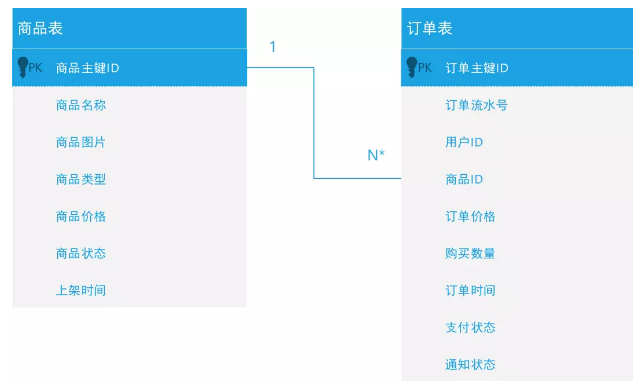
# Mybatis-Plus的应用场景及注入SQL原理分析

# 一、背景

## 1.1 传统Mybatis的弊端

### 1.1.1 场景描述

假设有两张表：一张商品表、一张订单表，具体表的字段如下：



**现有如下需求：**

* 分别根据id查询商品表和订单表所有信息
* 根据支付状态和通知状态查询订单表信息
* 对订单表增加一个订单状态，根据订单状态查询订单信息

### 1.1.2 需求

**需求a：**根据id查询商品表：

​@Select(" SELECT p.id ,p.name ,p.picture ,p.type ,p.price, p.type, p.time

FROM product p where id = #{id} ")

List<Product> getProductsBYId(@Param("id") Integer id);

根据id查询订单表所有信息：

@Select(" SELECT o.id ,o.pay\_no ,o.user\_id ,o.product\_id ,o.pay\_price, o.num, o.pay\_time, o.order\_type, o.notif\_type

FROM order o where id = #{id} ")

List<Order> getOrderBYId(@Param("id") Integer id);

**需求b:**根据支付状态和通知状态查询订单表信息

​@Select(" SELECT o.id ,o.pay\_no ,o.user\_id ,o.product\_id ,o.pay\_price, o.num, o.pay\_time, o.order\_type, o.notif\_type

FROM order o where order\_type= #{orderType} ")

List<Order> getOrderBYId(@Param("orderType") Integer orderType);

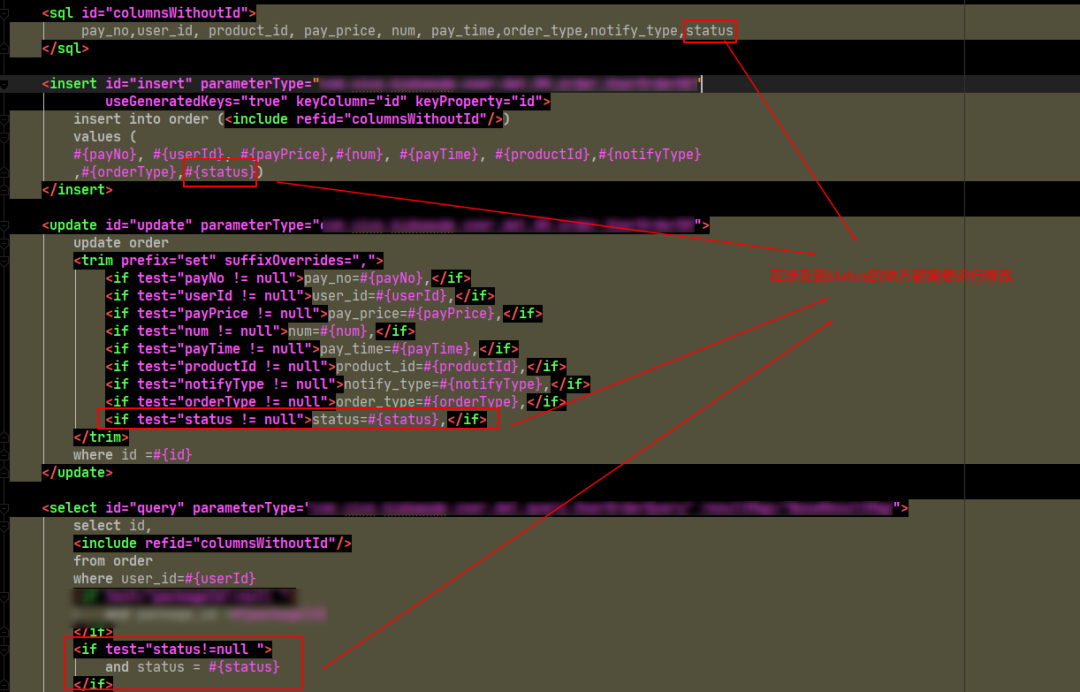
@Select(" SELECT o.id ,o.pay\_no ,o.user\_id ,o.product\_id ,o.pay\_price, o.num, o.pay\_time, o.order\_type, o.notif\_type

FROM order o where notify\_type= #{notifyType} ")

List<Order> getOrderBYId(@Param("notifyType") Integer notifyType);

**需求c:**对订单表增加一个订单状态status，根据订单状态查询订单信息。

传统mybaits需要三步：首先需要在订单表里加个字段，然后在订单的实体类添加这个属性，并且将所有dao层设计该状态的的查询sql都修改一遍，加上这个字段



### 1.1.3 上述方式有什么问题呢？

需求a：对于不同的实体类，即使查询的目的一致，仍然需要重复构造类似的sql语句，仅仅是表字段和表信息不同。

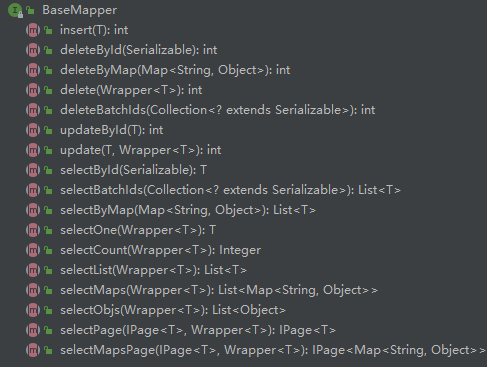
需求b：对于相似的查询条件，针对某个单一场景必须构造不同的sql，造成sql语句的大量冗余。

需求c：将dao层所有涉及到新增字段的sql都需要修改一遍，这个过程比较繁琐且容易出错。

使用mybatis-plus就可以解决上述问题。

### 1.1.4 Myatis-plus的解决方案

首先让ProductMapper和OrderMapper继承BaseMapper类：



public interface ProductMapper extends BaseMapper<Product> {

}

public interface OrderMapper extends BaseMapper<Order> {

}

**需求a：**

分别根据id查询商品表和订单表：由于BaseMapper中提供了selectById的方法，可以直接根据具体业务场景，传入指定的参数例如（id=1）即可。无需书写具体的sql语句，至于sql自动生成的原理将在下面介绍；

productMapper.selectById(1);

orderMapper.selectById(1);

**需求b：**

此时利用BaseMapper.selectList(Wapper queryWrapper)方法直接构造查询条件，例如查询支付状态为2和通知状态为1的订单信息

orderMapper.selectList(new QueryWrapper<Order>().eq("orderType",2));

orderMapper.selectList(new QueryWrapper<Order>().eq("notifyType",1));

orderMapper.selectList(new QueryWrapper<Order>().eq("orderType",2));orderMapper.selectList(new QueryWrapper<Order>().eq("notifyType",1));

此时我们可以发现：使用了Mybatis-plus以后，我们更加聚焦于业务本身，对于上述相似的应用场景，无需构造雷同的SQL，利用包装器直接传入查询条件。

**需求c：**

前两步与传统mybatis一致，由于MyBatis-plus无需手动创建SQL，因此减少了大量的重复劳动。

## 1.2 MyBatis-Plus的定位

MyBatis-Plus (opens new window)（简称 MP）是一个 MyBatis (opens new window)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。

## 1.3 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](https://cloud.tencent.com/product/tencentdb-catalog?from_column=20065&from=20065)：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、[SQLServer](https://cloud.tencent.com/product/sqlserver?from_column=20065&from=20065) 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。

## 1.4 原理解析

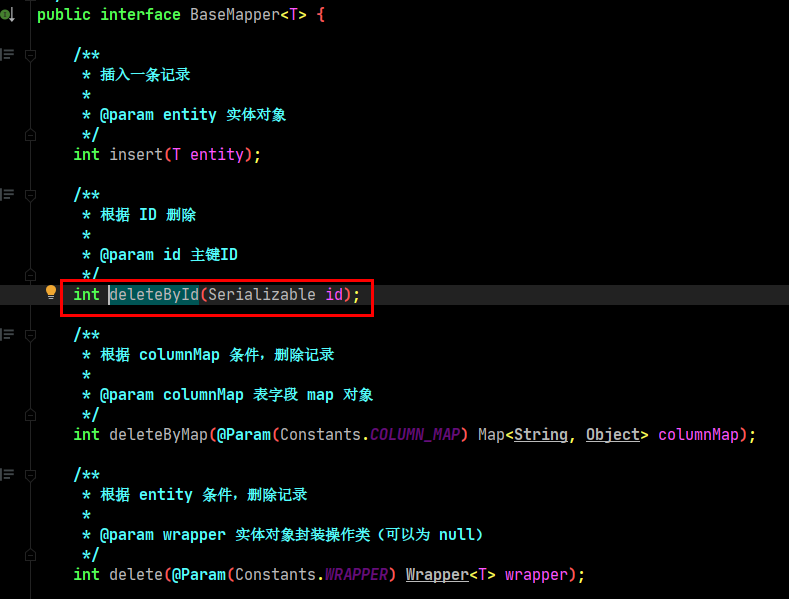
本文针对MyBatis-plus中的核心功能：SQL自动注入功能，进行流程分析及原理探究。

# 二、准备工作

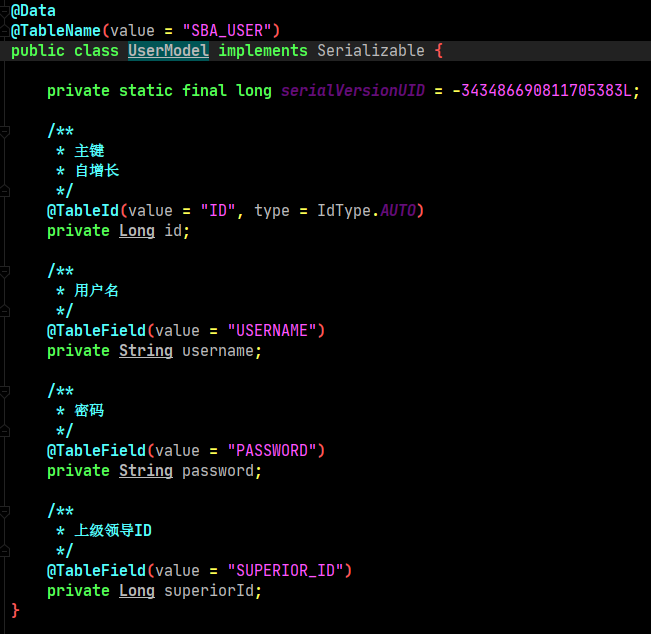
先从一个简单的demo入手，感受一下MyBatis-plus的便捷性。

## 2.1 基础接口BaseMapper

源码中提供了一个基础接口，里面包含了基本的增删改查方法。

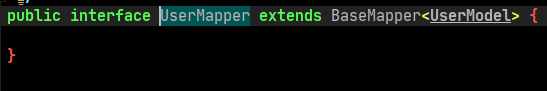


## 2.2 创建实体类对象

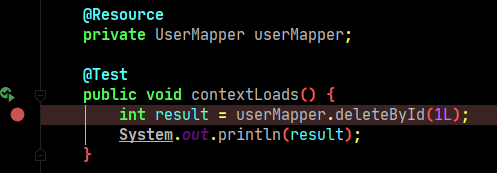


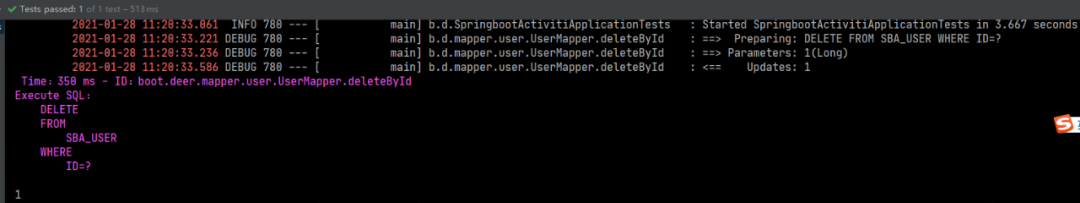
## 2.3 业务接口UserMapper

业务中根据具体实体对象，继承该抽象接口。



## 2.4 测试用例





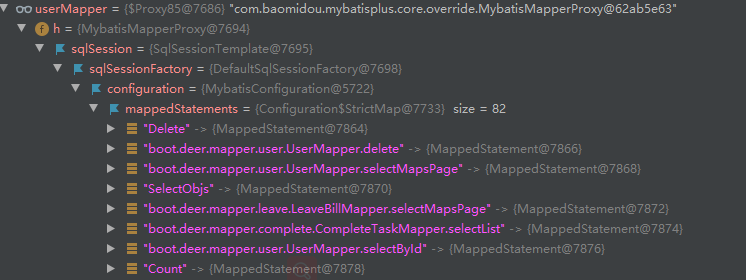
控制台显示：MyBatis-plus最终为我们自动生成了SQL语句。根据上述操作分析：UserMapper继承了BaseMapper，拥有了deleteById的方法，但是MyBatis-plus是基于mybatis的增强版，**关键在于最终仍然需要提供具体的SQL语句，来进行数据库操作**。

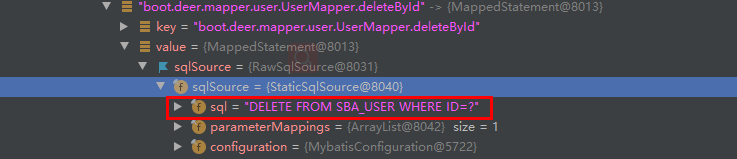
下面就通过debug由上而下分析mybatis-plus是如何生成业务sql以及自动注入的。

# 三、SQL语句存储在哪里？

mappedStatements:描述sql信息

如下图所示：mybatis为我们生成了一个代理对象，里面包含了一些重要的属性。





**具体如下：**

userMapper——>mybatisMapperProxy——>sqlSession——>sqlSessionFactory

——>configuration——>mappedStatements——>mappedStatement——>sql语句

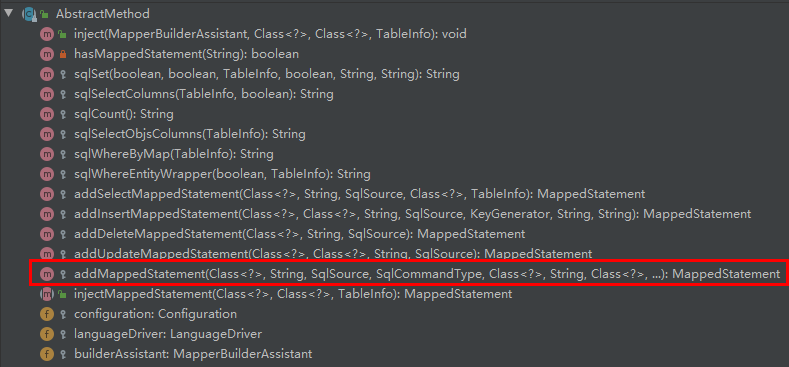
至此我们可以发现**每一个SQL语句对应一个mappedStatement**，mappedstatements存储在configuration文件（configuration是mybatis的全局配置文件，包含数据源、mapper、其他配置信息）中。

# 四、SQL语句是什么时候注入的？

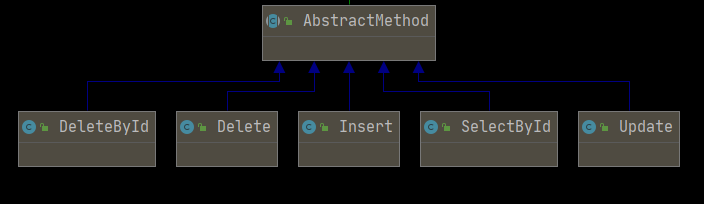
## 4.1 AbstractMethod.addMappedStatement

基于上面的分析，想要知道SQL语句什么时候拿到的，就是要找到mappedStatement被添加的位置。追踪到AbstractMethod的抽象方法中。

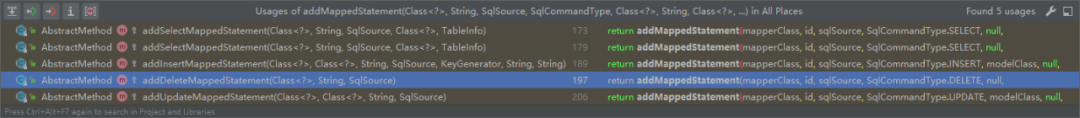
找到了addMappedStatement()方法

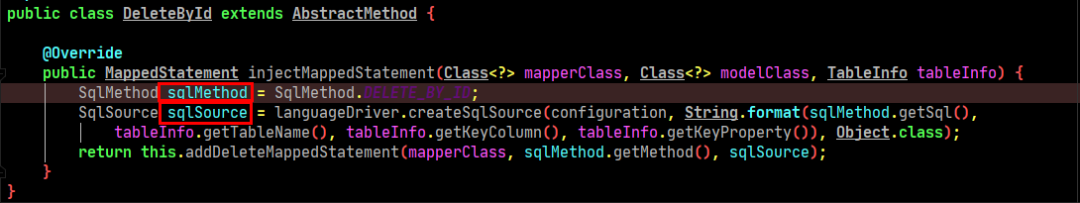


而BaseMapper的所有方法（deleteById、delete、insert、select、update等）都继承了该抽象方法。



根据mapper方法（deleteById）显然是调用addDeleteMappedStatement方法。

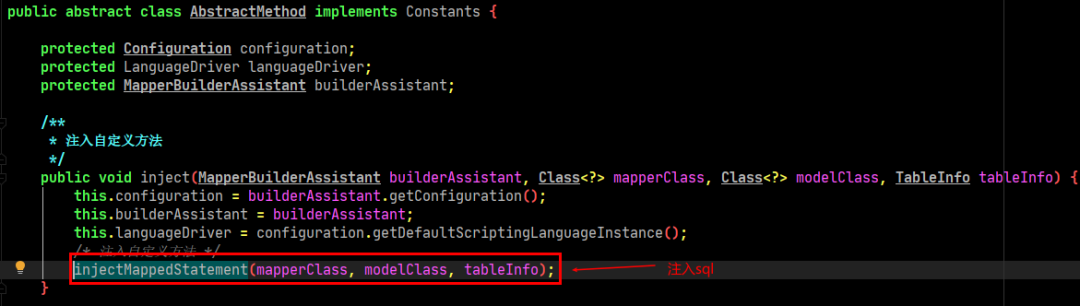


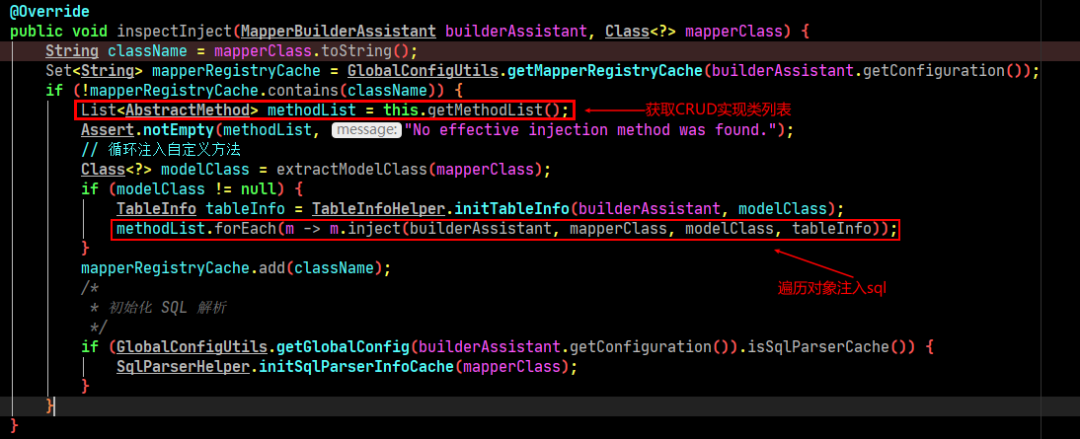


这里我们可以发现，源码中根据不同的方法继承AbstractMethod实现了不同的实现类，并且实现了injectMappedStatement方法，sqlSource也是在这个地方被添加进配置文件。

## 4.2 AbstractMethod.inject

继续研究AbstractMethod抽象类，inject方法实现了自动注入sql的动作。





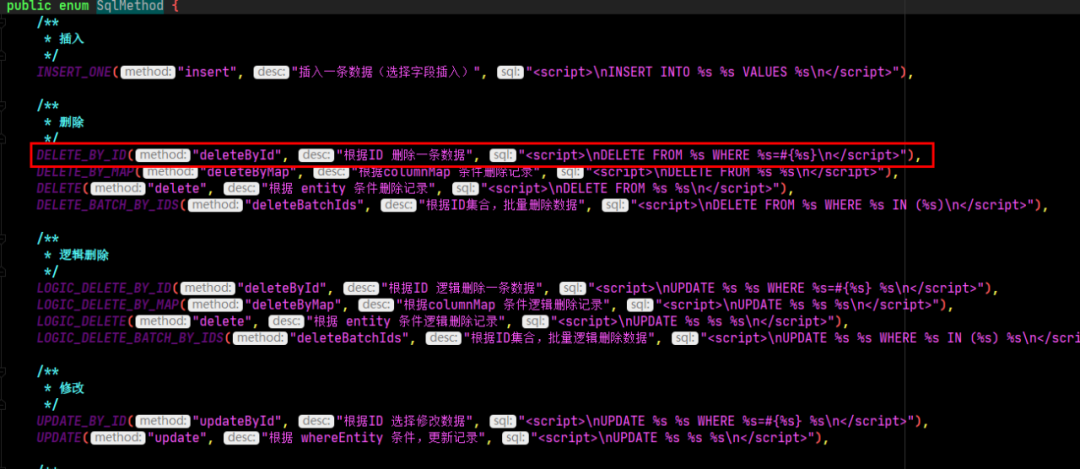


有上述源码可知，项目启动时，首先由默认注入器生成基础CRUD实现类对象，其次遍历实现类列表，依次注入各自的模板SQL，最后将其添加至mappedstatement。

# 五、SQL语句是怎么生成的？

## 5.1 SQL模板

上述方法中有两个关键的参数：SqlMethod、SqlSource；

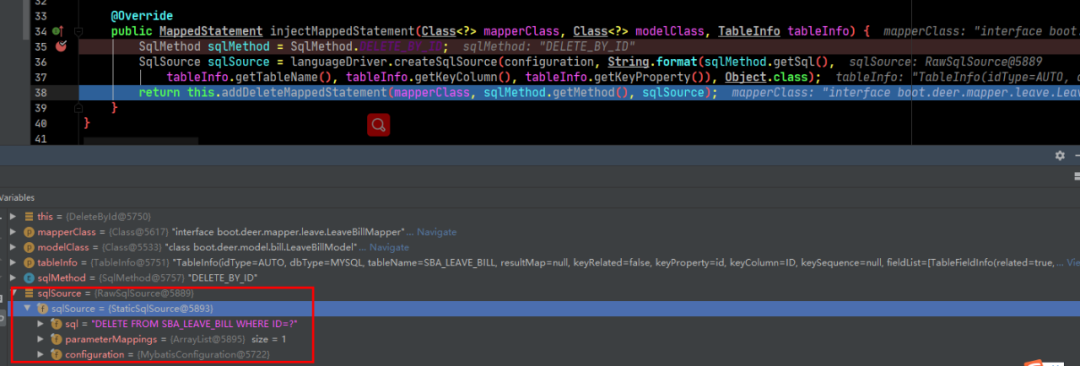


继续研究源码发现：sqlMethod本质上是一个枚举类，存储了两个关键的元素：

BaseMapper中的方法名； 方法名对应的sql语句模板（即被<scripe>标签包裹的字符串）。

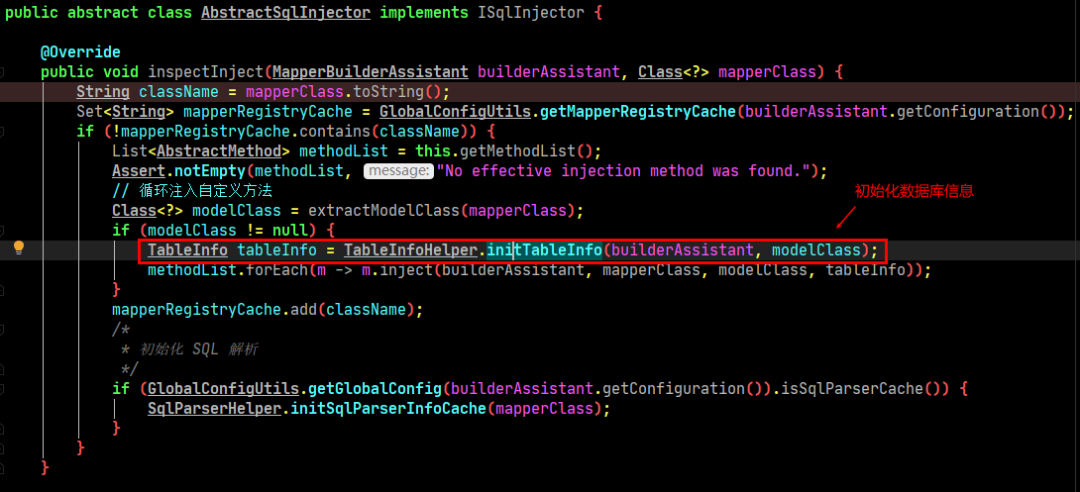
到这里我们基本了解了mybaits-plus实现sql自动生成的本质：根据不同的方法来提供一些通用的模板，项目启动后再加载进mappedStatement。

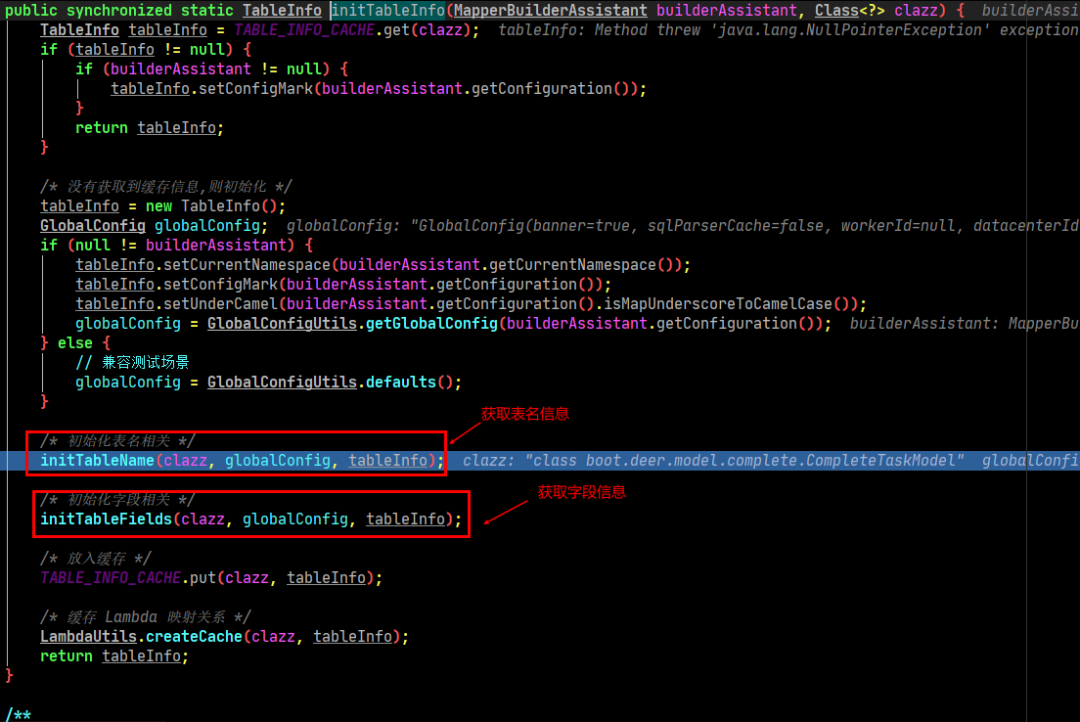
## 5.2 SqlSource



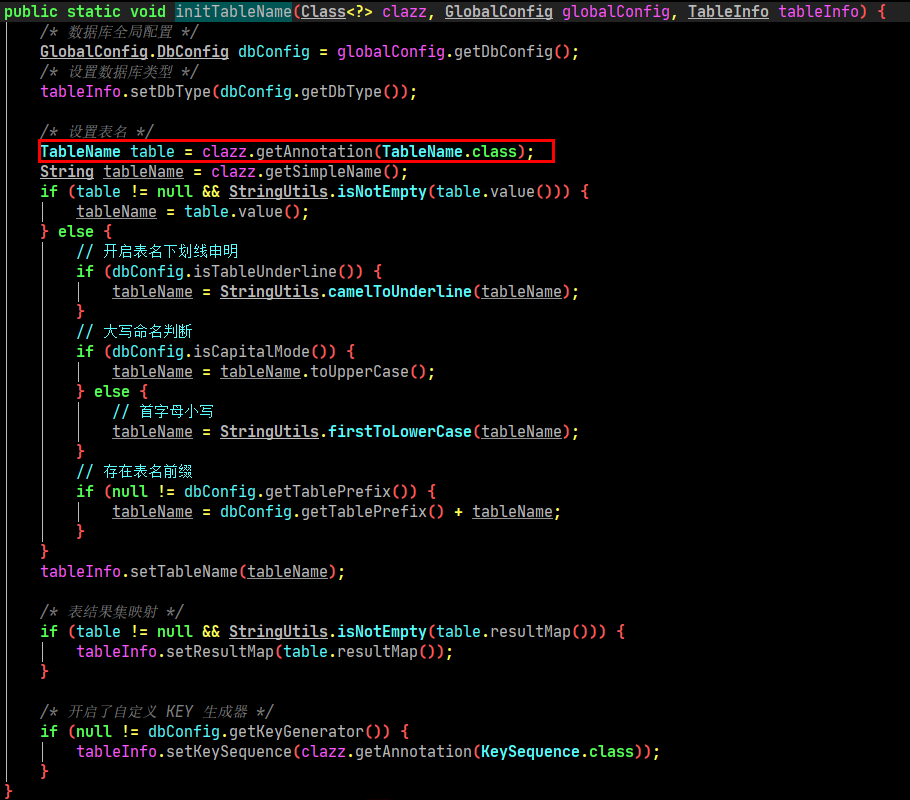
此时SqlSource通过解析SQL模板、以及传入的表信息和主键信息构建出了SQL语句。

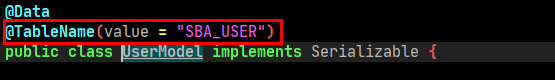
## 5.3 数据库表信息是如何获取的？





分析initTableName()方法：获取表名信息源码中传入了实体类信息clazz，其实就是通过实体上的@TableName注解拿到了表名；

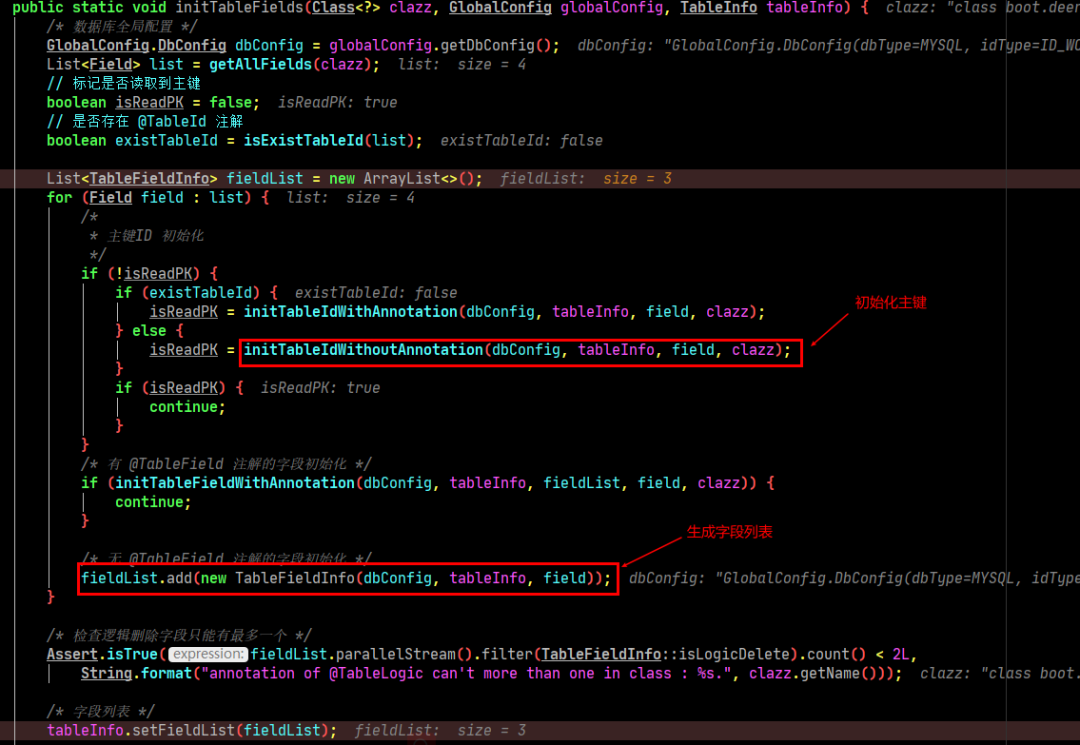




我们在定义实体类的同时，指定了该实体类对应的表名。

**分析initTableFields()方法：**

获取主键及其他字段信息

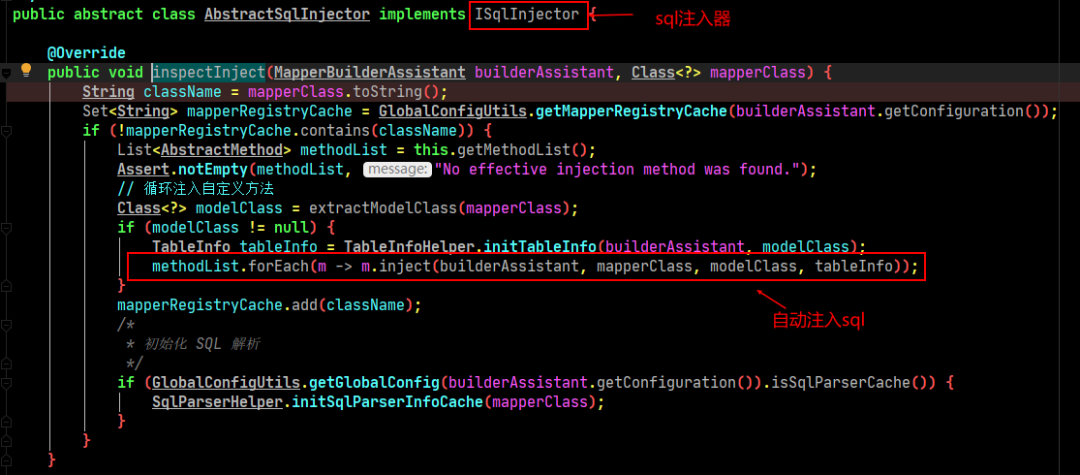


至此tableInfo的信息已经注入完成了。

在研究完解析mapper的核心过程之后，我们再简单看下mapper文件被添加到configuration（mybatis核心配置文件）的过程。

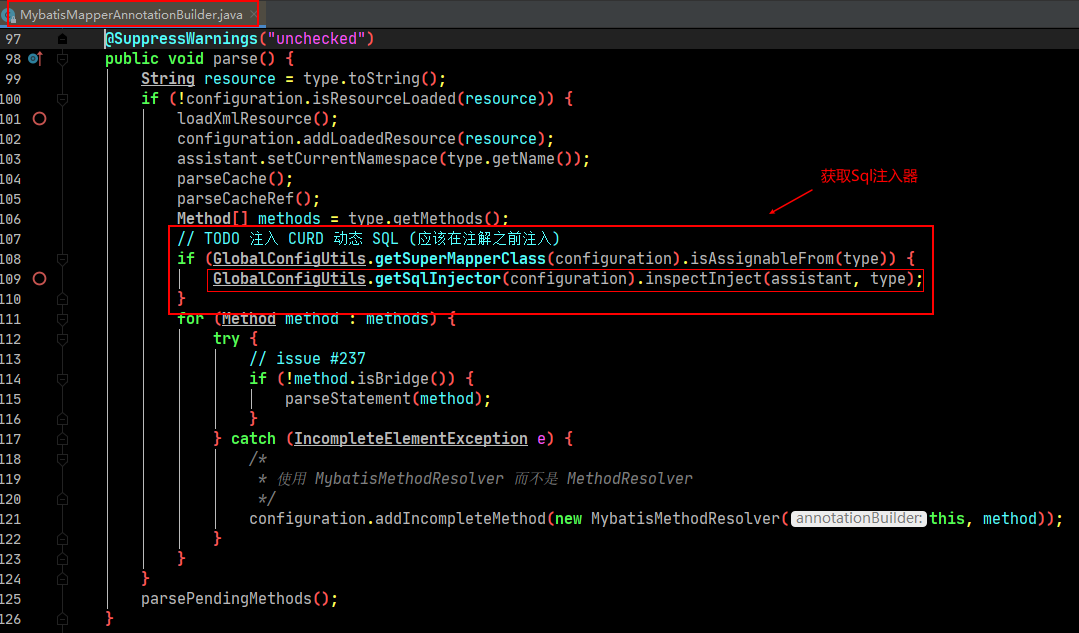
# 六、mapper文件被添加的过程

**ISqlInjector：Sql注入器**



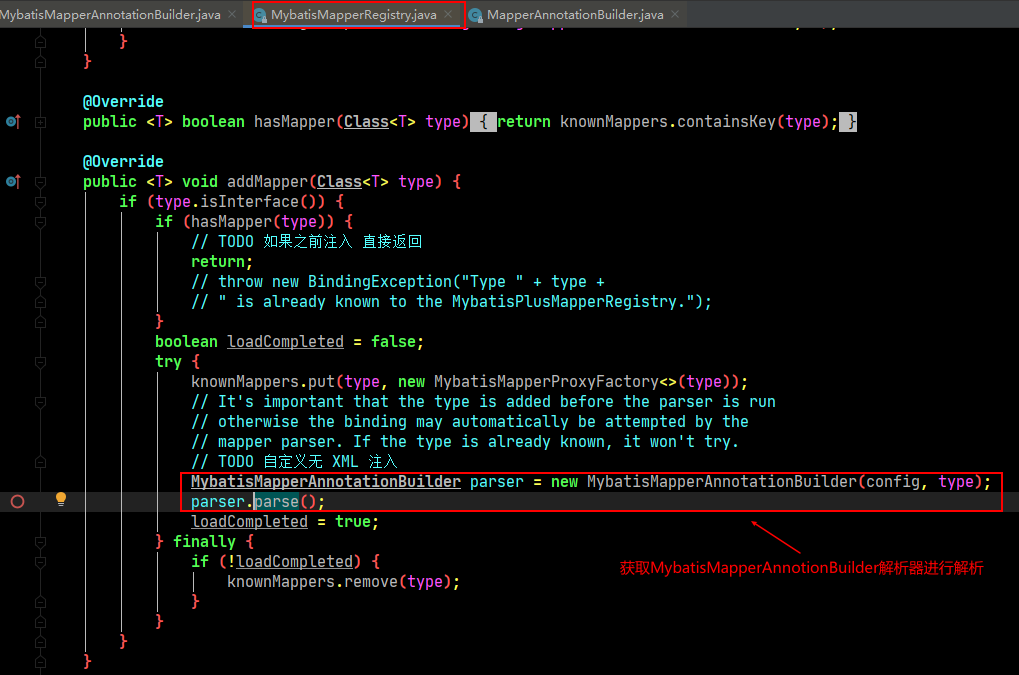
**MybatisMapperAnnotationBuilder:mapper解析器**

MybatisMapperAnnotationBuilder中的parse方法获取了sqlInjector（Sql注入器）来进行SQL注入。

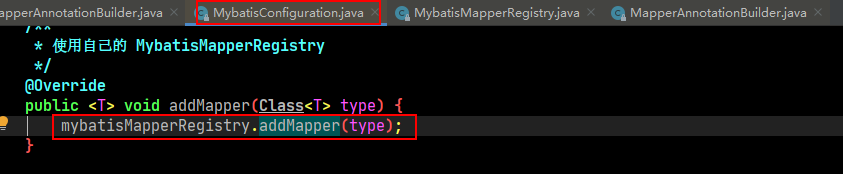


**Mybatis添加mapper的固有流程：**MybatisMapperRegistry

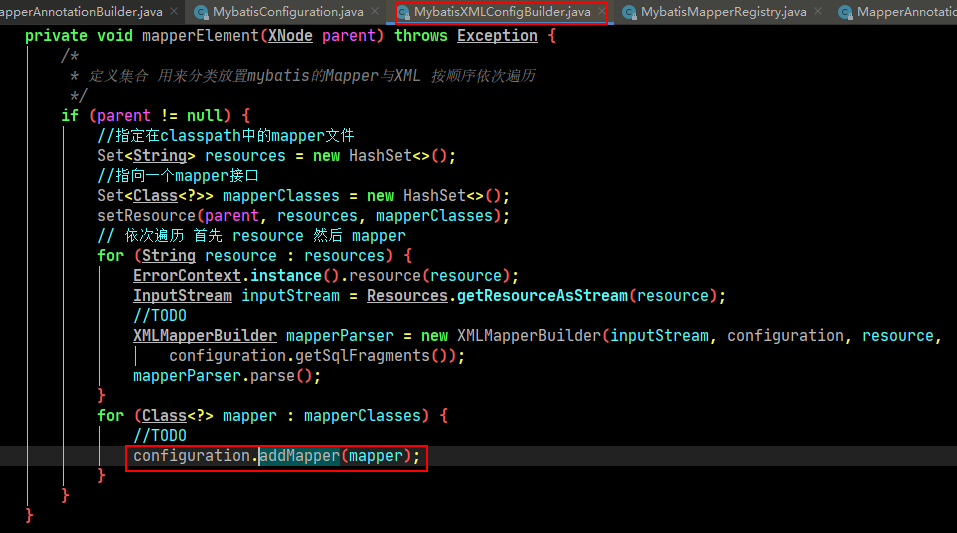
调用MapperAnnotionBuilder解析器进行解析



**MybatisConfiguration.addMapper**

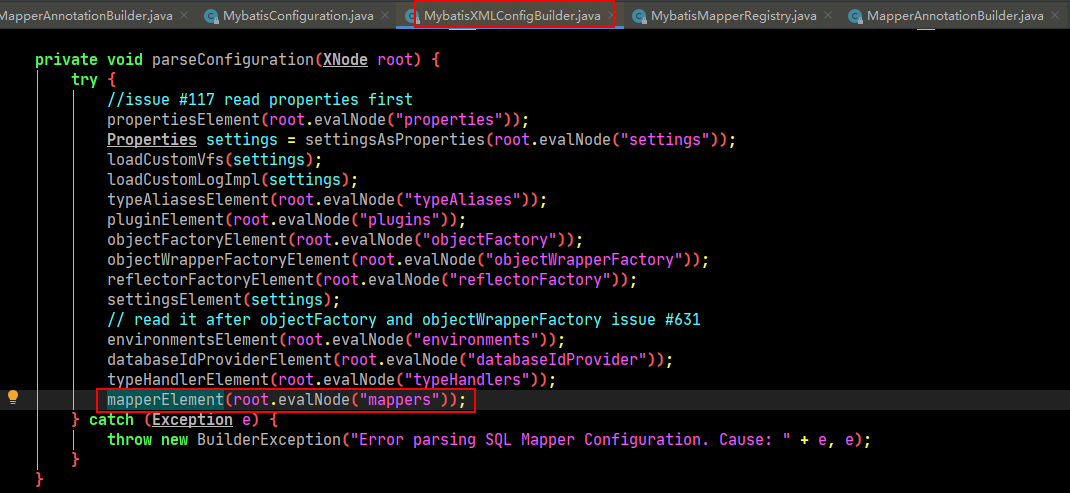


**MybatisXMLConfigBuilder.mapperElemnt**



**MybayisXMLConfigBuilder.parseConfiguration**

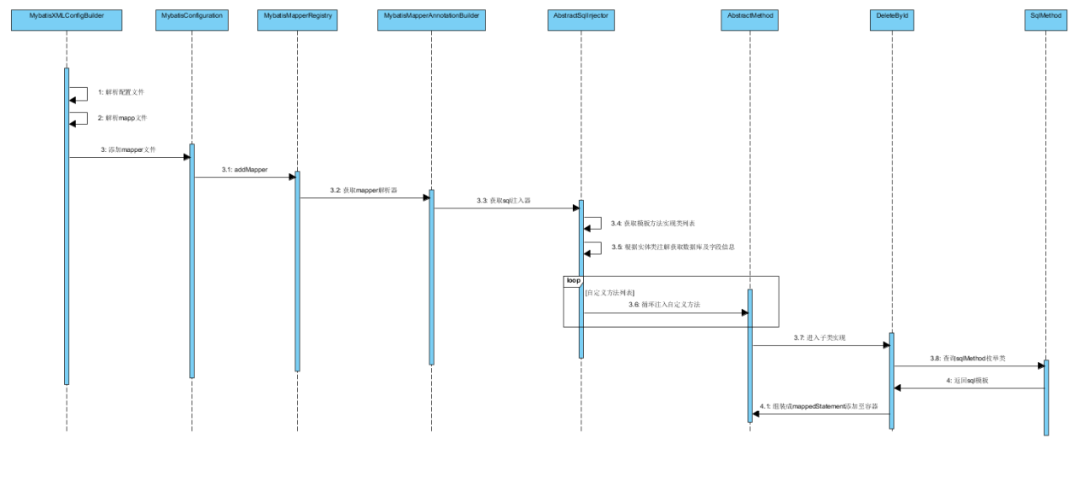
添加mapper文件的过程分析到这里就完成了。



# 七、总结

## 7.1 流程梳理

下面总结梳理了一下mybatis-plus解析mapper文件自动注入sql的主要流程。



## 7.2 Mybatis-plus的ORM的核心思想

1）实体类和数据库表通过自定义注解来完成一一映射。

2）对象属性和字段同样使用注解来一一对应（命名注意要相同）。

3）为了提高复用性使得具体的mapper继承通用的增删改查接口。

4）利用模板方法和对象属性值动态拼接SQL。

# 八、参考文档

MyBatis-plus官方文档：[https://mp.baomidou.com/](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fmp.baomidou.com%2F&objectId=1828212&objectType=1&isNewArticle=undefined)