

Mybatis-Plus



MyBatis-Plus

为简化开发而生

为什么要用Mybatis-plus

- **无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑
- **损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作
- **强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求
- **支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错
- **支持多种数据库**：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer2005、SQLServer 等多种数据库
- **支持主键自动生成**：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题
- **支持 XML 热加载**：Mapper 对应的 XML 支持热加载，对于简单的 CRUD 操作，甚至可以无 XML 启动
- **** ****：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强大的 CRUD 操作

- **支持自定义全局通用操作**：支持全局通用方法注入（Write once, use anywhere）
- **支持关键词自动转义**：支持数据库关键词（order、key.....）自动转义，还可自定义关键词
- **内置代码生成器**：采用代码或者 Maven 插件可快速生成 Mapper、Model、Service、Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用
- **内置分页插件**：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询
- **内置性能分析插件**：可输出 Sql 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出慢查询
- **内置全局拦截插件**：提供全表 delete、update 操作智能分析阻断，也可自定义拦截规则，预防误操作
- **内置 Sql 注入剥离器**：支持 Sql 注入剥离，有效预防 Sql 注入攻击























[文档](#) [常见问题](#)

MybatisPlus 和 原生Mybatis对比

MybatisPlus			
Integer selectCount(@Param(Constants.WRAPPER) Wrapper queryWrapper);	查询满足条件数据的数量	int countByExample(UserExample example);	根据example获取满足条件的数量
int delete(@Param(Constants.WRAPPER) Wrapper wrapper);	根据Wrapper删除	int deleteByExample(UserExample example);	根据example删除
int deleteById(Serializable id);	根据主键删除	int deleteByPrimaryKey(Long id);	根据主键删除
int insert(T entity);	插入操作	int insert(User record);	插入User
List selectList(@Param(Constants.WRAPPER) Wrapper queryWrapper);	根据Wrapper查询	List selectByExample(UserExample example);	根据example查询满足条件的列
		int insertSelective(User record);	可选择的插入
T selectById(Serializable id);	根据主键查询	User selectByPrimaryKey(Long id);	根据主键查询
int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper updateWrapper);		int updateByExampleSelective(@Param("record") User record, @Param("example") UserExample example);	根据example可选择的更新
		int updateByExample(@Param("record") User record, @Param("example") UserExample example);	根据example进行更新
int updateById(@Param(Constants.ENTITY) T entity);		int updateByPrimaryKeySelective(User record);	根据主键可选择更新
int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper updateWrapper);		int updateByPrimaryKey(User record);	根据主键更新
T selectOne(@Param(Constants.WRAPPER) Wrapper queryWrapper);	返回满足条件的单个对象		
List selectObjs(@Param(Constants.WRAPPER) Wrapper queryWrapper);			
List selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);	根据Map查询		
IPage selectPage(IPage page, @Param(Constants.WRAPPER) Wrapper queryWrapper);	分页查询		
IPage<Map<String, Object>> selectMapsPage(IPage page, @Param(Constants.WRAPPER) Wrapper queryWrapper);	分页查询		
List selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);	根据主键批量查询		
int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);	根据主键批量删除		

可见MybatisPlus 提供更为丰富的方法，支持根据Map **查询更新** 操作，支持根据**主键的批量操作**, 支持**分页**,不必自己再去实现.

下面是实现的**Service**类提供的一些方法

- m  save(T): boolean
- m  saveBatch(Collection<T>, int): boolean
- m  saveOrUpdate(T): boolean
- m  saveOrUpdateBatch(Collection<T>, int): boolean
- m  removeById(Serializable): boolean
- m  removeByMap(Map<String, Object>): boolean
- m  remove(Wrapper<T>): boolean
- m  removeByIds(Collection<? extends Serializable>): boolean
- m  updateById(T): boolean
- m  update(T, Wrapper<T>): boolean
- m  updateBatchById(Collection<T>, int): boolean
- m  getById(Serializable): T
- m  listByIds(Collection<? extends Serializable>): Collection<T>
- m  listByMap(Map<String, Object>): Collection<T>
- m  getOne(Wrapper<T>, boolean): T
- m  getMap(Wrapper<T>): Map<String, Object>
- m  count(Wrapper<T>): int
- m  list(Wrapper<T>): List<T>
- m  page(IPage<T>, Wrapper<T>): IPage<T>
- m  listMaps(Wrapper<T>): List<Map<String, Object>>
- m  listObjs(Wrapper<T>, Function<? super Object, V>): List<V>
- m  pageMaps(IPage<T>, Wrapper<T>): IPage<Map<String, Object>>

一些代码层面的对比

分页查询

```
/**
 * mybaits-plus
 */

@Override
public IPage getAllByPage(long rows, long offset) {
    return userMapper.selectPage(new Page<User>(rows, offset), null);
}
```

```

/**
 * mybaits
 */
@Override
public PageInfo<User> selectByPage(int row, int offset) {
    PageHelper.startPage(row, offset);
    List<User> list = userMapper.selectAll();
    return new PageInfo<>(list);
}

```

Mybatis-plus 自己实现了分页查询,而且是**物理分页**,物理分页就是在进行查询时带上了 limit 参数,内存分页是指把所有的满足条件的数据查询出来然后在进行出来.

需要注意的是实现物理分页需要向Spring 容器注册一个Bean

```

@Configuration
/**
 * 需要加上分页的拦截器
 * 这是Springboot 的写法 如果是Spring 可以在xml 文件中进行配置
 */
public class config {
    @Bean
    public PaginationInterceptor paginationInterceptor() {
        PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
        return paginationInterceptor;
    }
}

```

多条件查询

```

@Override
public List<User> getUser(Object userName, Object email) {
    return userMapper.selectList(new QueryWrapper<User>()
        .eq("email", email)
        .eq("name", userName)
        .lt("id", 3));
}

```

等价于

```

SELECT *
FROM user
WHERE (email='' AND name='' AND id < 3)

```

```

@Override
public List<User> selectByUserName(String userName) {
    UserExample userExample = new UserExample();
    userExample.createCriteria()
        .andNameEqualTo("userName")
        .andIdBetween(1L, 5L);
    return userMapper.selectByExample(userExample);
}

```

Mybatis-plus 查询时可以直接new QueryWrapper一个对象 采用Build模式, 底层也是根据这个Wrapper对象拼装出sql语句 进行执行

每一个实体类都要对应一个Example, 而且 不能直接new 需要 new完成之后然后在进行设置.

//todo debug一下进行展示

```

public int selectCount() {
    Date startDate = new Date();
    Date currentDate = new Date();
    int buyCount = userMapper.selectCount(new QueryWrapper<User>()
        .select("sum(quantity)")
        .isNull("order_id")
        .eq("user_id", 1)
        .eq("type", 1)
        .in("status", new Integer[]{0, 1})
        .eq("product_id", 1)
        .between("created_time", startDate, currentDate)
        .eq("weal", 1));
    return buyCount;
}

```

查询方式	说明
select	设置 SELECT 查询字段
where	WHERE 语句, 拼接 + WHERE 条件
and	AND 语句, 拼接 + AND 字段=值
andNew	AND 语句, 拼接 + AND (字段=值)
or	OR 语句, 拼接 + OR 字段=值
orNew	OR 语句, 拼接 + OR (字段=值)
eq	等于=
allEq	基于 map 内容等于=
ne	不等于<>
gt	大于>
ge	大于等于>=
lt	小于<
le	小于等于<=
like	模糊查询 LIKE
notLike	模糊查询 NOT LIKE
in	IN 查询
notIn	NOT IN 查询
isNull	NULL 值查询
isNotNull	IS NOT NULL
groupBy	分组 GROUP BY
having	HAVING 关键词
orderBy	排序 ORDER BY
orderAsc	ASC 排序 ORDER BY
orderDesc	DESC 排序 ORDER BY
exists	EXISTS 条件语句
notExists	NOT EXISTS 条件语句
between	BETWEEN 条件语句
notBetween	NOT BETWEEN 条件语句

查询方式	说明
addFilter	自由拼接 SQL
last	拼接在最后, 例如: last("LIMIT 1")

批量查询

```
@Override
public List<User> getUserByIdList(List<Long> idList) {
    return userMapper.selectBatchIds(idList);
}
```

```
@Override
public List<User> selectBatchIds(List<Long> idList) {
    List<User> users = new ArrayList<>();
    for (Long id : idList) {
        users.add(userMapper.selectByPrimaryKey(id));
    }
    return users;
}
```

```
<select id="selectByIds"parameterType="java.util.List"resultMap="BaseResultMap">
    select * from user where
    <foreach collection="idList" item="id" open="(" close=")" separator=",">
        #{id}
    </foreach>
</select>
```

批量删除

```
@Override
public int deleteBatchIds(List<Integer> idList) {
    return userMapper.deleteBatchIds(idList);
}
```

Lambda写法

```
@Override
public List<User> getUserByLambda(String age, String email) {
    return userMapper.selectList(new QueryWrapper<User>().lambda()
        .eq(User::getAge, Integer.valueOf(age))
        .eq(User::getEmail, email)
    );
}
```



```
@Override
public List<User> getUserByLambdaAnother(Integer age, String email) {
    return userMapper.selectList(new QueryWrapper<User>().lambda()
        .and(obj -> obj.eq(User::getEmail, email)
        .eq(User::getAge, age)));
}
```

总结

1. Mybatis-plus提供了很多**基础的查询**，对单表的简单查询也没必要重新在Mapper定义方法，然后在xml 文件写sql，直接在Service层进行操作
2. Mybatis-plus 提供了**分页接口**，而且是物理分页，不必在自己实现。
3. 提供了代码生成工具，可以一键生成Mapper， xml， Service， Controller
4. 3.0.7增加了lambda推导列名 不用很麻烦的写("xxx"),也避免了输入出错的可能性。

我的总结就是减少了大量的单表SQL，而且提供了一些不错的插件，还有主键生成器，性能分析拦截器（用于输出每条 SQL 语句及其执行时间），支持多数据源。

缺点可能是会存在一些**Bug**，Mapper层和Service层都继承了 mybatis-plus的类，对自己的扩展有一些限制。

注意

1. 代码生成工具pom需要引入模板依赖

```
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.7</version>
</dependency>
```

2. SpringMvc**分页**需要在Mybatis的配置文件加入以下的配置

```
<plugins> <!-- | 分页插件配置 | 插件提供二种方言选择：1、默认方言 2、自定义方言实现类，两者均未配置则抛出异常！ | overflowCurrent 溢出总页数，设置第一页 默认false | optimizeType Count优化方式 （版本2.0.9 改为使用 jsqparser 不需要配置） | --> <!-- 注意！！如果要支持二级缓存分页使用类CachePaginationInterceptor 默认、建议如下！！ --> <plugin
interceptor="com.baomidou.mybatisplus.plugins.PaginationInterceptor"> <property
name="sqlParser" ref="自定义解析类、可以没有" /> <property name="localPage" value="默认 false
改为 true 开启了 pageHelper 支持、可以没有" /> <property name="dialectClazz" value="自定义方言
类、可以没有" /> </plugin> </plugins>
```

3. 注意 xml文件如果不在resources文件下，可能不会被编译

```
<resources>
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>**/*.xml</include>
    </includes>
    <filtering>true</filtering>
  </resource>
</resources>
```