

Wrapper 是条件构造器，用于动态生成 SQL 条件。

IPage 是分页参数接口，结合分页插件实现分页查询。

**代码示例**

@Mapper

public interface UserMapper extends BaseMapper<User> {

// 无需手写基本CRUD方法

}

​

// 使用示例

User user = new User();

user.setName("Tom");

user.setAge(25);

userMapper.insert(user);

# 二、IService 核心功能

IService 是 MyBatis-Plus 提供的业务层接口，主要封装了对 BaseMapper 的进一步抽象，提供一些常用的业务逻辑方法。

**核心原理**

依赖 Service 层：IService 内部调用 BaseMapper 的方法，通过组合的方式封装业务逻辑。

增强业务扩展性：通过继承 IService，可以更容易添加自定义业务逻辑，同时不影响基础 CRUD 功能。

**常用 CRUD 方法一览**

IService 方法大部分是对 BaseMapper 方法的封装，并增加了一些便利操作：

**新增操作**

保存一条记录，忽略非空字段。

boolean save(T entity)

批量插入记录，提高插入效率。

boolean saveBatch(Collection<T> entityList)

**删除操作**

根据主键删除

boolean removeById(Serializable id)

根据 Map 条件删除

boolean removeByMap(Map<String, Object> columnMap)

根据主键批量删除

boolean removeBatchIds(Collection<? extends Serializable> idList)

**更新操作**

根据主键更新

boolean updateById(T entity)

根据条件更新

boolean update(T entity, Wrapper<T> updateWrapper)

批量更新记录，支持根据主键更新多个实体。

boolean updateBatchById(Collection<T> entityList)

**使用示例**

**UserService接口**

public interface UserService extends IService<User> {

}

**UserServiceImpl**

@Service

public class UserServiceImpl extends ServiceImpl<UserMapper, User>

implements UserService{

}

**保存操作示例**

User user = new User();

user.setName("Alice");

user.setAge(30);

userService.save(user);

**查询操作示例**

List<User> users = userService.listByIds(Arrays.asList(1, 2, 3));

users.forEach(System.out::println);

**分页查询示例**

IPage<User> page = new Page<>(1, 5); // 第1页，每页5条记录

QueryWrapper<User> queryWrapper = new QueryWrapper<>();

queryWrapper.gt("age", 18); // 年龄大于18

IPage<User> result = userService.page(page, queryWrapper);

result.getRecords().forEach(System.out::println);

# 三、操作数据库的工作机制

接下来我们深入讲解一下 MyBatis-Plus 是如何通过代码生成和动态代理机制实现数据库操作的

代码生成：MP 基于实体类生成 SQL 映射文件，通过 BaseMapper 或 IService 动态生成常用方法的实现。

动态代理：MP 使用了 MyBatis 提供的 MapperProxy 动态代理机制，当调用 BaseMapper 的方法时，会动态解析 SQL 并执行。

**1. 代码生成机制**

MyBatis-Plus 的代码生成器并不是简单的“生成代码文件”，它更像一个“动态 SQL 工厂”。通过结合实体类和数据库表的映射关系，它在运行时动态生成 SQL 并执行，具体如下：

**核心实现逻辑**

实体类与表映射： 实体类通过 @TableName 和字段注解（如 @TableField）指定表和字段的对应关系。没有显式注解时，MP 会根据约定（如类名与表名的驼峰转下划线规则）推导映射。

自动注入 SQL 操作方法： MP 根据 BaseMapper 的泛型实体类，结合数据库元信息（表名、字段名等），生成对应的 CRUD 方法所需的 SQL 模板。例如：

selectById 会生成类似如下的 SQL。

SELECT \* FROM user WHERE id = #{id}

insert 会根据实体类的字段生成

INSERT INTO user (name, age) VALUES (#{name}, #{age})

**动态生成的好处：**

开发者无需手写基础 SQL。

避免手写 SQL 错误，提升效率。

结合条件构造器（Wrapper），还能动态生成复杂 SQL。

**2. 动态代理机制**

MyBatis-Plus 使用 MyBatis 的 MapperProxy 动态代理机制，拦截 Mapper 接口的调用，将方法转化为 SQL 并执行。这是其实现核心数据库操作的基础。

**MapperProxy 的核心逻辑**

代理对象生成： 每一个 BaseMapper 实现类，实际上是由 MyBatis 动态代理生成的代理对象。在程序运行时，调用 BaseMapper 的方法并不会直接执行，而是由代理类的 invoke 方法处理。

**方法调用的拦截与解析：**

方法匹配：当调用 BaseMapper 方法时，MapperProxy 会根据方法签名（方法名、参数类型等）找到对应的 MappedStatement。

SQL 解析：MP 扩展了 MyBatis 的 MappedStatement，可以根据方法名（如 selectById）和注解信息动态解析成 SQL。

执行查询：SQL 解析完成后，通过 MyBatis 的 Executor 层（如 SimpleExecutor 或 BatchExecutor）将 SQL 提交到数据库并返回结果。

**与 MyBatis 的区别**

MyBatis：开发者需要手动在 XML 或注解中定义每个 SQL 语句，方法名和 SQL 没有直接关系。

MyBatis-Plus：通过方法名和泛型实体类自动生成 SQL，例如 selectById、insert 等方法会根据命名约定生成标准的 CRUD SQL，无需额外配置。

**3. 从调用到执行的全过程**

以下是一个 BaseMapper.selectById 方法的调用到执行的完整流程，帮助更好理解动态代理和代码生成的协同作用：

调用阶段： 开发者调用 userMapper.selectById(1)，此时 userMapper 是 MyBatis 的代理对象。

方法拦截： 代理对象的 invoke 方法被触发。它根据方法签名找到对应的 MappedStatement，selectById 对应的 SQL 模板为：

SELECT \* FROM {tableName} WHERE id = ?

SQL 生成：

MP 通过 条件构造器Wrapper和元数据（实体类中的元信息（字段名、类型、注解等）），将泛型 User 解析为表名 user。

将实体类字段（id、name 等）解析为表字段（id、name 等）。

**动态拼接 SQL：**

SELECT \* FROM user WHERE id = 1

执行阶段：

MyBatis 的 Executor 层负责将拼接好的 SQL 提交给数据库。

数据库返回结果集，MyBatis 的 ResultHandler 负责将结果集映射为实体对象（User 实例）。

返回结果： 最终，selectById 返回的是一个封装了查询结果的 User 对象。

**4. 设计优势**

MyBatis-Plus 的动态代理和代码生成机制在设计上有以下优势：

开发效率：通过自动生成 SQL 和动态代理调用，开发者不需要关心基础 SQL 的编写，大幅减少重复工作。

安全性：动态生成 SQL 时，会自动处理 SQL 注入风险（如参数使用 PreparedStatement 的形式绑定）。

扩展性：通过 Wrapper 构造器，支持复杂 SQL 动态生成；同时允许开发者自定义方法扩展。

统一性：提供了一套一致的 CRUD 方法名和调用规范，降低了团队协作成本

# 四、总结

【BaseMapper】

封装了最基础的 CRUD 操作，例如 insert、selectById、updateById、deleteById 等，直接操作数据库，适合处理简单的增删改查逻辑。

提供了一种零配置的开发体验，让开发者无需编写重复的 SQL 语句。

【IService】

对 BaseMapper 进行了进一步封装，包含了一些常用的业务逻辑扩展方法，例如批量插入、分页查询等。

通过继承 IService，开发者可以更方便地在 Service 层添加自定义的业务逻辑，同时利用已有的 CRUD 功能。