

一、insert

1、插入操作

```
1 public class CRUDTests {
2
3     @Autowired
4     private UserMapper userMapper;
5
6     @Test
7     public void testInsert(){
8
9         User user = new User();
10        user.setName("Helen");
11        user.setAge(18);
12        user.setEmail("55317332@qq.com");
13
14        int result = userMapper.insert(user);
15        System.out.println(result); //影响的行数
16        System.out.println(user); //id自动回填
17    }
18 }
```

注意：数据库插入id值默认为：全局唯一id

id	name	age	email
1	Jone	18	test1@baomidou.com
2	Jack	20	test2@baomidou.com
3	Tom	28	test3@baomidou.com
4	Sandy	21	test4@baomidou.com
5	Billie	24	test5@baomidou.com
1058193612635922434	Helen	18	55317332@qq.com

2、主键策略

(1) ID_WORKER

MyBatis-Plus默认的主键策略是：ID_WORKER 全局唯一ID

参考资料：分布式系统唯一ID生成方案汇总：

<https://www.cnblogs.com/haoxinyue/p/5208136.html>

(2) 自增策略

- 要想主键自增需要配置如下主键策略
 - 需要在创建数据表的时候设置主键自增
 - 实体字段中配置 @TableId(type = IdType.AUTO)

```
1 @TableId(type = IdType.AUTO)
2 private Long id;
```

其它主键策略：分析 IdType 源码可知

```
1 public enum IdType {
2     /**
3      * 数据库ID自增
4      */
5     AUTO(0),
6
7     /**
8      * 该类型为未设置主键类型
9      */
10    NONE(1),
11
12    /**
13     * 用户输入ID
14     * 该类型可以通过自己注册自动填充插件进行填充
15     */
16    INPUT(2),
17
18    /**
19     * 全局唯一ID
20     */
```

```

21     ASSIGN_ID(3),
22
23     /**
24      * 全局唯一ID (UUID)
25      */
26     ASSIGN_UUID(4),
27
28     /** @deprecated */
29     @Deprecated
30     ID_WORKER(3),
31     /** @deprecated */
32     @Deprecated
33     ID_WORKER_STR(3),
34     /** @deprecated */
35     @Deprecated
36     UUID(4);
37     private final int key;
38     private IdType(int key) {
39         this.key = key;
40     }
41     public int getKey() {
42         return this.key;
43     }
44 }

```

二、update

1、根据Id更新操作

注意：update时生成的sql自动是动态sql：UPDATE user SET age=? WHERE id=?

```

1     @Test
2     public void testUpdateById(){
3
4         User user = new User();
5         user.setId(1L);
6         user.setAge(28);
7

```

```
8         int result = userMapper.updateById(user);
9         System.out.println(result);
10
11     }
```

2、自动填充

项目中经常会遇到一些数据，每次都使用相同的方式填充，例如记录的创建时间，更新时间等。我们可以使用MyBatis Plus的自动填充功能，完成这些字段的赋值工作：

(1) 数据库表中添加自动填充字段

在User表中添加datetime类型的新的字段 `create_time`、`update_time`

(2) 实体上添加注解

```
1 @Data
2 public class User {
3     .....
4
5     @TableField(fill = FieldFill.INSERT)
6     private Date createTime;
7
8     //@TableField(fill = FieldFill.UPDATE)
9     @TableField(fill = FieldFill.INSERT_UPDATE)
10    private Date updateTime;
11 }
```

(3) 实现元对象处理器接口

注意：不要忘记添加 `@Component` 注解

```
1 @Component
2 public class MyMetaObjectHandler implements MetaObjectHandler {
3
```

```

4      @Override
5      public void insertFill(MetaObject metaObject) {
6          this.setFieldValByName("createTime", new Date(), metaObject);
7          this.setFieldValByName("updateTime", new Date(), metaObject);
8      }
9
10     @Override
11     public void updateFill(MetaObject metaObject) {
12         this.setFieldValByName("updateTime", new Date(), metaObject);
13     }
14 }

```

(4) 测试

3、乐观锁

主要适用场景：当要更新一条记录的时候，希望这条记录没有被别人更新，也就是说实现线程安全的数据更新

乐观锁实现方式：

- 取出记录时，获取当前version
- 更新时，带上这个version
- 执行更新时， set version = newVersion where version = oldVersion
- 如果version不对，就更新失败

(1) 数据库中添加version字段

```
1 ALTER TABLE `user` ADD COLUMN `version` INT
```

version	int	11	0	<input type="checkbox"/>	
---------	-----	----	---	--------------------------	--

(2) 实体类添加version字段

并添加 @Version 注解

```

1 @Version
2 @TableField(fill = FieldFill.INSERT)
3 private Integer version;

```

(3) 元对象处理器接口添加version的insert默认值

```

1 @Override
2 public void insertFill(MetaObject metaObject) {
3     .....
4     this.setFieldValByName("version", 1, metaObject);
5 }

```

特别说明:

- 支持的数据类型只有 int,Integer,long,Long,Date,Timestamp,LocalDateTime
- 整数类型下 `newVersion = oldVersion + 1`
- `newVersion` 会回写到 `entity` 中
- 仅支持 `updateById(id)` 与 `update(entity, wrapper)` 方法
- 在 `update(entity, wrapper)` 方法下, `wrapper` 不能复用!!!

(4) 在 MybatisPlusConfig 中注册 Bean

创建配置类

```

1 package com.atguigu.mybatisplus.config;
2
3 import com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor;
4 import org.mybatis.spring.annotation.MapperScan;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.transaction.annotation.EnableTransactionManagement;
8
9 @EnableTransactionManagement
10 @Configuration
11 @MapperScan("com.atguigu.mybatis_plus.mapper")
12 public class MybatisPlusConfig {
13

```

```

14     /**
15      * 乐观锁插件
16      */
17     @Bean
18     public OptimisticLockerInterceptor optimisticLockerInterceptor() {
19         return new OptimisticLockerInterceptor();
20     }
21 }

```

(5) 测试乐观锁可以修改成功

测试后分析打印的sql语句，将version的数值进行了加1操作

```

1  /**
2   * 测试 乐观锁插件
3   */
4  @Test
5  public void testOptimisticLocker() {
6
7      //查询
8      User user = userMapper.selectById(1L);
9      //修改数据
10     user.setName("Helen Yao");
11     user.setEmail("helen@qq.com");
12     //执行更新
13     userMapper.updateById(user);
14 }

```

(5) 测试乐观锁修改失败

```

1  /**
2   * 测试乐观锁插件 失败
3   */
4  @Test
5  public void testOptimisticLockerFail() {
6
7      //查询
8      User user = userMapper.selectById(1L);
9      //修改数据
10     user.setName("Helen Yao1");

```

```
11     user.setEmail("helen@qq.com1");
12
13     //模拟取出数据后，数据库中version实际数据比取出的值大，即已被其它线程修改并更新了version
14     user.setVersion(user.getVersion() - 1);
15
16     //执行更新
17     userMapper.updateById(user);
18 }
```

三、select

1、根据id查询记录

```
1 @Test
2 public void testSelectById(){
3
4     User user = userMapper.selectById(1L);
5     System.out.println(user);
6 }
```

2、通过多个id批量查询

完成了动态sql的foreach的功能

```
1 @Test
2 public void testSelectBatchIds(){
3
4     List<User> users = userMapper.selectBatchIds(Arrays.asList(1, 2, 3));
5     users.forEach(System.out::println);
6 }
```


3、简单的条件查询

通过map封装查询条件

```
1 @Test
2 public void testSelectByMap(){
3
4     HashMap<String, Object> map = new HashMap<>();
5     map.put("name", "Helen");
6     map.put("age", 18);
7     List<User> users = userMapper.selectByMap(map);
8
9     users.forEach(System.out::println);
10 }
```

注意：map中的key对应的是数据库中的列名。例如数据库user_id，实体类是userId，这时map的key需要填写user_id

4、分页

MyBatis Plus自带分页插件，只要简单的配置即可实现分页功能

(1) 创建配置类

此时可以删除主类中的 `@MapperScan` 扫描注解

```
1 /**
2  * 分页插件
3  */
4 @Bean
5 public PaginationInterceptor paginationInterceptor() {
6     return new PaginationInterceptor();
7 }
```

(2) 测试selectPage分页

测试：最终通过page对象获取相关数据

```

1 @Test
2 public void testSelectPage() {
3
4     Page<User> page = new Page<>(1,5);
5     userMapper.selectPage(page, null);
6
7     page.getRecords().forEach(System.out::println);
8     System.out.println(page.getCurrent());
9     System.out.println(page.getPages());
10    System.out.println(page.getSize());
11    System.out.println(page.getTotal());
12    System.out.println(page.hasNext());
13    System.out.println(page.hasPrevious());
14 }

```

控制台sql语句打印: SELECT id,name,age,email,create_time,update_time FROM user LIMIT 0,5

(3) 测试selectMapsPage分页: 结果集是Map

```

1 @Test
2 public void testSelectMapsPage() {
3     //Page不需要泛型
4     Page<Map<String, Object>> page = new Page<>(1, 5);
5     Page<Map<String, Object>> pageParam = userMapper.selectMapsPage(page, null);
6     List<Map<String, Object>> records = pageParam.getRecords();
7     records.forEach(System.out::println);
8     System.out.println(pageParam.getCurrent());
9     System.out.println(pageParam.getPages());
10    System.out.println(pageParam.getSize());
11    System.out.println(pageParam.getTotal());
12    System.out.println(pageParam.hasNext());
13    System.out.println(pageParam.hasPrevious());
14 }

```

四、delete

1、根据id删除记录

```
1 @Test
2 public void testDeleteById(){
3
4     int result = userMapper.deleteById(8L);
5     System.out.println(result);
6 }
```

2、批量删除

```
1 @Test
2 public void testDeleteBatchIds() {
3
4     int result = userMapper.deleteBatchIds(Arrays.asList(8, 9, 10));
5     System.out.println(result);
6 }
```

3、简单的条件查询删除

```
1 @Test
2 public void testDeleteByMap() {
3
4     HashMap<String, Object> map = new HashMap<>();
5     map.put("name", "Helen");
6     map.put("age", 18);
7
8     int result = userMapper.deleteByMap(map);
9     System.out.println(result);
10 }
```

4、逻辑删除

- **物理删除：真实删除**，将对应数据从数据库中删除，之后查询不到此条被删除数据
- **逻辑删除：假删除**，将对应数据中代表是否被删除字段状态修改为“被删除状态”，之后在数据库中仍旧能看到此条数据记录

(1) 数据库中添加 deleted 字段

```
1 ALTER TABLE `user` ADD COLUMN `deleted` boolean
```

deleted	tinyint	1	0	<input type="checkbox"/>	
---------	---------	---	---	--------------------------	--

(2) 实体类添加deleted 字段

并加上 @TableLogic 注解 和 @TableField(fill = FieldFill.INSERT) 注解

```
1 @TableLogic
2 @TableField(fill = FieldFill.INSERT)
3 private Integer deleted;
```

(3) 元对象处理器接口添加deleted的insert默认值

```
1 @Override
2 public void insertFill(MetaObject metaObject) {
3     .....
4     this.setFieldValByName("deleted", 0, metaObject);
5 }
```

(4) application.properties 加入配置

此为默认值，如果你的默认值和mp默认的一样,该配置可无

```
1 mybatis-plus-global-config.db-config.logic-delete-value=1
2 mybatis-plus-global-config.db-config.logic-not-delete-value=0
```

(5) 测试逻辑删除

- 测试后发现，数据并没有被删除，deleted字段的值由0变成了1
- 测试后分析打印的sql语句，是一条update
- **注意：**被删除数据的deleted 字段的值必须是 0，才能被选取出来执行逻辑删除的操作

```
1 /**
2  * 测试 逻辑删除
3  */
4 @Test
5 public void testLogicDelete() {
6
7     int result = userMapper.deleteById(1L);
8     System.out.println(result);
9 }
```

(7) 测试逻辑删除后的查询

MyBatis Plus中查询操作也会自动添加逻辑删除字段的判断

```
1 /**
2  * 测试 逻辑删除后的查询：
3  * 不包括被逻辑删除的记录
4  */
5 @Test
6 public void testLogicDeleteSelect() {
7     User user = new User();
8     List<User> users = userMapper.selectList(null);
9     users.forEach(System.out::println);
10 }
```

测试后分析打印的sql语句，包含 WHERE deleted=0

`SELECT id,name,age,email,create_time,update_time,deleted FROM user WHERE deleted=0`

五、其它

如果想进行复杂条件查询，那么需要使用条件构造器 Wrapper，涉及到如下方法

- 1、 delete
- 2、 selectOne
- 3、 selectCount
- 4、 selectList
- 5、 selectMaps
- 6、 selectObjs
- 7、 update