

```
1 // 官网
2 https://baomidou.com/introduce/
```

## 1.引入依赖

### 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：

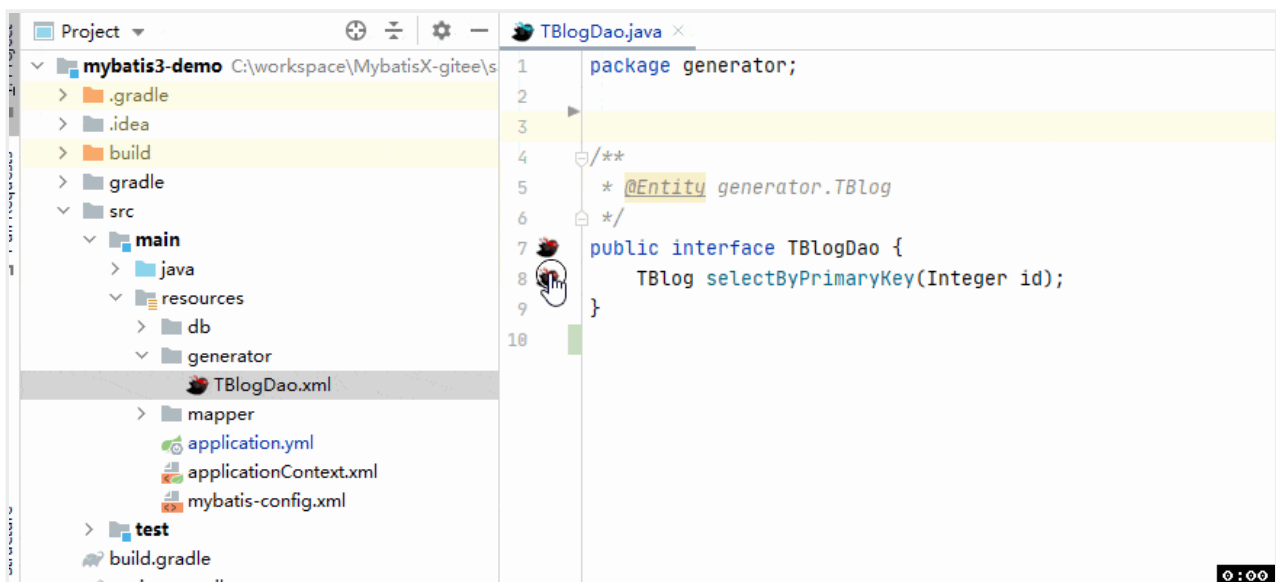
```
<!--MybatisPlus-->
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.5.3.1</version>
</dependency>
```

喜大普奔

## 2.Mybatis X 插件

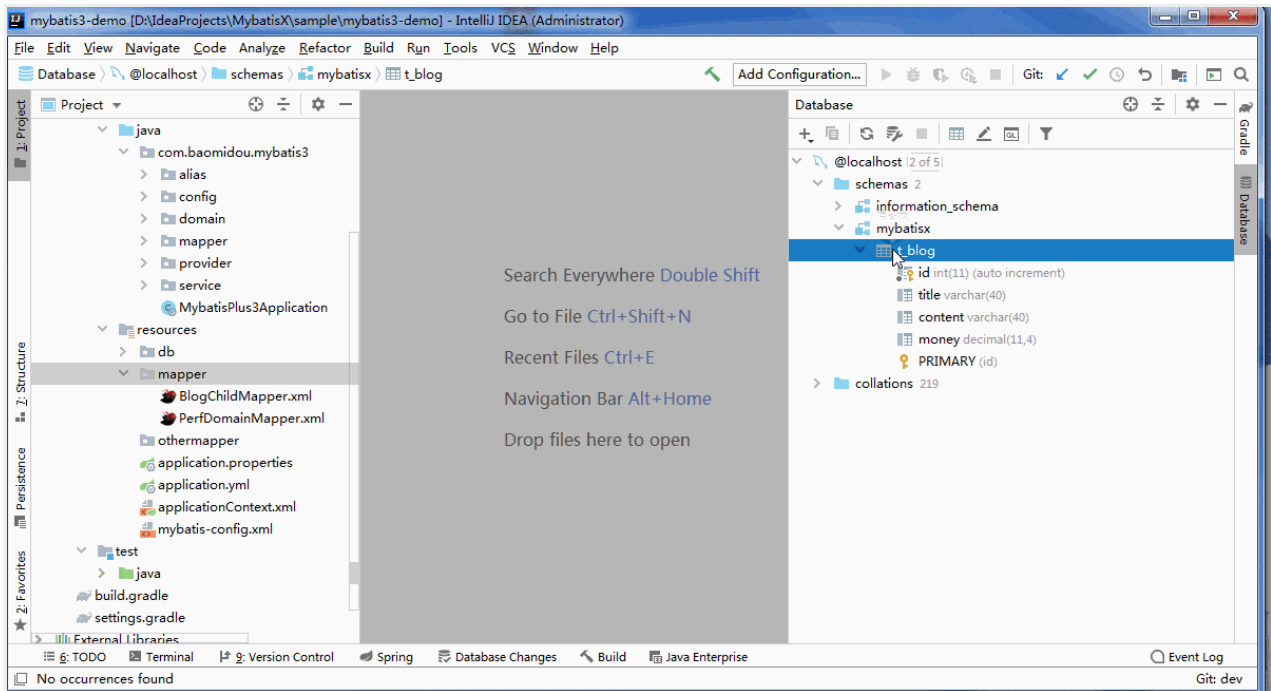
### ①XML 映射跳转

MybatisX 提供了便捷的 XML 映射文件与 Java 接口之间的跳转功能，让开发者能够快速地在两者之间切换，提高开发效率。



## ②代码生成

通过 MybatisX，您可以轻松地根据数据库表结构生成对应的 Java 实体类、Mapper 接口及 XML 映射文件。

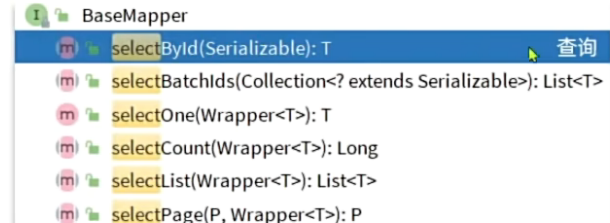
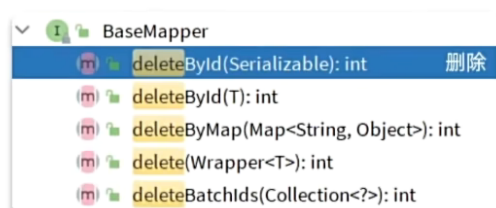
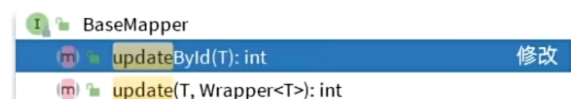


## 3.定义Mapper

### 2. 定义Mapper

自定义的Mapper继承MybatisPlus提供的BaseMapper接口：

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



## 4. 常见注解



多一句没有，少一句不行，用更短时间，教会更实用的技术！

### 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

```
@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

它不是数据库的字段就行了

## 5. 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：

```
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
```

## 6.案例

```
@Test
void testQueryWrapper() {
    // 1. 构建查询条件
    QueryWrapper<User> wrapper = new QueryWrapper<User>()
        .select("id", "username", "info", "balance")
        .like(column: "username", val: "o")
        .ge(column: "balance", val: 1000);
    // 2. 查询
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
```

```
@Test
void testUpdateByQueryWrapper() {
    // 1. 要更新的数据
    User user = new User();
    user.setBalance(2000);
    // 2. 更新的条件
    QueryWrapper<User> wrapper = new QueryWrapper<User>().eq(column: "username", val: "jack");
    // 3. 执行更新
    userMapper.update(user, wrapper);
}
```

① 查询出名字中带o的，存款大于等于1000元的人的id、us

```
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")
```

需求：更新id为1,2,4的用户的余额，扣200

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

```
@Test
void testUpdateWrapper() {
    List<Long> ids = List.of(1L, 2L, 4L);
    UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
        .setSql("balance = balance - 200")
        .in(column: "id", ids);
    userMapper.update(entity: null, wrapper);
}
```

## 7.条件构造器(适用于复杂条件)

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

```
void testQueryWrapper() {
    // 1. 创建查询条件构造器
    QueryWrapper<User> wrapper = new QueryWrapper<User>()
        .select( ...columns: "id", "username", "info", "balance")
        .like( column: "username", val: "o")
        .ge( column: "balance", val: 1000);
    // 2. 查询
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}

@Test
void testLambdaQueryWrapper() {
    // 1. 创建查询条件构造器
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>()
        .select(User::getId, User::getUsername, User::getInfo, User::getBalance)
        .like(User::getUsername, val: "o")
        .ge(User::getBalance, val: 1000);
    // 2. 查询
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
```

## 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>
```

高知软件

## 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
```

## 总结

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 {}
```

要去指定mapper的类型和实体类的类型

## 8.分页插件

```
1 // com.itheima.mp
2 //    config
3 //    MybatisConfig.java
```

### 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：

```
@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;
    }
}
```

也就是分页插件

```
1 @Test
2 void testPage() {
3     int pageNo = 1;
4     int pageSize = 2;
5     // 1. 准备分页条件
6     // 1.1. 分页条件
7     Page<User> page = Page.of(pageNo, pageSize);
8     // 1.2. 排序条件
9     page.addOrder(new OrderItem("balance", true));
10    page.addOrder(new OrderItem("id", true));
11    // 2. 执行分页查询
12    Page<User> p = userService.page(page);
13    // 3. 解析
14    long total = p.getTotal();
15    System.out.println("total = " + total);
16    long pages = p.getPages();
17    System.out.println("pages = " + pages);
18    List<User> users = p.getRecords();
19    users.forEach(System.out::println);
20 }
```