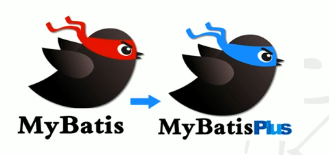
# 简介

## 介绍

MyBatisPlus是MyBatis的一个增强工具包，只做增强不做改变，为了简化开发工作，提高生产效率而生。



注意：MyBatisPlus的使用的前提是基于MyBatis。

## 相关网址

官网：<https://mp.baomidou.com/>

码云：<https://gitee.com/baomidou/mybatis-plus>

GitHub：<https://github.com/baomidou/mybatis-plus>

可以在官网查看MyBatisPlus的教程文档。

## 特性

只在MyBatis基础上做增强，不做改变，引入它不会对现有工程产生影响，还内置了通用Mapper。



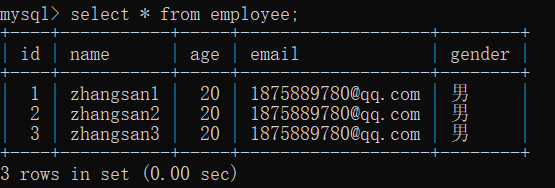
## 前置知识

在学习MyBatisPlus之前，你需要掌握MyBatis、Spring、Maven的使用。

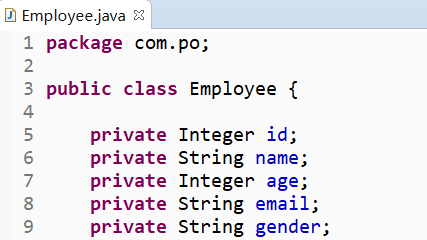
# 集成MyBatisPlus

## 准备工作

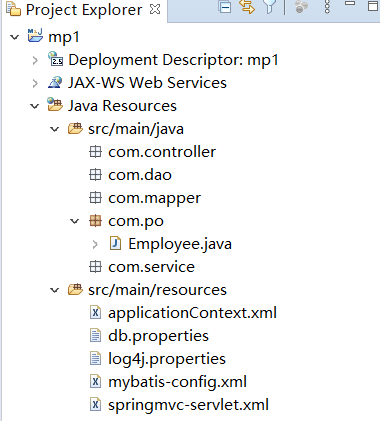
1）在集成之前，在数据库中创建一个employee表用于测试



2）创建Maven工程，将其完善为一个SSM项目，并为employee表编写实体Bean：

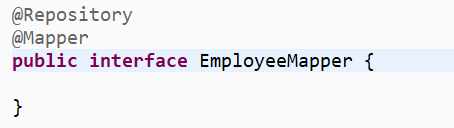


目录结构如下：



3）进一步完善项目，编写mapper文件和dao接口：



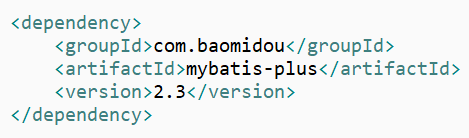


注意：记得扫描Mapper文件

以上准备工作完成后，测试一下项目是否能够正常运行。

## 进行集成

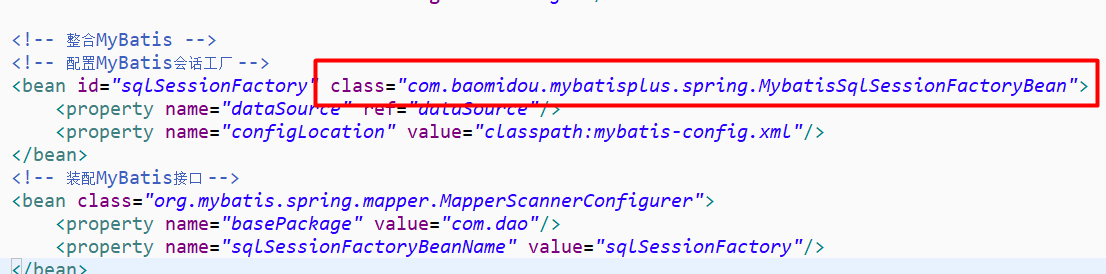
1）引入MyBatisPlus依赖：



注意：引入该依赖后，不用引入mybatis包以及mybatis-spring的整合包，

因为mybatisplus依赖包会帮我们维护这两个依赖。

2）替换SqlSession类：



先前使用的是：org.mybatis.spring.SqlSessionFactoryBean；

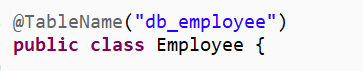
这样，MyBatisPlus集成成功。

# 常用注解

## @TableName

在默认情况下，MyBatisPlus会认为实体类的类名与对应的表名一致，如果实体类名与表名不一致，则需要通过@TableName注解来指定此实体类对应的真实表名。

例如，存在表名为db\_employee的表，编写的实体类名称为Employee，为此我们需要通过@TableName为该实体类指定其对应的表名：



否则在使用MyBatisPlus提供的CRUD方法时将会产生异常。

## @TableId

@TableId用于标注实体类中表示主键的属性，该注解有两个属性：value、type

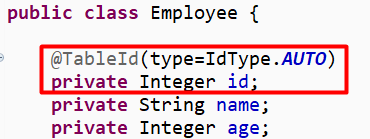
1. value：如果该实体类中主键属性的名称与表中主键字段名不一致时，则可以通

过该属性来指定；

1. type：指定主键的生成策略（即主键是如何产生的）：



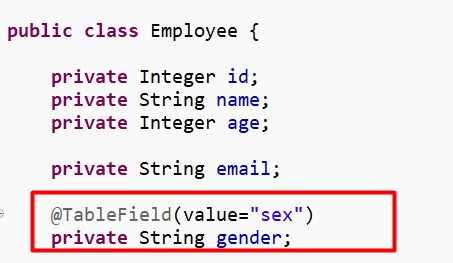
通常使用第一种，表示自增主键。



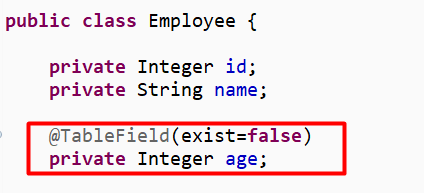
## @TableFiled

如果实体类中某属性的属性名与该实体类对应表中某字段的字段名不一致，则可以使用@TableFiled注解进行指定，另外，如果实体类中某个属性并不是用来表示表中对应的某个字段，则也可以使用@TableField注解将该属性标注为不是表字段。

1. 指定字段名：例如，存在Employee表，表中存在sex字段，对应的实体类Employee中使用gender属性来表示sex字段，此时就需要在该属性上使用@TableField注解来指明该属性表示的表中的sex字段。



1. 标注非表字段：例如表employee有两个字段id、name，其对应的实体类有三个属性id、name、age，很明显，age属性不是表employee的字段，所以，需要使用@TableField注解标注该属性非表字段。



# 全局策略配置

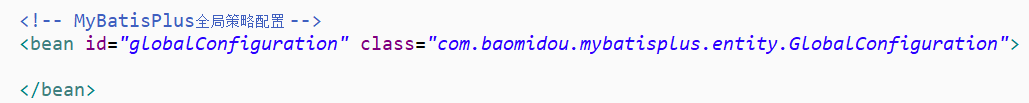
## 介绍

全局策略配置是针对全局而言的，我们可以通过全局策略配置来配置MyBatisPlus的一些运行时配置，如是否关闭驼峰式命名（默认是开启的）等等。

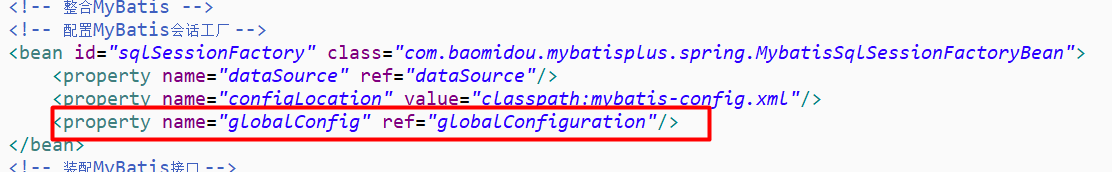
## 开启全局策略配置

全局策略配置并不是必须的。

1）全局策略配置在Spring配置文件中定义：

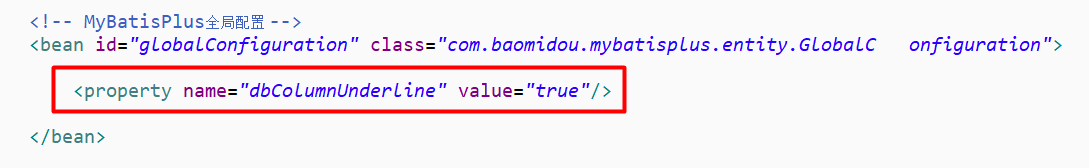


2）将其注入到SQL会话工厂中：



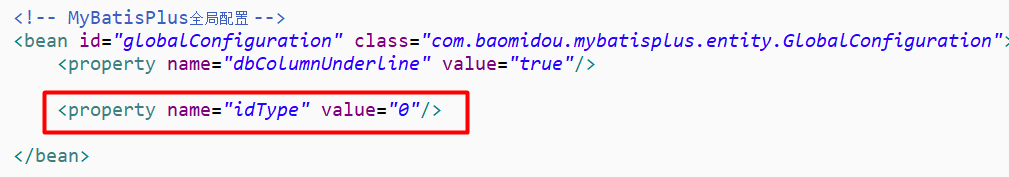
## 常用的全局配置

1. 开启下划线命名：



下划线命名的作用与MyBatis驼峰命名相同，因为默认值就是true，所以也可以不配置。

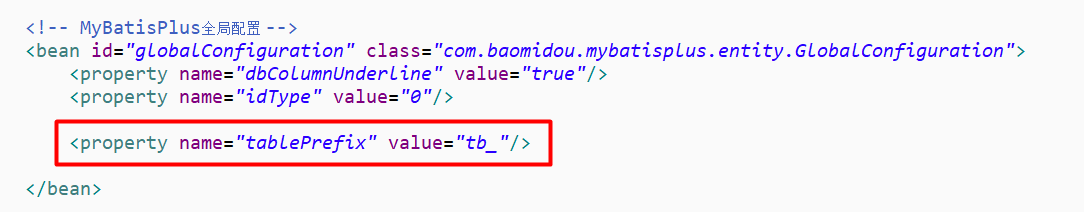
1. 配置全局的主键策略，这样就不需要在每个实体类中通过@TableId注解来标注主键属性了：





IdType.AUTO可以用0表示，IdType.INPUT可以用1表示，以此类推（因为它们是枚举类型）。

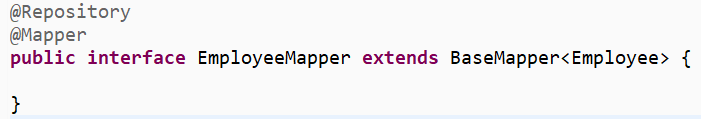
1. 全局的表前缀配置，通过全局表前缀配置，则实体类对应的表名为“前缀+表名”的形式。



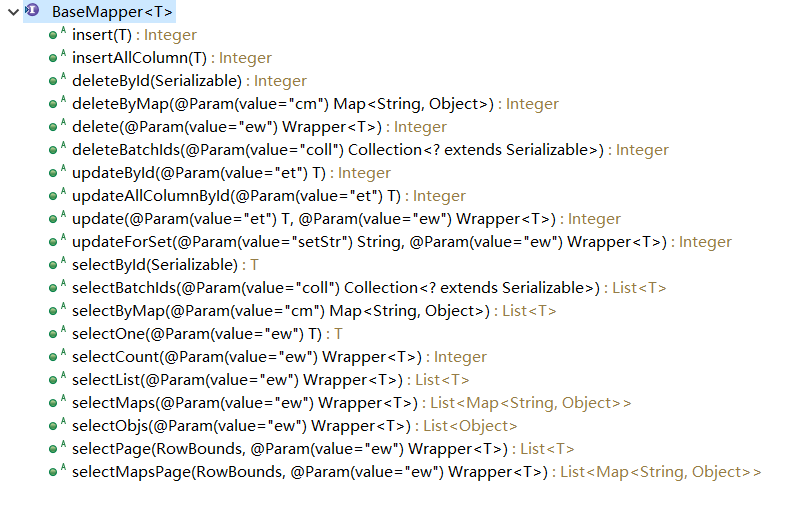
当实体类名与数据表名不一致时，这样就不用在实体类上使用@TableName注解了，但要求这些表名的前缀必须相同。

# 基本使用

集成MyBatisPlus后，会默认提供一些CRUD操作方法，通过这些方法可以完成基本的增删查改，使用这些方法，Mapper接口需要继承BaseMapper接口：



其中，泛型的类型为实体类类型，Mapper接口继承BaseMapper接口，会拥有基本CRUD方法，如下：



## 插入操作

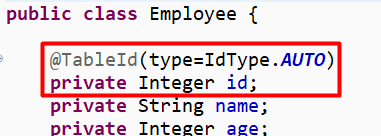
BaseMapper接口提供了insert( )方法和insertAllAllColumn( )方法插入数据，在插入数据之前，我们需要使用@TableId注解标注实体类中的主键属性；

@TableId注解有两个属性：value、type

1. 当实体类属性名与表中对应的注解字段名不一致时，使用value指定主键字段名（如两者一致，则可以省略该属性）；
2. type用于指定主键生成策略，也就是实体类对应表的主键的生成策略，可以取以下值：



将Employee实体类加上主键注解：

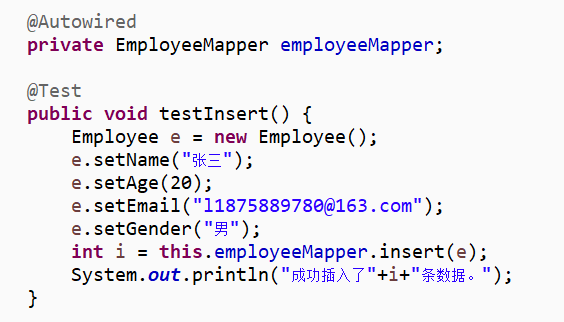


### 插入数据

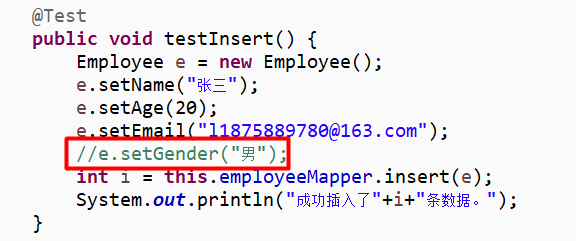
插入数据可以使用insert( )方法或insertAllColumn( )方法，都会返回一个Interger类型的

值，表示影响的行数；

1）使用insert( )方法插入数据：

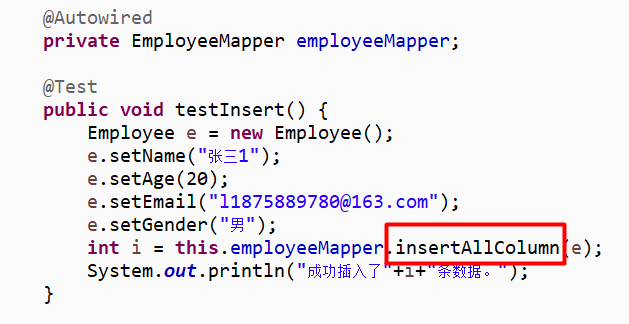


使用insert( )方法插入数据时，可以不插入某个属性：



但前提是这个属性在表中对应的字段的约束可以为空或者有非空默认值。

2）使用insertAllColumn( )方法插入



使用这种方式插入数据的必须设置所有的属性（不包括自动生成主键）。

### 主键回填

在插入数据库时，会默认主键回填，回填到插入的实体类对象的主键属性中：



## 更新操作

### 更新数据

1）使用updateById( )方法根据ID修改数据：



可以使用此方法修改部分字段的数据：



2）使用updateAllColumnById( )方法根据ID修改数据：

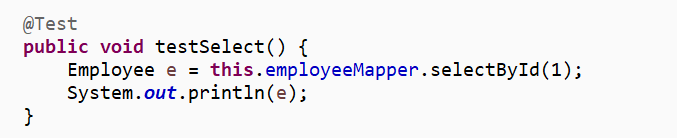


使用此方法必须修改所有字段的值，ID字段除外（ID字段是主键）。

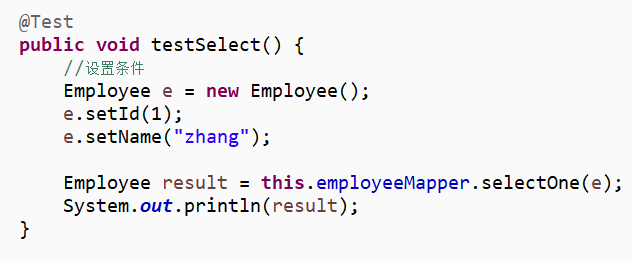
## 查询操作

### 查询数据

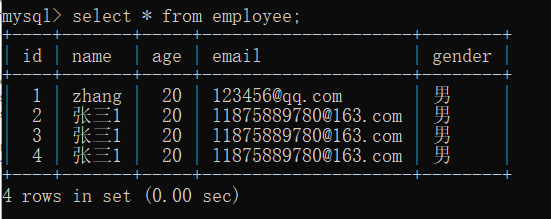
1. 使用selectById( )根据ID查询单条数据：



1. 使用selectOne( )方法根据条件查询（AND连接）单条数据：



使用这个查询只能出一条数据，所以在设置条件时，条件中必须至少指定一个唯一字段，否则查询会失败，如表中存在以下数据：



使用selectOne( )方法查询时，将查询条件设置为name=”张三” and age=20，查询时就会出现错误，因为会查询到多条数据，而selectOne( )方法只能返回一条数据，所以在设置查询条件时可以加上一个唯一字段值，如字段id的值。

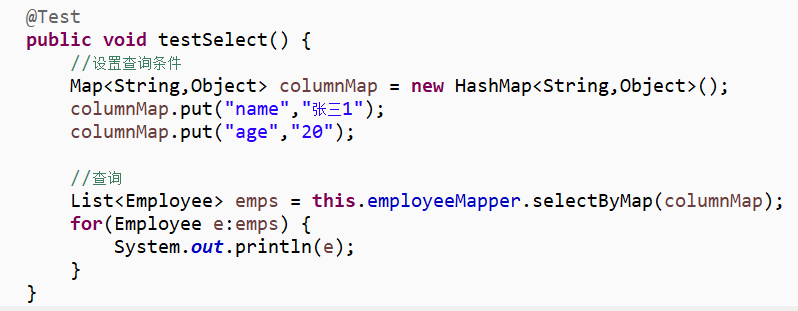
1. 使用selectBatchIds( )方法根据主键ID列表（IN表达式）查询多条数据：



其生成的查询语句为：



1. 使用selectByMap( )方法根据条件（AND关键字）查询多条数据：



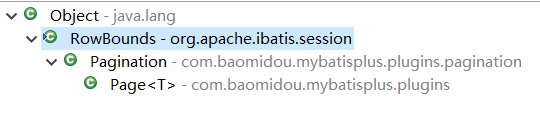
注意：Map中的key必须是以表的字段名为准，而不是实体类属性名（当实体类属性与表字段名不一致时需要特别注意，包括驼峰命名），否则找不到字段时会报错。

### 分页查询

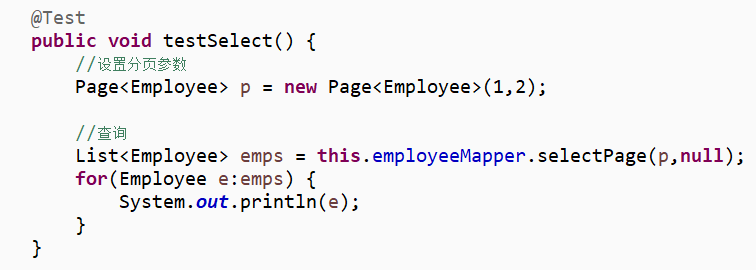
MyBatisPlus分页查询底层依赖于MyBatis，使用selectPage ( )方法可以实现分页查询，该方法接收一个RowBounds类型的对象：



RowBounds类的继承结构如下：



它有一个子类Pagination，Pagination类有很多与分页相关的属性，如每页显示条数、当前页、总数、排序等等，通常是利用 Pagination类的子类Page实现分页：



声明Page对象时，构造方法的第一个参数表示当前页，第二个参数表示每页要显示的条数，但观察打印的SQL语句：



它还是会去数据库中查找全部的记录，然后再进行分页处理，所以这种分页方式不是很好，属于假分页，也叫内存分页，当数据量特别大的时候是非常可怕的。在后面学习分页插件的时候可以对其进行改善。

## 删除操作

### 删除数据

1. 使用deleteById( )方法根据ID删除单条数据：



1. 使用deleteByMap( )方法根据条件（AND关键字连接）删除多条数据：



注意：Map中的key必须是以表的字段名为准，而不是实体类属性名（当实体类属性与表字段名不一致时需要特别注意，包括驼峰命名），否则找不到字段时会报错。

1. 使用deleteBatchIds( )方法根据主键ID列表（IN表达式）删除多条数据：



# 条件构造器

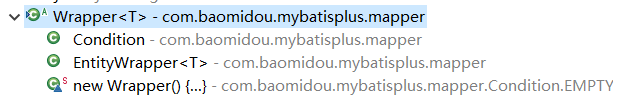
1. Mapper接口继承BaseMapper接口后，拥有了常用的CRUD方法，通过这些方法可以完成大部分的单表CRUD操作，但是在进行一些复杂操作时（如模糊查询），这些方法是完成不了的，为此MyBatisPlus提供了条件构造器，允许自定义复杂条件。

2）在BaseMapper中也可以看到很多方法上有一个Wrapper类型的参数：



它就是为了使用条件构造器而存在的。

## EntityWrapper简介

1. EntntityWrapper是Wrapper类的子类：

MyBatisPlus正是通过EntryWrapper类或者Condition类来让开发者自由的构建条件，提高开发效率。

1. EntityWrapper类称为实体包装器，主要用于处理SQL拼接，排序，实体参数查询等，但需要注意的是，使用的字段名是表中的字段名，而不是实体类的属性名。
2. 使用条件构造器主要是通过以下方法构造条件：



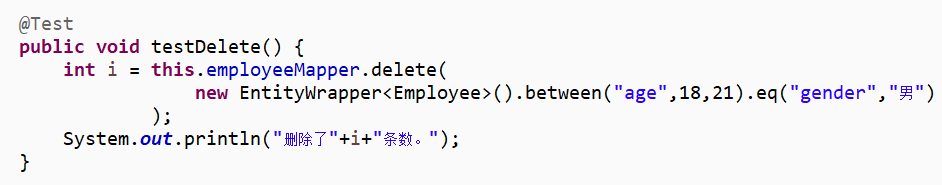


## 使用

1. 删除满足条件的记录（当wrapper为null时表示删除表中所有的记录）



删除age在18和21之间的，且gender为男的所有员工：



1. 修改满足条件的记录（可只修改部分字段，当wrapper为null时表示修改全部记录）



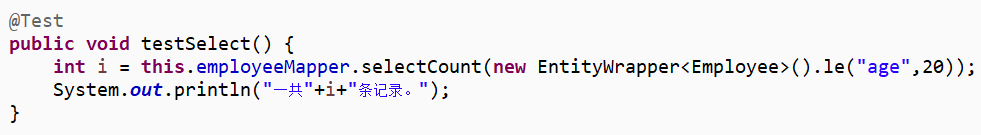
age>=20的记录，将age修改为18：



1. 查询满足条件的记录数（wrapper为null时表示查询全部记录数）



查询年龄<=20的记录数：



1. 查询满足条件的记录（wrapper为null时表示查询全部记录）



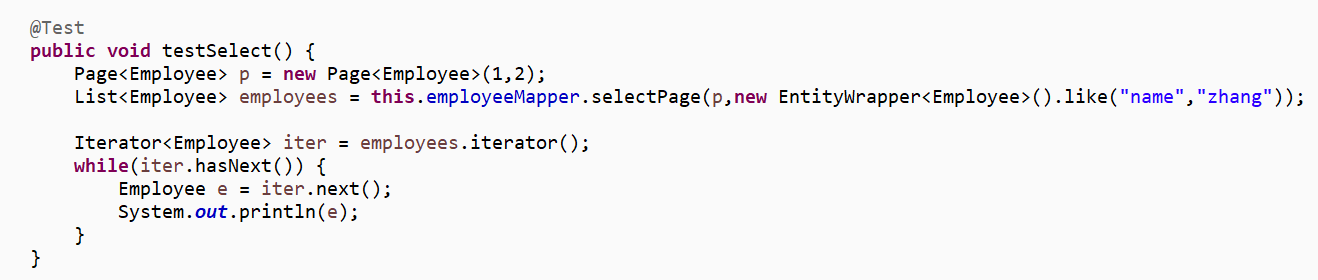
查询名字中带有zhang的员工：



1. 查询满足条件的记录并进行分页（当wrapper为null表示查询全部记录）



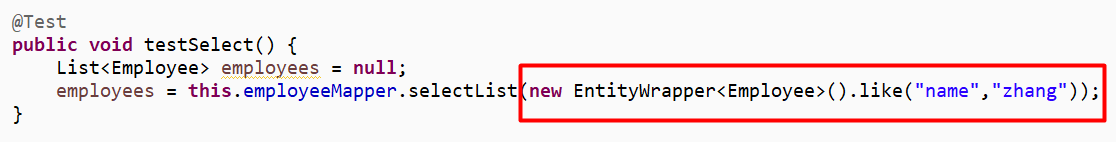
查询名字中带有zhang的员工，并分页显示：



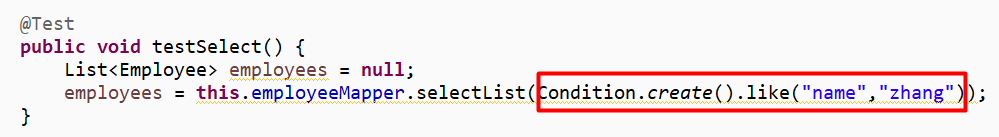
## Condition

Condition和EntityWrapper一样，也是Wrapper类的子类，它与EntityWrapper的关系类似于兄弟，使用方法几乎相同。

如使用EntityWrapper类查询名字中带有zhang的员工：



而使用Condition类作为参数，如下：



所以说，Condition类与EntityMapper类的用法几乎相同，都继承了Wrapper类中的条件方法，只不过创建对象的方式不同而已，EntityWrapper通过new关键字创建对象，Condition通过静态方法创建对象。

## 小结

使用条件构造器的关键点在于Wrapper类中的条件构造方法的使用。

# ActiveRecord（活动记录）

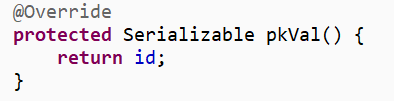
ActiveRecord（活动记录）是一种领域模型模式，特点是一个模型类对应关系型数据库中的一个表，而模型类的一个实例对应表中一条记录。

## 开启ActiveRecord模式

1. 让实体Bean继承Model类：



指定的泛型为当前实体类类型；

1. 继承Model类后，实体类中需要覆写pkVal( )方法，并在此方法中返回主键属性：

实体类继承Model类后，将拥有了基本的CRUD操作方法。

## 使用

**插入数据**

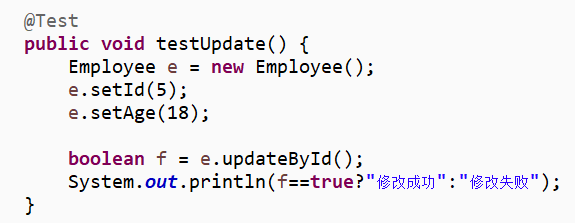
使用insert( )方法插入数据：



更多插入数据的方法请查看Model类。

**更新数据**

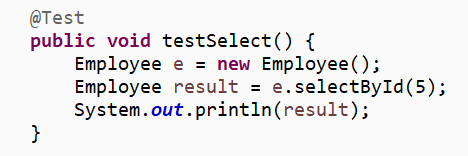
使用updateById( )根据条件更新单条数据：



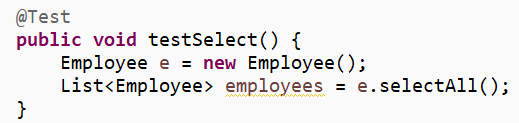
更多更新数据的方法请查看Model类。

**查询数据**

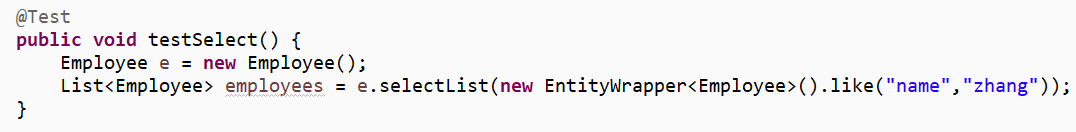
1）使用selectById( )方法根据ID查询单条数据：



2）使用selectAll( )方法查询所有记录：



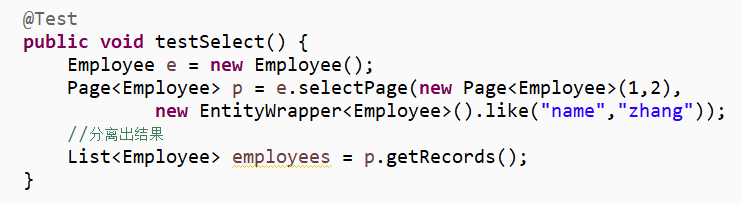
1. 使用selectList( )方法根据条件构造器查询满足条件的记录（wrapper为null时表示查询所有记录）：



1. 使用selectCount方法根据条件构造器查询满足条件的记录数（wrapper为空表示查询所有记录数）：



1. 使用selectPage( )方法根据条件构造进行分页查询（wrapper为空表示查询所有的记录）：

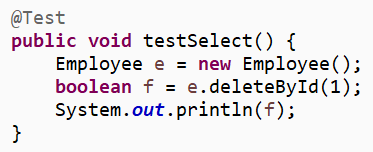


注意：查询返回的是一个Page对象，可以调用getRecords( )方法分离查询结果；

关于更多查询数据的方法可以查看Model类。

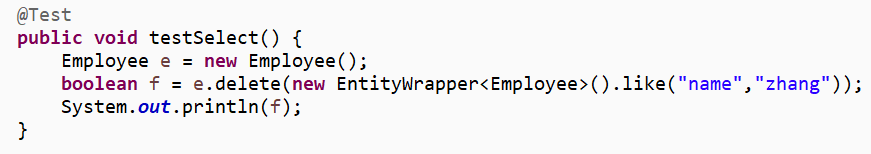
**删除数据**

1. 使用deleteById( )方法根据ID删除单条数据：



注意：不管是否删除了数据，返回的boolean值始终为true；

1. 使用delete( )方法根据条件构造器删除满足条件的数据：



注意：不管是否删除了数据，返回的boolean值始终为true；

## 小结

1. ActiveRecord了解即可，因为使用起来有些别扭。
2. AR底层使用的依然是MyBatis的语法，其本质还是调用MyBatis对应的方法，类似于语法糖，（语法糖的定义如下：）



1. 到此，我们已经领略了MyBatisPlus的魅力与高效率。

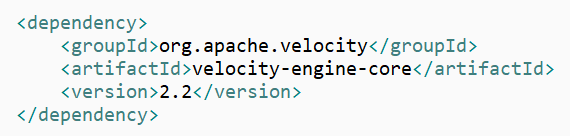
# 代码生成器

1. MyBatis Generator（MBG）是基于XML文件来生成代码的，而MyBatisPlus（MP）是基于Java代码来生成代码的。
2. MyBatis Generator可以生成实体类、Mapper接口、Mapper映射文件，而MyBatisPlus可以生成实体类、Mapper接口、Mapper映射文件、Service层、Controller层。所以，MyBatisPlus代码生成器比MyBatis Generator更强大。
3. 在使用代码生成器时，需要注意一些小事项，在MyBatisPlus中，建议数据库和表字段名采用驼峰式命名方式，而不是以下滑线进行命名，如果采用的是下划线命名方式，就需要开启全局下划线开关（默认就是开启的），如果表字段名命名与实体类不一致就需要使用注解进行指定，采用下划线时，在生成代码时需要配置dbColumnUnderLine属性；

这么做的原因是为了避免在对应实体类时产生的性能操作，这样字段不用做映射就能直接和实体类对应，当然，如果项目不用考虑这些小性能，你采用下划线也是没问题的，在生成代码时配置dbColunmUnderLine属性就可以了。

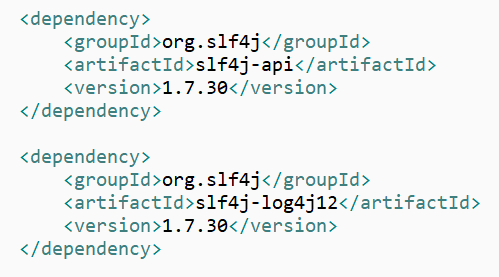
## 加入依赖

1）MyBatisPlus代码生成器需要依赖模板，常用的是Apache的Velocity模板：



当然也可以使用其他模板；

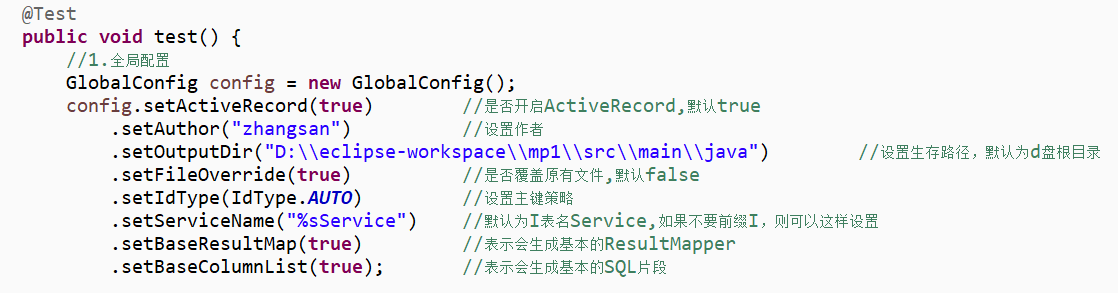
2）加入日志包：



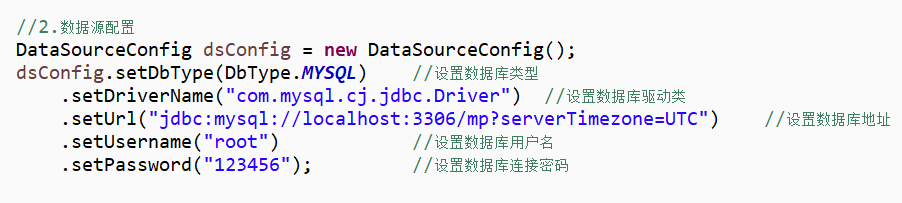
## 使用

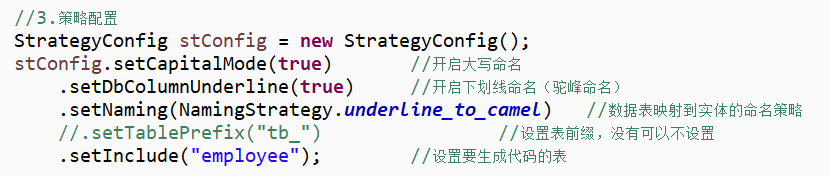
编写一个主方法或一个测试方法：

1. 全局配置：



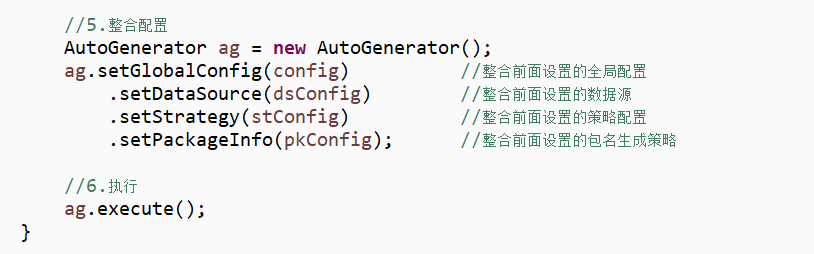
2）数据源配置：



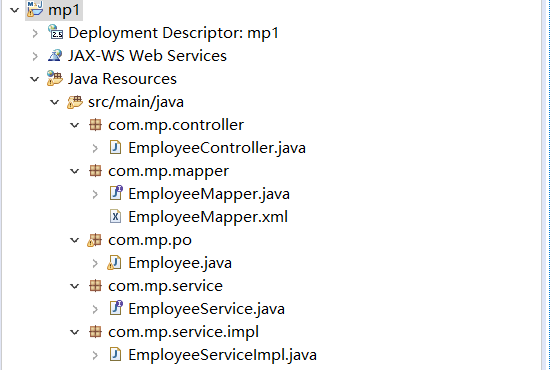
1. 策略配置：
2. 包名配置：



1. 整合前面所有的配置，执行：

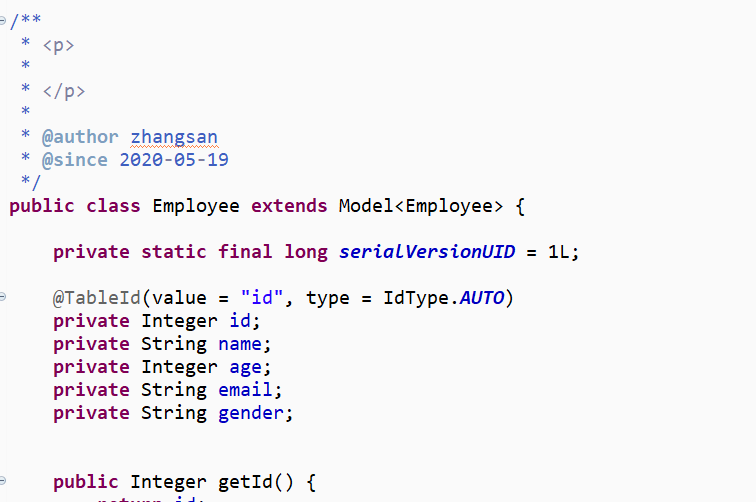


以上代码写完后，执行此单元测试方法后，会在指定位置生成相应的包和文件：

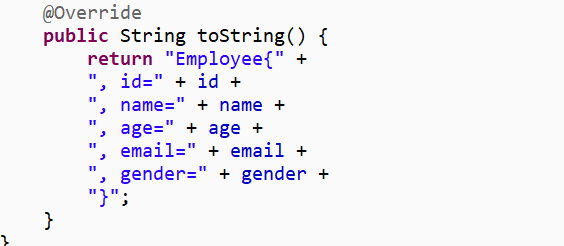


## 分析

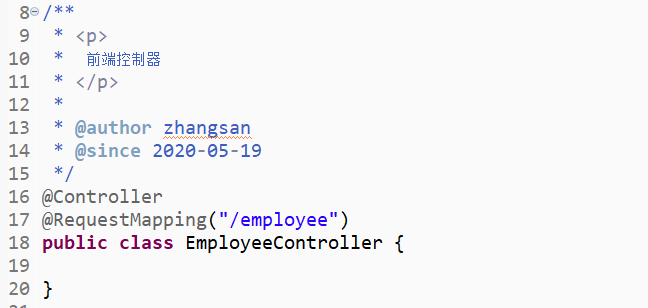
1. 自动生成的实体类：



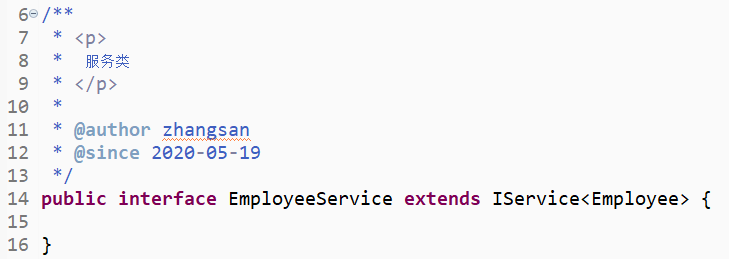
开启了ActiveRecord，覆写了toString( )方法：



1. 生成的控制器：

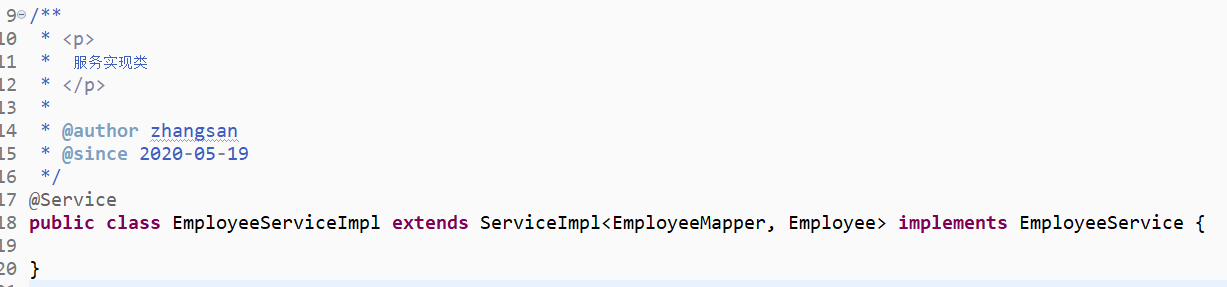


1. 生成的业务层接口：

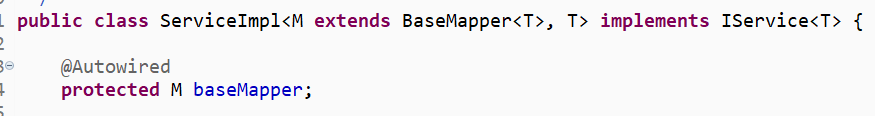


生成的业务层接口继承了IService接口，可以观察该接口中的源码发现它有什么作用；

1. 生成的业务层实现类：

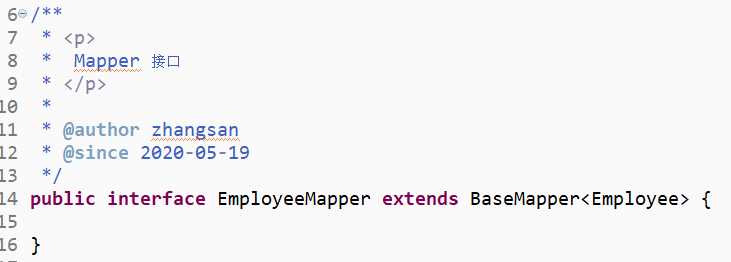


它实现了业务层接口时，也继承了ServiceImpl类，在ServiceImpl类中，会发现它嵌入了Mapper接口对象：



所以，在业务层无需注入Mapper接口bean；

5）生成的Mapper接口：



继承了BaseMapper接口，拥有了基本的CRUD方法；

1. 生成的Mapper映射文件：



它开启了二级缓存；

## 小结

以上只是MyBatisPlus自动代码生成器的基本使用，你可以参照官方文档深入了解它的用法[https://baomidou.gitee.io/mybatis-plus-doc/#/generate-code](https://baomidou.gitee.io/mybatis-plus-doc/" \l "/generate-code)，也可以观察**GlobalConfig、DataSourceConfig、StrategyConfig、PackageConfig、AutoGenerator**等类的内部源码来发现更多可以设置的属性，进一步熟悉此代码生成器的用法。

# 插件扩展

## MyBatis插件机制简介

1. 插件机制

MyBatis通过插件（interceptor）可以做到拦截四大对象相关方法的执行，根据需求，完成相关数据的动态变化。

**Executor**

**StatementHandler**

**ParameterHandler**

**ResultSetHandler**

1. 插件原理

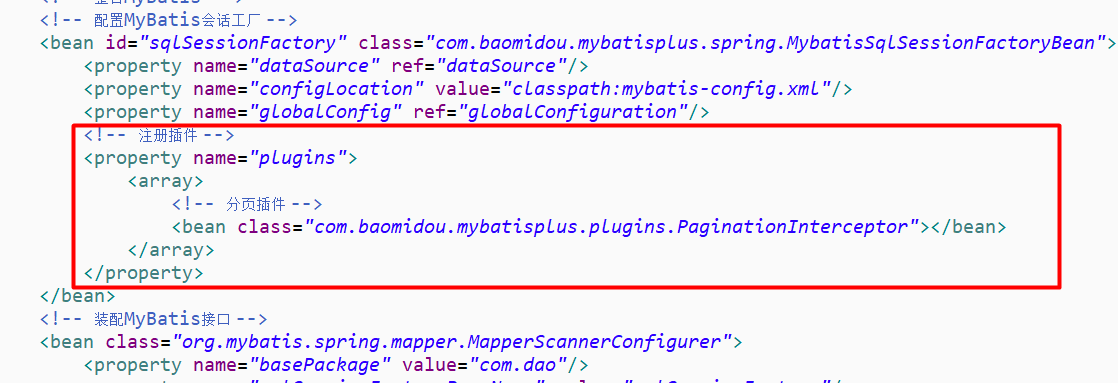
四大对象的每个对象在创建时，都会执行interceptorChain.pluginAll( )方法，会经过每个插件对象的plugin( )方法，目的是为了为当前的四大对象创建代理。代理对象就可以拦截到四大对象相关方法的执行，在存在插件的情况下，执行四大对象的方法都需要经过代理。

3）MyBatisPlus提供的插件使用的也是MyBatis的插件机制与原理。

## 分页插件

在前面的学习中是单纯的通过BaseMapper接口中的selectPage( )方法实现分页的，但是这种分页是假分页，属于内存分页，在查询时，不会使用到LIMIT，它会把所有的结果都查出来，然后进行筛选，这是非常不好的，为此，我们可以使用MyBatisPlus提供的PaginationInterceptor插件结合selectPage( )方法实现真正意义上的分页，也就是物理分页，PaginationInterceptor分页插件在查询时会使用LIMIT。

1. 在Spring配置文件中注册MyBatisPlus分页插件

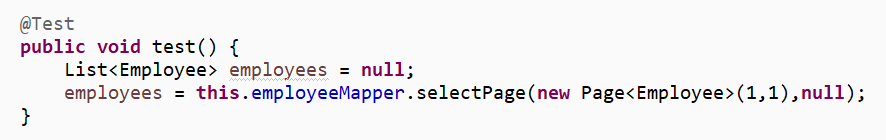


如果是在SpringBoot中使用，则自定义类，提供一个Bean：



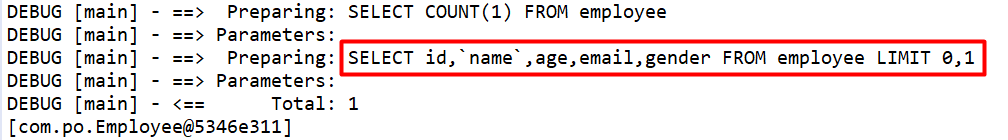
注意：类上需要使用@Configuratin注解进行标注。

1. 编写方法进行测试：



在selectPage( )方法中，第一个参数 Page对象表示查询第一页，每页显示一条，第二个参数是条件构造器，为null时表示查询所有记录；

3）执行测试方法，观察打印的SQL：



使用LIMIT查询，真正的实现了分页。

1. 从Page对象可以获取与分页相关的信息：



1. 也可以将查询的分页结果封装到Page对象中：



这样，只需要将Page对象传入到前端页面即可。

## 执行分析插件

SQL执行分析插件com.baomidou.mybatisplus.plugins.SqlExplainInterceptor也称SQL执行分析拦截器，只支持MySQL5.6.3以上版本，其作用是分析DELETE、UPDATE语句，防止小白或恶意进行DELETE、UPDATE全表操作，该插件只建议在开发环境中使用，不建议在生产环境中使用；该插件的底层，通过SQL语句分析命令：Explain分析当前的SQL语句，根据结果集中的Extra列来判断当前SQL执行是否是全表操作。

1. 注册执行分析插件



stopProceed属性的值为true，表示分析当前SQL是否是全表操作，如果是，则停止执行该SQL语句；

2）编写测试方法进行测试：

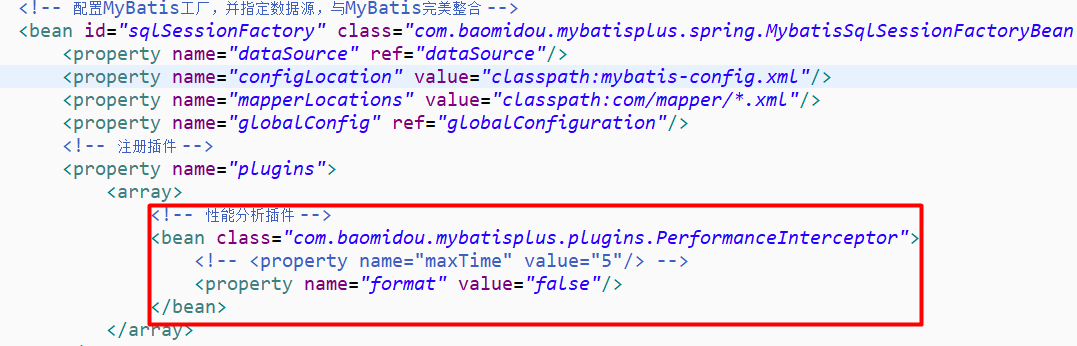


执行测试方法，会发现SQL执行无效，因为这是全表操作；

## 性能分析插件

性能分析插件com.baomidou.mybatisplus.plugins.PerformanceInterceptor用于分析SQL的执行性能，通过该插件也可以指定一个时间，当SQL执行超时，则会停止运行，有助于发现问题。（只建议在开发环境中使用）

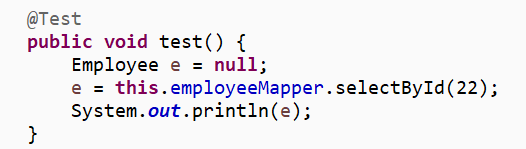
1. 注册性能分析插件：



属性maxTime为SQL执行超时时间，单位为毫秒，当SQL执行超时，会抛出异常，有助于发现问题，这个超时时间要合理设置，这里设置的是5毫秒是非常夸张的，一条SQL语句不可能执行只需要5毫秒时间，所以会抛出异常；

属性format值为true表示格式化输出打印的SQL语句，默认为false；

1. 编写测试方法进行测试：

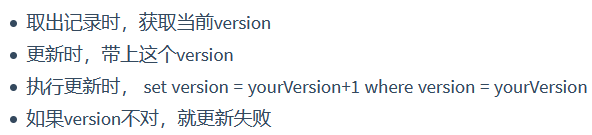


性能分析信息：



## 乐观锁插件

如果想要实现一个需求：当要更新一条记录时，希望这条记录没有被别人更新。乐观锁的实现原理：



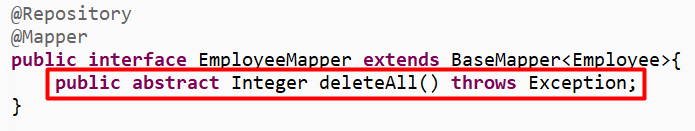
@Version注解用于标注实体字段，必须要有；

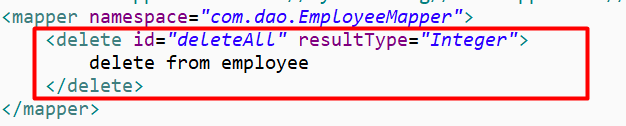
乐观锁插件可以保证用户在修改一条数据时，其他用户无法修改。

# 自定义全局操作

## 简介

当MyBatisPlus提供的基本CRUD方法不能满足开发需求时，可以在Mapper接口和Mapper文件中定义：

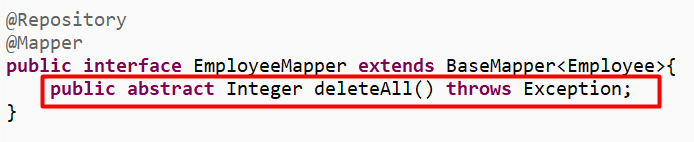




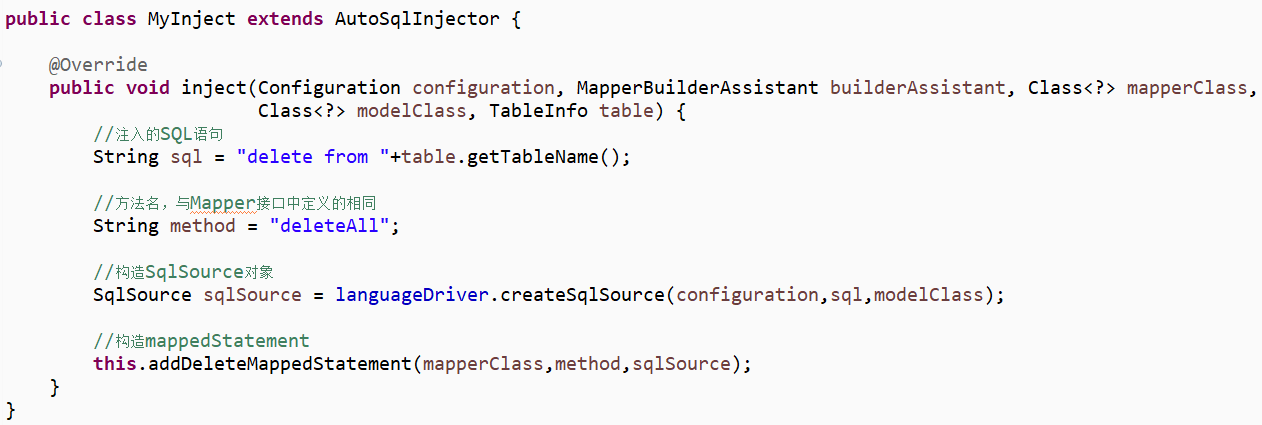
但除了这种方式外，还可以通过另一种方式来定义，即自定义全局操作（AutoSqlInjector），通过自定义全局操作（AutoSqlInjector）自定义SQL操作，将其注入到MyBatisPlus中，相当于对MyBatisPlus原有的CRUD方法进行了扩充。

## 使用

1. 在mapper接口中定义相关的CRUD方法：



1. 自定义类继承AutoSqlInjector扩展AutoSqlInjector的inject( )方法，实现mapper接口中定义的CRUD方法的功能：



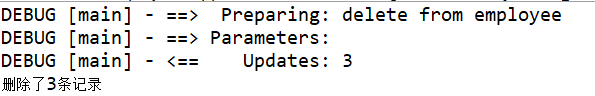
1. 将自定义的扩展类注入到全局策略中：



1. 编写测试方法进行测试：



测试结果如下：



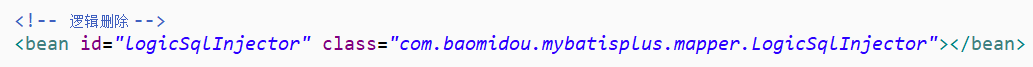
# 逻辑删除

## 简介

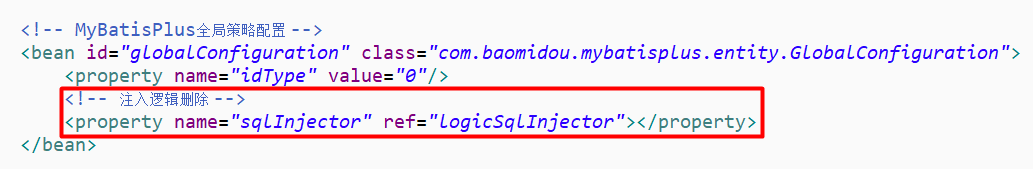
1. 逻辑删除也称假删除，它并不会从数据库中真正的将数据删除，而是将当前被删除的这条数据中的一个逻辑删除字段置为删除状态。如在创建用户表的时候，预置了一个名为lock的字段，用户活跃，设置为1，用户注销账户时，将lock字段值修改了-1，表示该用户已被删除，但在表中并没有真正的删除。
2. MyBatisPlus提供了com.baomidou.mybatisplus.mapper.LogicSqlInjector实现逻辑删除。

## 使用

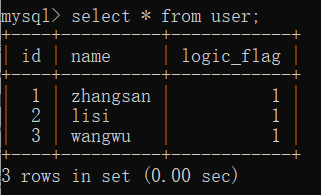
1. 在Spring配置文件中配置逻辑删除：



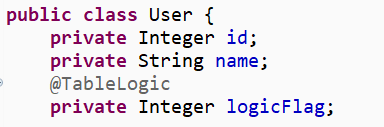
1. 在MyBatisPlus全局策略配置中注入逻辑删除：



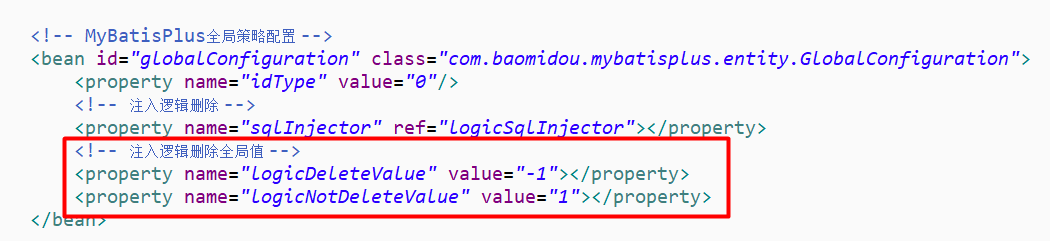
1. 存在user表，表中数据如下：



其中，loginc\_flag字段及逻辑字段，为1表示这个用户是正常使用的，没有被删除。

1. 创建实体类，需要使用@TableLogic注解标注逻辑删除字段：

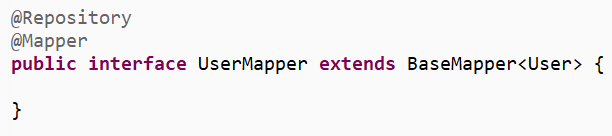
5）在MyBatisPlus全局策略配置中配置全局逻辑删除值：



logicDeleteValue值为-1，表示logic\_flag值为-1时，此用户是删除状态；

logicNotDeleteValue值为1，表示logic\_flag值为1时，此用户是非删除状态；

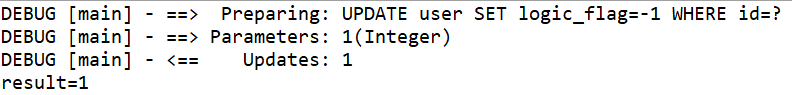
1. 为user表创建Mapper接口：



1. 进行测试：



执行测试方法，观察运行结果：



明明执行的删除操作，但是执行的SQL却是更新操作，将逻辑删除字段修改为预先指定的逻辑删除全局值-1，这就实现了逻辑删除。

# 公共字段填充

## 简介

1. 在进行插入或修改操作时，有一些字段我们没有给它提供值，这个时候我们希望有默认值给它填充上去，如果想要实现这个功能，则可以使用MyBatisPlus提供的公共字段填充功能。
2. 使用公共字段填充功能时，需要认识一个接口：

com.baomidou.mybatisplus.mapper.MetaObjectHandler

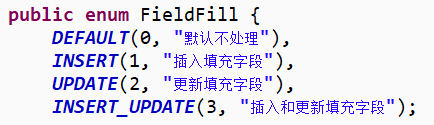
在这个接口中，需要关注两个方法：



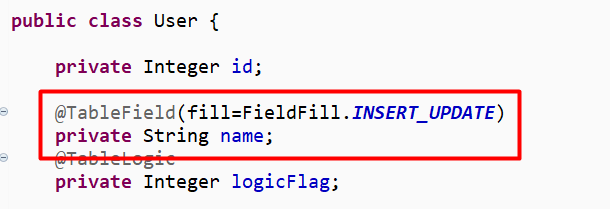
insertFill( )方法用于在插入时进行填充；updateFill( )方法用于在更新时进行填充；它们都有一个MetaObject对象参数，MetaObject表示的是元对象，是MyBatis提供的一个用于更加方便，更加优雅的访问对象的属性，给对象的属性设置值的对象，它还可以用于包装对象，支持Object、Map、Collection等等对象进行包装。使用MetaObject获取对象的属性值或设置对象的属性值其原理是反射技术。

## 使用

1. 在实体类中在需要填充的属性上使用@TableField注解标注，通过它的fill属性指定填充策略，fill属性值是一个FieldFill枚举类型值：



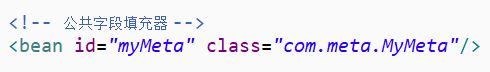
DEFAULT表示不填充字段（默认的），INSERT表示插入时会填充字段，UPDATE表示更新时会填入字段，INSERT\_UPDATE表示插入和更新时都会填充字段；

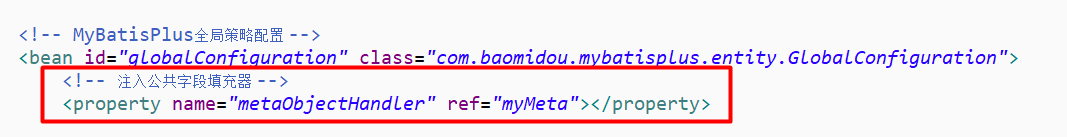


1. 自定义公共字段处理器，指定填充条件和填充数据：

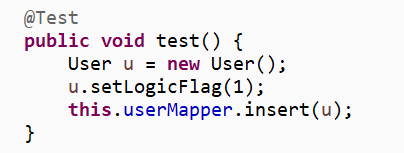


1. 在Spring配置文件中配置公共字段处理器，并将其注入到MyBatisPlus全局策略配置中：

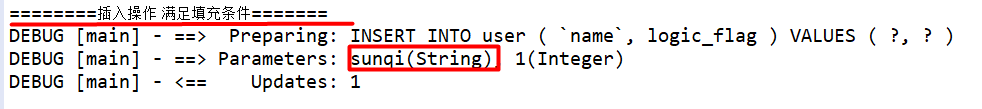




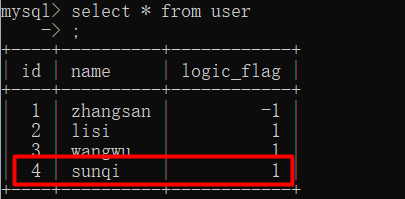
1. 进行测试，执行插入操作：



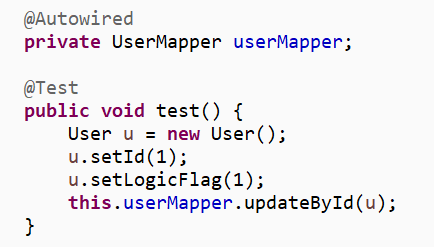
故意不设置name属性的值，观察是否会填充，先观察打印的SQL：



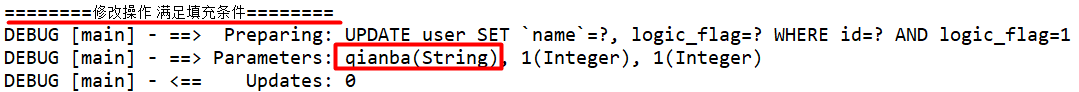
填充成功，也可以观察数据库：



1. 进行测试，执行更新操作：



观察打印的SQL：



填充成功，但在使用updateById( )更新数据时，如果不指定属性的值则表示不更新这个字段的值，而你又通过字段填充功能向属性中填充了，这时候它就会修改字段的值，达到了事与愿违的弊端。