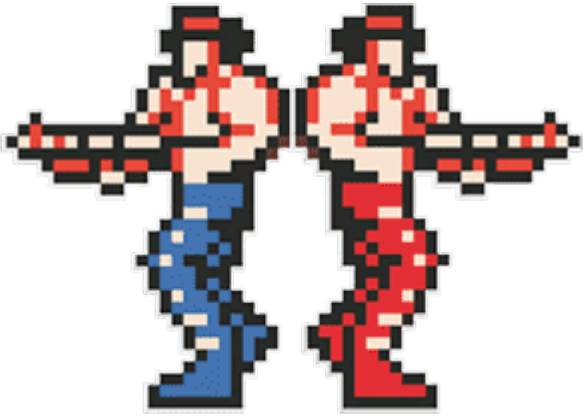
MybatisPlus

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

MybatisPlus

MyBatis LIKE



TO BE THE BEST PARTNER OF MYBATIS

目录

Contents

- 快速入门
- 核心功能
- 扩展功能
- 插件功能

快速入门

目录

Contents

- 入门案例
  - ✓ 学会MP的基本用法
  - ✓ 体会MP的无侵入和方便快捷的特点
- 常见注解
- 常见配置

案例

入门案例

需求：基于课前资料提供的项目，实现下列功能：

① 新增用户功能

② 根据id查询用户

③ 根据id批量查询用户

④ 根据id更新用户

⑤ 根据id删除用户

步骤

1. 引入MybatisPlus的起步依赖

MyBatisPlus官方提供了starter，其中集成了Mybatis和MybatisPlus的所有功能，并且实现了自动装配效果。

因此我们可以用MybatisPlus的starter代替Mybatis的starter：

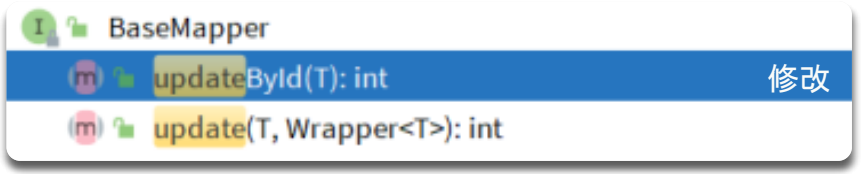
```
<!--MybatisPlus-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3.1</version>
</dependency>
```

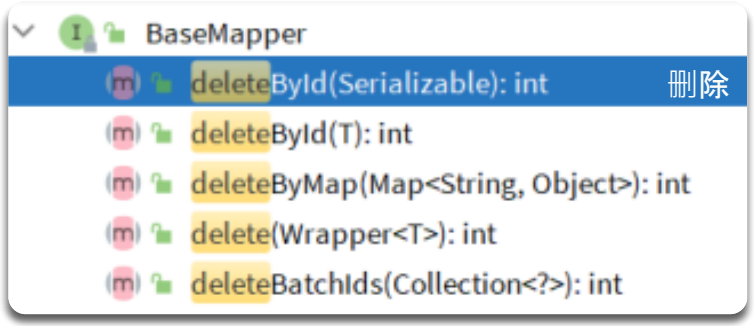
步骤

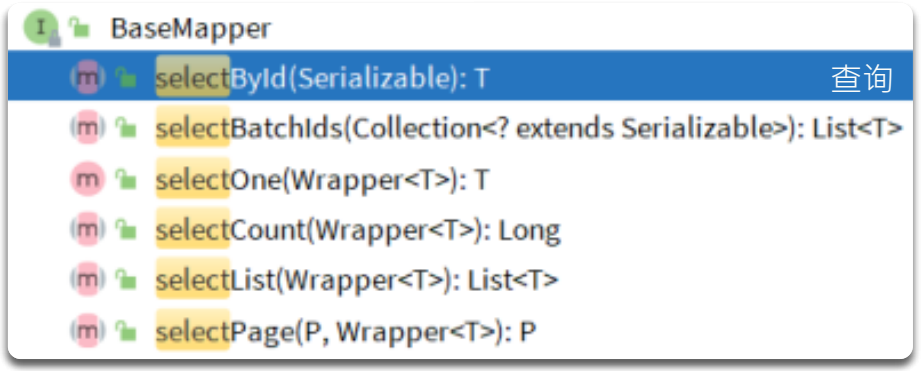
2. 定义Mapper

自定义的Mapper继承MybatisPlus提供的BaseMapper接口：

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

}
```







使用MybatisPlus的基本步骤：

① 引入MybatisPlus依赖，代替Mybatis依赖

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3.1</version>
</dependency>
```

② 定义Mapper接口并继承BaseMapper

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

}
```

目录

Contents

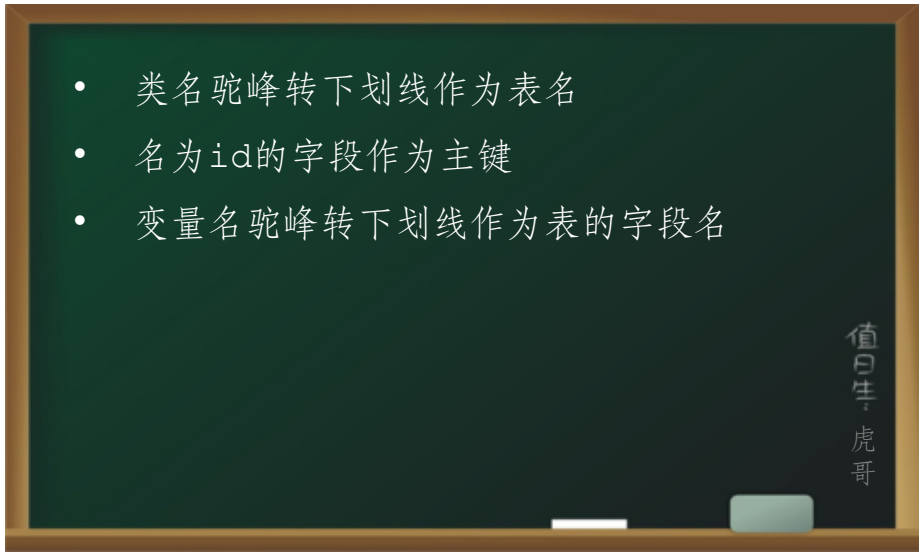
- 入门案例
- 常见注解
- 常见配置

常见注解

MyBatisPlus通过扫描实体类，并基于反射获取实体类信息作为数据库表信息。

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

}
```

```java
@Data
public class User {
    private Long id;
    private String username;
    private String password;
    private String phone;
    private String info;
    private Integer status;
    private Integer balance;
    private LocalDateTime createTime;
    private LocalDateTime updateTime;
}
```



常见注解

MybatisPlus中比较常用的几个注解如下：

- **@TableName**：用来指定表名
- **@TableId**：用来指定表中的主键字段信息
- **@TableField**：用来指定表中的普通字段信息

```java
@TableName("tb_user")
public class User {
    @TableId(value="id", type= IdType.AUTO)
    private Long id;
    @TableField("username")
    private String name;
    @TableField("is_married")
    private Boolean isMarried;
    @TableField("`order`")
    private Integer order;
    @TableField(exist = false)
    private String address;
}
```

IdType枚举：

- AUTO：数据库自增长
- INPUT：通过set方法自行输入
- ASSIGN_ID：分配 ID，接口IdentifierGenerator的方法nextId来生成id，默认实现类为DefaultIdentifierGenerator雪花算法

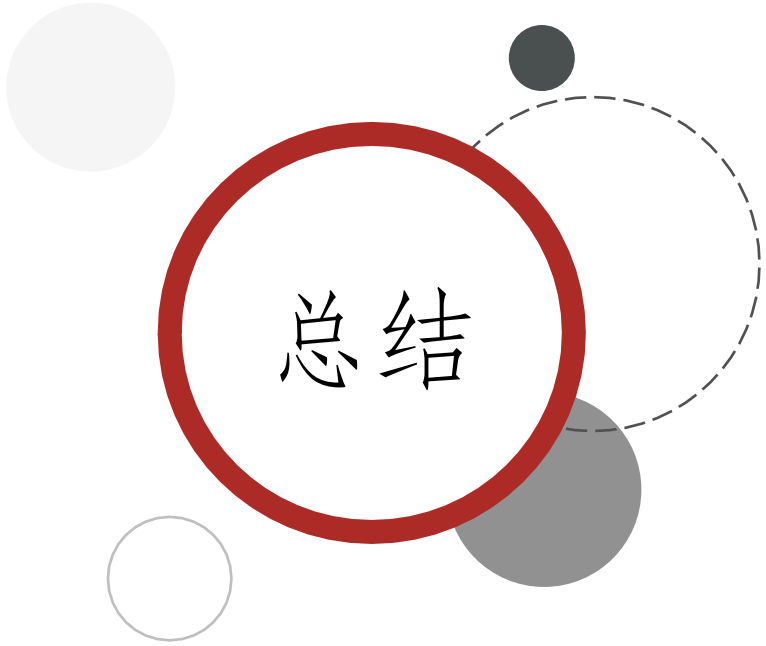
使用@TableField的常见场景：

- 成员变量名与数据库字段名不一致
- 成员变量名以is开头，且是布尔值
- 成员变量名与数据库关键字冲突
- 成员变量不是数据库字段

值日生：虎哥

名称：tb_user　　注释：用户表

#	名称	数据类型	注释	默认
1	id	BIGINT	用户id	AUTO_INCREMENT
2	username	VARCHAR	用户名	无默认值
3	is_married	BIT	密码	0
4	order	TINYINT	序号	NULL



MybatisPlus是如何获取实现CRUD的数据库表信息的?

- 默认以类名驼峰转下划线作为表名
- 默认把名为id的字段作为主键
- 默认把变量名驼峰转下划线作为表的字段名

MybatisPlus的常用注解有哪些?

- @TableName：指定表名称及全局配置
- @TableId：指定id字段及相关配置
- @TableField：指定普通字段及相关配置

IdType的常见类型有哪些?

- AUTO、ASSIGN_ID、INPUT

使用@TableField的常见场景是?

- 成员变量名与数据库字段名不一致
- 成员变量名以is开头，且是布尔值
- 成员变量名与数据库关键字冲突
- 成员变量不是数据库字段

目录

Contents

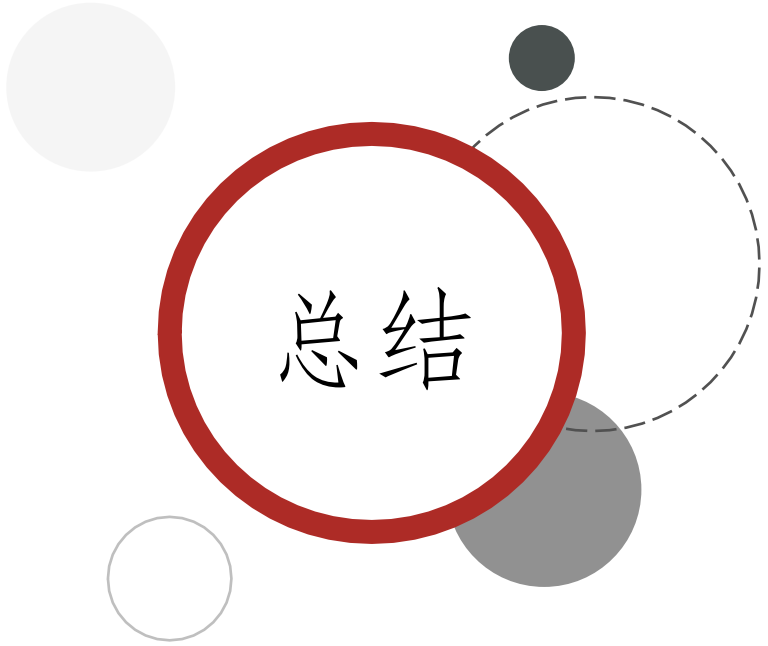
- 入门案例
- 常见注解
- 常见配置

常见配置

MyBatisPlus的配置项继承了MyBatis原生配置和一些自己特有的配置。例如：

```yaml
mybatis-plus:
  type-aliases-package: com.itheima.mp.domain.po # 别名扫描包
  mapper-locations: "classpath*:/mapper/**/*.xml" # Mapper.xml文件地址，默认值
  configuration:
    map-underscore-to-camel-case: true # 是否开启下划线和驼峰的映射
    cache-enabled: false # 是否开启二级缓存
  global-config:
    db-config:
      id-type: assign_id # id为雪花算法生成
      update-strategy: not_null # 更新策略：只更新非空字段
```

具体可参考官方文档：[使用配置 | MyBatis-Plus (baomidou.com)]



MyBatisPlus使用的基本流程是什么?

① 引入起步依赖

② 自定义Mapper基础BaseMapper

③ 在实体类上添加注解声明 表信息

④ 在application.yml中根据需要添加配置

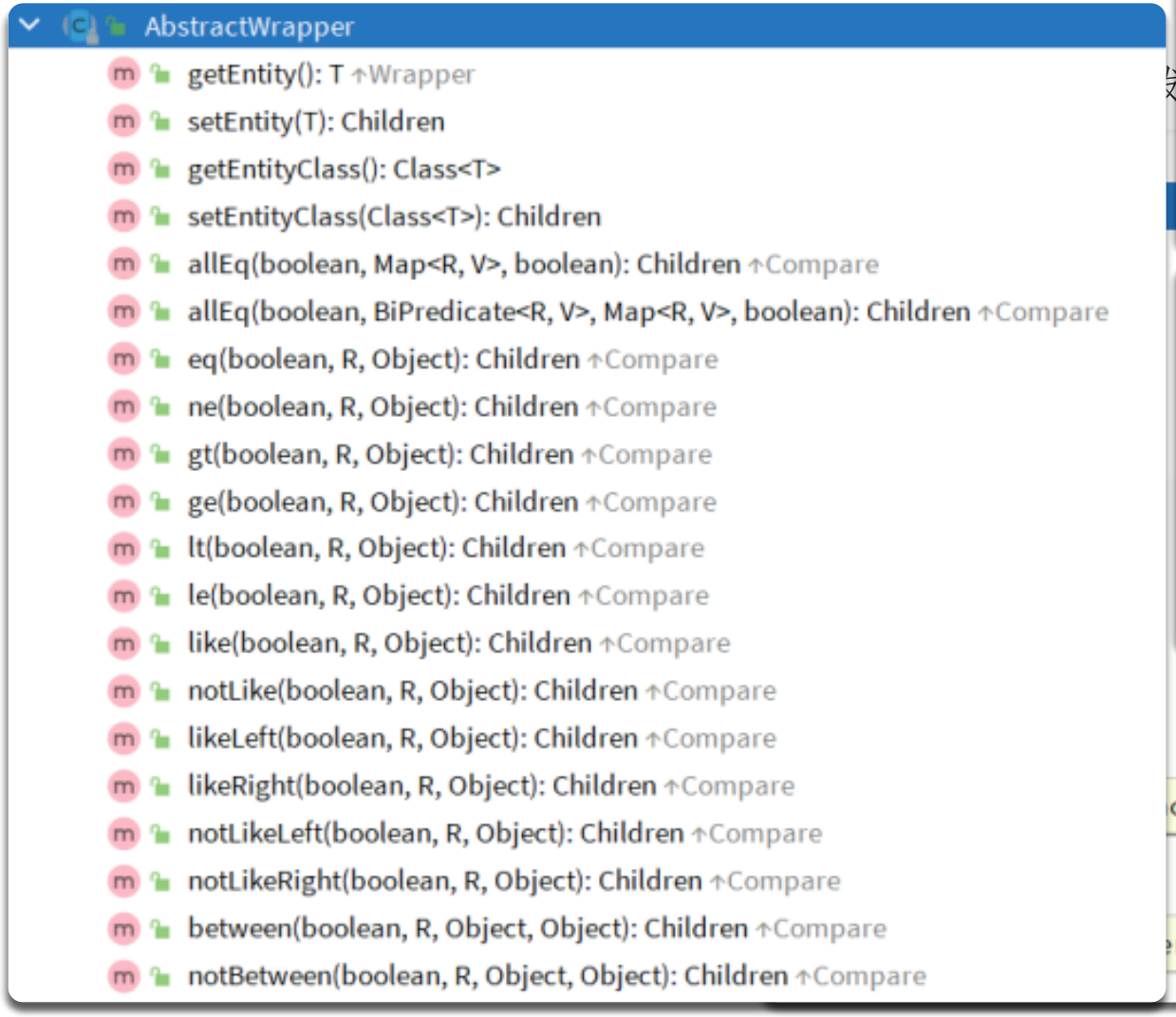
02 核心功能

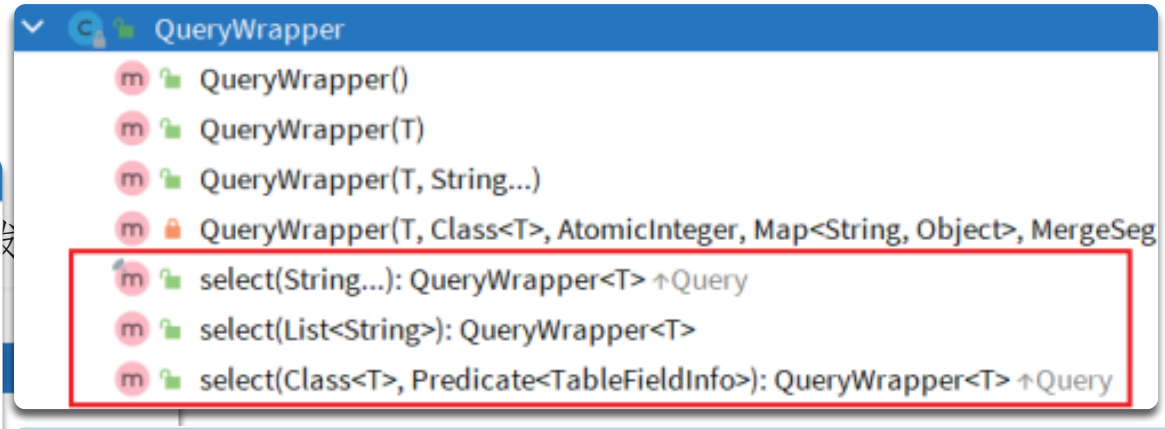
目录

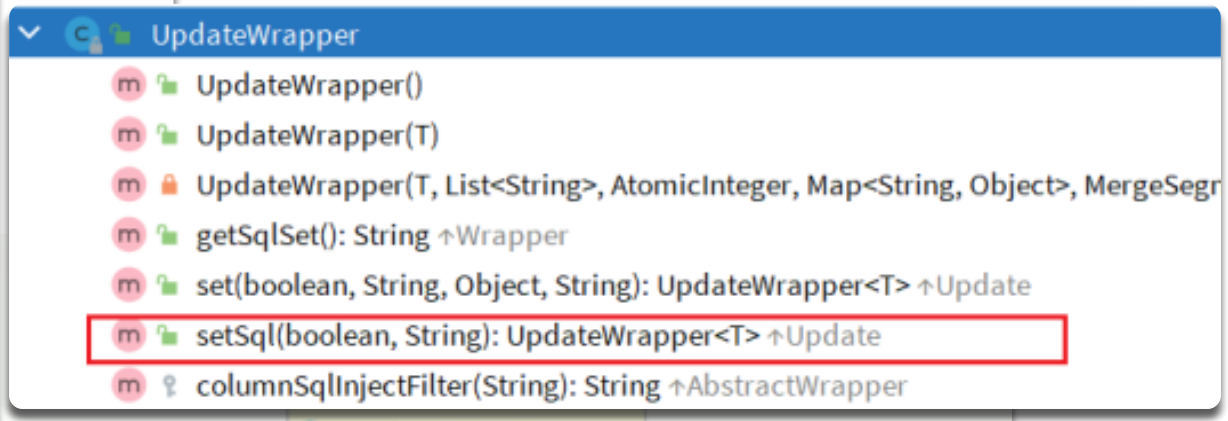
Contents

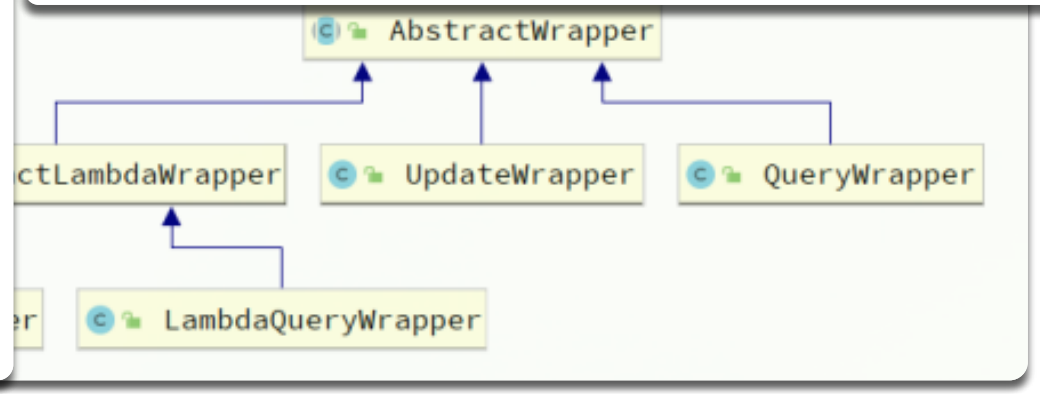
- 条件构造器
- 自定义SQL
- Service接口

条件构造器


AbstractWrapper
getEntity(): T ↑Wrapper
setEntity(T): Children
getEntityClass(): Class<T>
setEntityClass(Class<T>): Children
allEq(boolean, Map<R, V>, boolean): Children ↑Compare
allEq(boolean, BiPredicate<R, V>, Map<R, V>, boolean): Children ↑Compare
eq(boolean, R, Object): Children ↑Compare
ne(boolean, R, Object): Children ↑Compare
gt(boolean, R, Object): Children ↑Compare
ge(boolean, R, Object): Children ↑Compare
lt(boolean, R, Object): Children ↑Compare
le(boolean, R, Object): Children ↑Compare
like(boolean, R, Object): Children ↑Compare
notLike(boolean, R, Object): Children ↑Compare
likeLeft(boolean, R, Object): Children ↑Compare
likeRight(boolean, R, Object): Children ↑Compare
notLikeLeft(boolean, R, Object): Children ↑Compare
notLikeRight(boolean, R, Object): Children ↑Compare
between(boolean, R, Object, Object): Children ↑Compare
notBetween(boolean, R, Object, Object): Children ↑Compare


QueryWrapper
QueryWrapper()
QueryWrapper(T)
QueryWrapper(T, String...)
QueryWrapper(T, Class<T>, AtomicInteger, Map<String, Object>, MergeSeg
select(String...): QueryWrapper<T> ↑Query
select(List<String>): QueryWrapper<T>
select(Class<T>, Predicate<TableFieldInfo>): QueryWrapper<T> ↑Query


UpdateWrapper
UpdateWrapper()
UpdateWrapper(T)
UpdateWrapper(T, List<String>, AtomicInteger, Map<String, Object>, MergeSegr
getSqlSet(): String ↑Wrapper
set(boolean, String, Object, String): UpdateWrapper<T> ↑Update
setSql(boolean, String): UpdateWrapper<T> ↑Update
columnSqlInjectFilter(String): String ↑AbstractWrapper


AbstractWrapper
ctLambdaWrapper
UpdateWrapper
QueryWrapper
LambdaQueryWrapper

案例 基于QueryWrapper的查询

需求：

① 查询出名字中带o的，存款大于等于1000元的人的id、username、info、balance字段

```
SELECT id,username,info,balance
FROM user
WHERE username LIKE ? AND balance >= ?
```

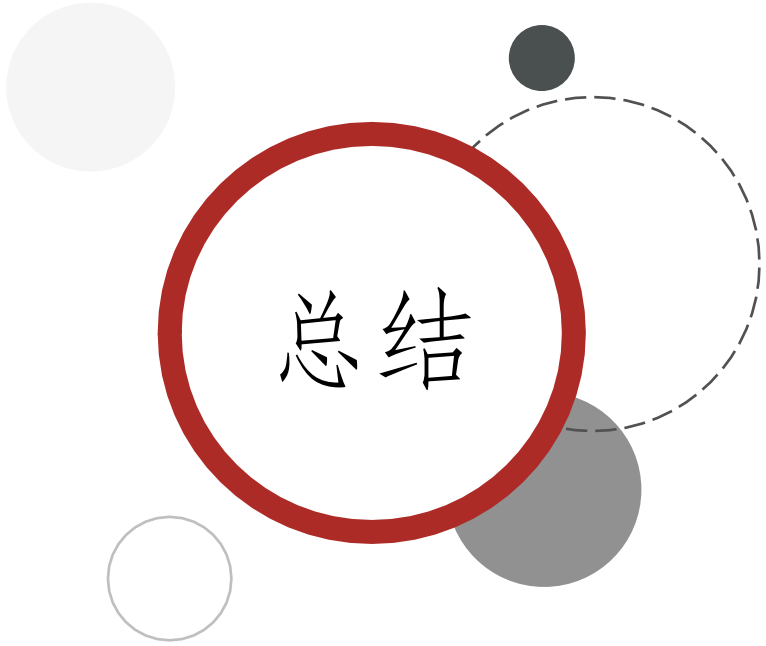
② 更新用户名为jack的用户的余额为2000

```
UPDATE user
    SET balance = 2000
    WHERE (username = "jack")
```

案例 基于UpdateWrapper的更新

需求：更新id为1,2,4的用户的余额，扣200

```sql
UPDATE user
    SET balance = balance - 200
    WHERE id in (1, 2, 4)
```



条件构造器的用法：

- QueryWrapper和LambdaQueryWrapper通常用来构建select、delete、update的where条件部分
- UpdateWrapper和LambdaUpdateWrapper通常只有在set语句比较特殊才使用
- 尽量使用LambdaQueryWrapper和LambdaUpdateWrapper，避免硬编码

目录

Contents

- 条件构造器
- 自定义SQL
- Service接口

自定义SQL

我们可以利用MyBatisPlus的Wrapper来构建复杂的Where条件，然后自己定义SQL语句中剩下的部分。

案例 自定义SQL

需求：将id在指定范围的用户（例如1、2、4 ）的余额扣减指定值

```
<update id="updateBalanceByIds">
    UPDATE user
    SET balance = balance - #{amount}
    WHERE id IN
    <foreach collection="ids" separator="," item="id" open="(" close=")">
        #{id}
    </foreach>
</update>
```

自定义

MP构建

```
SELECT status, COUNT(id) AS total
FROM tb_user
<where>
    <if test="name != null">AND username LIKE #{name}</if>
    <if test="ids != null">
        AND id IN
        <foreach collection="ids" open="(" close=")" item="id" separator=",">
            #{id}
        </foreach>
    </if>
</where>
GROUP BY status
```

自定义SQL

我们可以利用MyBatisPlus的Wrapper来构建复杂的Where条件，然后自己定义SQL语句中剩下的部分。

① 基于Wrapper构建where条件

```
List<Long> ids = List.of(1L, 2L, 4L);
int amount = 200;
// 1.构建条件
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>().in(User::getId, ids);
// 2.自定义SQL方法调用
userMapper.updateBalanceByIds(wrapper, amount);
```

② 在mapper方法参数中用Param注解声明wrapper变量名称，必须是ew

```
void updateBalanceByIds(@Param("ew") LambdaQueryWrapper<User> wrapper, @Param("amount") int amount);
```

③ 自定义SQL，并使用Wrapper条件

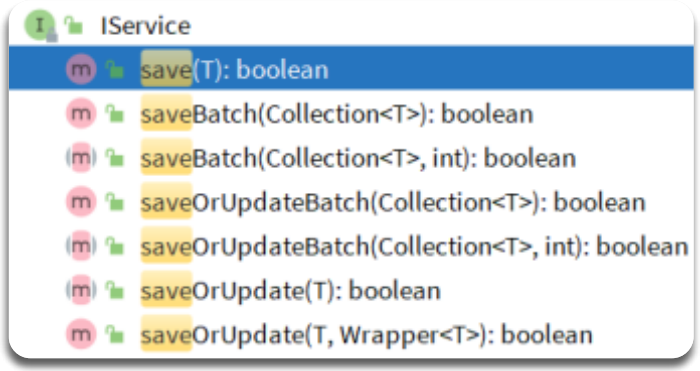
```
<update id="updateBalanceByIds">
    UPDATE tb_user SET balance = balance - #{amount} ${ew.customSqlSegment}
</update>
```

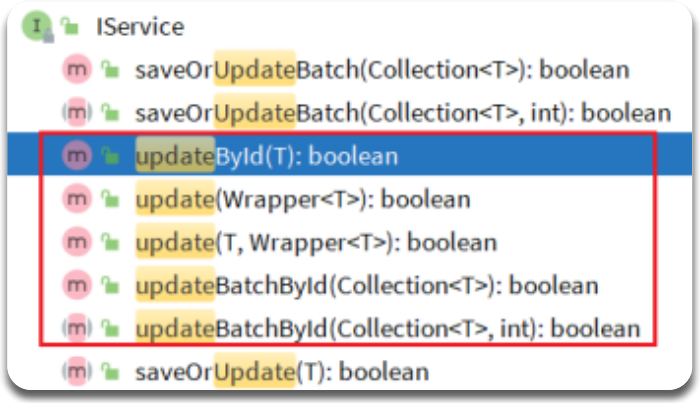
目录

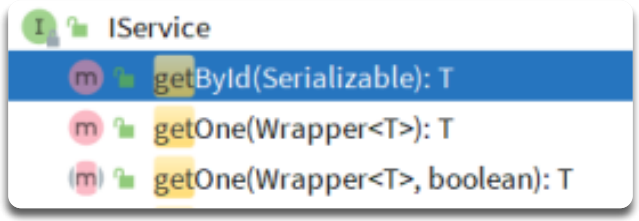
Contents

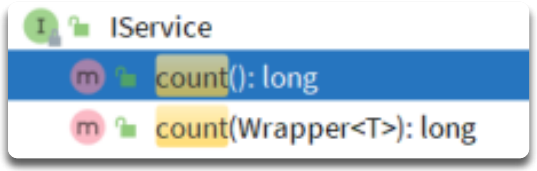
- 条件构造器
- 自定义SQL
- Service接口

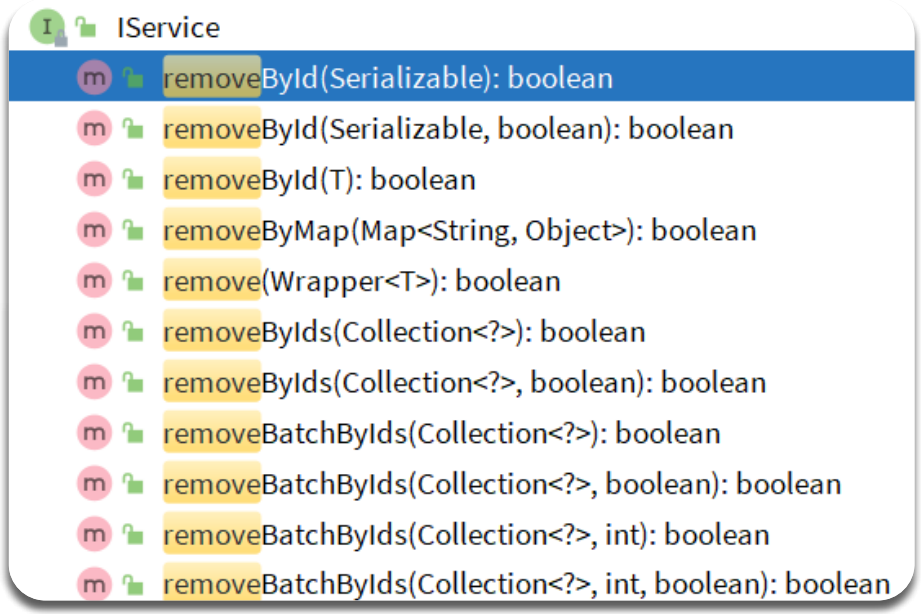
Service接口

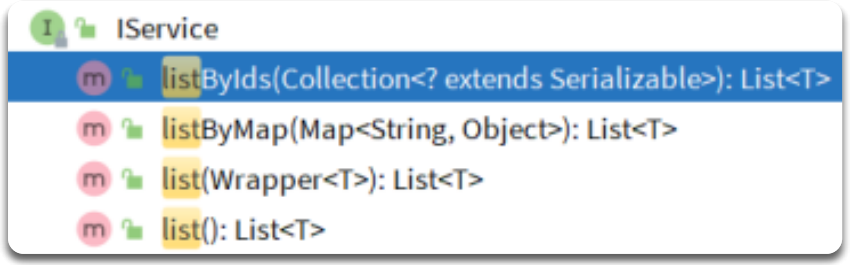

IService
save(T): boolean
saveBatch(Collection<T>): boolean
saveBatch(Collection<T>, int): boolean
saveOrUpdateBatch(Collection<T>): boolean
saveOrUpdateBatch(Collection<T>, int): boolean
saveOrUpdate(T): boolean
saveOrUpdate(T, Wrapper<T>): boolean

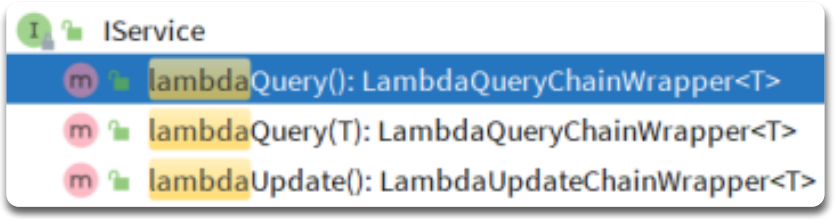

IService
saveOrUpdateBatch(Collection<T>): boolean
saveOrUpdateBatch(Collection<T>, int): boolean
updateById(T): boolean
update(Wrapper<T>): boolean
update(T, Wrapper<T>): boolean
updateBatchById(Collection<T>): boolean
updateBatchById(Collection<T>, int): boolean
saveOrUpdate(T): boolean

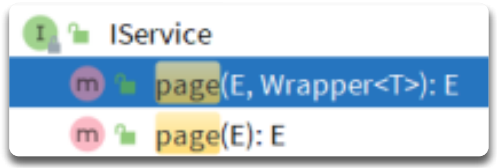

IService
getById(Serializable): T
getOne(Wrapper<T>): T
getOne(Wrapper<T>, boolean): T


IService
count(): long
count(Wrapper<T>): long

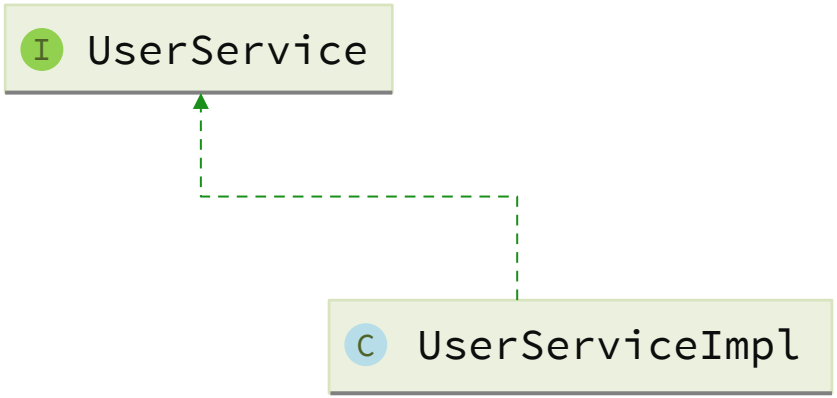

IService
removeById(Serializable): boolean
removeById(Serializable, boolean): boolean
removeById(T): boolean
removeByMap(Map<String, Object>): boolean
remove(Wrapper<T>): boolean
removeByIds(Collection<?>): boolean
removeByIds(Collection<?>, boolean): boolean
removeBatchByIds(Collection<?>): boolean
removeBatchByIds(Collection<?>, boolean): boolean
removeBatchByIds(Collection<?>, int): boolean
removeBatchByIds(Collection<?>, int, boolean): boolean


IService
listByIds(Collection<? extends Serializable>): List<T>
listByMap(Map<String, Object>): List<T>
list(Wrapper<T>): List<T>
list(): List<T>

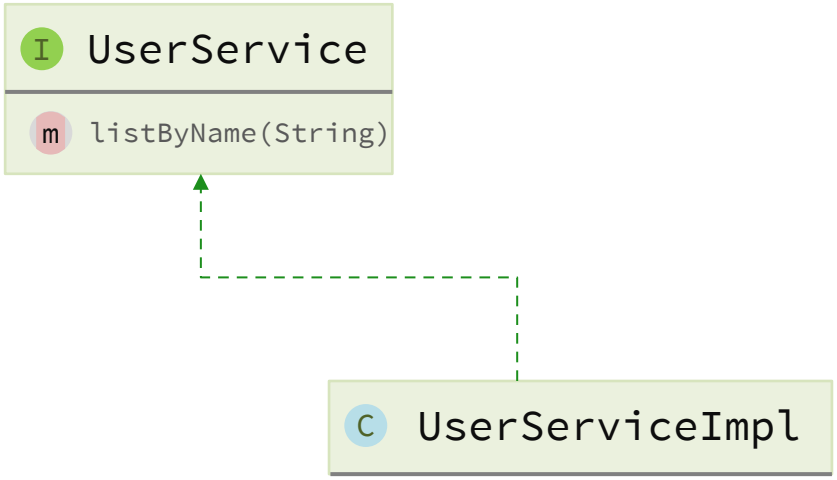

IService
lambdaQuery(): LambdaQueryChainWrapper<T>
lambdaQuery(T): LambdaQueryChainWrapper<T>
lambdaUpdate(): LambdaUpdateChainWrapper<T>


IService
page(E, Wrapper<T>): E
page(E): E

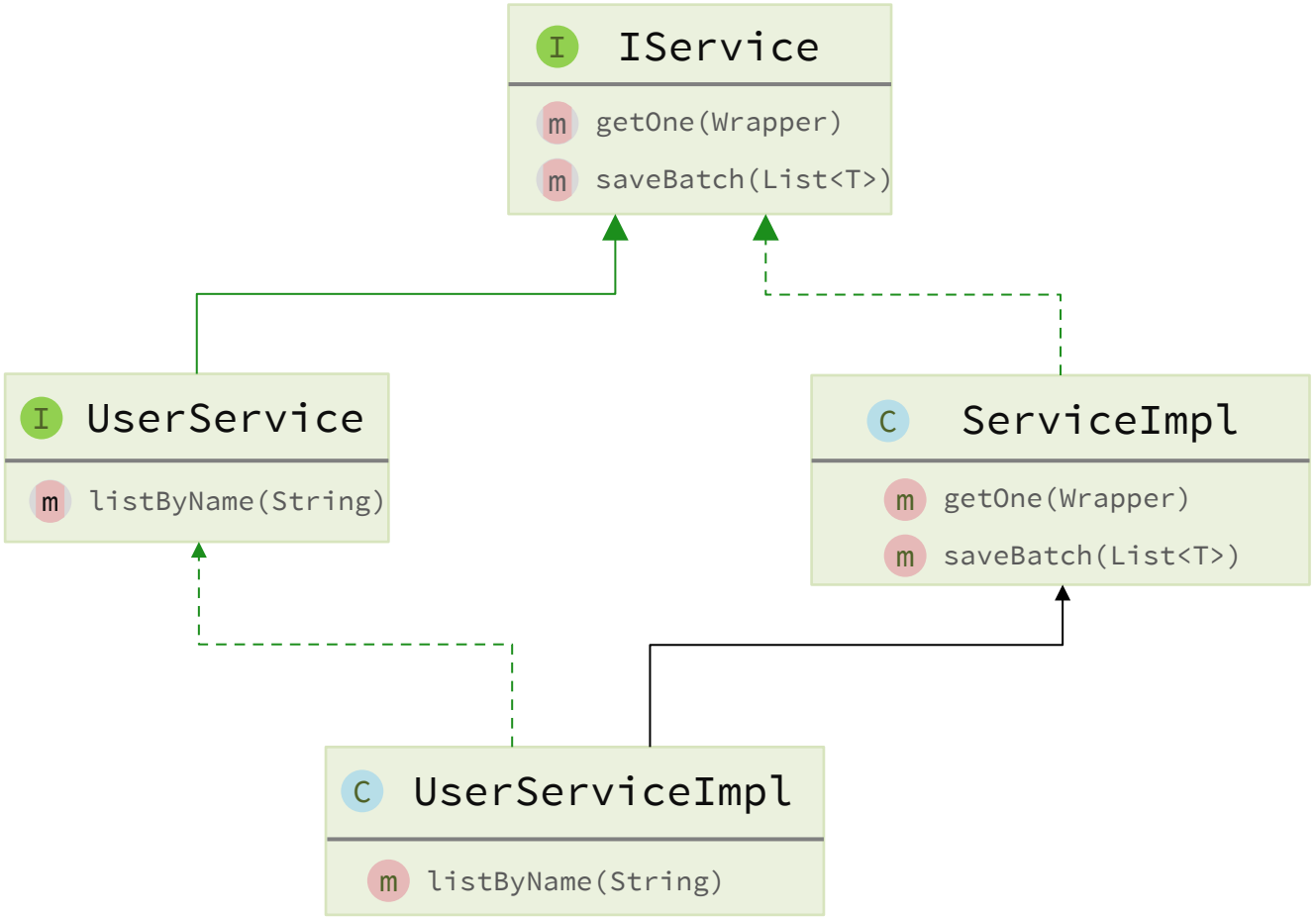
Service接口

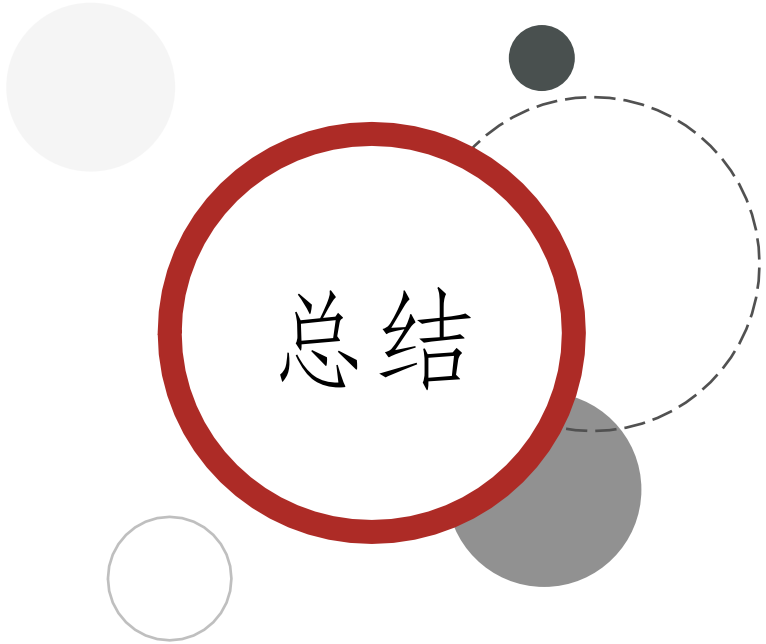


Service接口



Service接口


IService
getOne(Wrapper)
saveBatch(List<T>)
UserService
listByName(String)
ServiceImpl
getOne(Wrapper)
saveBatch(List<T>)
UserServiceImpl
listByName(String)



MP的Service接口使用流程是怎样的?

- 自定义Service接口继承IService接口

```
public interface IUserService
                    extends IService<User> {}
```

- 自定义Service实现类，实现自定义接口并继承

ServiceImpl类

```
public class UserServiceImpl
        extends ServiceImpl<UserMapper, User>
        implements IUserService {
}
```

案例

基于Restful风格实现下列接口

需求：基于Restful风格实现下面的接口：

编号	接口	请求方式	请求路径	请求参数	返回值
1	新增用户	POST	/users	用户表单实体	无
2	删除用户	DELETE	/users/{id}	用户id	无
3	根据id查询用户	GET	/users/{id}	用户id	用户VO
4	根据id批量查询	GET	/users	用户id集合	用户VO集合
5	根据id扣减余额	PUT	/users/{id}/deduction/{money}	• 用户id • 扣减金额	无

案例 IService的Lambda查询

需求：实现一个根据复杂条件查询用户的接口，查询条件如下：

- name：用户名关键字，可以为空
- status：用户状态，可
- minBalance：最小余
- maxBalance：最大余

```
<select id="queryUsers" resultType="com.itheima.mp.domain.po.User">
    SELECT *
    FROM tb_user
    <where>
        <if test="name != null">
            AND username LIKE #{name}
        </if>
        <if test="status != null">
            AND `status` = #{status}
        </if>
        <if test="minBalance != null and maxBalance != null">
            AND balance BETWEEN #{minBalance} AND #{maxBalance}
        </if>
    </where>
</select>
```

案例

IService的Lambda更新

需求：改造根据id修改用户余额的接口，要求如下

① 完成对用户状态校验

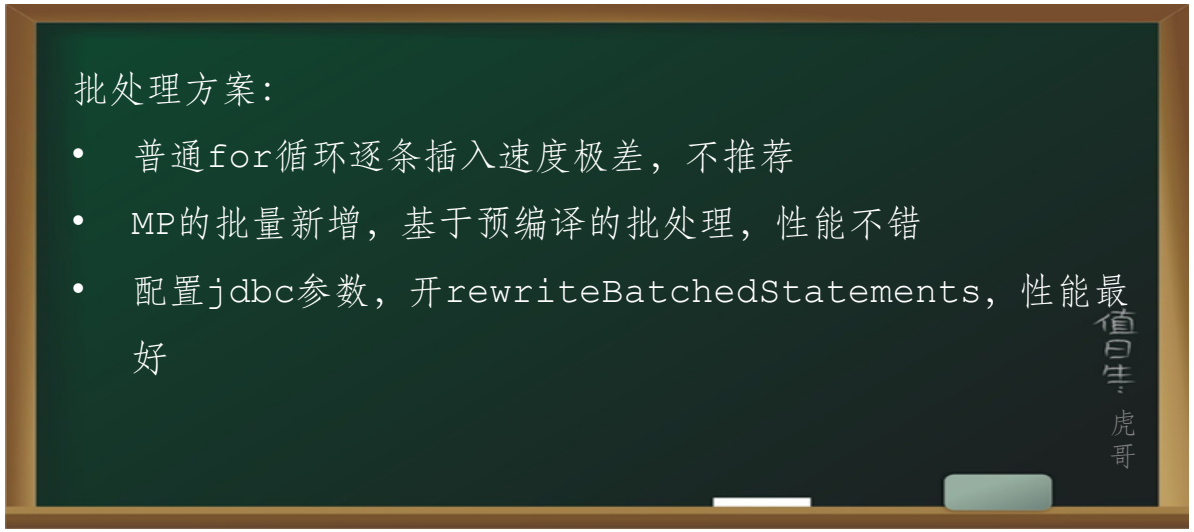
② 完成对用户余额校验

③ 如果扣减后余额为0，则将用户status修改为冻结状态（2）

案例 IService批量新增

需求：批量插入10万条用户数据，并作出对比：

- 普通for循环插入
- IService的批量插入
- 开启rewriteBatchedStatements=true参数



扩展功能

目录

Contents

- 代码生成
- 静态工具
- 逻辑删除
- 枚举处理器
- JSON处理器

代码生成

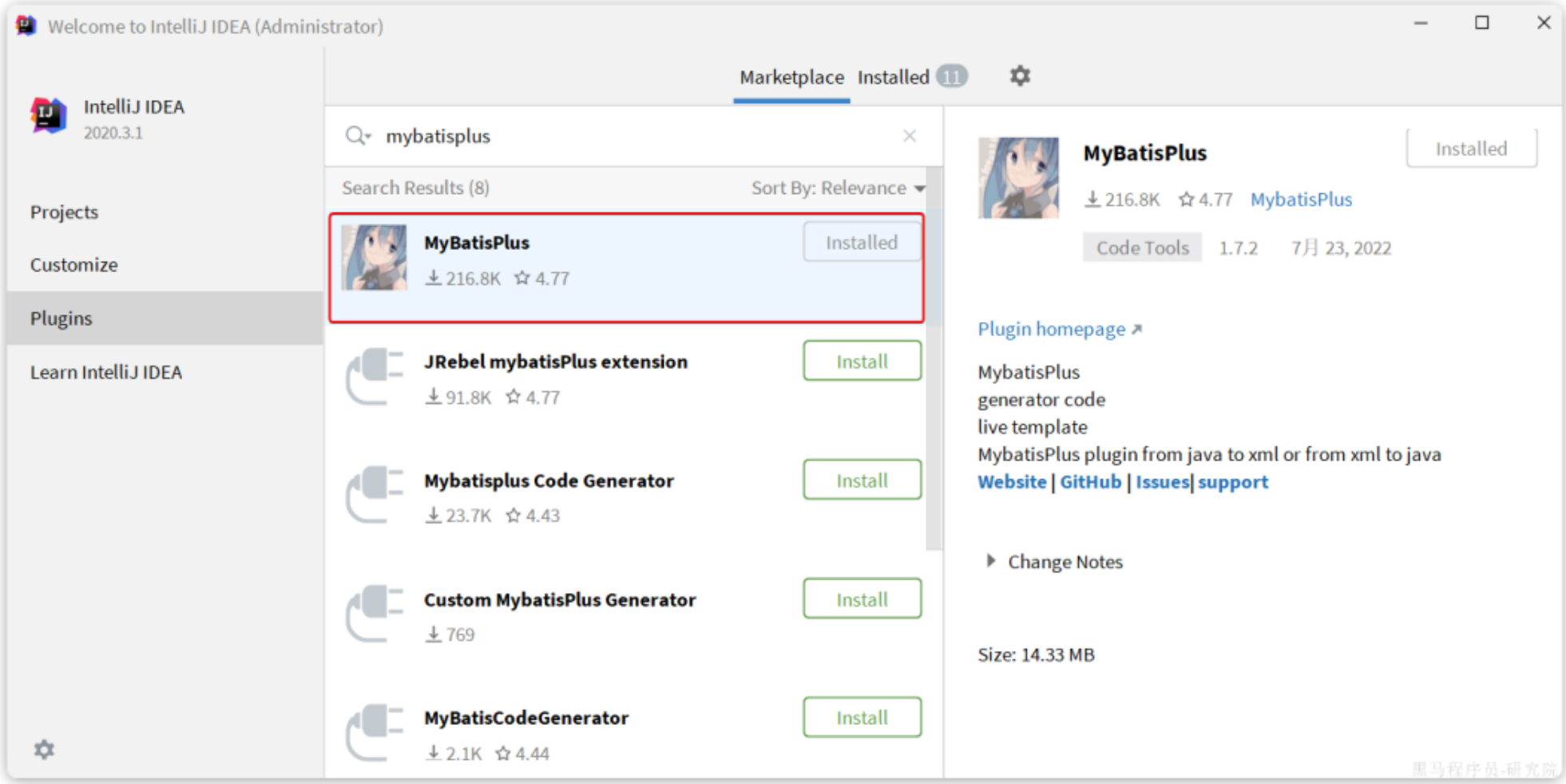
```java
@TableName("user")
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    private String name;
    private Integer age;
    private Boolean isMarried;
    private Integer order;
}
```

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

}
```

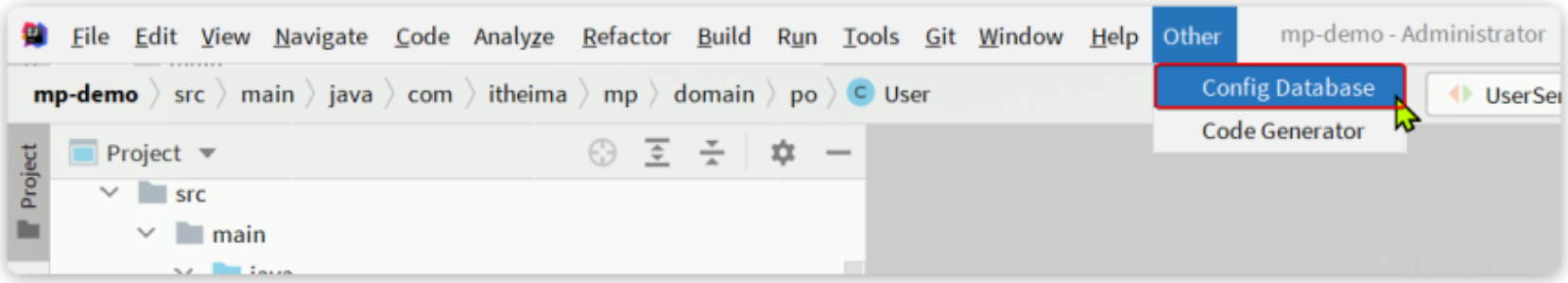
```java
public interface IUserService extends IService<User> {

}
```

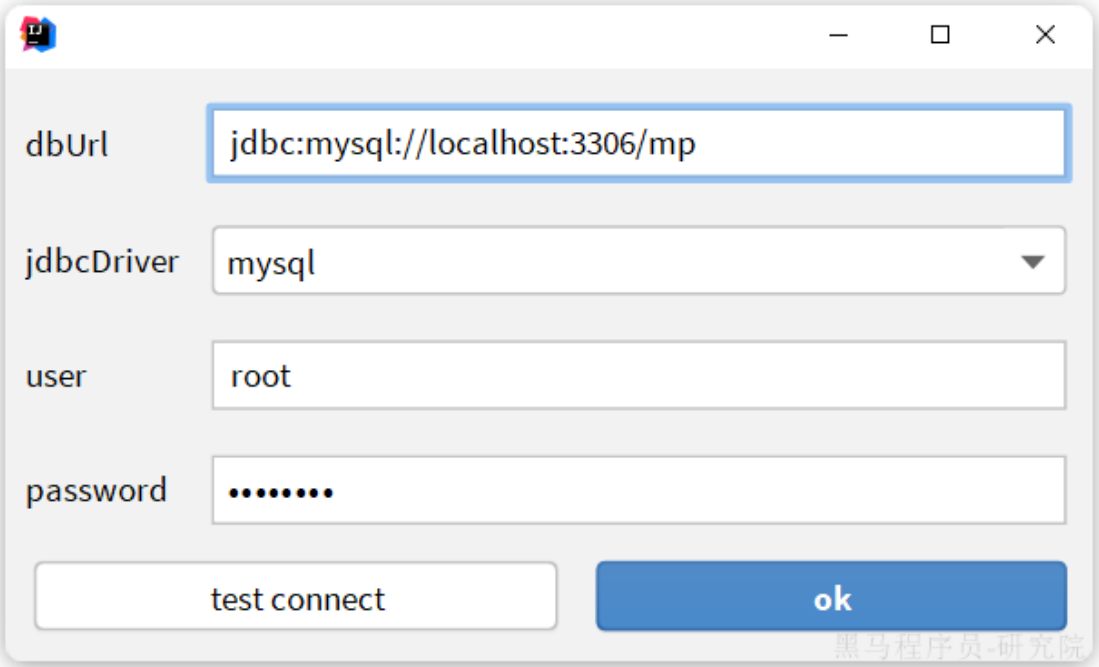
```java
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements IUserService{

}
```

代码生成

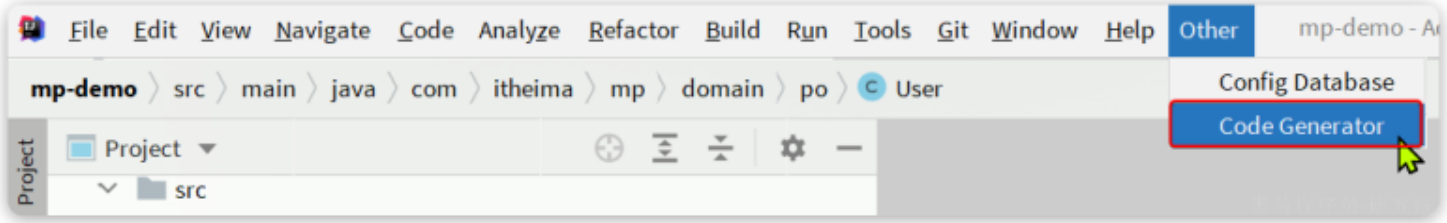

Welcome to IntelliJ IDEA (Administrator)
IntelliJ IDEA
2020.3.1
Projects
Customize
Plugins
Learn IntelliJ IDEA
Marketplace
Installed 11
mybatisplus
Search Results (8)
Sort By: Relevance
MyBatisPlus
Installed
216.8K ☆4.77
JRebel mybatisPlus extension
Install
91.8K ☆4.77
Mybatisplus Code Generator
Install
23.7K ☆4.43
Custom MybatisPlus Generator
Install
769
MyBatisCodeGenerator
Install
2.1K ☆4.44
MyBatisPlus
Installed
216.8K ☆4.77 MybatisPlus
Code Tools 1.7.2 7月 23, 2022
Plugin homepage ↗
MybatisPlus
generator code
live template
MybatisPlus plugin from java to xml or from xml to java
Website | GitHub | Issues | support
Change Notes
Size: 14.33 MB

代码生成


File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help Other
mp-demo - Administrator
mp-demo src main java com itheima mp domain po User
Config Database
Code Generator
Project
src
main


dbUrl
jdbc:mysql://localhost:3306/mp
jdbcDriver
mysql
user
root
password
test connect
ok

代码生成


File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help Other
mp-demo - A
mp-demo src main java com itheima mp domain po User
Config Database
Code Generator
Project
Project
src

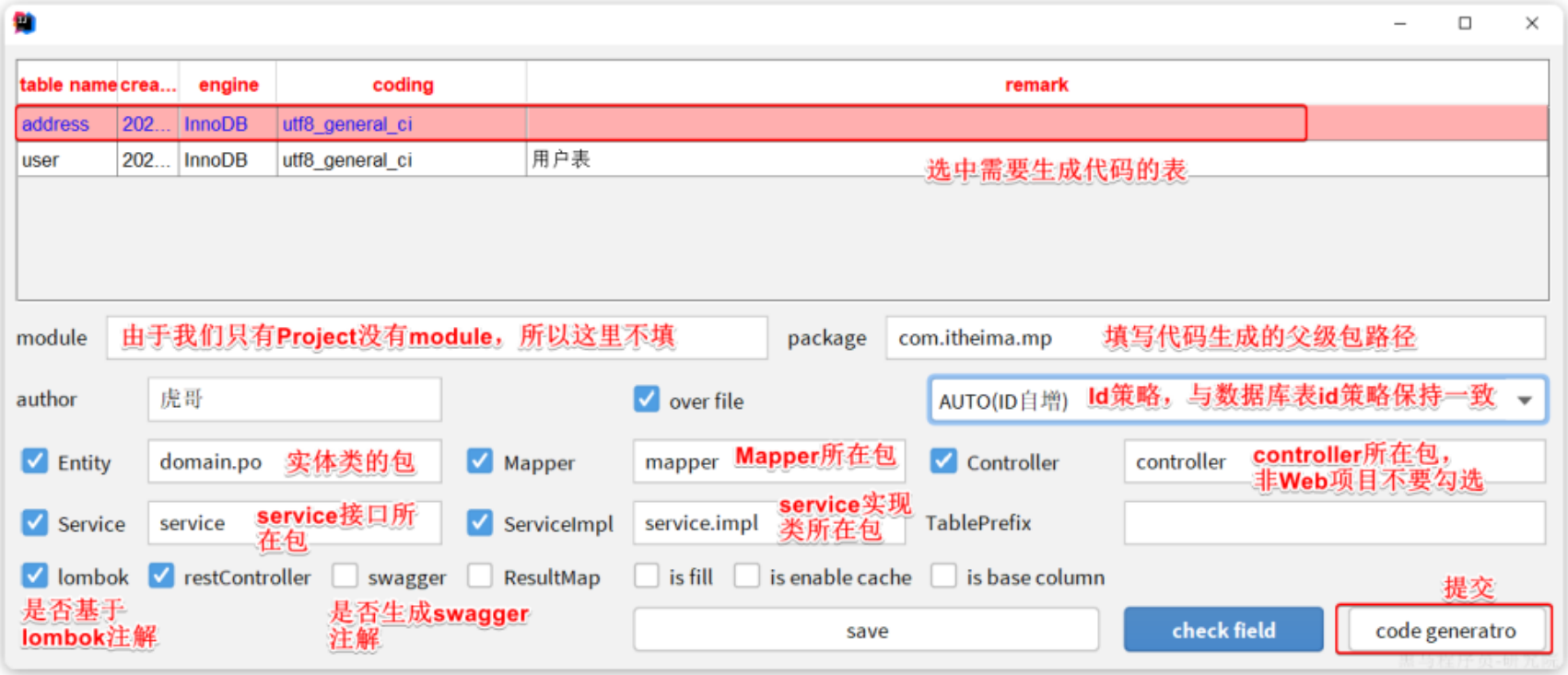


table name crea... engine coding remark
address 202... InnoDB utf8_general_ci
user 202... InnoDB utf8_general_ci 用户表
选中需要生成代码的表
module
由于我们只有Project没有module，所以这里不填
package
com.itheima.mp
填写代码生成的父级包路径
author
虎哥
over file
AUTO(ID自增)
Id策略，与数据库表id策略保持一致
Entity
domain.po
实体类的包
Mapper
mapper
Mapper所在包
Controller
controller
controller所在包，非Web项目不要勾选
Service
service
service接口所在包
ServiceImpl
service.impl
service实现类所在包
TablePrefix
lombok
restController
swagger
ResultMap
is fill
is enable cache
is base column
是否基于lombok注解
是否生成swagger注解
save
check field
提交
code generatro

目录

Contents

- 代码生成
- 静态工具
- 逻辑删除
- 枚举处理器
- JSON处理器

静态工具

Db
- save(T): boolean
- saveBatch(Collection<T>): boolean
- saveBatch(Collection<T>, int): boolean
- saveOrUpdateBatch(Collection<T>): boolean
- saveOrUpdateBatch(Collection<T>, int): boolean
- saveOrUpdate(T): boolean
- saveOrUpdate(T, AbstractWrapper<T, ?, ?>): boolean

Db
- removeById(Serializable, Class<T>): boolean
- removeById(T): boolean
- remove(AbstractWrapper<T, ?, ?>): boolean
- removeByIds(Collection<? extends Serializable>, Class<T>): boolean

Db
- listByMap(Map<String, Object>, Class<T>): List<T>
- listByIds(Collection<? extends Serializable>, Class<T>): List<T>
- list(AbstractWrapper<T, ?, ?>): List<T>
- list(Class<T>): List<T>

Db
- saveOrUpdateBatch(Collection<T>): boolean
- saveOrUpdateBatch(Collection<T>, int): boolean
- updateById(T): boolean
- update(AbstractWrapper<T, ?, ?>): boolean
- update(T, AbstractWrapper<T, ?, ?>): boolean
- updateBatchById(Collection<T>): boolean
- updateBatchById(Collection<T>, int): boolean
- saveOrUpdate(T): boolean

Db
- count(Class<T>): long
- count(AbstractWrapper<T, ?, ?>): long

Db
- getById(Serializable, Class<T>): T
- getOne(AbstractWrapper<T, ?, ?>): T
- getOne(AbstractWrapper<T, ?, ?>, boolean): T

Db
- lambdaQuery(Class<T>): LambdaQueryChainWrapper<T>
- lambdaUpdate(Class<T>): LambdaUpdateChainWrapper<T>

Db
- pageMaps(E, Class<T>): E
- pageMaps(E, AbstractWrapper<T, ?, ?>): E
- page(IPage<T>, Class<T>): IPage<T>
- page(IPage<T>, AbstractWrapper<T, ?, ?>): IPage<T>

案例 静态工具查询

需求：

① 改造根据id查询用户的接口，查询用户的同时，查询出用户对应的所有地址

② 改造根据id批量查询用户的接口，查询用户的同时，查询出用户对应的所有地址

③ 实现根据用户id查询收货地址功能，需要验证用户状态，冻结用户抛出异常（练习）

目录

Contents

- 代码生成
- 静态工具
- 逻辑删除
- 枚举处理器
- JSON处理器

逻辑删除

逻辑删除就是基于代码逻辑模拟删除效果，但并不会真正删除数据。思路如下：

- 在表中添加一个字段标记数据是否被删除
- 当删除数据时把标记置为1
- 查询时只查询标记为0的数据

例如逻辑删除字段为deleted：

- 删除操作：

```sql
UPDATE user SET deleted = 1 WHERE id = 1 AND deleted = 0
```

- 查询操作：

```sql
SELECT * FROM user WHERE deleted = 0
```

逻辑删除

MybatisPlus提供了逻辑删除功能，无需改变方法调用的方式，而是在底层帮我们自动修改CRUD的语句。我们要做的就是在application.yaml文件中配置逻辑删除的字段名称和值即可：

```yaml
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: flag # 全局逻辑删除的实体字段名，字段类型可以是boolean、integer
      logic-delete-value: 1 # 逻辑已删除值(默认为 1)
      logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

注意

逻辑删除本身也有自己的问题，比如：

- 会导致数据库表垃圾数据越来越多，影响查询效率
- SQL中全都需要对逻辑删除字段做判断，影响查询效率

因此，我不太推荐采用逻辑删除功能，如果数据不能删除，可以采用把数据迁移到其它表的办法。

目录

Contents

- 代码生成
- 静态工具
- 逻辑删除
- 枚举处理器
- JSON处理器

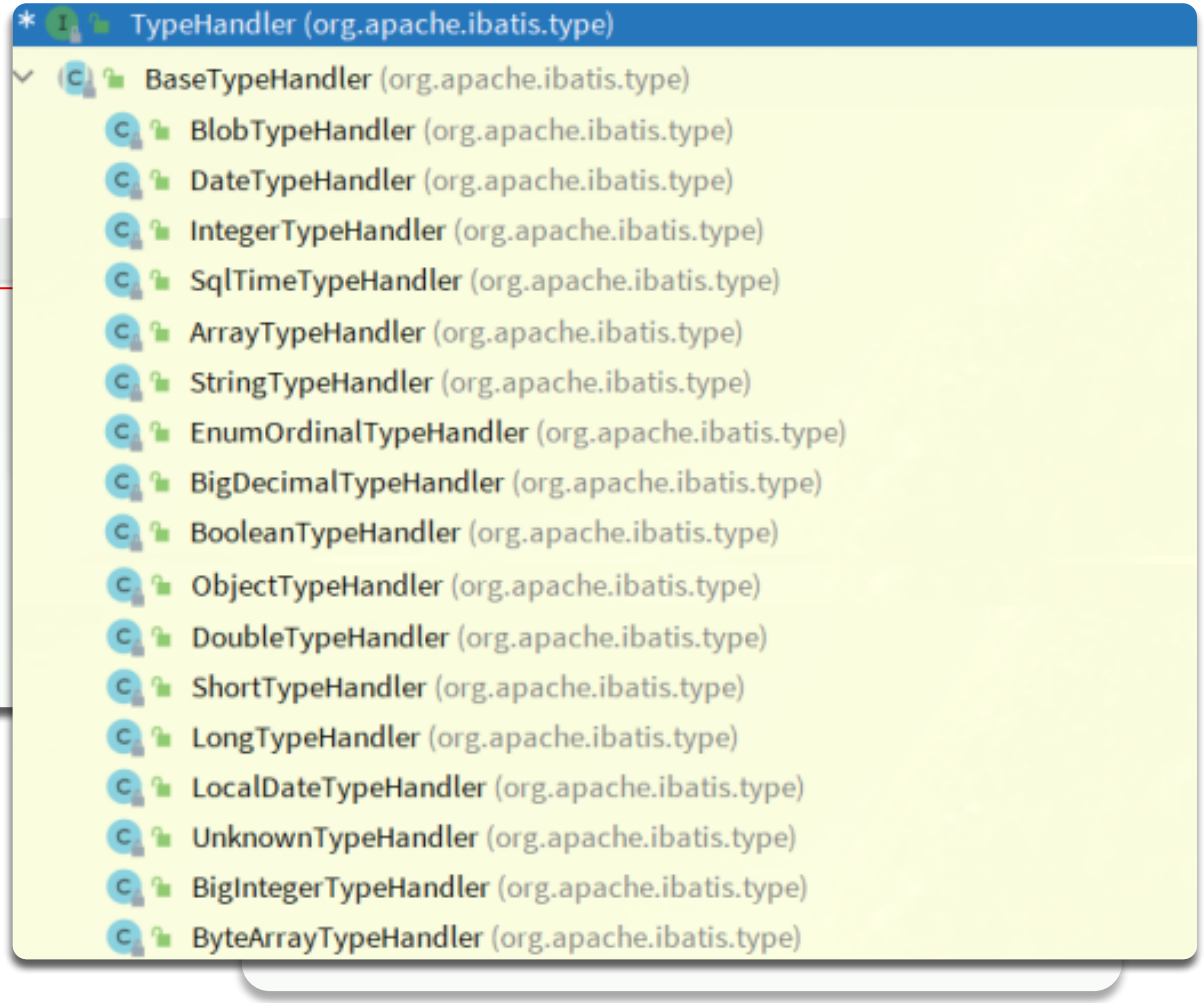
枚举处理器

User类中有一个用户状态字段：

User.java

```
34
        详细信息
38      private String info;
39
        使用状态（1正常 2冻结）
43      private UserStatusatus;
44
```

#	名称	数据类型	注释
1	**id**	**BIGINT**	**用户id**
2	username	VARCHAR	用户名
3	password	VARCHAR	密码
4	phone	VARCHAR	注册手机号
5	info	JSON	详细信息
6	status	INT	使用状态（1正常 2冻结）
7	balance	INT	账户余额



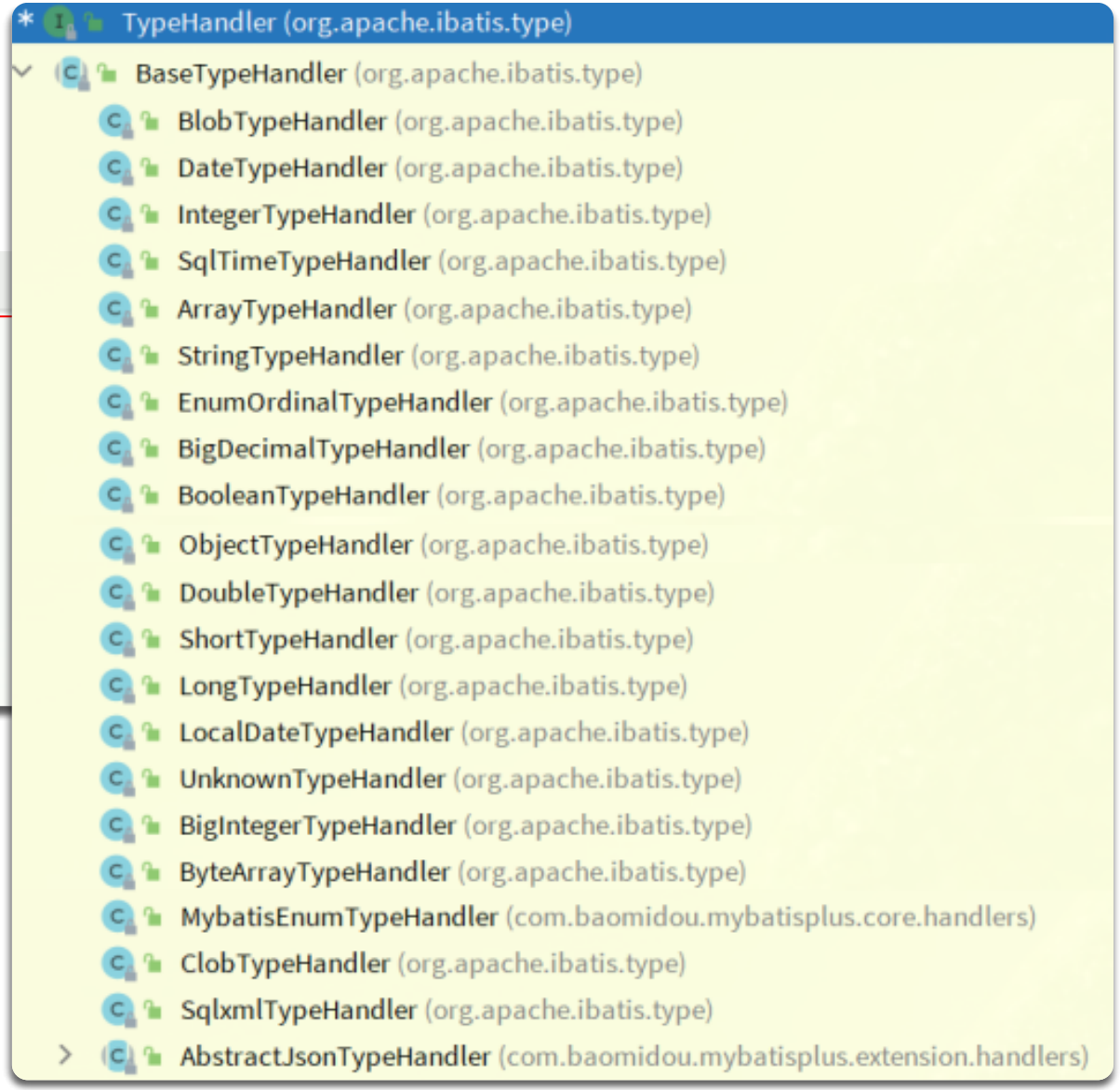
枚举处理器

User类中有一个用户状态字段：

User.java

```
详细信息
38  private String info;
39
使用状态（1正常 2冻结）
43  private UserStatus status;
44
```

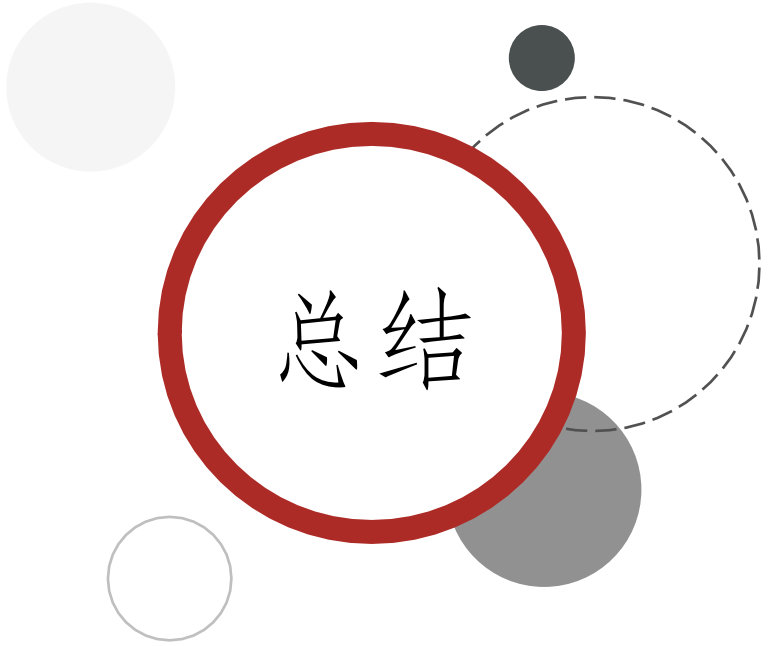
#	名称	数据类型	注释
1	id	BIGINT	用户id
2	username	VARCHAR	用户名
3	password	VARCHAR	密码
4	phone	VARCHAR	注册手机号
5	info	JSON	详细信息
6	status	INT	使用状态（1正常 2冻结）
7	balance	INT	账户余额



通用枚举

在application.yml中配置全局枚举处理器：

```yaml
mybatis-plus:
  configuration:
    default-enum-type-handler: com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler
```



如何实现PO类中的枚举类型变量与数据库字段的转换?

① 给枚举中的与数据库对应value值添加@EnumValue注解

```
@EnumValue
private final int value;
@JsonValue
private final String desc;
```

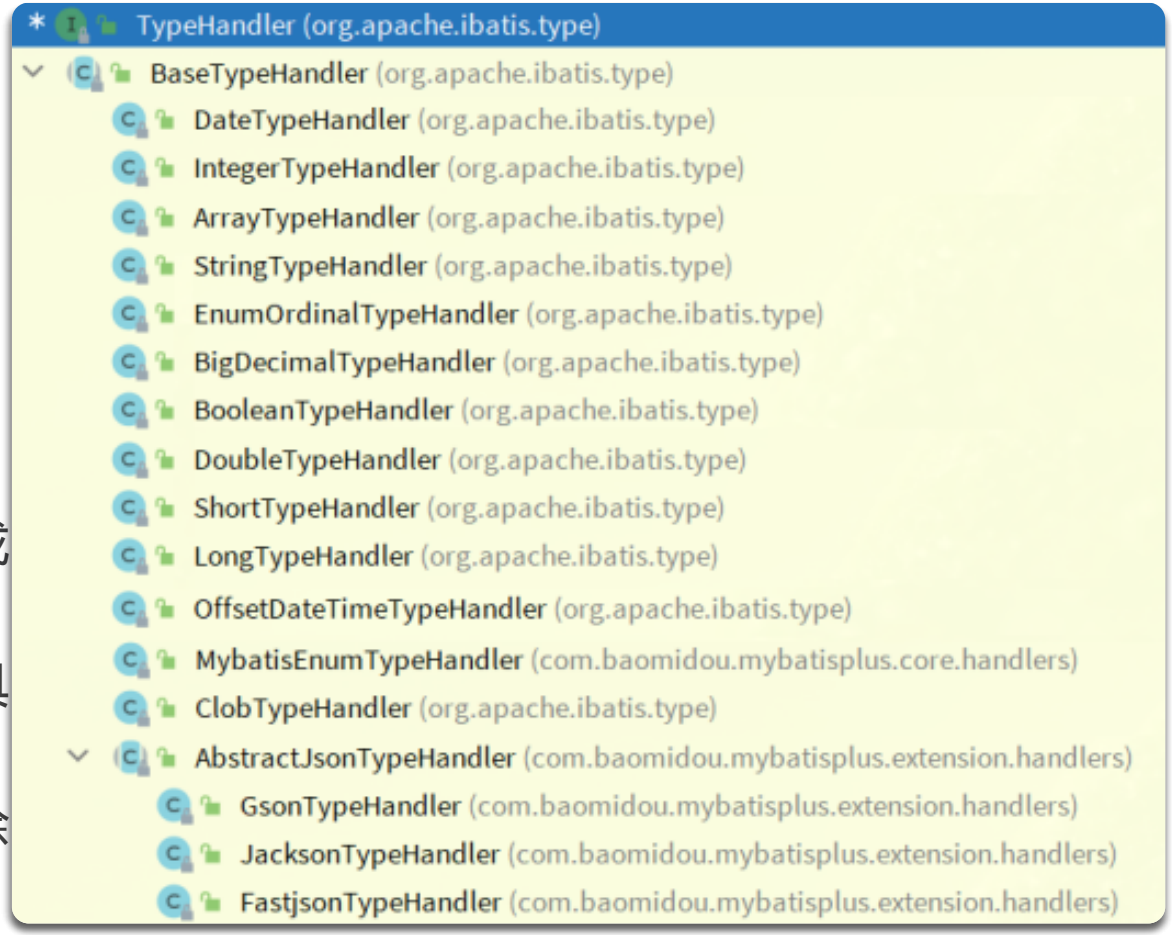
② 在配置文件中配置统一的枚举处理器，实现类型转换

```
mybatis-plus:
  configuration:
    default-enum-type-handler: com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler
```

目录

Contents

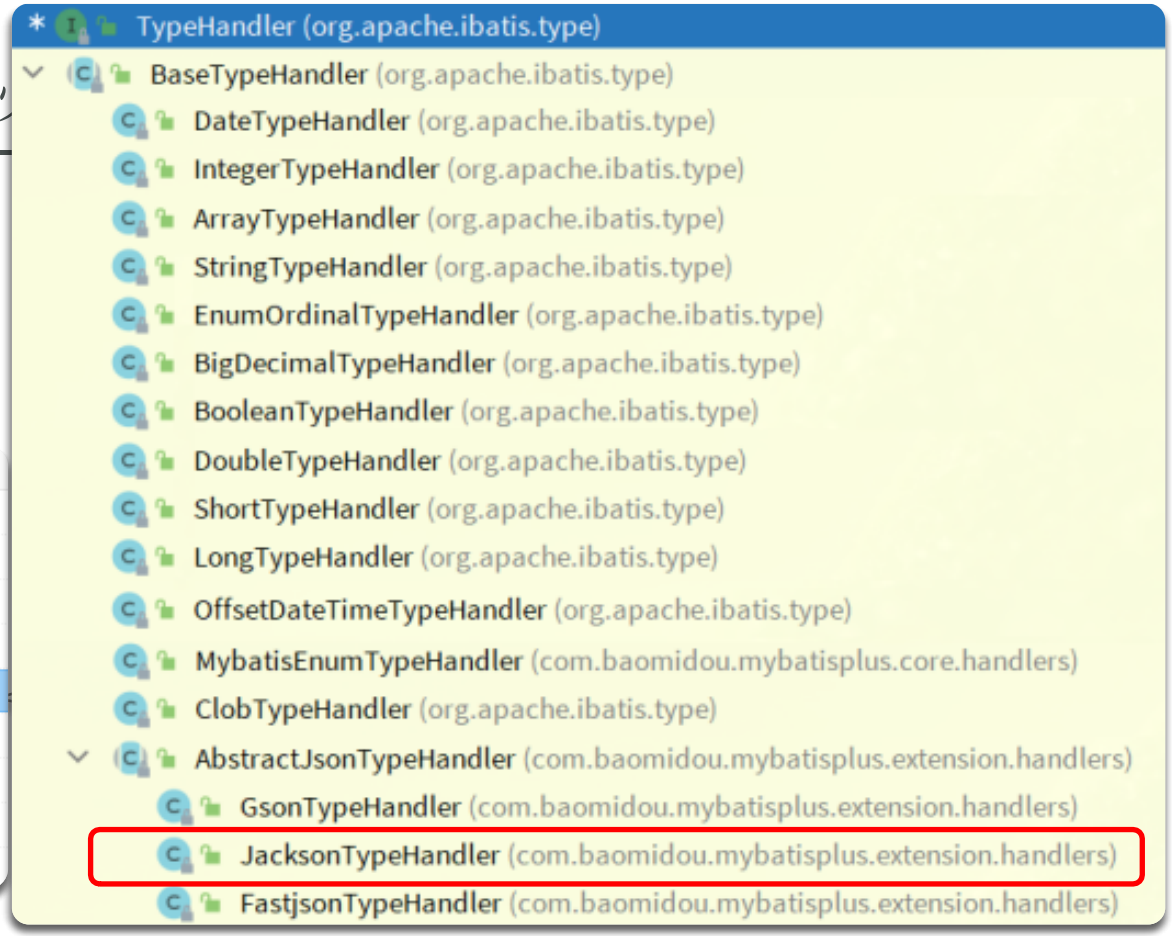
- 代码生成
- 静态工具
- 逻辑删除
- 枚举处理器
- JSON处理器



JSON处理器

数据库中user表中有一个json类型的字段：

#	名称	数据类型	注释	长度/集合
1	id	BIGINT	用户id	19
2	username	VARCHAR	用户名	50
3	password	VARCHAR	密码	128
4	phone	VARCHAR	注册手机号	20
5	info	JSON	详细信息	
6	status	INT	使用状态（1正常 2冻结）	10
7	balance	INT	账户余额	10
8	create_time	DATETIME	创建时间	
9	update_time	DATETIME	更新时间	




```java
@Data
@TableName(value = "user", autoResultMap = true)
public class User {
    private Long id;

    private String username;
    @TableField(typeHandler = JacksonTypeHandler.class)
    private UserInfo info;
}
```

```java
@Data
public class UserInfo {
    private Integer age;
    private String intro;
    private String gender;
}
```

插件功能

插件功能

MyBatisPlus提供的内置拦截器有下面这些：

序号	拦截器	描述
1	TenantLineInnerInterceptor	多租户插件
2	DynamicTableNameInnerInterceptor	动态表名插件
3	PaginationInnerInterceptor	分页插件
4	OptimisticLockerInnerInterceptor	乐观锁插件
5	IllegalSQLInnerInterceptor	SQL性能规范插件，检测并拦截垃圾SQL
6	BlockAttackInnerInterceptor	防止全表更新和删除的插件

目录

Contents

- 分页插件
- 通用分页实体

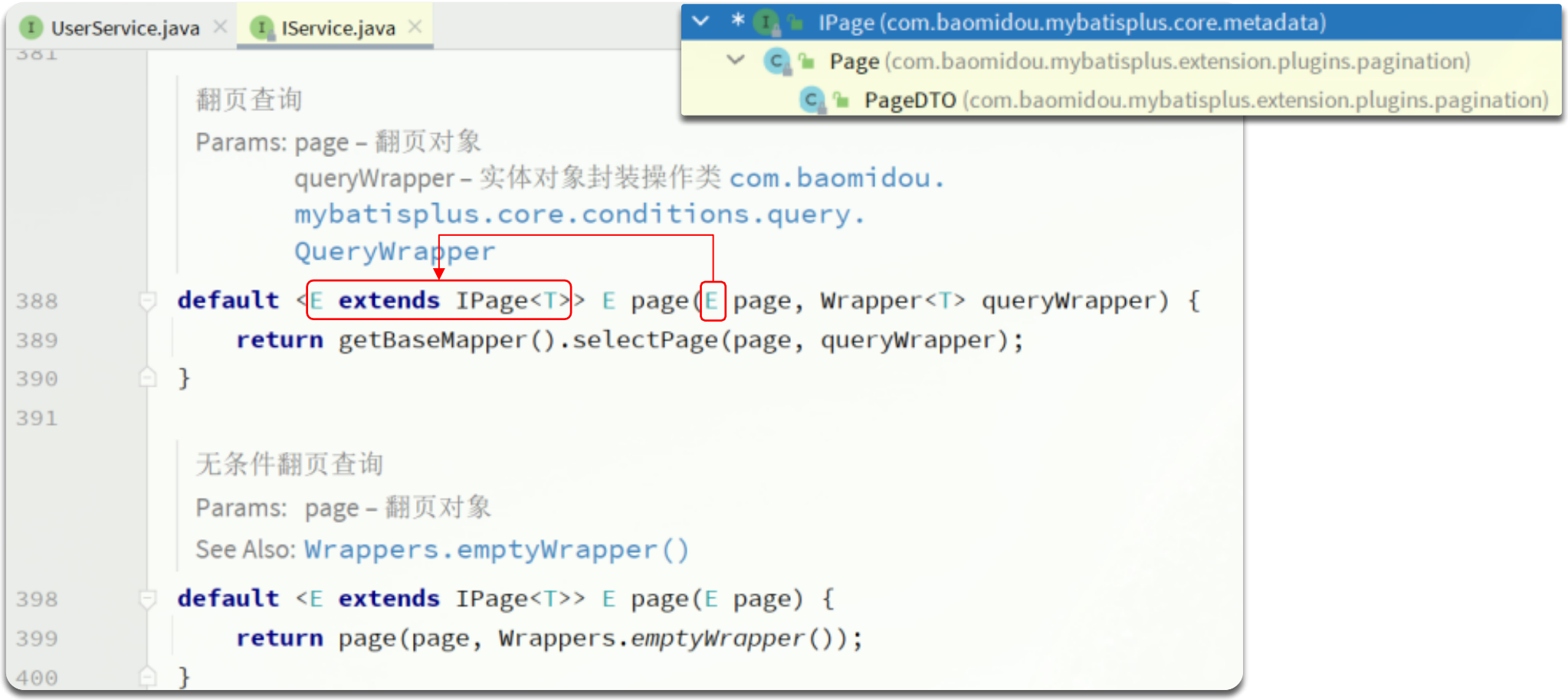
分页插件

首先，要在配置类中注册MyBatisPlus的核心插件，同时添加分页插件：

```java
@Configuration
public class MybatisConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        // 1.初始化核心插件
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        // 2.添加分页插件
        PaginationInnerInterceptor pageInterceptor = new PaginationInnerInterceptor(DbType.MYSQL);
        pageInterceptor.setMaxLimit(1000L); // 设置分页上限
        interceptor.addInnerInterceptor(pageInterceptor);
        return interceptor;
    }
}
```

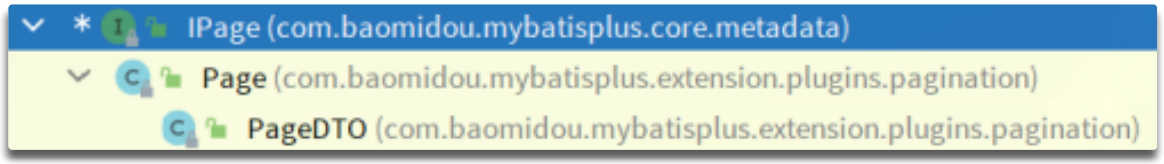
分页插件

接着，就可以使用分页的API了：



分页插件

接着，就可以使用分页的API了：




```java
@Test
void testPageQuery() {
    // 1.查询
    int pageNo = 1, pageSize = 5;
    // 1.1.分页参数
    Page<User> page = Page.of(pageNo, pageSize);
    // 1.2.排序参数，通过OrderItem来指定
    page.addOrder(new OrderItem("balance", false));
    // 1.3.分页查询
    Page<User> p = userService.page(page);
    // 2.总条数
    System.out.println("total = " + p.getTotal());
    // 3.总页数
    System.out.println("pages = " + p.getPages());
    // 4.分页数据
    List<User> records = p.getRecords();
    records.forEach(System.out::println);
}
```

目录

Contents

- 分页插件
- 通用分页实体

案例 简单分页查询案例

需求：遵循下面的接口规范，编写一个UserController接口，实现User的分页查询

参数	说明
请求方式	GET
请求路径	/users/page

请求参数	{
	"pageNo": 1,
	"pageSize": 5,
	"sortBy": "balance",
	"isAsc": false,
	"name": "jack",
	"status": 1
	}

返回值	
特殊说明	• 如果排序字段为空，默认按照更新时间排序 • 排序字段不为空，则按照排序字段排序

案例

通用分页实体

需求：

- 在PageQuery中定义方法，将PageQuery对象转为MyBatisPlus中的Page对象
- 在PageDTO中定义方法，将MyBatisPlus中的Page结果转为PageDTO结果