1 MyBatis-Plus简介

Mybatis-Plus官网：https://baomidou.com/

MyBatis-Plus（简称 MP）是一个 MyBatis的增强工具，在 MyBatis 的基础上只做增强不做改变，为 简化开发、提高效率而生。

> 愿景:
>
> 我们的愿景是成为 MyBatis 最好的搭档，就像 魂斗罗 中的 1P、2P，基友搭配，效率翻倍。

1.1 Mybatis-Plus的特性

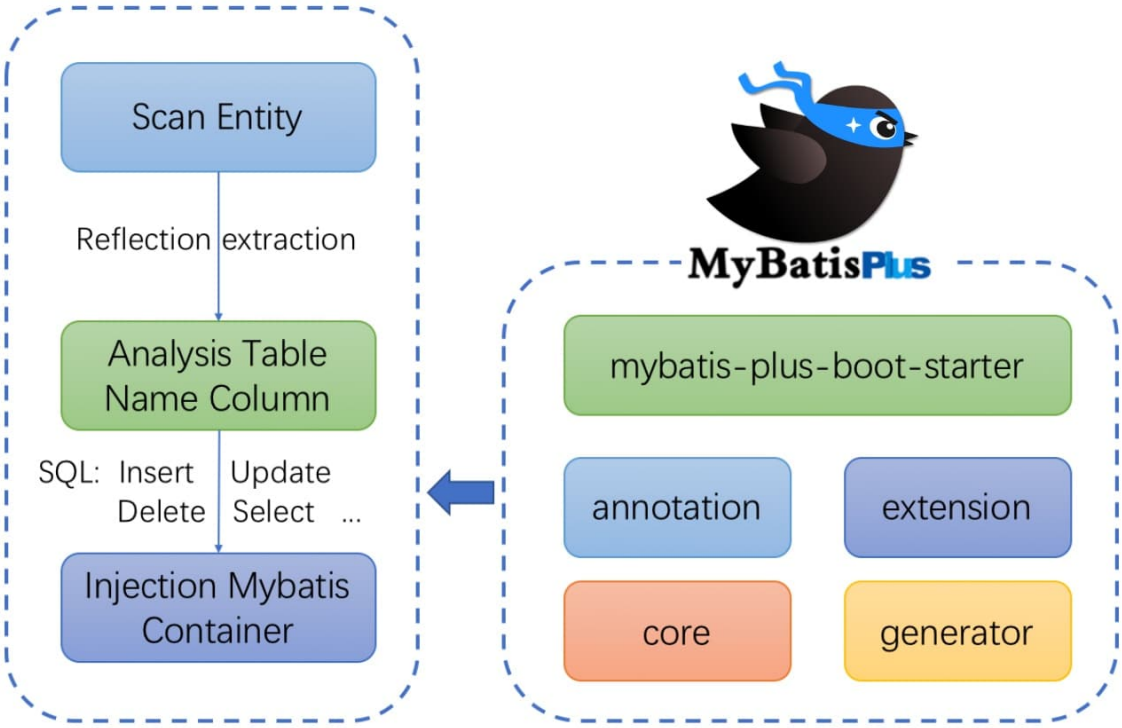
- **无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑。
- **损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作。
- **强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求。
- **支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错。
- **支持主键自动生成**：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题。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

1.2 Mybatis-Plus支持的数据库

任何能使用 MyBatis进行 CRUD, 并且支持标准 SQL 的数据库，具体支持情况如下，如果不在下列表查看分页部分教程 PR 您的支持。

- MySQL，Oracle，DB2，H2，HSQL，SQLite，PostgreSQL，SQLServer，Phoenix，Gauss，ClickHouse，Sybase，OceanBase，Firebird，Cubrid，Goldilocks，csiidb，informix，TDengine，redshift
- 达梦数据库，虚谷数据库，人大金仓数据库，南大通用(华库)数据库，南大通用数据库，神通数据库，瀚高数据库，优炫数据库，星瑞格数据库

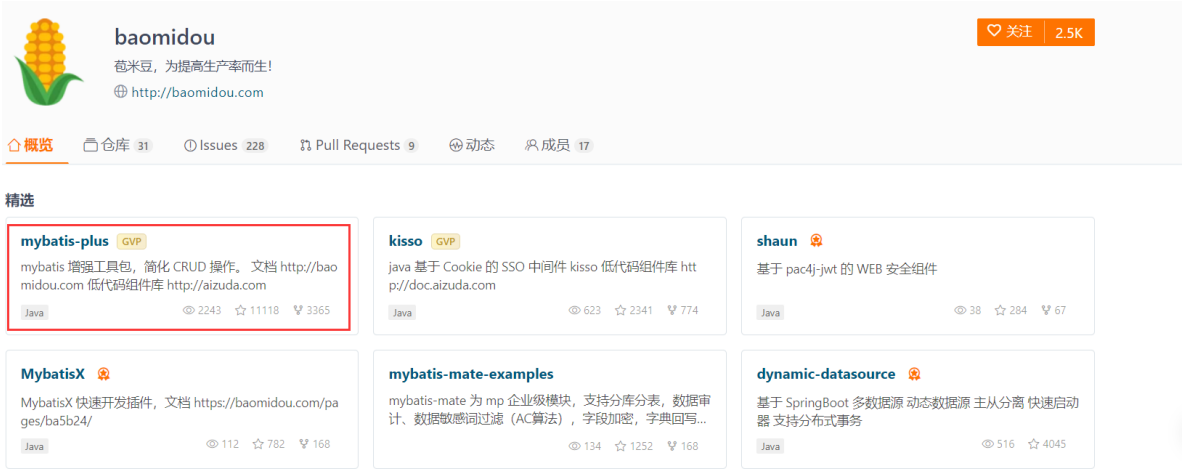
1.3 Mybatis-Plus的整体架构



1.4 Mybatis-Plus的开发作者

Mybatis-Plus是由baomidou（苞米豆）组织开发并且开源的，目前该组织大概有17人左右。

码云地址：https://gitee.com/organizations/baomidou



2 Mybatis-Plus环境搭建

对于Mybatis整合MP有常常有三种用法，分别是Mybatis-Plus、Spring+Mybatis-Plus、SpringBoot+Mybatis-Plus。我们先用前两种。

SpringBoot整合Mybatis-Plus是我们课程的核心。

2.1 Mybatis-Plus单独使用

第一步：创建数据表

```
CREATE TABLE `user` (
`id` BIGINT(20) NOT NULL COMMENT '主键ID',
`name` VARCHAR(30) DEFAULT NULL COMMENT '姓名',
`age` INT(11) DEFAULT NULL COMMENT '年龄',
`email` VARCHAR(50) DEFAULT NULL COMMENT '邮箱',
PRIMARY KEY (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8;

INSERT INTO USER (id, NAME, age, email) VALUES
(1, 'Jone', 18, 'jone@baomidou.com'),
(2, 'Jack', 20, 'jack@baomidou.com'),
(3, 'Tom', 28, 'tom@baomidou.com'),
(4, 'Sandy', 21, 'sandy@baomidou.com'),
(5, 'Billie', 24, 'billie@baomidou.com');
```

第二步：创建工程引入依赖

创建一个maven工程，工程名为mybatis-plus-quickstart。然后引入相关依赖：

```
<dependencies>
    <!-- mybatis-plus插件依赖 -->
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus</artifactId>
        <version>3.1.0</version>
    </dependency>
    <!-- MySql -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
    </dependency>
    <!-- 连接池 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.0.11</version>
    </dependency>
    <!--简化bean代码的工具包-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
        <version>1.18.4</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
```

```
        <version>1.6.4</version>
    </dependency>
</dependencies>
```

为了方便查看测试效果，引入log4j.properties配置文件：

```
log4j.rootLogger=DEBUG,A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%t] [%c]-[%p] %m%n
```

第三步：编写实体类及Mapper接口

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private Long id;
    private String name;
    private int age;
    private String email;

}
```

```
public interface UserMapper{

    // 方便在mybatis-plus环境下面测试原生mybatis的使用
    public User findById(Long id);
}
```

第四步：引入Mybatis的核心配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://192.168.10.140:3306/dbtest1"/>
                <property name="username" value="root"/>
                <property name="password" value="Admin123!"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
```

```
        <package name="com.xq.mapper"></package>
    </mappers>
</configuration>
```

为了在Mybatis-plus的环境下面测试原生Mybatis的使用，所以我们也引入Mybatis的映射文件。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xq.mapper.UserMapper">

    <select id="findById" parameterType="long" resultType="com.xq.pojo.User">
        select * from user where id = #{id}
    </select>
</mapper>
```

第五步：测试原生Mybatis的功能

```
public class AppTest {

    //使用原生的Mybatis测试
    @Test
    public void test01() throws Exception{
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(in);
        SqlSession sqlSession = factory.openSession();
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        User user = userMapper.findById(1L);
        System.out.println(user);
        sqlSession.close();
    }
}
```

第六步：测试Mybatis-Plus的使用

- 修改接口文件

```
package com.xq.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.xq.pojo.User;
                                        接口中的泛型是实体类的数据类型
6 usages
public interface UserMapper extends BaseMapper<User> {

    1 usage
    public User findById(Long id);          BaseMapper中封装了单表操作的方法
}
```

- 编写测试类

```
@Test
public void test02() throws Exception{
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    MybatisSqlSessionFactoryBuilder builder = new MybatisSqlSessionFactoryBuilder();
    SqlSessionFactory factory = builder.build(in);
    SqlSession sqlSession = factory.openSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user = userMapper.selectById(1L);
    System.out.println(user);                   mybatis-plus框架提供的方法，我们自己无需实现。
    sqlSession.close();
}
```

思考：为什么我们的接口继承了BaseMapper，就不用写单表的CRUD方法了?

首先我们来看一下BaseMapper中的方法：

```
package com.baomidou.mybatisplus.core.mapper;
public interface BaseMapper<T> extends Mapper<T> {
    /**
    * 插入一条记录
    * @param entity 实体对象
    */
    int insert(T entity);

    /**
    * 根据 ID 删除
    * @param id 主键ID
    */
    int deleteById(Serializable id);

    /**
    * 根据实体(ID)删除
    * @param entity 实体对象
    * @since 3.4.4
    */
    int deleteById(T entity);

    /**
    * 根据 columnMap 条件，删除记录
    * @param columnMap 表字段 map 对象
    */
    int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);

    /**
    * 根据 entity 条件，删除记录
    * @param queryWrapper 实体对象封装操作类（可以为 null,里面的 entity 用于生成 where
    语句）
    */
    int delete(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

    /**
    * 删除（根据ID 批量删除）
    * @param idList 主键ID列表(不能为 null 以及 empty)
    */
    int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends
Serializable> idList);

    /**
    * 根据 ID 修改
    * @param entity 实体对象
    */
    int updateById(@Param(Constants.ENTITY) T entity);

    /**
    * 根据 whereEntity 条件，更新记录
    * @param entity 实体对象 (set 条件值,可以为 null)
    * @param updateWrapper 实体对象封装操作类（可以为 null,里面的 entity 用于生成
    where 语句）
    */
```

```
    int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER)
Wrapper<T> updateWrapper);

    /**
    * 根据 ID 查询
    * @param id 主键ID
    */
    T selectById(Serializable id);

    /**
    * 查询（根据ID 批量查询）
    * @param idList 主键ID列表(不能为 null 以及 empty)
    */
    List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends
Serializable> idList);

    /**
    * 查询（根据 columnMap 条件）
    * @param columnMap 表字段 map 对象
    */
    List<T> selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object>
columnMap);

    /**
    * 根据 entity 条件，查询一条记录
    * <p>查询一条记录，例如 qw.last("limit 1") 限制取一条记录, 注意：多条数据会报异常</p>
    * @param queryWrapper 实体对象封装操作类（可以为 null）
    */
    default T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper) {
        List<T> ts = this.selectList(queryWrapper);
        if (CollectionUtils.isNotEmpty(ts)) {
            if (ts.size() != 1) {
                throw ExceptionUtils.mpe("One record is expected, but the query
result is multiple records");
            }
            return ts.get(0);
        }
        return null;
    }

    /**
    * 根据 Wrapper 条件，查询总记录数
    * @param queryWrapper 实体对象封装操作类（可以为 null）
    */
    Long selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

    /**
    * 根据 entity 条件，查询全部记录
    * @param queryWrapper 实体对象封装操作类（可以为 null）
    */
    List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

    /**
    * 根据 Wrapper 条件，查询全部记录
    * @param queryWrapper 实体对象封装操作类（可以为 null）
    */
    List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T>
queryWrapper);
```

```
    /**
     * 根据 Wrapper 条件，查询全部记录
     * <p>注意： 只返回第一个字段的值</p>
     * @param queryWrapper 实体对象封装操作类（可以为 null）
     */
    List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

    /**
     * 根据 entity 条件，查询全部记录（并翻页）
     * @param page 分页查询条件（可以为 RowBounds.DEFAULT）
     * @param queryWrapper 实体对象封装操作类（可以为 null）
     */
    <P extends IPage<T>> P selectPage(P page, @Param(Constants.WRAPPER)
Wrapper<T> queryWrapper);

    /**
     * 根据 Wrapper 条件，查询全部记录（并翻页）
     * @param page 分页查询条件
     * @param queryWrapper 实体对象封装操作类
     */
    <P extends IPage<Map<String, Object>>> P selectMapsPage(P page,
@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
}
```

接下来我们打断点跟踪一下。

```
    @Test
    public void test02() throws Exception{
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");  in: BufferedInputStream@2235
        MybatisSqlSessionFactoryBuilder builder = new MybatisSqlSessionFactoryBuilder();  builder: MybatisSqlSessionFactoryBuilder@2236
        SqlSessionFactory factory = builder.build(in);  in: BufferedInputStream@2235  builder: MybatisSqlSessionFactoryBuilder@2236
        SqlSession sqlSession = factory.openSession();  factory: DefaultSqlSessionFactory@2237  sqlSession: DefaultSqlSession@2252
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  sqlSession: DefaultSqlSession@2252
        User user = userMapper.selectById(1L);
        System.out.println(user);
        sqlSession.close();
    }
}
```

找到mappedStatements:

简单说明：

由于使用了MybatisSqlSessionFactoryBuilder进行了构建，继承的BaseMapper中的方法就载入到了SqlSession中，所以就可以直接使用相关的方法；

2.2 在Spring环境下面使用Mybatis-Plus

创建工程mybatis-plus-spring，然后引入如下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- 连接池 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.2.8</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.27</version>
    </dependency>
    <!-- 日志 -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.30</version>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
    <!-- lombok用来简化实体类 -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.16</version>
    </dependency>
    <!--MyBatis-Plus的核心依赖-->
```

```
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus</artifactId>
        <version>3.4.3.4</version>
    </dependency>
</dependencies>
```

第二步：创建实体类和接口

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private Long id;
    private String name;
    private int age;
    private String email;

}
```

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

}
```

第三步：创建配置文件

- 创建数据库相关的配置文件db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://192.168.10.140:3306/dbtest1
jdbc.username=root
jdbc.password=Admin123!
```

- 创建spring核心配置文件applicationContext.xml，用来整合mybatis-plus

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">
    <!-- 引入jdbc.properties -->
    <context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>

    <!-- 配置Druid数据源 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>
```

```
    <!-- 配置用于创建SqlSessionFactory的工厂bean -->
    <bean
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
        <!-- 设置MyBatis配置文件的路径（可以不设置） -->
       <!-- <property name="configLocation" value="classpath:mybatis-
config.xml">
        </property>-->
        <!-- 设置数据源 -->
        <property name="dataSource" ref="dataSource"></property>
        <!-- 设置类型别名所对应的包 -->
        <property name="typeAliasesPackage" value="com.xq.pojo">
        </property>
        <!--
        设置映射文件的路径
        若映射文件所在路径和mapper接口所在路径一致，则不需要设置
        -->
        <!--
        <property name="mapperLocations" value="classpath:mapper/*.xml">
        </property>
        -->
    </bean>

    <!--
    配置mapper接口的扫描配置
    由mybatis-spring提供，可以将指定包下所有的mapper接口创建动态代理
    并将这些动态代理作为IOC容器的bean管理
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.xq.mapper"></property>
    </bean>
</beans>
```

- 创建日志文件logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <!--定义日志文件的存储地址 logs为当前项目的logs目录 还可以设置为../logs -->
    <property name="LOG_HOME" value="logs" />
    <!--控制台日志， 控制台输出 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符
            宽度,%msg：日志消息，%n是换行符-->
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}
                - %msg%n</pattern>
        </encoder>
    </appender>
    <!--myibatis log configure-->
    <logger name="com.apache.ibatis" level="TRACE"/>
    <logger name="java.sql.Connection" level="DEBUG"/>
    <logger name="java.sql.Statement" level="DEBUG"/>
    <logger name="java.sql.PreparedStatement" level="DEBUG"/>
    <!-- 日志输出级别 -->
    <root level="DEBUG">
        <appender-ref ref="STDOUT" />
    </root>
```

```
</configuration>
```

第四步：测试mybatis-plus的使用

```
//在Spring的环境中进行测试
@RunWith(SpringJUnit4ClassRunner.class)
//指定Spring的配置文件
@ContextConfiguration("classpath:applicationContext.xml")

public class AppTest {

    @Autowired
    private UserMapper userMapper;
    @Test
    public void testMyBatisBySpring(){
        List<User> userList = userMapper.selectList(null);
        userList.forEach(user ->{
            System.out.println(user);
        });
    }
}
```

2.3 在Springboot环境下面使用Mybatis-plus

第一步:创建工程引入相关依赖

创建工程springboot-mybatis-plus-project,并引入如下依赖:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.2</version>
    <relativePath/>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
```

```
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-boot-starter</artifactId>
        <version>3.5.1</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

第二步：编写启动类

```
@SpringBootApplication
@MapperScan(basePackages = "com.xq.mapper")
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class,args);
    }
}
```

第三步：编写实体类及相关的接口

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

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

第四步：编写配置文件

```yaml
spring:
  # 配置数据源信息
  datasource:
    # 配置数据源类型
    type: com.zaxxer.hikari.HikariDataSource
    # 配置连接数据库信息
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://192.168.10.140:3306/dbtest1
    username: root
    password: Admin123!
```

注意：

> **1. 驱动类 driver-class-name**
>
> spring boot 2.0（内置jdbc5驱动），驱动类使用： driver-class-name: com.mysql.jdbc.Driver
>
> spring boot 2.1及以上（内置jdbc8驱动），驱动类使用：driver-class-name: com.mysql.cj.jdbc.Driver
>
> 否则运行测试用例的时候会有 WARN 信息
>
> **2. 连接地址 url**
>
> MySQL5.7版本的url： jdbc:mysql://localhost:3306/mybatis_plus?characterEncoding=utf-8&useSSL=false
>
> MySQL8.0版本的url： jdbc:mysql://localhost:3306/mybatis_plus?serverTimezone=GMT%2B8&characterEncoding=utf-8&useSSL=false
>
> 否则运行测试用例报告如下错误：
>
> java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more

第五步：测试mybatis-plus

```java
@SpringBootTest
public class MybatisPlusTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testSelectList(){
        //selectList()根据MP内置的条件构造器查询一个list集合，null表示没有条件，即查询所有
        userMapper.selectList(null).forEach(System.out::println);
    }
}
```

经过测试，我们发现我们springboot整合mybatis-plus没有问题。但是和在spring环境下面使用mybatis-plus相比，少了日志相关的功能，我们也可以在springboot环境下面配置日志的整合，这样在测试的时候，可以在控制台下面观测mybatis-plus解析sql语句的具体细节。

```yaml
# 配置MyBatis日志
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

```
JDBC Connection [HikariProxyConnection@847141861 wrapping com.mysql.cj.jdbc.ConnectionImpl@1084ac45] will not be managed by Spring
==>  Preparing: SELECT id,name,age,email FROM user
==> Parameters:
<==    Columns: id, name, age, email
<==        Row: 1, Jone, 18, jone@baomidou.com
<==        Row: 2, Jack, 20, jack@baomidou.com
<==        Row: 3, Tom, 28, tom@baomidou.com
<==        Row: 4, Sandy, 21, sandy@baomidou.com
<==        Row: 5, Billie, 24, billie@baomidou.com
<==      Total: 5
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@48eb001a]
User(id=1, name=Jone, age=18, email=jone@baomidou.com)
User(id=2, name=Jack, age=20, email=jack@baomidou.com)
User(id=3, name=Tom, age=28, email=tom@baomidou.com)
User(id=4, name=Sandy, age=21, email=sandy@baomidou.com)
User(id=5, name=Billie, age=24, email=billie@baomidou.com)
2024-04-12 14:11:19.895  INFO 6308 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2024-04-12 14:11:19.932  INFO 6308 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

3 Mybatis-Plus基本CRUD操作

3.1 插入操作

```java
@Test
public void testInsert(){
    User user = new User(null, "张三", 23, "zhangsan@qq.com");
    //INSERT INTO user ( id, name, age, email ) VALUES ( ?, ?, ?, ? )
    int result = userMapper.insert(user);
    System.out.println("受影响行数："+result);
    //1778676561893969922
    System.out.println("id自动获取："+user.getId());
}
```

> 最终执行的结果，所获取的id为1778676561893969922
>
> 这是因为MyBatis-Plus在实现插入数据时，会默认基于雪花算法的策略生成id。如果我们想让我们的id实现自增策略，在后续我们使用@TableId注解来实现。

3.2 删除操作

在Mybatis-Plus中删除的方式有3种，一种是根据id删除；还有一种是根据id批量删除；最后还可以根据条件删除。

3.2.1 通过id删除

deleteById方法：根据id删除记录。

```java
@Test
public void testDeleteById(){
    //通过id删除用户信息
    //DELETE FROM user WHERE id=?
    int result = userMapper.deleteById(1778676561893969922L);
    System.out.println("受影响行数："+result);
}
```

3.2.2 通过id批量删除记录

deleteBatchIds方法：根据id批量删除记录。

```
@Test
public void testDeleteBatchIds(){
    //通过多个id批量删除
    //DELETE FROM user WHERE id IN ( ? , ? , ? )
    List<Long> idList = Arrays.asList(1L, 2L, 3L);
    int result = userMapper.deleteBatchIds(idList);
    System.out.println("受影响行数："+result);
}
```

3.2.3 通过map条件删除记录

deleteByMap方法：将条件封装成map。根据条件删除记录。

```
@Test
public void testDeleteByMap(){
    //根据map集合中所设置的条件删除记录
    //DELETE FROM user WHERE name = ? AND age = ?
    Map<String, Object> map = new HashMap<>();
    map.put("age", 23);
    map.put("name", "张三");
    int result = userMapper.deleteByMap(map);
    System.out.println("受影响行数："+result);
}
```

3.3 修改操作

在Mybatis-Plus中，更新操作有2种，一种是根据id更新，另一种是根据条件更新。

3.3.1 根据id更新

updateById方法: 根据id进行修改

```
@Test
public void testUpdateById(){
    User user = new User(4L, "admin", 22, null);
    //UPDATE user SET name=?, age=? WHERE id=?
    int result = userMapper.updateById(user);
    System.out.println("受影响行数："+result);
}
```

3.3.2 根据条件更新

update方法：根据条件修改用户信息。

```
@Test
public void testUpdateUser1(){
    User user = new User();
    user.setAge(22); //更新的字段
    user.setEmail("admin@qq.com");
    //更新的条件
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.eq("id", 4L);
    //执行更新操作 UPDATE user SET age=?, email=? WHERE (id = ?)
    int result = this.userMapper.update(user, wrapper);
    System.out.println("受影响行数：" + result);
}
```

还可以这么写：

```
@Test
public void testUpdate2() {
    //更新的条件以及字段
    UpdateWrapper<User> wrapper = new UpdateWrapper<>();
    wrapper.eq("id", 4L).set("age", 28).set("email","admin@163.com");
    //执行更新操作 UPDATE user SET age=?,email=? WHERE (id = ?)
    int result = this.userMapper.update(null, wrapper);
    System.out.println("受影响行数：" + result);
}
```

3.4 查询操作

用户查询操作也有多种场景，比如可以根据id查询；根据id进行批量查询；查询所有信息；根据条件查询指定的信息。

3.4.1 根据主键id查询

selectById方法:根据主键id查询

```
@Test
public void testSelectById(){
    //根据id查询用户信息
    //SELECT id,name,age,email FROM user WHERE id=?
    User user = userMapper.selectById(4L);
    System.out.println(user);
}
```

3.4.2 根据多个id查询多个用户信息

selectBatchIds: 根据id批量查询

```
@Test
public void testSelectBatchIds(){
    //根据多个id查询多个用户信息
    //SELECT id,name,age,email FROM user WHERE id IN ( ? , ? )
    List<Long> idList = Arrays.asList(4L, 5L);
    List<User> list = userMapper.selectBatchIds(idList);
    list.forEach(System.out::println);
}
```

3.4.3 通过map条件查询用户信息

selectByMap: 将查询条件封装成map集合进行查询。

```
@Test
public void testSelectByMap(){
    //通过map条件查询用户信息
    //SELECT id,name,age,email FROM user WHERE name = ? AND age = ?
    Map<String, Object> map = new HashMap<>();
    map.put("age", 20);
    map.put("name", "Jack");
    List<User> list = userMapper.selectByMap(map);
    list.forEach(System.out::println);
}
```

3.4.4 查询所有数据

selectList: 查询所有记录，前提是selectList方法中的条件为null

```
@Test
public void testSelectList(){
    //查询所有用户信息
    //SELECT id,name,age,email FROM user
    List<User> list = userMapper.selectList(null);
    list.forEach(System.out::println);
}
```

4 通用Service

什么是通用Service，官方给出的解释是：

通用 Service CRUD 封装IService接口，进一步封装 CRUD 采用get 查询单行 remove 删除 list 查询集合 page 分页前缀命名方式区分 Mapper 层避免混淆。
1、泛型 T 为任意实体对象
2、建议如果存在自定义通用 Service 方法的可能，请创建自己的 IBaseService 继承 Mybatis-Plus 提供的基类

```
public interface IService<T> {
    no usages
    int DEFAULT_BATCH_SIZE = 1000;

    default boolean save(T entity) { return SqlHelper.retBool(this.getBaseMapper().insert(entity)); }

    no usages
    @Transactional(
        rollbackFor = {Exception.class}
    )
    default boolean saveBatch(Collection<T> entityList) { return this.saveBatch(entityList, batchSize: 1000); }

    no usages  1 implementation
    boolean saveBatch(Collection<T> entityList, int batchSize);
```

IService接口被ServiceImpl所实现：

```
public class ServiceImpl<M extends BaseMapper<T>, T> implements IService<T> {
    protected Log log = LogFactory.getLog(this.getClass());
    no usages
    @Autowired
    protected M baseMapper;
    no usages
    protected Class<T> entityClass = this.currentModelClass();
    no usages
    protected Class<M> mapperClass = this.currentMapperClass();

    no usages
    public ServiceImpl() {
    }

    no usages
    public M getBaseMapper() { return this.baseMapper; }

    no usages
    public Class<T> getEntityClass() { return this.entityClass; }

    /** @deprecated */
    no usages
    @Deprecated
    protected boolean retBool(Integer result) { return SqlHelper.retBool(result); }
```

第一步：创建业务接口：

```
/**
 * UserService继承IService模板提供的基础功能
 */
public interface UserService extends IService<User> {

}
```

第二步：创建业务接口实现类：

```
/**
 * ServiceImpl实现了IService，提供了IService中基础功能的实现
 * 若ServiceImpl无法满足业务需求，则可以使用自定的UserService定义方法，并在实现类中实现
 */
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {

}
```

接下来我们来测试通用Service业务接口。

(1) 测试查询次数

```
@Autowired
private UserService userService;

@Test
public void testGetCount(){
    long count = userService.count();
    System.out.println("总记录数：" + count);
}
```

(2) 测试批量插入

```
@Test
public void testSaveBatch(){
    ArrayList<User> users = new ArrayList<>();
    for (int i = 0; i < 5; i++) {
        User user = new User();
        user.setName("abc" + i);
        user.setAge(20 + i);
        users.add(user);
    }
    // SQL:INSERT INTO t_user ( username, age ) VALUES ( ?, ? ),(?,?)
    userService.saveBatch(users);
}
```

(3) 修改操作

```
@Test
public void testUpdate(){
    User user = new User(1778690964123090947L,"miller",30,"miller@qq.com");
    // SQL:UPDATE user SET name=?, age=?, email=? WHERE id=?
    userService.updateById(user);
}
```

(4) 删除操作

```
@Test
public void testDelete(){
    List<Long> ids = new ArrayList<>();
    ids.add(1778690963879821313L);
    ids.add(1778690964114702337L);
    ids.add(1778690964114702338L);
    ids.add(1778690964123090946L);
    ids.add(1778690964123090947L);
    // SQL:DELETE FROM user WHERE id=?
    userService.removeBatchByIds(ids);
}
```

5 Mybatis-Plus的常用注解

5.1 @TableName注解

经过以上的测试，在使用MyBatis-Plus实现基本的CRUD时，我们并没有指定要操作的表，只是在Mapper接口继承BaseMapper时，设置了泛型User，而操作的表为user表。由此得出结论，**MyBatis-Plus在确定操作的表时，由BaseMapper的泛型决定，即实体类型决定，且默认操作的表名和实体类型的类名一致。**

问题：

若实体类类型的类名和要操作的表的表名不一致，会出现什么问题？我们将表user更名为t_user，测试查询功能。程序抛出异常，Table 'mybatis_plus.user' doesn't exist，因为现在的表名为tb_user，而默认操作的表名和实体类型的类名一致，即user表

执行查询操作，我们发现报错：

```
org.springframework.jdbc.BadSqlGrammarException:
### Error querying database.  Cause: java.sql.SQLSyntaxErrorException: Table 'dbtest1.user' doesn't exist
### The error may exist in com/xq/mapper/UserMapper.java (best guess)
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: SELECT id,name,age,email FROM user WHERE id=?
### Cause: java.sql.SQLSyntaxErrorException: Table 'dbtest1.user' doesn't exist
; bad SQL grammar []; nested exception is java.sql.SQLSyntaxErrorException: Table 'dbtest1.user' doesn't exist
```

此时解决方案有两种：

(1) 我们可以实体@TableName注解解决问题。

在实体类类型上添加@TableName("tb_user")，标识实体类对应的表，即可成功执行SQL语句

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("tb_user")
public class User {

    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

(2) 通过全局配置解决问题

在开发的过程中，我们经常遇到以上的问题，即实体类所对应的表都有固定的前缀，例如`t_`或`tbl_`此时，可以使用MyBatis-Plus提供的全局配置，为实体类所对应的表名设置默认的前缀，那么就 不需要在每个实体类上通过@TableName标识实体类对应的表

```
# 配置MyBatis日志
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      # 配置MyBatis-Plus操作表的默认前缀
      table-prefix: tb_
```

5.2 @TableId注解

前面我们说过，MyBatis-Plus在实现CRUD时，会默认将id作为主键列，并在插入数据时，默认 基于雪花算法的策略生成id。

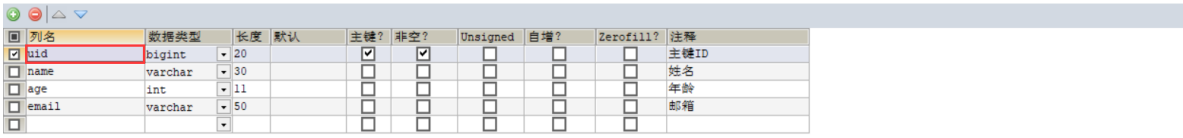
问题：

若实体类和表中表示主键的不是id，而是其他字段，例如uid，MyBatis-Plus会自动识别uid为主 键列吗？ 我们实体类中的属性id改为uid，将表中的字段id也改为uid，测试添加功能。

此时程序抛出异常，Field 'uid' doesn't have a default value，说明MyBatis-Plus没有将uid作为主键赋值

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    private Long uid; // 将id修改成uid，使实体类和数据表字段对不上。
    private String name;
    private Integer age;
    private String email;
}
```

紧接着我们将数据表中的字段也改成uid



列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	注释
uid	bigint	20		☑	☑	☐	☐	☐	主键ID
name	varchar	30		☐	☐	☐	☐	☐	姓名
age	int	11		☐	☐	☐	☐	☐	年龄
email	varchar	50		☐	☐	☐	☐	☐	邮箱
				☐	☐	☐	☐	☐	

我们再执行插入操作：

```
org.springframework.dao.DataIntegrityViolationException:
### Error updating database.  Cause: java.sql.SQLException: Field 'id' doesn't have a default value
### The error may exist in com/xq/mapper/UserMapper.java (best guess)
### The error may involve com.xq.mapper.UserMapper.insert-Inline
### The error occurred while setting parameters
### SQL: INSERT INTO tb_user  ( name, age, email )  VALUES  ( ?, ?, ? )
### Cause: java.sql.SQLException: Field 'id' doesn't have a default value
; Field 'id' doesn't have a default value; nested exception is java.sql.SQLException: Field 'id' doesn't have a default value
```

此时可以通过@TableId解决问题。

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId // 使用@TableId注解手动将插入数据的主键设置为uid
    private Long uid;
    private String name;
    private Integer age;
    private String email;
}
```

再次执行插入操作，即可解决问题。

5.2.1 @TableId的value属性

现在又遇到一个问题，如果前后字段名称不一致呢，比如实体类字段为id，数据表主键字段为uid。此时需要指定@TableId的value属性。

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId(value = "uid") //如果前后字段不一致,可以添加@TableId注解,通过value属性指定uid的值
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

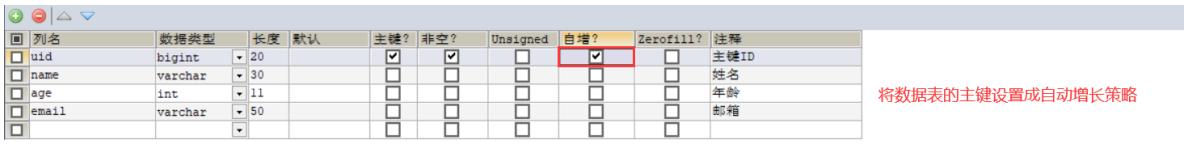
5.2.2 @TableId的type属性

type属性还可以指定主键的生成策略。在默认的情况下，我们主键基于雪花算法生成。如果此时我们不想使用雪花算法生成主键，比如想使用自增策略的主键。

首先我们修改数据表的主键为自动增长策略：

先清空表中的数据：

```
TRUNCATE TABLE tb_user;
```



接下来新增记录：

```
INSERT INTO tb_user (uid, NAME, age, email) VALUES
(1, 'Jone', 18, 'jone@baomidou.com'),
(2, 'Jack', 20, 'jack@baomidou.com'),
(3, 'Tom', 28, 'tom@baomidou.com'),
(4, 'Sandy', 21, 'sandy@baomidou.com'),
(5, 'Billie', 24, 'billie@baomidou.com');
```

此外我们还需要配置主键的增长策略：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId(value = "uid",type = IdType.AUTO) //如果前后字段不一致,可以添加@TableId注解,通过value属性指定uid的值
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

或者可以在配置文件中全局配置：

```yaml
# 配置MyBatis日志
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      # 配置MyBatis-Plus操作表的默认前缀
      table-prefix: tb_
      # 配置MyBatis-Plus的主键策略为自增长
      id-type: auto
```

常用的主键策略：

mybatis-plus通过枚举类描述主键的生成策略：

```java
public enum IdType {
    AUTO(0),
    NONE(1),
    INPUT(2),
    ASSIGN_ID(3),
    ASSIGN_UUID(4);

    private final int key;

    private IdType(int key) {
        this.key = key;
    }

    public int getKey() {
        return this.key;
    }
}
```

值	描述
AUTO	数据库 ID自增，这种情况下将表中主键设置为自增，否则，没有设置主动设置id值进行插入时会报错
NONE	无状态，该类型为未设置主键类型（注解里等于跟随全局，全局里默认ASSIGN_ID）
INPUT	insert 前自行 set 主键值，在采用IKeyGenerator类型的ID生成器时必须为INPUT
ASSIGN_ID	分配 ID(主键类型为 Number(Long 和 Integer)或 String)(since 3.3.0),使用接口IdentifierGenerator的方法nextId(默认实现类为DefaultIdentifierGenerator雪花算法)
ASSIGN_UUID	分配 UUID,主键类型为 String(since 3.3.0),使用接口IdentifierGenerator的方法nextUUID(默认 default 方法)

5.2.3 雪花算法

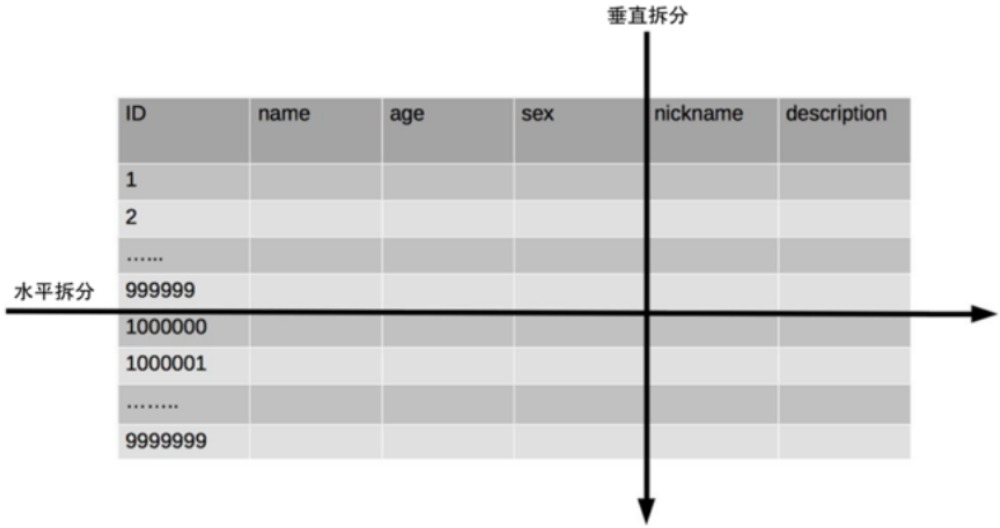
背景：

需要选择合适的方案去应对数据规模的增长，以应对逐渐增长的访问压力和数据量。 数据库的扩展方式主要包括：**业务分库、主从复制，数据库分表。**

数据库分表：

将不同业务数据分散存储到不同的数据库服务器，能够支撑百万甚至千万用户规模的业务，但如果业务继续发展，同一业务的单表数据也会达到单台数据库服务器的处理瓶颈。例如，淘宝的几亿用户数据，如果全部存放在一台数据库服务器的一张表中，肯定是无法满足性能要求的，此时就需要对单表数据进行拆分。

单表数据拆分有两种方式：垂直分表和水平分表。示意图如下：



垂直分表：

垂直分表适合将表中某些不常用且占了大量空间的列拆分出去。

例如，前面示意图中的 nickname 和 description 字段，假设我们是一个婚恋网站，用户在筛选其他用户的时候，主要是用 age 和 sex 两个字段进行查询，而 nickname 和 description 两个字段主要用于展示，一般不会在业务查询中用到。description 本身又比较长，因此我们可以将这两个字段独立到另外一张表中，这样在查询 age 和 sex 时，就能带来一定的性能提升。

水平分表：

水平分表适合表行数特别大的表，有的公司要求单表行数超过 5000 万就必须进行分表，这个数字可以作为参考，但并不是绝对标准，关键还是要看表的访问性能。对于一些比较复杂的表，可能超过 1000 万就要分表了；而对于一些简单的表，即使存储数据超过 1 亿行，也可以不分表。

但不管怎样，当看到表的数据量达到千万级别时，作为架构师就要警觉起来，因为这很可能是架构的性能瓶颈或者隐患。

水平分表相比垂直分表，会引入更多的复杂性，例如要求全局唯一的数据id该如何处理

(1)：主键自增

以最常见的用户 ID 为例，可以按照 1000000 的范围大小进行分段，1 ~ 999999 放到表 1中，1000000 ~ 1999999 放到表2中，以此类推。

① 复杂点：分段大小的选取。分段太小会导致切分后子表数量过多，增加维护复杂度；分段太大可能会导致单表依然存在性能问题，一般建议分段大小在 100 万至 2000 万之间，具体需要根据业务选取合适的分段大小。

② 优点：可以随着数据的增加平滑地扩充新的表。例如，现在的用户是 100 万，如果增加到 1000 万，只需要增加新的表就可以了，原有的数据不需要动。

③ 缺点：分布不均匀。假如按照 1000 万来进行分表，有可能某个分段实际存储的数据量只有 1 条，而另外一个分段实际存储的数据量有 1000 万条。

(2): 取模

同样以用户 ID 为例，假如我们一开始就规划了 10 个数据库表，可以简单地用 user_id % 10 的值来 表示数据所属的数据库表编号，ID 为 985 的用户放到编号为 5 的子表中，ID 为 10086 的用户放到编号为 6 的子表中。

① 复杂点：初始表数量的确定。表数量太多维护比较麻烦，表数量太少又可能导致单表性能存在问题。

② 优点：表分布比较均匀。

③ 缺点：扩充新的表很麻烦，所有数据都要重分布。

(3): 雪花算法

雪花算法是由Twitter公布的分布式主键生成算法，它能够保证不同表的主键的不重复性，以及相同表的主键的有序性。

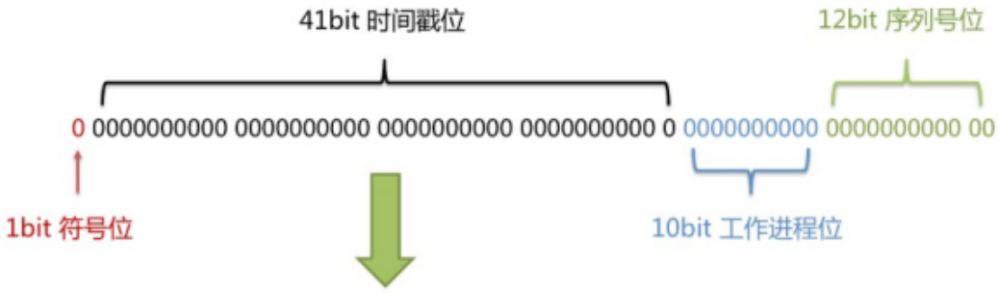
① 核心思想：

长度共64bit（一个long型）。 首先是一个符号位，1bit标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负 数是1，所以id一般是正数，最高位是0。

41bit时间截(毫秒级)，存储的是时间截的差值（当前时间截 - 开始时间截)，结果约等于69.73年。

10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID，可以部署在1024个节点）。

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）。



② 优点：整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞，并且效率较高。

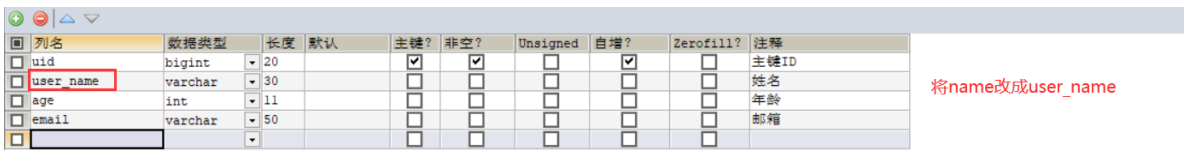
5.3 @TableField注解

经过以上的测试，我们可以发现，MyBatis-Plus在执行SQL语句时，要保证实体类中的属性名和 表中的字段名一致。如果实体类中的属性名和字段名不一致的情况，会出现什么问题呢?

第一种情况:

若实体类中的属性使用的是驼峰命名风格，而表中的字段使用的是下划线命名风格。例如实体类属性userName，表中字段user_name。

此时MyBatis-Plus会自动将下划线命名风格转化为驼峰命名风格。



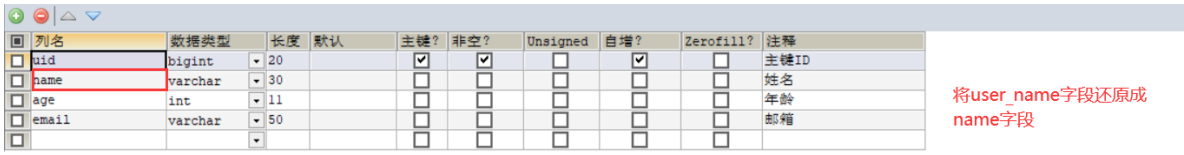
接着将实体类的字段也修改一下：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId(value = "uid")
    private Long id;
    private String userName;// 设置成驼峰命名规则
    private Integer age;
    private String email;
}
```

再执行查询操作，没有问题。

第二种情况：

若实体类中的属性和表中的字段不满足情况1。例如实体类属性name，表中字段username。此时需要在实体类属性上使用@TableField("username")设置属性所对应的字段名



列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	注释
uid	bigint	20		☑	☑	☐	☑	☐	主键ID
name	varchar	30		☐	☐	☐	☐	☐	姓名
age	int	11		☐	☐	☐	☐	☐	年龄
email	varchar	50		☐	☐	☐	☐	☐	邮箱
				☐	☐	☐	☐	☐	

将user_name字段还原成name字段

此时实体对应的字段需要使用@TableField进行手动映射：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId(value = "uid") //如果前后字段不一致,可以添加@TableId注解,通过value属性指定uid的值
    private Long id;
    @TableField("name") // 前后字段不一致,需要手动进行映射
    private String userName;
    private Integer age;
    private String email;
}
```

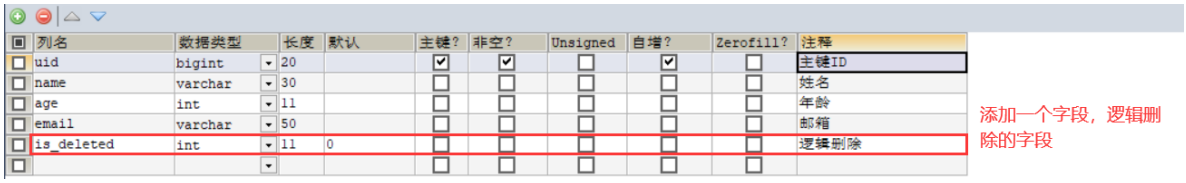
5.4 @TableLogic注解

该主键是实现逻辑删除的注解。对数据表中记录删除有两种。一种是物理删除，还有一种就是逻辑删除。

> 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的数据。
>
> 逻辑删除：假删除，将对应数据中代表是否被删除字段的状态修改为“被删除状态”，之后在数据库 中仍旧能看到此条数据记录。

下面我们使用@TableLogic注解实现逻辑删除。

第一步：数据库中创建逻辑删除状态列，设置默认值为0



列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	注释
uid	bigint	20		☑	☑	☐	☑	☐	主键ID
name	varchar	30		☐	☐	☐	☐	☐	姓名
age	int	11		☐	☐	☐	☐	☐	年龄
email	varchar	50		☐	☐	☐	☐	☐	邮箱
is_deleted	int	11	0	☐	☐	☐	☐	☐	逻辑删除
				☐	☐	☐	☐	☐	

添加一个字段，逻辑删除的字段

第二步：实体类中添加逻辑删除属性

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    @TableId(value = "uid")
    private Long id;
    @TableField("name")
    private String userName;
    private Integer age;
    private String email;

    @TableLogic // 逻辑删除的字段
    private Integer isDeleted;
}
```

第三步：测试逻辑删除

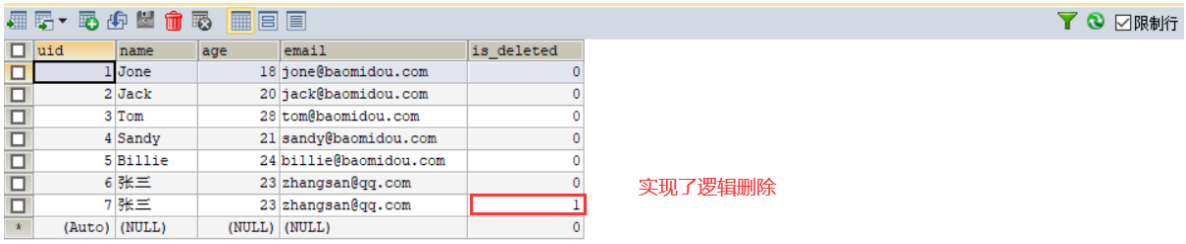
```
@Test
public void testDeleteById(){
    //通过id删除用户信息
    int result = userMapper.deleteById(7L);
    System.out.println("受影响行数："+result);
}
```

我们看看IDEA控制台解析对应的SQL语句：

```
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4fe533ff] was not registered for synchronization because synchronization is
2024-04-12 17:47:09.772  INFO 19768 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-04-12 17:47:09.779  WARN 19768 --- [           main] com.zaxxer.hikari.util.DriverDataSource  : Registered driver with driverClassName=
2024-04-12 17:47:11.102  INFO 19768 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
JDBC Connection [HikariProxyConnection@930641076 wrapping com.mysql.cj.jdbc.ConnectionImpl@14422d9d] will not be managed by Spring
==>  Preparing: UPDATE tb_user SET is_deleted=1 WHERE uid=? AND is_deleted=0
==> Parameters: 7(Long)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4fe533ff]
受影响行数：1
2024-04-12 17:47:11.158  INFO 19768 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2024-04-12 17:47:11.161  INFO 19768 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

是update语句，不是delete语句

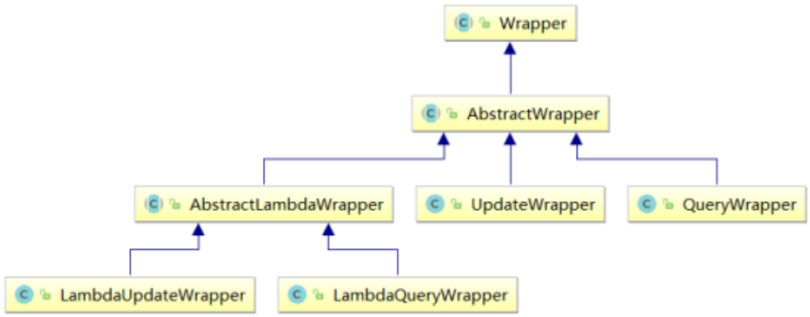
我们再看看数据库中的记录：



uid	name	age	email	is_deleted
1	Jone	18	jone@baomidou.com	0
2	Jack	20	jack@baomidou.com	0
3	Tom	28	tom@baomidou.com	0
4	Sandy	21	sandy@baomidou.com	0
5	Billie	24	billie@baomidou.com	0
6	张三	23	zhangsan@qq.com	0
7	张三	23	zhangsan@qq.com	1
(Auto)	(NULL)	(NULL)	(NULL)	0

实现了逻辑删除

6 条件构造器和常用接口



- Wrapper： 条件构造抽象类，最顶端父类
- AbstractWrapper： 用于查询条件封装，生成 sql 的 where 条件
  - QueryWrapper： 查询条件封装
  - UpdateWrapper： Update 条件封装
  - AbstractLambdaWrapper： 使用Lambda 语法
    - LambdaQueryWrapper：用于Lambda语法使用的查询Wrapper
    - LambdaUpdateWrapper： Lambda 更新封装Wrapper

6.1 QueryWrapper

(1) 组装查询条件

```
@Test
public void test01(){
    //查询用户名包含a，年龄在20到30之间，并且邮箱不为null的用户信息
    // SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE
    // is_deleted=0 AND (username LIKE ? AND age BETWEEN ? AND ? AND email IS
NOT NULL)
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.like("username", "a")
                .between("age", 20, 30)
                .isNotNull("email");
    List<User> list = userMapper.selectList(queryWrapper);
    list.forEach(System.out::println);
}
```

(2) 组装排序条件

```
@Test
public void test02(){
    // 按年龄降序查询用户，如果年龄相同则按id升序排列
    // SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE
    // is_deleted=0 ORDER BY age DESC,id ASC
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.orderByDesc("age")
                .orderByAsc("id");
    List<User> users = userMapper.selectList(queryWrapper);
    users.forEach(System.out::println);
}
```

(3) 组装删除条件

```
@Test
public void test03(){
    //删除email为空的用户
    //DELETE FROM t_user WHERE (email IS NULL)
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.isNull("email");
    //条件构造器也可以构建删除语句的条件
    int result = userMapper.delete(queryWrapper);
    System.out.println("受影响的行数：" + result);
}
```

(4) 条件的优先级

```
@Test
public void test04() {
QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    // 将（年龄大于20并且用户名中包含有a）或邮箱为null的用户信息修改
    // UPDATE t_user SET age=?, email=? WHERE (username LIKE ? AND age > ? OR
email IS NULL)
    queryWrapper.like("username", "a")
                .gt("age", 20)
                .or()
                .isNull("email");
    User user = new User();
    user.setAge(38);
    user.setEmail("kobe@qq.com");
    int result = userMapper.update(user, queryWrapper);
    System.out.println("受影响的行数：" + result);
}
```

```
@Test
public void test04() {
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    //将用户名中包含有a并且（年龄大于20或邮箱为null）的用户信息修改
    //UPDATE t_user SET age=?, email=? WHERE (username LIKE ? AND (age > ? OR
email IS NULL))
    //lambda表达式内的逻辑优先运算
    queryWrapper.like("username", "a")
                .and(i -> i.gt("age", 20)
                .or().isNull("email"));
    User user = new User();
    user.setAge(18);
    user.setEmail("oscar@163.com");
    int result = userMapper.update(user, queryWrapper);
    System.out.println("受影响的行数：" + result);
}
```

(5) 组装select子句

```
@Test
public void test05() {
    // 查询用户信息的username和age字段
    // SELECT username,age FROM t_user
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.select("username", "age");
    List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);
    maps.forEach(System.out::println);
}
```

(6) 实现子查询

```
@Test
public void test06() {
    //查询id小于等于3的用户信息
    //SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE (id
IN(select id from t_user where id <= 3))
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.inSql("id", "select id from t_user where id <= 3");
    List<User> list = userMapper.selectList(queryWrapper);
    list.forEach(System.out::println);
}
```

6.2 UpdateWrapper

```
@Test
public void test07() {
    // 将（年龄大于20或邮箱为null）并且用户名中包含有a的用户信息修改
    // 组装set子句以及修改条件
    UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();
    // lambda表达式内的逻辑优先运算
    updateWrapper
    .set("age", 18)
    .set("email", "user@qq.com")
    .like("username", "a")
    .and(i -> i.gt("age", 20).or().isNull("email"));
    // 这里必须要创建User对象，否则无法应用自动填充。如果没有自动填充，可以设置为null
    // UPDATE t_user SET username=?, age=?,email=? WHERE (username LIKE ? AND
(age > ? OR email IS NULL))
    // User user = new User();
    // user.setName("张三");
    // int result = userMapper.update(user, updateWrapper);
    // UPDATE t_user SET age=?,email=? WHERE (username LIKE ? AND (age > ? OR
email IS NULL))
    int result = userMapper.update(null, updateWrapper);
    System.out.println(result);
}
```

6.3 Condition

在真正开发的过程中，组装条件是常见的功能，而这些条件数据来源于用户输入，是可选的，因 此我们在组装这些条件时，必须先判断用户是否选择了这些条件，若选择则需要组装该条件，若 没有选择则一定不能组装，以免影响SQL执行的结果。

思路一：

```
@Test
public void test08() {
    // 定义查询条件，有可能为null（用户未输入或未选择）
    String username = null;
    Integer ageBegin = 10;
    Integer ageEnd = 24;
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    // StringUtils.isNotBlank()判断某字符串是否不为空且长度不为0且不由空白符
(whitespace)
    构成
    if(StringUtils.isNotBlank(username)){
    queryWrapper.like("username","a");
    }
    if(ageBegin != null){
    queryWrapper.ge("age", ageBegin);
    }
    if(ageEnd != null){
    queryWrapper.le("age", ageEnd);
    }
    // SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE (age >=
? AND age <= ?)
    List<User> users = userMapper.selectList(queryWrapper);
    users.forEach(System.out::println);
}
```

思路二：

上面的实现方案没有问题，但是代码比较复杂，我们可以使用带condition参数的重载方法构建查 询条件，简化代码的编写。

```
@Test
public void test08UseCondition() {
    // 定义查询条件，有可能为null（用户未输入或未选择）
    String username = null;
    Integer ageBegin = 10;
    Integer ageEnd = 24;
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    // StringUtils.isNotBlank()判断某字符串是否不为空且长度不为0且不由空白符
(whitespace)构成
    queryWrapper.like(StringUtils.isNotBlank(username), "username", "a")
                .ge(ageBegin != null, "age", ageBegin)
                .le(ageEnd != null, "age", ageEnd);
    // SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE (age >=
? AND age <= ?)
    List<User> users = userMapper.selectList(queryWrapper);
    users.forEach(System.out::println);
}
```

6.4 LambdaQueryWrapper

```
@Test
public void test09() {
    // 定义查询条件，有可能为null（用户未输入）
    String username = "a";
    Integer ageBegin = 10;
    Integer ageEnd = 24;
```

```
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    // 避免使用字符串表示字段，防止运行时错误
    queryWrapper
    .like(StringUtils.isNotBlank(username), User::getName, username)
    .ge(ageBegin != null, User::getAge, ageBegin)
    .le(ageEnd != null, User::getAge, ageEnd);
    List<User> users = userMapper.selectList(queryWrapper);
    users.forEach(System.out::println);
}
```

6.5 LambdaUpdateWrapper

```
@Test
public void test10() {
    // 组装set子句
    LambdaUpdateWrapper<User> updateWrapper = new LambdaUpdateWrapper<>();
    updateWrapper.set(User::getAge, 18)
                 .set(User::getEmail, "user@atguigu.com")
                 .like(User::getName, "a")
                  // lambda表达式内的逻辑优先运算
                 .and(i -> i.lt(User::getAge, 24).or().isNull(User::getEmail));
    User user = new User();
    int result = userMapper.update(user, updateWrapper);
    System.out.println("受影响的行数：" + result);
}
```

7 插件

7.1 分页插件

MyBatis Plus自带分页插件，只要简单的配置即可实现分页功能

(1) 添加配置类

```
@Configuration
public class MybatisPlusConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new
        PaginationInnerInterceptor(DbType.MYSQL));
        return interceptor;
    }
}
```

(2) 测试

```
@Test
public void testPage(){
    //设置分页参数
    Page<User> page = new Page<>(1, 5);
    userMapper.selectPage(page, null);
    //获取分页数据
    List<User> list = page.getRecords();
    list.forEach(System.out::println);
```

```
        System.out.println("当前页："+page.getCurrent());
        System.out.println("每页显示的条数："+page.getSize());
        System.out.println("总记录数："+page.getTotal());
        System.out.println("总页数："+page.getPages());
        System.out.println("是否有上一页："+page.hasPrevious());
        System.out.println("是否有下一页："+page.hasNext());
    }
```

测试结果如下：

```
JDBC Connection [HikariProxyConnection@1958242673 wrapping com.mysql.cj.jdbc.ConnectionImpl@1f9d4b0e] will not be managed by Spring
==>  Preparing: SELECT COUNT(*) AS total FROM tb_user WHERE is_deleted = 0
==> Parameters:
<==    Columns: total
<==        Row: 6
<==      Total: 1
==>  Preparing: SELECT uid AS id,name AS userName,age,email,is_deleted FROM tb_user WHERE is_deleted=0 LIMIT ?
==> Parameters: 5(Long)
<==    Columns: id, userName, age, email, is_deleted
<==        Row: 1, Jone, 18, jone@baomidou.com, 0
<==        Row: 2, Jack, 20, jack@baomidou.com, 0
<==        Row: 3, Tom, 28, tom@baomidou.com, 0
<==        Row: 4, Sandy, 21, sandy@baomidou.com, 0
<==        Row: 5, Billie, 24, billie@baomidou.com, 0
<==      Total: 5
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@234cff57]
User(id=1, userName=Jone, age=18, email=jone@baomidou.com, isDeleted=0)
User(id=2, userName=Jack, age=20, email=jack@baomidou.com, isDeleted=0)
User(id=3, userName=Tom, age=28, email=tom@baomidou.com, isDeleted=0)
User(id=4, userName=Sandy, age=21, email=sandy@baomidou.com, isDeleted=0)
User(id=5, userName=Billie, age=24, email=billie@baomidou.com, isDeleted=0)
当前页：1
每页显示的条数：5
总记录数：6
总页数：2
是否有上一页：false
是否有下一页：true
```

7.2 xml自定义分页

(1) UserMapper中定义接口方法

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

    /**
     * 根据年龄查询用户列表，分页显示
     * @param page 分页对象,xml中可以从里面进行取值,传递参数 Page 即自动分页,必须放在第一位
     * @param age 年龄
     * @return
     */

    Page<User> selectPageVo(@Param("page") Page<User> page, @Param("age") Integer age);
}
```

(2) UserMapper.xml中编写SQL

在resources目录下面创建/com/xq/mapper目录，在目录里面创建UserMapper.xml。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xq.mapper.UserMapper">

    <!--SQL片段，记录基础字段-->
    <sql id="BaseColumns">uid,name,age,email</sql>
```

```
    <!--IPage<User> selectPageVo(Page<User> page, Integer age);-->
    <select id="selectPageVo" resultType="com.xq.pojo.User">
        SELECT <include refid="BaseColumns"></include> FROM tb_user WHERE age >
#{age}
    </select>
</mapper>
```

(3) 测试分页

```
@Test
public void testSelectPageVo(){
    //设置分页参数
    Page<User> page = new Page<>(1, 5);
    userMapper.selectPageVo(page, 20);
    //获取分页数据
    List<User> list = page.getRecords();
    list.forEach(System.out::println);
    System.out.println("当前页："+page.getCurrent());
    System.out.println("每页显示的条数："+page.getSize());
    System.out.println("总记录数："+page.getTotal());
    System.out.println("总页数："+page.getPages());
    System.out.println("是否有上一页："+page.hasPrevious());
    System.out.println("是否有下一页："+page.hasNext());
}
```

7.3 SQL性能分析插件

性能分析拦截器，用于输出每条 SQL 语句及其执行时间，可以设置最大执行时间，超过时间会抛出异常。

该插件只用于开发环境，不建议生产环境使用。

我们就在spring整合mybatis-plus的环境里面使用这个插件。由于高版本的mybatis-plus移除了这个插件的使用。所以我们需要将mybatis的版本下调：

```
 <dependency>
     <groupId>com.baomidou</groupId>
     <artifactId>mybatis-plus</artifactId>
     <version>3.1.1</version>
</dependency>
```

创建mybatis-plus的核心配置文件mybatis-config.xml，并引入SQL性能分析插件：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <plugins>
        <!-- SQL 执行性能分析，开发环境使用，线上不推荐。 maxTime 指的是 sql 最大执行时长
-->
        <plugin
interceptor="com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor">
            <property name="maxTime" value="100" />
            <!--SQL是否格式化 默认false-->
            <property name="format" value="true" />
```

```
        </plugin>
    </plugins>
</configuration>
```

然后在applicationContext.xml中引入配置文件：

```
<!-- 配置用于创建SqlSessionFactory的工厂 bean -->
<bean class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
    <!-- 设置MyBatis配置文件的路径（可以不设置） -->
    <property name="configLocation" value="classpath:mybatis-config.xml"></property>
    <!-- 设置数据源 -->
    <property name="dataSource" ref="dataSource"></property>
    <!-- 设置类型别名所对应的包 -->
    <property name="typeAliasesPackage" value="com.xq.pojo">
    </property>
    <!--
    设置映射文件的路径
    若映射文件所在路径和mapper接口所在路径一致，则不需要设置
    -->
    <!--
    <property name="mapperLocations" value="classpath:mapper/*.xml">
    </property>
    -->
</bean>
```

接下来我们测试：

```
 Time：23 ms - ID：com.xq.mapper.UserMapper.selectList
Execute SQL：
    SELECT
        uid AS id,
        name AS userName,
        age,
        email,
        is_deleted
    FROM
        tb_user
    WHERE
        is_deleted=0
```

此时会显示出SQL执行的信息，比如执行时间

如果我们把拦截SQL执行的时间缩短，此时执行SQL语句会报错。

```
### The error may exist in com/xq/mapper/UserMapper.java (best guess)
### The error may involve com.xq.mapper.UserMapper.selectList
### The error occurred while handling results
### SQL: SELECT  uid AS id,name AS userName,age,email,is_deleted  FROM tb_user   WHERE is_deleted=0
### Cause: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException:  The SQL execution time is too large, please optimize !

    at org.mybatis.spring.MyBatisExceptionTranslator.translateExceptionIfPossible(MyBatisExceptionTranslator.java:77)
    at org.mybatis.spring.SqlSessionTemplate$SqlSessionInterceptor.invoke(SqlSessionTemplate.java:446) <1 internal line>
    at org.mybatis.spring.SqlSessionTemplate.selectList(SqlSessionTemplate.java:230)
    at com.baomidou.mybatisplus.core.override.MybatisMapperMethod.executeForMany(MybatisMapperMethod.java:158)
    at com.baomidou.mybatisplus.core.override.MybatisMapperMethod.execute(MybatisMapperMethod.java:76)
    at com.baomidou.mybatisplus.core.override.MybatisMapperProxy.invoke(MybatisMapperProxy.java:62) <1 internal line>
    at com.xq.AppTest.testMyBatisBySpring(AppTest.java:24) <8 internal lines>
    at org.springframework.test.context.junit4.statements.RunBeforeTestExecutionCallbacks.evaluate(RunBeforeTestExecutionCallbacks.java:74)
    at org.springframework.test.context.junit4.statements.RunAfterTestExecutionCallbacks.evaluate(RunAfterTestExecutionCallbacks.java:84)
    at org.springframework.test.context.junit4.statements.RunBeforeTestMethodCallbacks.evaluate(RunBeforeTestMethodCallbacks.java:75)
    at org.springframework.test.context.junit4.statements.RunAfterTestMethodCallbacks.evaluate(RunAfterTestMethodCallbacks.java:86)
    at org.springframework.test.context.junit4.statements.SpringRepeat.evaluate(SpringRepeat.java:84) <1 internal line>
    at org.springframework.test.context.junit4.SpringJUnit4ClassRunner.runChild(SpringJUnit4ClassRunner.java:251)
    at org.springframework.test.context.junit4.SpringJUnit4ClassRunner.runChild(SpringJUnit4ClassRunner.java:97) <5 internal lines>
    at org.springframework.test.context.junit4.statements.RunBeforeTestClassCallbacks.evaluate(RunBeforeTestClassCallbacks.java:61)
    at org.springframework.test.context.junit4.statements.RunAfterTestClassCallbacks.evaluate(RunAfterTestClassCallbacks.java:70) <1 internal line>
    at org.springframework.test.context.junit4.SpringJUnit4ClassRunner.run(SpringJUnit4ClassRunner.java:190) <7 internal lines>
```

7.4 乐观锁插件

场景：

> 一件商品，成本价是80元，售价是100元。老板先是通知小李，说你去把商品价格增加50元。小李正在玩游戏，耽搁了一个小时。正好一个小时后，老板觉得商品价格增加到150元，价格太高，可能会影响销量。又通知小王，你把商品价格降低30元。
>
> 此时，小李和小王同时操作商品后台系统。小李操作的时候，系统先取出商品价格100元；小王也在操作，取出的商品价格也是100元。小李将价格加了50元，并将100+50=150元存入了数据库；小王将商品减了30元，并将100-30=70元存入了数据库。是的，如果没有锁，小李的操作就完全被小王的覆盖了。
>
> 现在商品价格是70元，比成本价低10元。几分钟后，这个商品很快出售了1千多件商品，老板亏1万多。

上面的故事，如果是悲观锁，小李取出数据后，小王只能等小李操作完之后，才能对价格进行操作，也会保证最终的价格是120元。如果是乐观锁，小王保存价格前，会检查下价格是否被人修改过了。如果被修改过 了，则重新取出的被修改后的价格，150元，这样他会将120元存入数据库。

7.4.1 模拟修改数据出现冲突

第一步：创建数据表，并插入数据

```
CREATE TABLE t_product
(
id BIGINT(20) NOT NULL COMMENT '主键ID',
NAME VARCHAR(30) NULL DEFAULT NULL COMMENT '商品名称',
price INT(11) DEFAULT 0 COMMENT '价格',
VERSION INT(11) DEFAULT 0 COMMENT '乐观锁版本号',
PRIMARY KEY (id)
);

INSERT INTO t_product (id, NAME, price) VALUES (1, '外星人笔记本', 100);
```

第二步：创建实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("t_product")
public class Product {

    private Long id;
    private String name;
    private Integer price;
    private Integer version;
}
```

第三步：定义Mapper接口

```
public interface ProductMapper extends BaseMapper<Product> {

}
```

第四步：测试

```
@Autowired
ProductMapper productMapper;

@Test
public void testConcurrentUpdate() {
    //1、小李
    Product p1 = productMapper.selectById(1L);
    System.out.println("小李取出的价格：" + p1.getPrice());

    //2、小王
    Product p2 = productMapper.selectById(1L);
    System.out.println("小王取出的价格：" + p2.getPrice());

    //3、小李将价格加了50元，存入了数据库
```

```
        p1.setPrice(p1.getPrice() + 50);
        int result1 = productMapper.updateById(p1);
        System.out.println("小李修改结果：" + result1);

        //4、小王将商品减了30元，存入了数据库
        p2.setPrice(p2.getPrice() - 30);
        int result2 = productMapper.updateById(p2);
        System.out.println("小王修改结果：" + result2);

        //最后的结果
        Product p3 = productMapper.selectById(1L);
        //价格覆盖，最后的结果：70
        System.out.println("最后的结果：" + p3.getPrice());
    }
```

7.4.2 乐观锁实现流程

数据库中添加version字段。取出记录时，获取当前version

```
SELECT id,`name`,price,`version` FROM product WHERE id=1
```

更新时，version + 1，如果where语句中的version版本不对，则更新失败

```
UPDATE product SET price=price+50, `version`=`version` + 1 WHERE id=1 AND
`version`=1
```

7.4.3 Mybatis-Plus实现乐观锁

第一步：修改实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("t_product")
public class Product {

    private Long id;
    private String name;
    private Integer price;

    @Version // 标识当前字段是版本号字段。
    private Integer version;
}
```

第二步：添加乐观锁插件配置

```
@Configuration
public class MybatisPlusConfig {
    no usages
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        //添加分页插件
        interceptor.addInnerInterceptor(new
                PaginationInnerInterceptor(DbType.MYSQL));
        //添加乐观锁插件
        interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
        return interceptor;
    }
}
```

第三步：测试修改冲突

> ① 小李查询商品信息： SELECT id,name,price,version FROM t_product WHERE id=?
>
> ② 小王查询商品信息： SELECT id,name,price,version FROM t_product WHERE id=?
>
> ③ 小李修改商品价格，自动将version+1
>
> UPDATE t_product SET name=?, price=?, version=? WHERE id=? AND version=? Parameters: 外星人笔记本(String), 150(Integer), 1(Integer), 1(Long), 0(Integer)
>
> ④ 小王修改商品价格，此时version已更新，条件不成立，修改失败
>
> UPDATE t_product SET name=?, price=?, version=? WHERE id=? AND version=? Parameters: 外星人笔记本(String), 70(Integer), 1(Integer), 1(Long), 0(Integer)
>
> 最终，小王修改失败，查询价格：150 SELECT id,name,price,version FROM t_product WHERE id=?

第四步：优化流程

把数据库中的商品价格重新设置为100再测试：

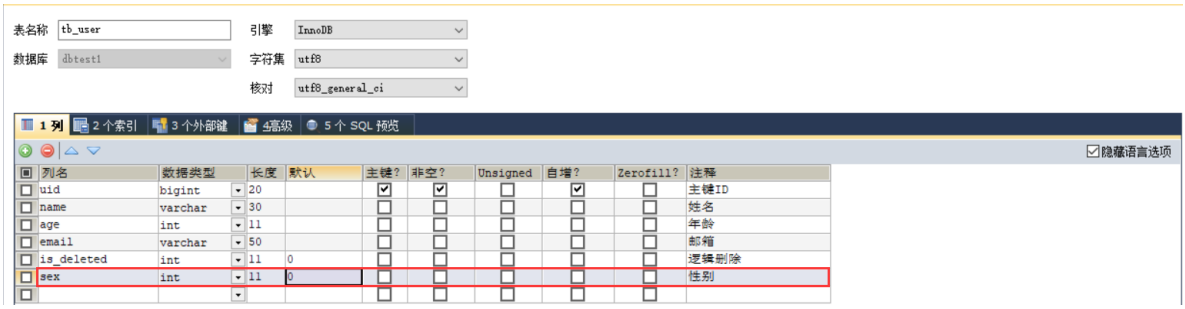
```
@Test
public void testConcurrentVersionUpdate() {
    //小李取数据
    Product p1 = productMapper.selectById(1L);
    //小王取数据
    Product p2 = productMapper.selectById(1L);
    //小李修改 + 50
    p1.setPrice(p1.getPrice() + 50);
    int result1 = productMapper.updateById(p1);
    System.out.println("小李修改的结果：" + result1);
    //小王修改 - 30
    p2.setPrice(p2.getPrice() - 30);
    int result2 = productMapper.updateById(p2);
    System.out.println("小王修改的结果：" + result2);
    if (result2 == 0) {
        //失败重试，重新获取version并更新
        p2 = productMapper.selectById(1L);
        p2.setPrice(p2.getPrice() - 30);
        result2 = productMapper.updateById(p2);
    }
    System.out.println("小王修改重试的结果：" + result2);
    //老板看价格
    Product p3 = productMapper.selectById(1L);
    System.out.println("老板看价格：" + p3.getPrice());
}
```

查看数据表信息：

8 通用枚举

表中的有些字段值是固定的，例如性别（男或女），此时我们可以使用MyBatis-Plus的通用枚举来实现。

第一步：数据库表添加字段sex



第二步：创建通用枚举类型

```
public enum SexEnum {
    MALE(1, "男"),
    FEMALE(2, "女");
    @EnumValue
    private Integer sex;
    private String sexName;
    SexEnum(Integer sex, String sexName) {
        this.sex = sex;
        this.sexName = sexName;
    }
}
```

修改User类：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
// @TableName("tb_user")
public class User {

    no usages
    @TableId(value = "uid") //如果前后字段不一致，可以添加@TableId注解，通过value属性指定uid的值
    private Long id;
    no usages
    @TableField("name") // 前后字段不一致，需要手动进行映射
    private String userName;
    no usages
    private Integer age;
    no usages
    private String email;
    no usages
    private SexEnum sex;

    no usages
    @TableLogic
    private Integer isDeleted;
}
```

使用枚举类描述性别字段。

第三步：配置扫描通用枚举

```
spring:
  # 配置数据源信息
  datasource:
    # 配置数据源类型
    type: com.zaxxer.hikari.HikariDataSource
    # 配置连接数据库信息
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://192.168.10.140:3306/dbtest1
    username: root
    password: Admin123!
# 配置MyBatis日志
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      # 配置MyBatis-Plus操作表的默认前缀
      table-prefix: tb_
      # 配置MyBatis-Plus的主键策略为自增长
      id-type: auto
  # 配置扫描通用枚举
  type-enums-package: com.xq.pojo
```

扫描通用枚举类

第四步：测试

```
@Test
public void testSexEnum(){
    User user = new User();
    user.setUserName("Enum");
    user.setAge(20);
    // 设置性别信息为枚举项，会将@EnumValue注解所标识的属性值存储到数据库
    user.setSex(SexEnum.MALE);
    // INSERT INTO t_user ( username, age, sex ) VALUES ( ?, ?, ? )
    // Parameters: Enum(String), 20(Integer), 1(Integer)
    userMapper.insert(user);
}
```

查看数据表，数据添加情况：

uid	name	age	email	is_deleted	sex
1	Jone	18	jone@baomidou.com	0	0
2	Jack	20	jack@baomidou.com	0	0
3	Tom	28	tom@baomidou.com	0	0
4	Sandy	21	sandy@baomidou.com	0	0
5	Billie	24	billie@baomidou.com	0	0
6	张三	23	zhangsan@qq.com	0	0
7	张三	23	zhangsan@qq.com	1	0
8	Enum	20	(NULL)	0	1
(Auto)	(NULL)	(NULL)	(NULL)	0	0

基于通用枚举给sex字段添加数据

9 AR模式

Active Record(活动记录)，是一种领域模型模式，特点是一个模型类对应关系型数据库中的一个表，而模型类的一个实例对应表中的一行记录。

在Active Record模式中，对象中既有持久存储的数据，也有针对数据的操作，Active Record模式把数据增删改查的逻辑作为对象的一部分，处理对象的用户知道如何读写数据，提升了开发效率。

其实底层仍然使用的是Mapper层在完成数据库操作。只不过由我们自己调用Mapper对象操作数据库，变成了通过实体类对象来调用Mapper完成数据库操作。从代码的物理视图上我们是看不到实体类调用Mapper的过程的。也就说，本质上仍然是Mapper层在操作数据库实体类型操作数据掩盖了底层的mapper的方法的调用。

9.1 AR模式的使用

第一步：创建数据表并插入数据

```
CREATE TABLE department(
  id INT PRIMARY KEY AUTO_INCREMENT,
  dept_name VARCHAR(20),
  location VARCHAR(50)
)

INSERT INTO `department`(dept_name,location) VALUES('技术部','北京'),('市场部','上海'),('财务部','杭州')
```

第二步：创建实体类

创建实体类，并且需要继承Model这个抽象类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("department")
```

```
public class Department extends Model<Department> {

    @TableId(type = IdType.AUTO)
    private  int id;
    private String deptName;
    private String location;

    @Override
    public Serializable pkVal() {
        return id;
    }
}
```

第三步：创建接口

```
public interface DepartmentMapper extends BaseMapper<Department> {

}
```

第四步：测试

```
@SpringBootTest
public class ActiveRecordTest {

    //新增操作
    @Test
    public void test01(){
        Department department = new Department();
        department.setDeptName("人事部");
        department.setLocation("武汉");
        boolean result = department.insert();
        System.out.println("插入的结果是:" + result);
    }

    //查询操作
    @Test
    public void test02(){
        Department department = new Department();
        List<Department> departmentList = department.selectAll();
        departmentList.forEach(System.out::println);
    }

    //修改操作
    @Test
    public void test03(){
        Department department = new Department();
        department.setId(3);
        department.setLocation("南京");
        department.updateById();
    }

   //删除操作
   @Test
   public void test04(){
       Department department = new Department();
       department.deleteById(4);
```

```
    }

    //根据条件查询
    @Test
    public void test05(){
        Department department = new Department();
        QueryWrapper<Department> userQueryWrapper = new QueryWrapper<>();
        userQueryWrapper.eq("location","上海");
        List<Department> departmentList =
department.selectList(userQueryWrapper);
        departmentList.forEach(System.out::println);
    }
}
```

10 代码生成器

通过代码生成器可以帮助我们快速生成项目所需要的实体类、Dao、Service、Controller还有Mapper映射文件。类似于我们前面学习的mybatis逆向工程，但是比逆向工程生成的内容更加全面。

第一步：引入依赖

```
<!--代码生成器需要的依赖-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.5.1</version>
</dependency>

<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.31</version>
</dependency>
```

第二步：创建测试类

```
public class FastAutoGeneratorTest {

    public static void main(String[] args) {
        FastAutoGenerator.create("jdbc:mysql://192.168.10.140:3306/dbtest1",
"root", "Admin123!")
                        .globalConfig(builder -> {
                            builder.author("krisswen") // 设置作者
                                    //.enableSwagger() // 开启 swagger 模式
                                    .fileOverride() // 覆盖已生成文件
                                    .outputDir("D://mybatis_plus"); // 指定输出目
录
                        })
                        .packageConfig(builder -> {
                            builder.parent("com.xq") // 设置父包名
                                    .moduleName("mybatisplus") // 设置父包模块名

.pathInfo(Collections.singletonMap(OutputFile.mapperXml, "D://mybatis_plus"));
// 设置mapperXml生成路径
                        })
                        .strategyConfig(builder -> {
```

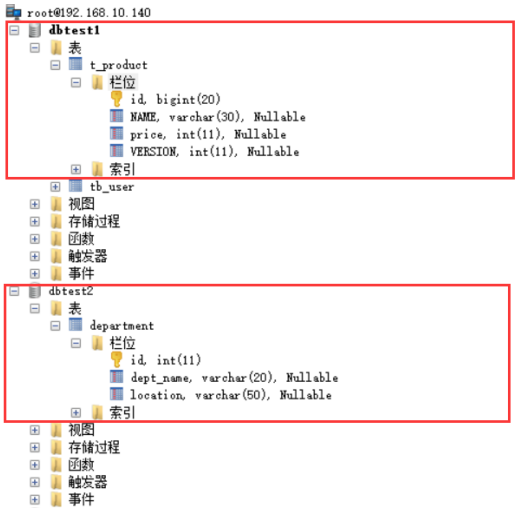
```
                            builder.addInclude("tb_user") // 设置需要生成的表名
                                    .addTablePrefix("tb_", "t_"); // 设置过滤表前
缀
                        })
                        .templateEngine(new FreemarkerTemplateEngine()) // 使用
Freemarker引擎模板，默认的是Velocity引擎模板
                        .execute();
    }
}
```

11 多数据源配置

适用于多种场景：纯粹多库、 读写分离、 一主多从、 混合模式等。

目前我们就来模拟一个纯粹多库的一个场景：

场景说明： 我们创建两个库，分别为：dbtest1与dbtest2（新建），将 dbtest1库的department表移动到dbtest2库，这样每个库一张表，通过一个测试用例分别获取商品数据与部门数据，如果获取到说明多库模拟成功。



模拟多数据源

现在我们就来模拟多数据源的配置。

第一步：创建项目，并引入相关依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.2</version>
    <relativePath/>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

```
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-boot-starter</artifactId>
        <version>3.5.1</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
    </dependency>

    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>dynamic-datasource-spring-boot-starter</artifactId>
        <version>3.5.0</version>
    </dependency>
</dependencies>
```

第二步：创建启动类

```
@SpringBootApplication
@MapperScan(basePackages = "com.xq.mapper")
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class,args);
    }
}
```

第三步：创建配置文件

```
spring:
# 配置数据源信息
  datasource:
    dynamic:
      # 设置默认的数据源或者数据源组,默认值即为master
      primary: master
      # 严格匹配数据源,默认false.true未匹配到指定数据源时抛异常,false使用默认数据源
      strict: false
      datasource:
        master:
          url: jdbc:mysql://192.168.10.140:3306/dbtest1
          driver-class-name: com.mysql.jdbc.Driver
          username: root
          password: Admin123!
        slave_1:
          url: jdbc:mysql://192.168.10.140:3306/dbtest2
          driver-class-name: com.mysql.jdbc.Driver
          username: root
          password: Admin123!
```

第四步：创建实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("t_product")
public class Product {
    private Long id;
    private String name;
    private Integer price;
    @Version
    private Integer version;
}
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("department")
public class Department{

    @TableId(type = IdType.AUTO)
    private  int id;
    private String deptName;
    private String location;
}
```

第五步：创建Mapper

```
public interface ProductMapper extends BaseMapper<Product> {

}
```

```
public interface DepartmentMapper extends BaseMapper<Department> {

}
```

第六步：创建Service接口及实现类

```
public interface DepartmentService extends IService<Department> {

}
```

```
public interface ProductService extends IService<Product> {

}
```

```
@DS("slave_1") // 指定数据源
@Service
public class DepartmentServiceImpl extends ServiceImpl<DepartmentMapper,
Department> implements DepartmentService {

}
```

```java
@DS("master") // 指定数据源
@Service
public class ProductServiceImpl extends ServiceImpl<ProductMapper, Product>
implements ProductService {

}
```

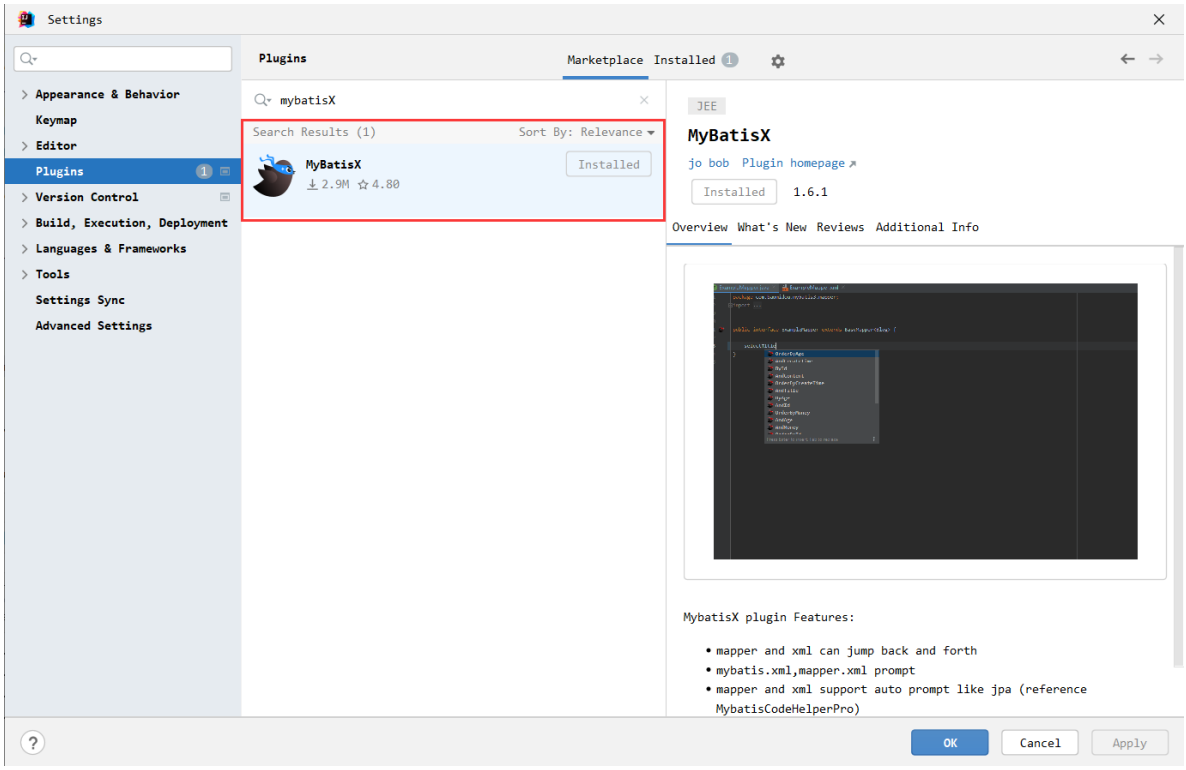
第七步：测试多数据源

```java
@SpringBootTest
public class DynamicDataSourceTest {

    @Autowired
    DepartmentService departmentService;

    @Autowired
    ProductService productService;

    @Test
    public void testDynamicDataSource(){
        System.out.println(departmentService.getById(1));
        System.out.println(productService.getById(1L));
    }
}
```

12 MybatisX插件

MyBatis-Plus为我们提供了强大的mapper和service模板，能够大大的提高开发效率 但是在真正开发过程中，MyBatis-Plus并不能为我们解决所有问题，例如一些复杂的SQL，多表联查，我们就需要自己去编写代码和SQL语句，我们该如何快速的解决这个问题呢，这个时候可 以使用MyBatisX插件 MyBatisX一款基于 IDEA 的快速开发插件，为效率而生。

首先我们需要再IDEA中去安装这款插件：



12.1 使用MybatisX插件生成代码

接下来我们使用MybatisX这款插件帮助我们生成Mybatis-plus相关的代码。

第一步：创建工程引入依赖

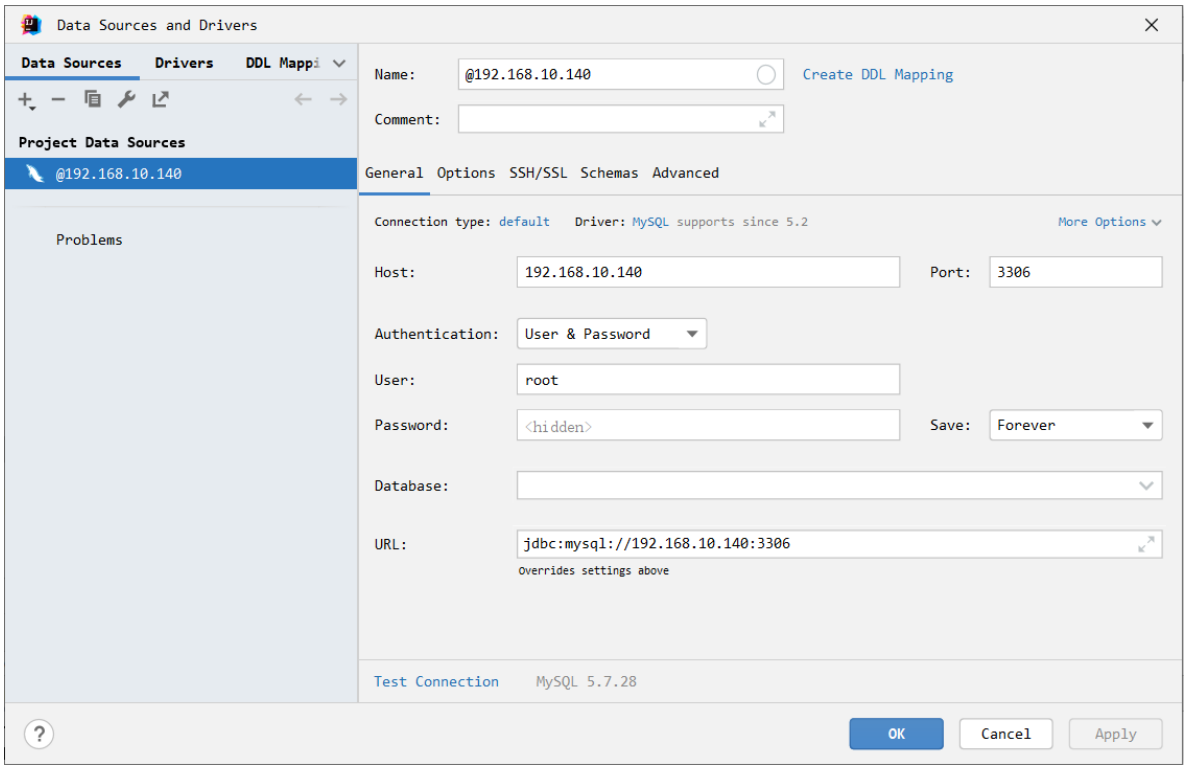
```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.2</version>
    <relativePath/>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-boot-starter</artifactId>
        <version>3.5.1</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
    </dependency>
</dependencies>
```

第二步：定义启动类和配置文件application.yml

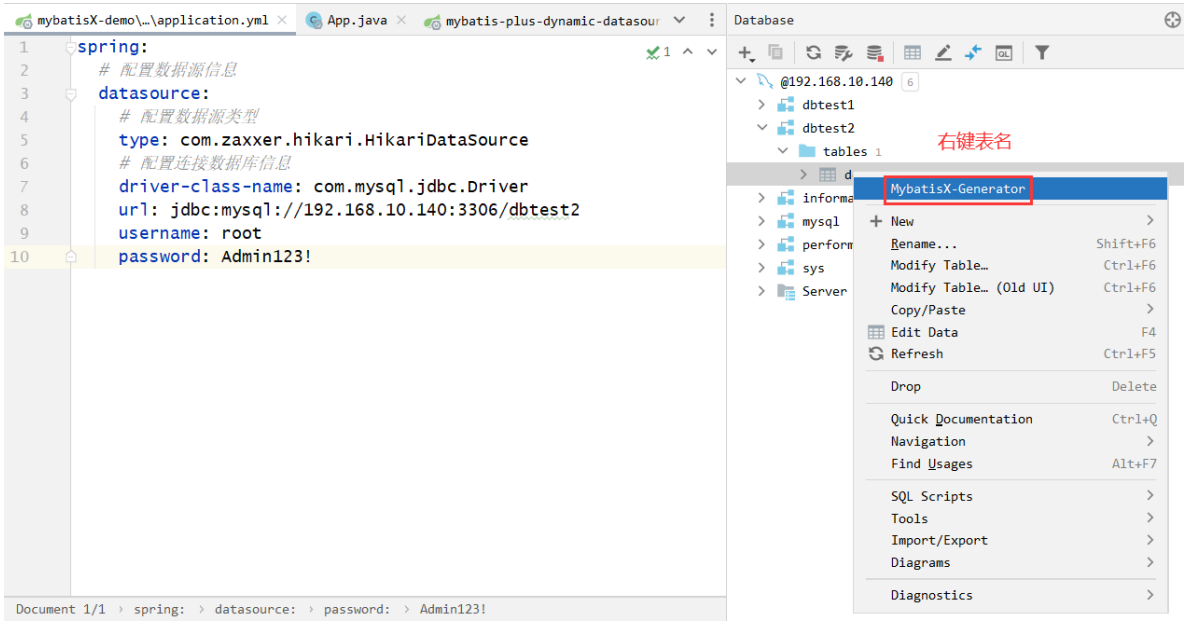
```
@SpringBootApplication
@MapperScan(basePackages = "com.xq.mapper")
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class,args);
    }
}
```

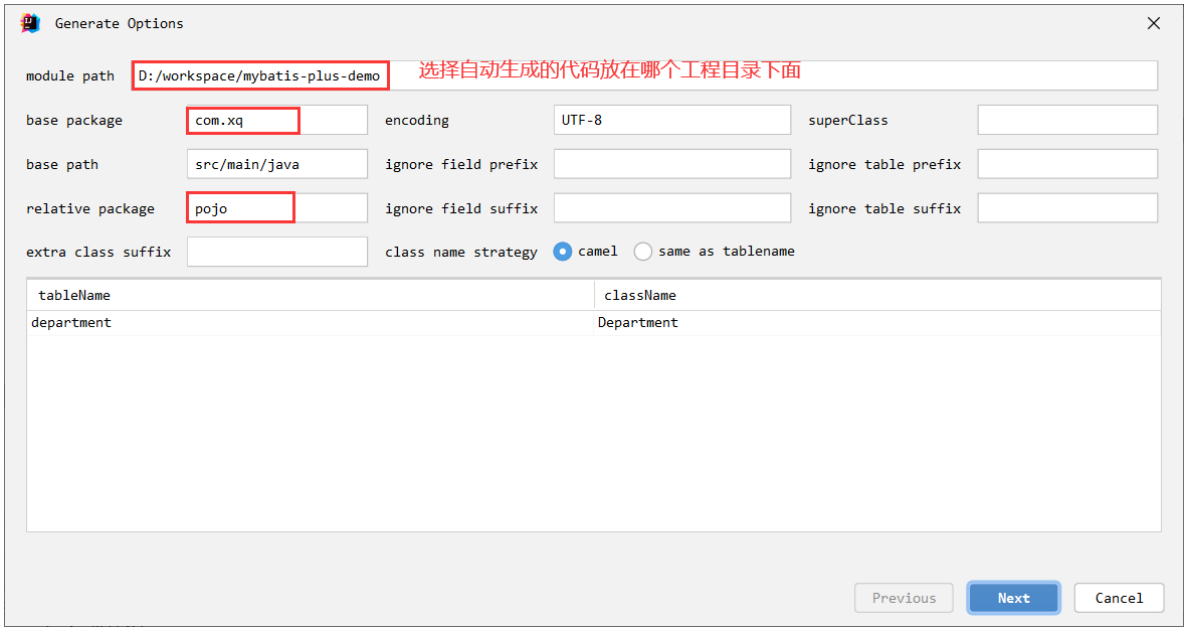
```yaml
spring:
  # 配置数据源信息
  datasource:
    # 配置数据源类型
    type: com.zaxxer.hikari.HikariDataSource
    # 配置连接数据库信息
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://192.168.10.140:3306/dbtest2
    username: root
    password: Admin123!
```

第三步：在IDEA中连接数据库

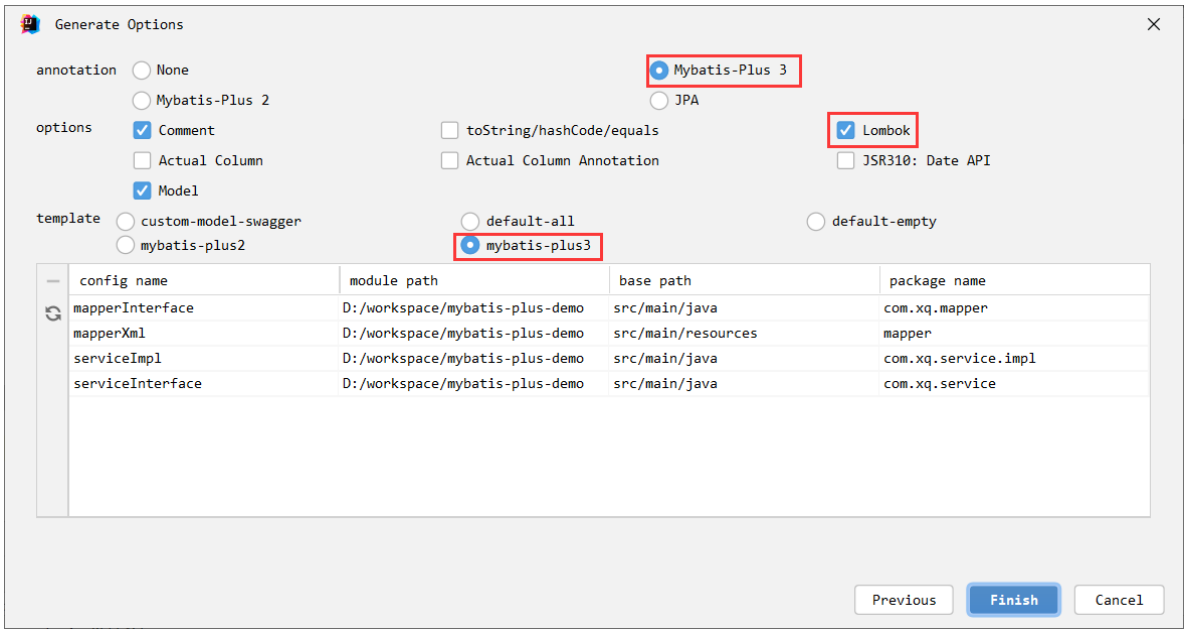


第四步：进行MybatisX的配置



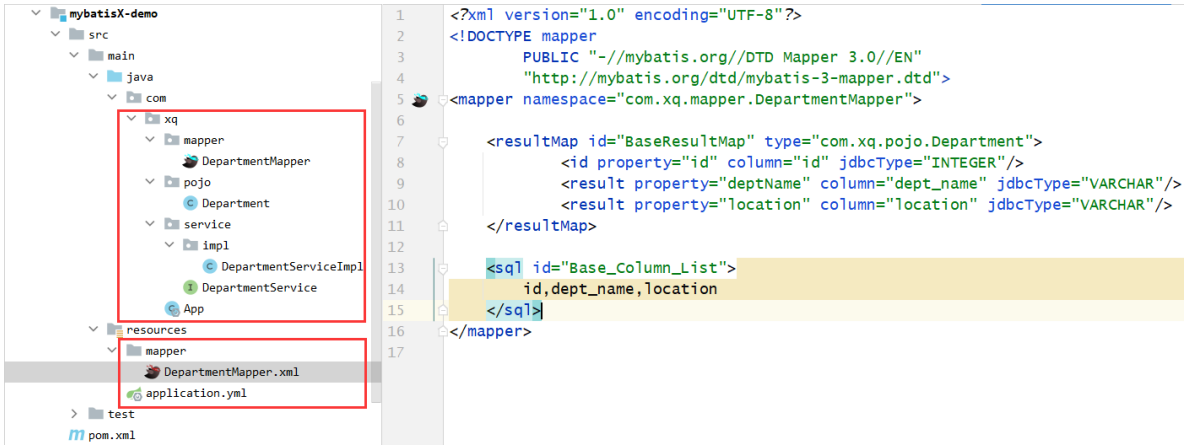


点击Next:



点击Finish完成。

最后生成内容如下：



12.2 基于MybatisX只定义接口

此时我们发现MybatisX插件不仅帮助我们生成了实体、mapper还有service代码。此外我们还可以自定义接口，基于MybatisX提供的模板自动帮助我们生成接口代码及对应的mapper映射文件。

```
public interface DepartmentMapper extends BaseMapper<Department> {

    no usages
    int insertSelective(Department department);

    no usages
    List<Department> searchById(@Param("id") Integer id);

    no usages
    int deleteById(@Param("id") Integer id);

    no usages
    int updateLocationById(@Param("location") String location, @Param("id") Integer id);
}
```

自定义接口

这些自定义接口对应Mapper映射文件也一同生成好了，我们可以查看：

```
<insert id="insertSelective">
    insert into department
    <trim prefix="(" suffix=")" suffixOverrides=",">
        <if test="id != null">id,</if>
        <if test="deptName != null">dept_name,</if>
        <if test="location != null">location,</if>
    </trim>
    values
    <trim prefix="(" suffix=")" suffixOverrides=",">
        <if test="id != null">#{id,jdbcType=INTEGER},</if>
        <if test="deptName != null">#{deptName,jdbcType=VARCHAR},</if>
        <if test="location != null">#{location,jdbcType=VARCHAR},</if>
    </trim>
</insert>
<select id="searchById" resultMap="BaseResultMap">
    select
    <include refid="Base_Column_List"/>
    from department
    where
    id = #{id,jdbcType=NUMERIC}
</select>
<delete id="deleteById">
    delete
    from department
    where id = #{id,jdbcType=NUMERIC}
</delete>
<update id="updateLocationById">
    update department
    set location = #{location,jdbcType=VARCHAR}
    where id = #{id,jdbcType=NUMERIC}
</update>
```