MyBatisPlus

1. MyBatisPlus概述

- MyBatisPlus可以节省我们大量的工作时间，所有的CRUD代码它都可以自动化完成。
- **简化MyBatis**: MyBatis-Plus是一个MyBatis的增强工具，在MyBatis的基础上只做增强不做改变。
- **强大的CRUD操作**，以后简单的CRUD操作，它不需要自己编写了
- **内置代码生成器**
- **内置分页插件**

2. 快速入门

传统方式： pojo - dao(连接mybatis, 配置mapper.xml文件) - service - controller

mybatis-plus后：

- pojo

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
/**
* @author Kai
* @date 2020/4/21 17:02
*/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
  private Long id;
  private String name;
  private Integer age;
  private String email;
}
```

- mapper接口

```
/**
* @author Kai
* @date 2020/4/21 17:04
*/
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.zth.pojo.User;
import org.springframework.stereotype.Repository;
// 相当于DAO层
// 在对应的Mapper上面继承基本的类BaseMapper
```

```
@Repository //代表持久层
public interface UserMapper extends BaseMapper<User> {
  //所有的CRUD操作都已经编写完成了
}
```

- 使用 测试类

```
@SpringBootTest
class MybatisPlusApplicationTests {
  // 继承了BaseMapper 所有的方法都来自于父类,
  // 我们也可以编写自己的扩展方法
  @Autowired
  private UserMapper userMapper;

  @Test
  void contextLoads() {
      //查询全部用户
      //参数是一个Wrapper 条件构造器
      List<User> users = userMapper.selectList(null);
      users.forEach(System.out::println);
  }
}
```

- 注意 我们需要在主启动类上去扫描我们的mapper包下的所有接口

```
@MapperScan("com.zth.mapper") //扫描mapper文件夹
@SpringBootApplication
public class MybatisPlusApplication {
  public static void main(String[] args) {
      SpringApplication.run(MybatisPlusApplication.class, args);
  }
}
```

- 结果

*User(id=1, name=Jone, age=18, email=**test1@baomidou.com**)*
*User(id=2, name=Jack, age=20, email=**test2@baomidou.com**)*
*User(id=3, name=Tom, age=28, email=**test3@baomidou.com**)*
*User(id=4, name=Sandy, age=21, email=**test4@baomidou.com**)*
*User(id=5, name=Billie, age=24, email=**test5@baomidou.com**)*

---

3. 配置日志

所有的sql现在是不可见的，我们希望知道它是怎么执行的，所以在开发时我们需要日志。

```
#  数据库连接配置
spring.datasource.username=root
spring.datasource.password=
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis_plus?
useUnicode=true&&characterEncoding=utf8&useSSL=false&serverTimezone=UTC
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# 配置日志  输出到控制台
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

日志输出

```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@642413d4] was not
registered for synchronization because synchronization is not active
JDBC Connection [HikariProxyConnection@421632334 wrapping
com.mysql.cj.jdbc.ConnectionImpl@43c87306] will not be managed by Spring
==>  Preparing: SELECT id,name,age,email FROM user
==> Parameters:
<==    Columns: id, name, age, email
<==        Row: 1, Jone, 18, test1@baomidou.com
<==        Row: 2, Jack, 20, test2@baomidou.com
<==        Row: 3, Tom, 28, test3@baomidou.com
<==        Row: 4, Sandy, 21, test4@baomidou.com
<==        Row: 5, Billie, 24, test5@baomidou.com
<==      Total: 5
Closing non transactional SqlSession
[org.apache.ibatis.session.defaults.DefaultSqlSession@642413d4]
User(id=1, name=Jone, age=18, email=test1@baomidou.com)
User(id=2, name=Jack, age=20, email=test2@baomidou.com)
User(id=3, name=Tom, age=28, email=test3@baomidou.com)
User(id=4, name=Sandy, age=21, email=test4@baomidou.com)
User(id=5, name=Billie, age=24, email=test5@baomidou.com)
```

---

4. CRUD扩展

插入操作

> Insert 插入

```
//测试插入
@Test
void testInsertUSer(){
    User user = new User();
    user.setName("zth");
    user.setAge(21);
    user.setEmail("kai");

    int result = userMapper.insert(user);    //帮我们自动生成id
    System.out.println(result);              // 输出： 1    （受影响的行数）
    System.out.println("user = " + user);    //输出：user = User(id=1252532505785446401,
name=zth, age=21, email=kai)
}
```

> 数据库插入的id的默认值为：全局的唯一id

主键生成策略

- **雪花算法：**snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），最后还有一个符号位，永远是0，几乎可以保证全球唯一

```
public class User {
    // 对应数据库的主键 （uuid、自增id、雪花算法、redis、zookeeper）
    @TableId(type = IdType.ID_WORKER)  //采用全局唯一的id生成策略
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

其余的源码解释：

```
public enum IdType {
    AUTO(0),            //数据库id自增
    NONE(1),            //未设置主键
    INPUT(2),           //手动输入
    ID_WORKER(3),       //默认的全局唯一id
    UUID(4),            //全局唯一id uuid
    ID_WORKER_STR(5);   //ID_WORKER 字符串表示法
}
```

更新操作

```java
//测试更新
@Test
public void testUpdate(){
    User user = new User();
    // 通过条件自动拼接动态sql
    user.setId(4L);
    user.setName("updated name");
    // 注意 updateById 但是参数是一个对象
    int result = userMapper.updateById(user);
    System.out.println(result);
}
```

自动填充

创建时间、修改时间

阿里巴巴开发手册：所有的数据库表：gmt_create、gmt_modified几乎所有的表都要配置上，而且需要自动化

> 方式一：数据库级别 （工作中不建议修改数据库）

1. 在表中新增字段create_time, update_time

字段名	类型	默认值
create_time	datetime	CURRENT_TIMESTAMP
update_time	datetime	CURRENT_TIMESTAMP 并勾选根据当前时间戳更新

> 方式二：代码级别

1. 删除数据库create_time、 update_time默认值
2. 实体类的字段属性上需要增加注解

```java
//字段添加填充内容
@TableField(fill = FieldFill.INSERT)  //插入时更新
private Date createTime;

@TableField(fill = FieldFill.INSERT_UPDATE) //插入以及更新时更新
private Date updateTime;
```

3. 编写处理器来处理注解

```java
import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import lombok.extern.slf4j.Slf4j;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.stereotype.Component;

import java.util.Date;

/**
 * @author Kai
 * @date 2020/4/22 16:02
 */
@Slf4j
@Component
```

```
public class MyMetaObjectHandler implements MetaObjectHandler {
    //插入时的填充策略
    @Override
    public void insertFill(MetaObject metaObject) {
        log.info("start insert fill ...");
        this.setFieldValByName("createTime", new Date(), metaObject);
        this.setFieldValByName("updateTime", new Date(), metaObject);
    }

    //更新时的填充策略
    @Override
    public void updateFill(MetaObject metaObject) {
        log.info("start update fill ...");
        this.setFieldValByName("updateTime", new Date(), metaObject);
    }
}
```

4. 测试插入、更新　create_time update_time 值都会更新

乐观锁

（面试常常被问到）

> 乐观锁：顾名思义十分乐观，它总是认为不会出现问题，无论干什么不去上锁！如果出现问题，再次更新值测试。
>
> 悲观锁：顾名思义十分悲观，它认为总是会出现问题，无论做什么都会上锁，再去操作。

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

```
乐观锁：1.先查询，获得版本号 version=1
-- A线程
update user set name = "zth", version = version + 1
where id = 2 and version = 1
-- B线程 抢先完成，这个时候version=2，会导致A修改失败，保证线程通信安全
update user set name = "zth", version = version + 1
where id = 2 and version = 1
```

> 测试一下Mybatis-Plus的乐观锁插件

1. 给数据库中增加version字段
2. 实体类加对应的字段

```
@Version//乐观锁注解
private Integer version;
```

3. 注册组件

```
@EnableTransactionManagement
@Configuration
public class MyBatisPlusConfig {
    //注册乐观锁插件
    @Bean
    public OptimisticLockerInterceptor optimisticLockerInterceptor() {
        return new OptimisticLockerInterceptor();
    }
}
```

4. 测试

```
@Test
public void testOptimisticLocker2(){
    //线程A
    User user = userMapper.selectById(1L);
    user.setName("kai");
    user.setEmail("990211");

    // 模拟另一个线程B执行了插队操作
    User user2 = userMapper.selectById(1L);
    user.setName("kai222");
    user.setEmail("9902112222");
    userMapper.updateById(user2);

    userMapper.updateById(user); //线程A更新 如果没有乐观锁就会覆盖插队线程B的值
}
```

查询操作

```
//测试批量查询
@Test
public void testSelectByBatchId(){
    List<User> ul = userMapper.selectBatchIds(Arrays.asList(1, 2,3));
    ul.forEach(System.out::println);
}

// 按条件查询之一 使用map操作
@Test
public void testSelectByBatchIds(){
    HashMap<String, Object> map = new HashMap<>();
    map.put("name", "zth");
    map.put("age", 21);

    List<User> ul = userMapper.selectByMap(map);
    ul.forEach(System.out::println);
}
```

分页查询

1. 原始的limit进行分页
2. pageHelper第三方插件
3. MyBatisPlusn也内置了分页插件

> 如何使用

1. 配置拦截器组件

```
@Bean
public PaginationInterceptor paginationInterceptor() {
    PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
    // 设置请求的页面大于最大页后操作， true调回到首页，false 继续请求  默认false
    // paginationInterceptor.setOverflow(false);
    // 设置最大单页限制数量，默认 500 条，-1 不受限制
    // paginationInterceptor.setLimit(500);
    // 开启 count 的 join 优化,只针对部分 left join
    paginationInterceptor.setCountSqlParser(new JsqlParserCountOptimize(true));
    return paginationInterceptor;
}
```

这里可以直接返回

```
//分页插件
@Bean
public PaginationInterceptor paginationInterceptor() {
    return new PaginationInterceptor();
}
```

2. 直接使用Page对象

```
//测试分页查询
@Test
public void testPage(){
    // 参数一：当前页
    // 参数二：页面大小
    Page<User> userpage = new Page<>(2, 5); //查询第一页 的五个数据
    userMapper.selectPage(userpage, null);

    userpage.getRecords().forEach(System.out::println);
    System.out.println("数据库中的数据总条数" + userpage.getTotal());
}
```

删除操作

```
//测试删除
@Test
public void testDeleteById(){
    userMapper.deleteById(4L);
}

//批量id批量删除
@Test
public void testDeleteBatchId(){
    userMapper.deleteBatchIds(Arrays.asList(1,2));
}

//通过map删除
@Test
public void testDeleteMap(){
    HashMap<String, Object> map = new HashMap<>();
    map.put("name", "Tom");
    userMapper.deleteByMap(map);
}
```

逻辑删除

> 物理删除：从数据库中直接移除
>
> 逻辑删除：在数据库中没有被移除，而是通过一个变量来让它失效
>
> deleted = 0 => deleted = 1

管理可以查看被删除的记录！ 防止数据的丢失，类似于回收站

1. 在数据表中增加一个deleted字段 (int 默认值为0)
2. 实体类中增加属性

```
@TableLogic //逻辑删除
private int deleted;
```

3. 配置

```
//逻辑删除组件
@Bean
public ISqlInjector sqlInjector() {
    return new LogicSqlInjector();
}
```

```
# 配置逻辑删除  没有删除是0 删除了是1
mybatis-plus.global-config.db-config.logic-delete-value=1
mybatis-plus.global-config.db-config.logic-not-delete-value=0
```

4. 测试

运行testDeleteById() 和testSelectById()

5. 性能分析插件

MyBatis-Plus提供了性能分析插件，如果超过了这个时间就停止运行

作用：性能分析拦截器，用于输出每条 SQL 语句及其执行时间

1. 导入插件

```
/**
     * SQL执行效率插件
     */
@Bean
@Profile({"dev","test"})// 设置 dev test 环境开启
public PerformanceInterceptor performanceInterceptor() {
    return new PerformanceInterceptor();
}
```

```
# 设置开发环境
# spring.profiles.active=dev
```

6. 条件查询器 Wrapper

写一些复杂的sql就可以使用它替代

```
@Test
void contextLoads() {
    //查询name不为空，并且邮箱不为空的用户，年龄大于12岁
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper
        .isNotNull("name")
        .isNotNull("email")
        .ge("age", 12);
    userMapper.selectList(wrapper).forEach(System.out::println);
}


@Test
void test2(){
    // 查询名字zth
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.eq("name", "zth");
    User user = userMapper.selectOne(wrapper);  // 查询一个数据， 出现多个结果使用List或Map
    System.out.println(user);
}

@Test
void test3(){
    // 查询年龄在 20~30岁的用户
    QueryWrapper<User> wrapper = new QueryWrapper();
    wrapper.between("age", 20, 30); //区间
    Integer count = userMapper.selectCount(wrapper);
    System.out.println(count);
}

//模糊查询
@Test
void test4(){
    QueryWrapper<User> wrapper = new QueryWrapper();
    wrapper
        .notLike("name", "e")
        .likeRight("email", "t");
    List<Map<String, Object>> maps = userMapper.selectMaps(wrapper);
    maps.forEach(System.out::println);
}
```

7. 代码自动生成器

AutoGenerator 是 MyBatis-Plus 的代码生成器，通过 AutoGenerator 可以快速生成 Entity、Mapper、Mapper XML、Service、Controller 等各个模块的代码，极大的提升了开发效率。

```
package com.zth;

import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.annotation.FieldFill;
import com.baomidou.mybatisplus.annotation.IdType;
import com.baomidou.mybatisplus.generator.AutoGenerator;
import com.baomidou.mybatisplus.generator.config.DataSourceConfig;
```

```java
import com.baomidou.mybatisplus.generator.config.GlobalConfig;
import com.baomidou.mybatisplus.generator.config.PackageConfig;
import com.baomidou.mybatisplus.generator.config.StrategyConfig;
import com.baomidou.mybatisplus.generator.config.po.TableFill;
import com.baomidou.mybatisplus.generator.config.rules.DateType;
import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;

import java.util.ArrayList;

/**
 * @author Kai
 * @date 2020/4/23 0:14
 */


//代码自动生成器
public class CodeGenerator {
    public static void main(String[] args) {
        //需要构建一个 代码自动生成器 对象
        AutoGenerator mpg = new AutoGenerator();
        //配置策略

        //1. 全局配置
        GlobalConfig gc = new GlobalConfig();
        String projectPath = System.getProperty("user.dir");
        gc.setOutputDir(projectPath + "/src/main/java");
        gc.setAuthor("zth");
        gc.setOpen(false);
        gc.setFileOverride(false); //是否覆盖
        gc.setServiceName("%sService"); // 去Service的I前缀
        gc.setIdType(IdType.AUTO); //主键策略
        gc.setDateType(DateType.ONLY_DATE); //日期类型
        gc.setSwagger2(true);
        mpg.setGlobalConfig(gc);

        //2. 设置数据源
        DataSourceConfig dsc = new DataSourceConfig();
        dsc.setUrl("jdbc:mysql://localhost:3306/mybatis_plus?
useUnicode=true&&characterEncoding=utf8&useSSL=false&serverTimezone=UTC");
        dsc.setDriverName("com.mysql.cj.jdbc.Driver");
        dsc.setUsername("root");
        dsc.setPassword("");
        dsc.setDbType(DbType.MYSQL);
        mpg.setDataSource(dsc);

        //3. 包配置
        PackageConfig pc = new PackageConfig();
        pc.setModuleName("module"); //模块名
        pc.setParent("com.zth");
        pc.setEntity("pojo");
        pc.setMapper("mapper");
        pc.setService("service");
        pc.setController("controller");
        mpg.setPackageInfo(pc);

        //4. 策略配置
        StrategyConfig strategy = new StrategyConfig();
        strategy.setInclude("user","table"); // 设置要映射的表名
```

```
        strategy.setNaming(NamingStrategy.underline_to_camel);  //设置包命名的规则 下划线转驼峰命名
        strategy.setColumnNaming(NamingStrategy.underline_to_camel); //设置列的命名
        strategy.setEntityLombokModel(true); //自动lombok
        strategy.setLogicDeleteFieldName("deleted"); //逻辑删除自动配置
        /* 自动填充策略 */
        TableFill createTime = new TableFill("create_time", FieldFill.INSERT);
        TableFill modifiedTime = new TableFill("modified_time", FieldFill.INSERT_UPDATE);
        ArrayList<TableFill> tableFills = new ArrayList<>();
        tableFills.add(createTime);
        tableFills.add(modifiedTime);
        strategy.setTableFillList(tableFills);
        /* 乐观锁 */
        strategy.setVersionFieldName("version");
        /* 有关Controller层的 */
        strategy.setRestControllerStyle(true); //开启Restful的驼峰命名格式
        strategy.setControllerMappingHyphenStyle(true); //eg.localhost:8080/hello_id_2
        mpg.setStrategy(strategy);

        mpg.execute(); //执行
    }
}
```