**Mybatis-plus-taste**

发现自己一直逛的超市也叫taste，巧

这里作为taste过程记录的新位置，在这里直接贴图，贴代码都会比较方便，希望能够节省出更多的时间。

# 201903151053

跑测试用例的时候，如果发现有相关Statement找不到的情况，可以检查一下target相应目录下面是否有XXXMapper.xml.若不存在，则需要手动复制一份对应的mapper.xml配置过去，才可以。

# 201903151345

发现一个问题，将数据库连接相关的配置存放到属性文件中，mybatis-config.xml用properties标签指定了属性文件配置，调试跟踪源码之后，发现也确实有去读取属性文件内容，但是配置连接池信息处，引用了属性文件(jdbc.properties)中的属性，却没有得到正确的转换，比如driver，它在配置DataSource对象的时候，拿到对应driver的值变成了“${jdbc.driver}”，而不是”com.mysql.jdbc.Driver”。

于是来了兴趣，找了mybatis配置DataSource对象的相关源码，它也没有去读出properties文件内的key，直接将字面量赋值到DataSource中的相应属性。这会是什么原因？难道是先后顺序吗?

# 201903152330

还是上面说的问题，对DataSource类成员变量设置值的代码有翻到了，觉得mybatis中实现中并没做相关处理——从属性文件中获取同名属性

# 201903171036

版本号：d5f17395ca91e37d2aeba03144090b272e6d5051

现在在测试分页插件.测试代码挺简单的，测试结果也十分准确。

这次测试的背后知识点是：Mybatis下RowBounds分页实现,在分页拦截器中禁用RowBounds内存分页，使用拼接后带分页语法的sql进行查询。

有关RowBounds，我引用别人的文章过来提供参考。

【原文地址】https://blog.csdn.net/u010077905/article/details/38469653

RowBounds类，用于实现分页查询。RowBounds中有两个属性，offset和limit。



RowBounds在处理分页时，只是简单的把offset之前的数据都skip掉，超过limit之后的数据不取出(内存分页)，上图中的代码取自MyBatis中的DefaultResultSetHandler类。跳过offset之前的数据是由方法skipRows处理，判断数据是否超过了limit则是由shouldProcessMoreRows方法进行判断。简单点说道，就是先把数据全部查询到ResultSet，然后从ResultSet中取出offset和limit之间的数据，这就实现了分页查询。

小结论：内存分页是从数据库中查出所有数据后，mybatis框架排除掉范围外数据。从性能上来说，这种分页方式不是很好。

# 201903171246

分页插件相关的开发以及测试程序都已经完整了，接下来分支版本是补全test用例，同步一下官方的Readme,提交一次git.

完成了上面的事情，那么框架的版本号也跟着推进到 4b6a7d4d9e08e1c987cfec73d4d78f593a3d28a0

# 201903171519

版本号：0bacdf4908fb4d8430edb9683e226e4840842dff

增加spring的支持，在wiki包下展示了结合spring的相关mybatis的配置，才能够如常使用plus增强的功能。

# 201903212259

版本号：3389a0a89d7d4e8e4eebebedbc246ea9b6d8ea3a

来到了这里，主要内容是提供Mybatis-plus在Spring多环境中的支持。但是我理解的可能有点不到位，也缺乏去试验的兴趣。在这个版本之后的commit说是增加对web mvc的支持，这就让我比较疑惑，为什么会有这些.？所以我决定观察一下，若没有相关的测试用例，我就不尝试那一部分了。

# 201903222117

版本号：7d999652e14e062d0efc433f28c1eec6fc87b821

这次是对分页插件中，查询总数量方法的修改，写得很明显，不支持含有”order by”关键字地count查询。虽然觉得处理得有点马虎，但是这还是版本的初期，肯定是以大方向为主，细节属于日常优化阶段处理。

这次的疑问是：

~~为什么order by对查询数据总数有影响？~~哈哈，这个问题本身就问的不对。count()会对order by产生什么影响？

在网上找到：count() 会造成order by 排序无效，这点倒是正确，count()把结果归并为一个数据，而order by的目的可能是为了弄一个有序数据集出来。

但是，这里貌似不影响结果。

原来，这里是不可以按照个人想法去盲猜的，下结论的根据是怎样，心里要明白才好。

我从count方法中找出查找总记录数的sql，实验一下：

SELECT COUNT(0) FROM ( originalSQL ) AS TOTAL;

这里sql和我盲猜用的查询总记录数sql，两条语句根本就不同，结果也很难走到一块过去。

实践一下mybatis-plus用的这一条就明白了。



经过实践，我看到这种select count(0)的方式并没有什么特别，照样能够计算出正确的结果。所以就不明白了官方为什么要做这次的修改。

# 201903232231

版本号:3adee82963762c966e6398d39a82b8ff44271f23

添加批量操作 hubin 2016/2/15 18:14

新的api: deleteByIds(List<Long>), insertBatch(List<T>)

# 201904021745

版本号: 3adee82963762c966e6398d39a82b8ff44271f23

还是这一个分支，这次跑通了测试，能够执行新api：批量删除与批量增加。

这次的taste中，发现原本作为 AutoInjector 类的静态常量（代表sql语句，mapper-statement-id），被抽取到SqlMethod枚举类中,代码逻辑变得更加清晰起来。

commit.

# 201904021758

版本号：8769c675ec212d85863a80f13b019886520e2cae

需要纠正一些，上一个版本中，批量删除api并不属于mapper增强的sql部分，所以我上面说的是有误的。

不过，在这次的版本中，大致地了解到mp开始为mapper类增强了批量删除，批量修改的功能。

# 201904004

[1159]

版本号：8769c675ec212d85863a80f13b019886520e2cae，6670ee72eb8d13baab6872578eae846d126f9bd4

结合这两个来看，mp一开始打算做批量修改的增强，后面我在taste这部分的过程中实在是别扭到极点，印象中mysql也没有类似可以用来修改多个对象的update语句啊.现在结合6670ee版本的修改后，也就释疑了。mp框架中后面也是舍弃掉批量修改这部分，毕竟正确的sql语句都没。。

[1211]

探索到一个地方，languageDriver.createSqlSource()方法，它是做什么的呢？

# 20190405

[1228]

taste了deleteBatch增强部分的测试，有一个要注意的地方。如果不了解的话，出错时候可能会找的一头雾水。



上图展示了解决问题的方法。这到底是一个怎样的问题？

像涉及到批量的增强操作：删除，插入。都是借助mybatis框架提供的<foreach>那些标签来实现动态数据需求。直接引用foreach标签的测试结果如下；



上面的错误中，仔细看多几次即可明白，它的报错原因是使用了参数#{item}，但是由于foreach标签并没有被解析，得不到参数item，可使用的参数只有collection或者list。在这里，只需要想办法让foreach标签得到解析就可以了。

在mp的做法中，我总结了该方法的步骤

1. 在有引用到mybatis特殊标签（暂时不知如何叫，明白是哪个就好）时候，需要做一下标记，用<script>标签包含起来。
2. 使用XMLLanguageDriver对象来构造SqlSource对象。它内部会判断sql字符串是否包含script标签，若有，会按照mybatis那一套标签解析去eval此字符串.



[2241]

版本号：6670ee72eb8d13baab6872578eae846d126f9bd4

此版本taste结束。

下一个版本号是：bd3d12237210640e2ca9acd5138e99788e84b4f7

这次是在说mp框架的版本成为了v1.1

[2332]

版本号：41eb310b89f0e8f16296453d1d3eeec5e9a1a178

这一次的版本，应该是发现原有insert部分的逻辑写法出现了问题，即部分对象的属性为空，但是数据库字段不允许为空的问题，为做更好的兼容，mp使用mybatis框架的trim标签解决这个问题。

trim标签属于mybatis框架的东西，自然要标记为需要mybatis框架的解析支持。

可以了解一下<trim>标签的作用:

trim标记是一个格式化的标记，可以完成set或者是where标记的功能，

prefix：前缀

prefixoverride：去掉第一个and或者是or

suffixoverride: 去掉最后一个指定的字符.

了解这两个属性之后，对trim标签的使用基本清晰。

这个版本插入数据，对空属性做了判断，若为空，不参与insert 操作.但是这前后都没有太大差别。

版本号：

这里版本希望从另外一个角度测试，比如主键名称就不是id，而是”test\_id”其它什么的，那这时候运行test会怎样，我尝试了一下，报错了：



为什么会出现这个错误呢？要怎么去避免？

一开始看到，那种莫名奇妙的感觉由开始上线。但什么复杂的问题，要去解决，就需要找到方法去捋顺它。

运行test之前，我按照此次版本中的sql文件修改了表结构，其中就把主键名称由id更改成test\_id.

所以，这里实体类中的属性名称id和表字段test\_id并没有对应上。

这里所注入sql中，insert into user (xx, xx, xx) ... 字段名称全部根据实体类属性转换过来，所以这里猜想属性名称和字段名称对应起来即可。

将实体类中的id属性修改成testId



看来并不是没有那么简单，上图中，貌似这里没有应用上驼峰转换呀，为什么了呢.驼峰转换没有应用上，我想直接在生成tableinfo对象时候，fieldList调用驼峰转换方式存放进去，结果是驼峰转换了，那#{aBcDe}这种有需要的却不支持了.我想不能每次使用到再进行一次转换，毕竟增删查改语句都会涉及一些，认为操作方式应该是新增一个成员属性column：List<String>，根据配置是否驼峰来确定这里是否调用驼峰转下划线的工具方法，然后所有sql字段的拼接使用转换后的，但即使这样操作也还是有问题的，怎么根据field找到column，再一次驼峰转换？如此，不就出来结论了吗？要么用到时候转换，要么使用具有映射的数据结构。

且看plus框架要怎么操作。

[2218]

版本号：ad62a641f01432c4bedd3244c12b5c101c3481be

上面的问题只需静观框架变化即可。



这一次中，它大致上是删除掉了 insertBatch 的实现，并且要在后面好多次commit之中才恢复回来。

新增了 IdKeyGenerator.java 文件，发现最近的commit都与自定义主键相关。前面的实现中，框架是支持自定义主键的好似，反而没有兼容下划线命名的字段是一个问题。

新增了 TableFieldInfo.java 文件。

接下来在代码中，会看到一些 keyProperty、keyColumn的词，他们代表主键的属性名、字段名，这两个属性被封装在TableInfo中。

在AutoSqlInjector类中，此次大部分改动主要围绕新文件TableFieldInfo.java展开。

没有主键，不给这个mapperClass增强通用sql语句.

# 20190418

[1247]

版本号同上，在同步TableInfoHelper类方法时候，更加明白了plus框架本次commit的意义。

框架要实现的是： 支持自定义字段名称。

在实体类中使用 @TableId @TableField时候可以自定义，

若没有使用value配置自定义的字段名称，框架最后按照列名和属性名一致进行处理。

若存在使用value配置自定义字段名称，框架使用 注解的value值作为该属性所对应的列名。

主键的属性/字段名称被封装在TableInfo类中，

其它属性/字段名称被封装在TableFieldInfo类中，并且以List<TableFieldInfo>形式存在于TableInfo类中。

# 20190501

[1234]

版本号同上,这次的taste中,明白plus框架自定义字段的真正作用.但是当前在Manjaro环境下,缺少数据库相关环境,放弃了这次的测试.

隔了十几天不看项目,确实又生疏了不少.

避免生疏的方式就是写类似todo的留言,这次有事离开,taste到哪里,就可以在这里简单描述一下,方便下回快速进入状态.

看项目,体验的是思想,有时候一些缺乏意义的代码就尽量copy,加快taste的速度.真正的踏实是有对应的输出.方便以后的记忆.

# 20190506

[1933]

版本号：5779bad4e65829bd283fd6c0c8d2deedadd07239

增加实体查询 hubin 2016/2/29 16:29

通过commit注释，也可以明白了，这次做的补充是实体对象中，按值不为空的属性进行单表查询。

明白了这点，自然了解又有新的sql语句又进行增强了。

**查询满足条件一条数据**

**SELECT %s FROM %s WHERE %s**

这一次，新登场的实体查询，作用于select one， select all之上。

select all这时候的命名就不是很合适了，他本就是用于查询全部数据的。

在下一次的版本中，他将被改为select list。

我个人感觉，hubin大佬在这次的修改中，已经开始展现后面 EntityWrapper的最初形态的最最最初形态。可参考

/AutoSqlInjector.java # sqlSelectColumns() 方法

/AutoSqlInjector.java # sqlWhere() 方法

# 20190507

[1933]

版本号：同上

大半个上午时间，我都在跑测试。一直失败，感觉很打击信心。

失败的原因主要有代码抄写疏漏，所以关键位置要多和plus官方的code仔细比较

数据包装类型可以进行空判断，mplus框架使用这点，在做实体查询时候，需要遵循一个约定，实体成员属性不包含基本数据类型。比如，不是 private int age; 而应该是 private Integer age;

其它基本没有问题，最后要勤劳调试，日志输出。比如inject了什么sql，可以确定是否语句本身问题

这回可以提交一次版本了

# 20190509

[0925]版本号：86824402afbc00d33c96050bfbf51ca9d9f999dc

commit message : 增加分页查询 hubin 2016/2/29 18:06

tasting...



左右两张图片，前后对比，除了方法名改的更加合适之外，还多了一个分页参数。

这里虽然添加了参数，但实际不影响 AutoInjector.java 中注入sql的逻辑。

因为分页的逻辑（判断分页参数部分），在分页插件拦截器中已经完成好了。所以这次的version让我感到这些东西真的巧妙。剩下的，或许就是跑动测试，验证结果就可以了。

--- % 分割线 % ---



同样，也发现了一个隐藏的小bug，上图黄色涂抹这个地方，在之前写错了，导致运行时候总是提示there is no getter for property ‘test\_type’ ... 这种错误。将其改正之后就什么都好了。

commit ✔

# 2019年5月20日星期一

[2344]

版本号：1445604cd0abbef57ddcf9226546d8c08e698902

Id 类型判断 青苗 2016/2/29 21:21

这次看到的内容是：移除TableId注解中auto属性，替换为枚举类型IdType，代表[主键自增, 用户自己传入，工具类生成]

新增的IdType看起来，在codeStyle上虽然我感觉不是很舒服，但是他的作用还是较为明确的，让人易懂。这也让我再次感受到枚举类的好处.

/\*\*

\* 主键ID，默认 ID 自增

\* {**@link im.lincq.mybatisplus.taste.annotations.IdType**}

\*/

**IdType** type**() default IdType.*AUTO\_INCREMENT*;**

上面就是注解TableId中的type属性，结合注释来看，是不是清晰易懂了呢..

[2356]

重命名文件

IdKeyGenerator → IdWorkerKeyGenerator

[2357]

toolkit包下的两个文件 【TableInfo.java，TableInfoHelper.java】

改动如下：

TableInfo类增加成员 IdType，移除autoIncrement:boolean属性;

TableInfoHelper负责将获取，填入具体的IdType值。

# 2019年5月21日星期二

[0002]

AutoSqlInjector.java

按照新的属性，对AutoSqlInjector类中就变量改为新变量引用。

[0008]

版本：6807afb1d56ff71ef8aa5262953944a96a2f49c9

Id动态生成 青苗 2016/2/29 22:11

这次commit是针对代码生成部分的.我跳过了代码生成这部分。

再往下看，两次commit，先是增加了回收map的操作，后是删除了gc,也可以跳过。



再往下是分页辅助类部分，新增了一个文件，内容不是很多。

[0749]

上午的问题出现分页插件count方法上面.对象不为空时候，条件查询没有注入值。导致报错了。

以后有时间再来仔细研究，这个问题现在没有去处理，以后肯定也是绕不开的。

# 2019年5月26日星期日

版本号：62039cc3504531b9a04aa50197c48a9b81a53fb7

增强条件删除记录 hubin 2016/3/9 17:44

这次的内容如commit所示; 他在AutoMapper.java中的体现为 int deleteSelective(T record);

根据条件来删除数据。

[1808]

版本号：029caff2d9cdaf4c1dd660b7ca6492ab154b6511

（去掉无效数据）

版本号：00fc4db60243e011608e505ed61b22dd517242fe

（添加selectCount查询）



上两个分支，一个是增加了 select count sql 语句注入，而后新的分支却又删除掉了这部分的东西。感觉增加这部分的count查询没什么毛病，为什么还要去掉呢.

[1820]

版本号：2adc631c1519a71469ad5fe44aa95d1d3a291bdd

实现自定义 id hubin 2016/3/11 18:13

这次的commit中，有将会有更大部分内容的修改和新增。仍然从测试用例方式开始入手较好。

发现，之前去除的insertBatch，这里算是给恢复回来了。从注释之中，看出了之前去除的原因。这种方式并不适用于Oracle。



发现，在AutoInjector.java中，应用了自定义的XmlLanguageDriver



测试insertBatch方法

# 2019年6月16日星期日

版本号：7fb67e83f4f1755420f478d8bc4be52ad6667498

升级 1.2.3 支持实体封装排序 hubin 2016/3/15 12:38

这次版本中，EntityWrapper的类开始出现了。这个类，在我们使用Mybatis-plus的过程中是最为常见的。它的使用也是比较多。对于它是如何实现，以及它在之后如何演化，我都有较多的兴趣。

EntityWrapper类的出现，提供了更加丰富，灵活的数据查询方式。比如，在当前的版本中，就能够通过它来指定我们的查询中，对指定字段进行排序的方式，在以后也一定会有对分组group的支持。

在AutoMapper.java中，添加关于EntityWrapper对象参数注入。

[1558]踩坑了,mybatis xml 中sql 参数，参数如果是string 类型，转化为可执行sql语句时候就会进行单引号的拼接，坑不在这里，而是plus框架为什么没有注意到这一点呢？

[1721]

版本：4dbc2f90123c595d89e4a10c8cff548a2072e8cd

去掉分页 total 为 0 判断 hubin 2016/3/23 10:38

为什么会去掉呢？

可能没有太多作用吧.

[2251]

版本：6d28064005ea7182badd8dc17d8eed1932c0fd38

添加 service 封装 hubin 2016/3/23 11:13

看到SuperServiceImpl没有直接实现ISuperService接口，感觉有点奇怪，往后多看来几个commit，才明白，后面的动作其实还大着呢（有关泛型），这种事情或许不宜过早考虑。

居然可以直接使用AutoWired注解去引入AutoMapper接口了，这是个出乎意料的点.

在文件中，可以注意到deleteById 和deleteBatchIds 方法，id的类型都是Object，这一点中可以看出框架作者已经考虑到了id可以有不同的类型（Int类型，Long类型，String类型），所以当前使用Object类型声明ID是合适的。

**public** *T* selectById**(Object** *id***)** {

**return** autoMapper**.**selectById**(***id***);**

}

[2347]

版本：ebc3ee1d9092907ce4ce32dcc0eba1693a628162

service 多重泛型 hubin 2016/3/23 17:09

这次的主要改变如下：

**public class SuperServiceImpl**<*T***,** *M* **extends AutoMapper**<*T*>> {

@Autowired

**protected** *M* autoMapper**;**

}

autoMapper属性的类型声明变为了泛型M,直观上就是变得简洁了，但是我有一个疑问，假如AMapper和BMapper都实现了AutoMapper接口，那么，spring注入类型对象时候，会选择谁呢?

我想，可以辅助决定的应该就是，在决定M具体类型的地方。我这么说好像是一句废话。但这是一个引导自己的过程，这么写是指向了未来继承该SuperServiceImpl类的子类，子类自然会给出泛型M的具体类型。

# 2019年6月19日星期三

版本：c9b5e53490bb7852ad7edbfe9ee543e6bb426f1a

优化分页 hubin 2016/3/25 18:11

PaginationInterceptor.java

如果countSql中包含order by 语句，则去掉order by 部分.再进行count查询。

前面有几个地方是其它的部分，感觉并没有和框架想接近，我就没有一起照搬了。如果以后需要，再一次性搬完吧。

# 2019年6月20日星期四

版本号：bbc74091a0fdb52d6319048a2bd4833a9f9a2973 && bf243123e0241065cf7251f78692be2e73e77dc5

升级 1.2.6 service 层封装 hubin 2016/3/28 14:59

更人性的封装 service 层 hubin 2016/3/28 14:50

这两个版本中，通过重载，提供了多种查询的方式。可供用户自由选择。

版本越趋于稳定，将会更多的放在bug,细节的维护上，或等待下一个灵感的到来。

[0017]

版本号：62c565b2f186cc04cc83eb048e0bff1c92aba4ea

修改 bug 青苗 2016/3/28 22:18

这个bug，我看的并不是特别明白，因为这里只是将驼峰单词转下划线，经过这次修改后，单词中每个字母都成为了大写形式。印象中小写的字段是没有区分的，mysql是这样，其它数据库就没有去了解了。

# 2019年6月22日星期六

版本号：b92567751a238ffc420bce5f26e0b54bc81a1b9e

新增3个选择更新方法 && 优化分页 hubin 2016/4/1 12:55

1. 新增了3个更新的方法，加上原有的算为4个.
2. 重新布局了insertBatch方法.（因为insertBatch本身不需要判断实体非空属性来拼接sql，那种是不符合实际的做法）

在这次taste中，了解到一个新的名词 selective ，条件.

这个名词对应是否选择实体类的非空属性，来进行sql拼接。

关于第1点，四个update分别是：



其实是两个方法，一个是根据id修改，另一个根据实体条件修改，我感觉第2个方法并没有什么具体作用，根据条件找出的实体，再修改，能改到什么呢..哈哈..

# 2019年6月23日星期日

版本号：b92567751a238ffc420bce5f26e0b54bc81a1b9e

新增3个选择更新方法 && 优化分页 hubin 2016/4/1 12:55

梳理一下：带selective的方法都是选择了非空字段来进行拼接sql，反之则使用了全部字段来拼接sql。

这两种不同的使用方式，对程序员，方法的使用者来说，带来了更多的选择，更加灵活。

update selective中，我昨天认为其无用的想法也得到了推翻。



看到有两个参数的方法，我也就明白了，一个参数用来做实际修改的，另外一个是作为数据匹配条件的。如此就打通了。

但要使用这样的方法也挺麻烦喔，要分别创建两个对象.可他的好处也是更加灵活，麻烦一些也是能够接受的。

[2250]

版本号：55acd76e6ec2e87f26b4106f4ee13c7c93c0bea7

分离 list page 方法 青苗 2016/4/7 22:15

这次的commit中，也确实看到了，plus框架开发者对原先可以兼容分页查询的selectList做了一些处理，让selectList专心做列表查询，将有关分页的职责分离出来，交给新增的selectPage接口。虽然底层调用都是一致的，但是对于框架的使用者而言，这样的区分使方法表示含义更加明了。

***UPDATE\_BATCH\_BY\_ID\_MYSQL*("updateBatchById", "mysql 根据ID 批量修改数据", "<script ...**

***UPDATE\_BATCH\_BY\_ID\_ORACLE*("updateBatchById", "oracle 根据ID 批量修改数据", "<script ---**

**这两句的sql id都是 updateBatchById，或许，将会在AutoInjector 中判断DBType来决定应用具体的SQL.**

# 2019年6月28日星期五

**injectUpdateBatchById这个方法的实现，并看不出来具体的批量修改在哪里体现出来.**

**但是看SqlMethod类中对应的sql.，就能够看出来了 <foreach>写在了里面.让我运行一下，看看这个inject-sql的输出将会是怎样的。**

**inject sql (updateBatchById) - MYSQL -**

**<script>**

**<foreach collection="list" item="item" index="index" separator=";">**

**UPDATE user**

**<trim prefix="SET" suffixOverrides=",">**

**name=#{name},age=#{age},test\_type=#{testType},**

**</trim> WHERE test\_id=#{id}**

**</foreach>**

**</script>**

**看出来了，使用最外层循环来生成多条update by id的sql语句。**

**上面的sql语句还不是最终的，比如 name = #{name} 是不能直接运行的，而应该是 name = #{item.name} ，其它使用的参数亦是如此。**

**再来看一个新版本的：**

**<script>**

**<foreach collection="list" item="item" index="index" separator=";">**

**UPDATE user**

**<trim prefix="SET" suffixOverrides=",">**

**<trim prefix="name=CASE test\_id" suffix="END,">**

**<foreach collection="list" item="i" index="index">**

**<if test="i.name!=null"> WHEN #{i.id} THEN #{i.name}**

**</if>**

**</foreach>**

**</trim>**

**<trim prefix="age=CASE test\_id" suffix="END,">**

**<foreach collection="list" item="i" index="index">**

**<if test="i.age!=null"> WHEN #{i.id} THEN #{i.age}**

**</if>**

**</foreach>**

**</trim>**

**<trim prefix="test\_type=CASE test\_id" suffix="END,">**

**<foreach collection="list" item="i" index="index">**

**<if test="i.testType!=null"> WHEN #{i.id} THEN #{i.testType}**

**</if>**

**</foreach>**

**</trim>**

**</trim> WHERE test\_id=#{id}**

**</foreach>**

**</script>**

**用测试例子，跑出的是下面多条由循环而出的这种sql语句。**

**它的工作机制是：当这条记录的主键值为指定参数值时候，则可以修改该字段。**

**UPDATE user**

**SET**

**name** = **CASE** test\_id **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **END**,

age = **CASE** test\_id **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **END**,

test\_type = **CASE** test\_id **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **WHEN** *?* **THEN** *?* **END**

**WHERE** test\_id = *?*;

# 2019年6月30日星期日

**版本号：b451215cb1c6e7cb17038b66d4085a05f783aed9**

**深入支持 oracle hubin 2016/4/26 14:44**

**这个commit中，修改了 oracle 分页方言的实现方式，对比起以前的实现，如今的版本看起来特别直观，早就忘记了oracle分页的自己，从这次的修改中找回了记忆。**

**SELECT \* FROM**

**( SELECT TMP.\*, ROWNUM ROW\_ID FROM ( originalSql ) TMP WHERE ROWNUM <= (offset>=1 ? offset \* limit : limit) )**

**WHERE ROW\_ID > offset**

**在originalSql中，自身就是一次select查询，oracle实现分页查询需要外面的两层select加持，最外层select和第二层一起把控数据行号范围。**

**[1403]**

**版本号：a87c0d56bc0b5dacc4c284a69c25973c8c308c12**

**支持 curd 二级缓存处理 青苗 2016/6/14 11:10**

**这次commit是青苗大神提交的，从中可以学习了解配置mybatis二级缓存的知识点。**

**这次commit中，改变了sql注入的时机。从MybatisConfiguration转移到了MyXMLMapperConfigBuilder中去。但是它们的作用其实都是一致的。**





**经过缓存作用之后，再次查询相同的对象数据，将会直接从内存中换回缓存对象，而不是再次执行select sql来查询用户。**

**思考：这种缓存配置是怎么生效的呢？是plus框架的功劳，亦或是mybatis所提供的功能呢？**

**在pom文件中引入mybatis-cache这个jar包，引入jar包后，配置一个ehcache.xml文件 （它并不是决定是否使用缓存的关键配置）**

**在UserMapper.xml文件中配置了：<cache type=”org.mybatis.caches.ehcache.LoggingEhcahe”>（它并不是决定是否使用缓存的关键配置）**

**经过了查找定位，是否决定使用缓存的关键在于 MapperBuilderAssistant 对象的addMappedStatement方法。这个方法的参数中就有是否使用缓存，清除缓存的参数开关。plus框架AutoSqlInjector类通过 此对象的 addMappedStatement 方法来注入mapperClass的CRUD-sql语句，它在这个地方做了数据缓存的逻辑处理：**

**若当前注入的mapperStatement 为SELECT类型，将不清除缓存，优先从缓存中查询数据，否则将清除缓存，从DB中查询数据。**

**那些引入的jar包，以及ehcache配置文件，我并不知道其具体作用在了哪里。但是后面plus框架中还有许多关于缓存的处理，敬请期待。**

**引用mybatis二级缓存的好处是减少了许多重复的查询，提高服务方法的响应速度，但是我看到后面有关缓存的commit中，有这么一条写着：修复了使用缓存导致无法计算count的错误，缓存这种东西需要慎用吧。**

**2019年7月15日星期一**

**[16:43]**

**代理模式：**

**动态代理，可以通过JDK中Proxy类的静态方法 newProxyInstance 来产生代理实例。该方法共需要3个参数：**

**1.类加载器**

**2.代理实例需要实现的接口数组**

**3.InvocationHandler接口实例（通过override invoke方法，实现具体的代理逻辑）**

**该方法返回的代理实例实现了我们参数中指定的所有接口。该代理实例身上的所有方法，被调用触发时候，都将经过代理逻辑（invoke）方法中的内容。**

**[1916]**

**版本：422682be3816ef6c7bdd0fb1709e7df0206d55d9**

**Commit：升级 1.3.0 支持 like 比较等查询 sqlSegment 实现 hubin 2016/7/6 17:02**

**这次的commit内容，主要内容是通过EntityWrapper 类的新成员属性 sqlSegment来完成sql语句中条件部分的自定义，并在其设置过程中，使用正则表达式匹配sql语句，防止出现sql语句注入的情况。**

**疑问：暂无疑问**

**2019年7月16日星期二**

**[1419]**

**版本：1528269463db14084e4194c1f19095235da5dace**

**Commit: 升级 1.3.1 新增 sql 执行性能分析 plugins 青苗 2016/7/7 21:37**

**这次的commit中，新增了一个插件(PerformanceInterceptor)，作用是sql性能分析相关的，说分析二字，让人觉得高大上，其实仅是输出sql的执行耗时，如果耗时大于指定的时间，将抛出异常标识预警。**

**[1510]**

**版本：5e130429a8721315b8f85e78f88ad76d89c3e5b8**

**Commit: 升级 1.3.2 新增 deleteByMap , selectByMap hubin 2016/7/8 17:23**

**这两个方法的支持，增加了sql的选择条目。使用Map对象来代替Entity实例，在某些特殊情况下.或许会特别有用。**

**Map对象中， key 代表 column 字段， value 代表实际数据值。**

|  |
| --- |
| **private** String sqlWhereByMap () {  *// # cm 是Map对象，keys代表 Map对象的键集合，foreach标签遍历到每个键，键名称代表 表字段名称*  StringBuilder where = **new** StringBuilder();  where.append(**"\n<foreach collection=\"cm.keys\" item=\"k\" separator=\"AND\"> "**);  where.append(**"\n<if test=\"cm[k]!=null\">"**).append(**"${k}=#{cm[${k}]}"**).append(**"</if>"**);  where.append(**"\n</foreach>"**);  **return** where.toString();  } |

**2019年7月18日星期四**

**说一说DefaultParameterHandler 和MybatisDefaultParameterHandler类**

**DefaultParameterHandler类由Mybatis框架提供，它实现了ParameterHandler接口。其作用同接口的注释一样:**

**A parameter handler sets the parameters of the PreparedStatement.**

**译：设置PreparedStatement对象sql参数的handler。**

**关键方法：setParameters(PreparedStatement ps);**

**这个方法从boundSql中获取所有ParamterMapping对象，并且循环遍历这部分数据，**

**循环内部是调用各自typeHandler对象的setParameter方法，完成对ps实例的参数设置；**

**关于 typeHandler.**

**Mybati基于jdbc执行 sql，使用PreparedSatement设置sql语句的参数的方式为：**

**ps.setInt (1, value: Integer);**

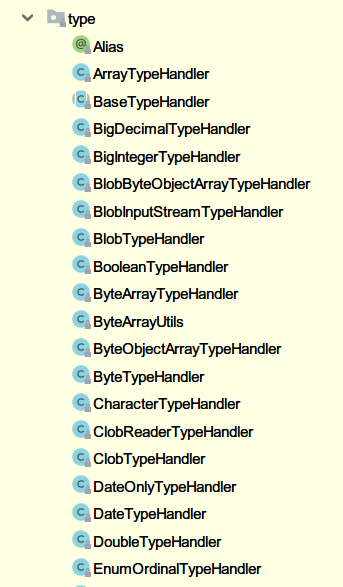
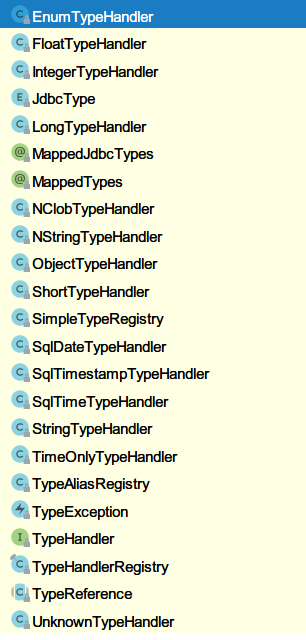
**ps.setLong(1, value: Long);**

**ps.setShort(1, value: Short);**

**…**

**不同的数据类型在mybatis框架中都有对应的TypeHandler，他们都实现了TypeHandler接口，实现运行时多态，且有效避免了多种不同关于类型判断的if语句。**

**有关typeHandler的代码，在mybatis的type包下。下面两张图中，可以看到有许多TypeHandler后缀的类**

**MybatisDefaultParameterHandler**

**构造方法： 直接调用父类DefaultParameterHandler的构造方法，但在第二个参数（parameterObject）中，做了一些小小的改变**

**processBatch 方法，参数如果是 实体对象，直接设置该对象的主键，如果是集合，遍历集合，设置集合中对象的主键。**

**说一说ParameterMode枚举类（IN、OUT、INOUT）**

**该类中有枚举成员 IN，OUT，INOUT，字面看起来，我觉得有一些熟悉，好似当时学习Oracle时候留下的一点印象。**

**上网了解之后，这三种在MySQL中，对应存储过程IN、OUT、INOUT的三种参数模式。**

**以下内容参考：**[**https://www.cnblogs.com/super-yu/p/9018512.html**](https://www.cnblogs.com/super-yu/p/9018512.html)**，**

**IN类型: 使用IN类型来传递信息，存储过程内部可以对参数的值进行修改，但是修改后的值调用者不可见。**

**OUT类型: 使用OUT类型来传递信息，在存储过程内部，该值的默认值为NULL，无论调用者是否传值给存储过程。**

**INOUT类型: 使用IN类型来传递信息，存储过程内部可以对参数的值进行修改，并将最终值返回给调用者。**

**通过这篇文章，可以明白，这三种参数模式的作用是：参数在存储过程内部修改后，决定调用者是否可见的配置。**

**[1701]**

**说一说SystemClock类和Sequence类**

**版本：0b78dffa76e12f9b70ea352bc2922607f4714252**

**Commit:** **优化 IdWorker 生成器 hubin 2016/8/1 14:50**

**使用了一个Scheduler线程池，重写线程工厂，设置产出的线程为守护线程。**

**制定了周期为1毫秒的任务，将System.currentMillis()的返回值（当前毫秒值）设置到该类的对象成员now上。**

**为什么是这两个类，这两个类是从 IdWorker 类中分离出来的. 并且按照注释上意思来说，还对获取当前毫秒值的操作上做了优化。这点可以查看SystemClock类。现在IdWorker类的前后版本对比中，可以明显看出当前版本的代码简洁了很多。**

**2019年7月19日星期五**

**版本：1956b585bf1c3f3329cd0d79e32b13a732a29d6e**

**commit:** **升级 1.3.10 增加自定义 select 结果集，优化 page 分页 青苗 2016/8/13 0:41**

**自定义select结果集**

**这个小节. 属于一个新的想法，目的是让plus框架辅助支持注入用户自定义查询字段的sql.**

**比如. 原本一直是select 全表字段 from 表名称 where xxx order by xx desc;**

**现在想要的效果就是，如果我只想查询name，和age两个字段的数据出来，其他的不想查，那么在这个版本之后，我就有这个方式的操作入口啦。**

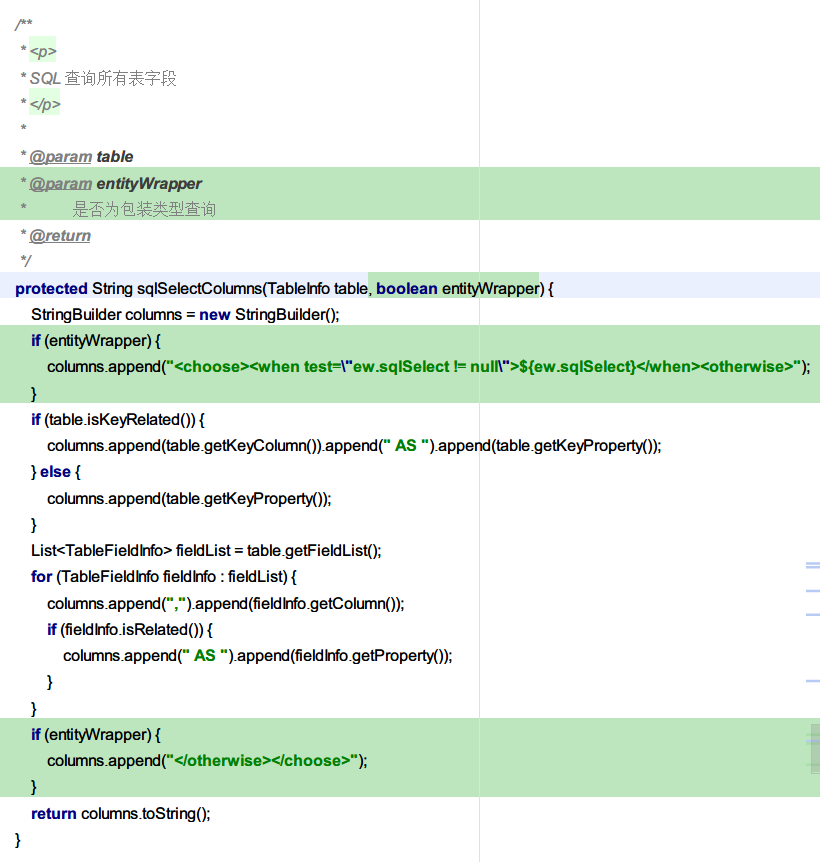
**IService接口，只保留一个selectPage方法，一个selectList方法。方法签名如下：**

|  |
| --- |
| List<T> selectList(T entity, String sqlSelect, String sqlSegment, String orderByField);  Page<T> selectPage(Page<T> page, String sqlSelect, T entity, String sqlSegment); |

**上面的selectPage方法中，参数看不到ordreByField，框架并不是移除了这个参数，只是将他移动到了Page类中，这样操作其实看起来没有实际的必要性，静观其变。**

**AutoInject.java 关键部分代码.**

**<choose><when test></when><otherwise>< /otherwise></choose>**



**这个方法是通过判断entityWrapper的来决定是否启用自定义查询字段。**

**所有调用这个方法的地方，除了 injectSelectList传true值之外，其他均传了false值。**

**也就是说，自定义查询字段的功能用在了selectList方法上。映射到service中，则是selectList和selectPage方法都能使用这样的功能。**

**这个功能确实挺好的，但也有其不足之处，是字段硬编码于字符串中，哪天字段发生了修改，忽略了这边字符串内字段信息的同步，项目便产生了潜在的错误，然后不知某时，程序运行在经过某串代码的时候被触发出来。**

**下面是关于这个功能的测试代码：**

|  |
| --- |
| System.***err***.println(**"\n------------------分页pagination查询 --- 查询页中 testType = 1 的所有数据----------------------"**);  Page<User> page = **new** Page<>(1, 2);  */\*\**  *\* 排序 test\_id desc*  *\*/*  page.setOrderByField(**"test\_id"**);  page.setAsc(**false**);  ew = **new** EntityWrapper<User>(**new** User(1));  ew.setSqlSegment(**"age,name"**);  */\*\**  *\* 查询条件，SQL片段（注意！程序会自动在 sqlSegment内容前面添加where或者and）*  *\*/*  ew.setSqlSegment(**"name like '%dateBatch%'"**);  List<User> paginList = userMapper.selectPage(page, ew);  paginList.forEach(UserMapperTest::*print*); |

**说说ContiPerf工具**

**参考文章：**[**https://blog.csdn.net/tomato\_\_/article/details/22060449**](https://blog.csdn.net/tomato__/article/details/22060449)

**ContiPerf是一个轻量级的测试工具，基于JUnit 4 开发，可用于效率测试等。**

**2019年8月13日星期二**

**版本：0fd38ce118c014a1a3b8dd509f79e1258555a5cf**

**升级 1.4.1 优化查询 青苗 2016/8/20 0:33**

**虽然以前自己写过单元测试，而且还写过不少，自己也觉得满意，但是我做单元测试的目的更多的是测试已经存在的一段代码。**

**在这次的版本中，第一次感觉我是在接触TDD测试驱动开发思想。**

**按照本次版本的 EntityWrapperTest.java ，我明白了有一种方式，可以先编写自己期望的功能使用代码，并编写自己预期该功能的正确运行结果，最终按照测试代码来驱动自己核心功能代码的开发。这种方式最终能够决定到的细节程度十分精确，举个例子就明白了，可以做到一个必要trim()实际未使用trim()的检测。**

**跟着框架开发路线走，开开脑洞，这就是我目前想要的。**

**我的commit：升级1.4.1 | EntityWrapper类增加 addFilter 方法. 方法内部使用MessageFormat类替换指定变量，形成sql语句中where条件部分。**

**2019年8月15日星期四**

**版本：6629afb7868b65912e0f3ef08f41f05904fd1f8c**

**Commit: 优化 yanghu pull reuqest T-SQL 青苗 2016/8/21 16:25**

**再次从EntityWrapperTest类测试方法中，发现前面taste的代码错误，纠正了了方向。**

**这个版本中存在一个问题，EntityWrapper类的方法存在交叉的调用歧义，将在下一次commit中得到解决。**

**歧义点是：**

**public EntityWrapper<T> addFilter (String keyWord, String filter, Object … params) {}**

**public EntityWrapper<T> addFilter (String filter, Object … params) {}**

**上面两个方法声明中，方法2的使用会产生调用歧义，在运行时违背了我们的期望，实际是方法1被调用了。**

**[0108]**

**版本：e447b0e440e978a4eb94aab182ee1fca2d555e0c**

**Commit: 调整addFilter 不能交叉使用 tsql 青苗 2016/8/21 21:25**

**这里解决了上面 addFilter 交叉使用的问题，它是通过将方法提取到父类中完成的，怪神奇的，没想到什么可以解释这个歪理。**

**[1812]**

**版本：a5003068ce011b887565051964c44a47b3498aa0**

**Commit:**

**1、修改TSQL语句构建逻辑。**

**2、添加LIKE、NOT\_LIKE、IS\_NULL、IS\_NOT\_NULL常用方法。 3、addFilter及addFilterIfNeed 可以使用 and() andIf() 方法直接替代，语义看起来更明确。**

**yanghu 2016/8/23 19:41**

**非常有难度，心不静，看代码则不行。看着新文件MybatisTSQLPlus类的代码，恍若天书。一开始可以理解，这次的变动基本将所有 sql conditon 都先放入集合中，然后在需要获取的地方进行统一拼接，返回最后符合用户想要的t-sql查询部分。后来看着作者组装sql代码的方法时候，看的不是很明白，有些受打击。后面，通过EntityWrapperTest类的测试方法，对这个过程进行了调试，最终基本搞明白了这个拼接sql过程的实现。**

**一系列TSQL语义的方法。先看一张截图，从截图中可以看到该类中一系列的方法，从框架上注释来看这是与t-sql 语义相关。如调用or(“status={0}”, 1)将对应”OR status=1”字符串。方法中参数部分是本次查询sql语句中的组成部分。如 conditions : status=1，columns：id, is\_deleted,gmt\_created 等。这些参数在方法中不会被直接拼接成结果字符串，而是先存放到与语义对应的集合中，之后可以通过build方法完成这些sq差选条件部分的组装。这些集合对象是内部类的成员。**



**SQLCondition类 是MybatisTSQLPlus的私有静态内部类。这个类定义了多个对应t-sql语义的集合类型成员变量，用于存放对应t-sql语义的sql关键字和原生sql字符串。这些集合在后面组装sql (where之后的部分) 时候用到。组装sql的方法是buildSQL()，它负责组织多个sqlCause方法，完成了 where + groupby + having + orderby + asc/desc等语句的拼接（空的集合将会略过）。**

**下面来看看这个类：类定义：**

**private static class SQLCondition**

**成员变量：**

**List**<**String**> where = **new** ArrayList<**String**>**();**

**List**<**String**> having = **new** ArrayList<**String**>**();**

**List**<**String**> groupBy = **new** ArrayList<**String**>**();**

**List**<**String**> orderBy = **new** ArrayList<**String**>**();**

**List**<**String**> lastList = **new** ArrayList<**String**>**();**

**List**<**String**> andOr = **new** ArrayList<**String**>**();**

**这个类的成员变量中，全部都是List<String>类型，用来存放前面t-sql方法中传递的参数，以及sql关键字，列表数据结构的有序性在这里起到了帮助作用。**

**构造函数：在这里对成员 andOr 放入了四个元素，它们都是Sql的关键字。这个数组在sqlCause方法中用到，可以看后面代码注释的地方。**

**private static class SQLCondition** {

**public SQLCondition()** {

andOr**.**add**(*AND*);**

andOr**.**add**(*OR*);**

andOr**.**add**(*AND\_NEW*);**

andOr**.**add**(*OR\_NEW*);**

}

**private void** sqlCause **(SafeAppendable** *builder***, String** *keyWord***, List**<**String**> *parts***, String** *open***, String** *close***, String** *conjunction***)** { …后面提到…. }

**private String** buildSQL **(SafeAppendable** *builder***)** {}

**public String** sql **(Appendable** *a***)** {}

}

**buildSQL方法：**

**//**按标准顺序链接并构建SQL

**private String** buildSQL **(SafeAppendable** *builder***)** {

sqlCause**(***builder***, "WHERE",** where**, "(", ")", *AND*);**

sqlCause**(***builder***, "GROUP BY",** groupBy**, "(", ")", ",");**

sqlCause**(***builder***, "HAVING",** having**, "(", ")", *OR*);**

sqlCause**(***builder***, "ORDER BY",** orderBy**, "(", ")", ",");**

**return** *builder***.**toString**();**

}

**最关键的sqlCause方法， sqlCause方法时执行具体sql拼接的核心逻辑，该方法的实现难点，我认为是拼接过程的顺序性。它巧妙利用了list结构的有序性，最终输出了正确的sql条件部分。**

**private void** sqlCause **(SafeAppendable** *builder***, String** *keyWord***, List**<**String**> *parts***, String** *open***, String** *close***, String** *conjunction***)** {

// parts集合不为空的情况下

**if (**!*parts***.**isEmpty**())** {

// 如果builder 不为空，则准备换行

**if (**!*builder***.**isEmpty**())** {

*builder***.**append**("\n");**

}

*builder***.**append**(***keyWord***);**

*builder***.**append**(" ");**

*builder***.**append**(***open***);**

**String** last = **"\_\_";**

// 遍历集合

**for (int** i = **0,** n = *parts***.**size**();** i < n**;** i++**)** {

**String** part = *parts***.**get**(**i**);**

**if (**i > **0)** {

// 若当前 part 是 sql关键字，则拼接关键字. 继续下一层循环

// 若不是，如 part = "stauts=1"，同样要拼接关键字(调用sqlCause方法时指定的关键字conjunction)，再拼接part. 然后才开始下一次循环。

**if (**andOr**.**contains**(**part**)** || andOr**.**contains**(**last**))** {

*builder***.**append**(**part**);**

last = part**;**

**continue;**

} **else** {

*builder***.**append**(***conjunction***);**

}

}

*builder***.**append**(**part**);**

}

*builder***.**append**(***close***);**

}

}

**对了，sql字符串存储在 SafeAppendable 类的成员属性a中。**

**private final Appendable** a**;**

**a是 Appendable接口类型，StringBuffer,StringBuilder的父类实现了Appendable接口，所以可以将appendable实例当作 StringBuilder/Buffer 同样对待使用，比如 Appendable a = new StringBuffer(); 这样的设计得以支持运行时多态。**

**测试代码，以及测试结果：**

**@Test**

**public void** testSQL11 **()** {

*/\**

*\* 实体待查询使用方法 输出看结果*

*\* \*/*

ew**.**setEntity**(new** User**(1));**

ew**.**where**("name={0}", "zhangsan").**and**("id=1")**

**.**orNew**("status={0}", "0").**or**("status=1")**

**.**notLike**("nlike", "notvalue")**

**.**andNew**("new=xx").**like**("hhh", "ddd")**

**.**andNew**("pwd=11").**isNotNull**("n1,n2").**isNull**("n3")**

**.**groupBy**("x1").**groupBy**("x2,x3")**

**.**having**("x1=11").**having**("x3=433")**

**.**orderBy**("dd").**orderBy**("d1,w2")**

**;**

**System.*out*.**println**(**ew**.**getSqlSegment**());**

}

**// 输出：**

**AND (name=zhangsan AND id=1)**

**OR (status=0 OR status=1 AND nlike NOT LIKE CONCAT(CONCAT('%', 'notvalue'), '%'))**

**AND (new=xx AND hhh LIKE CONCAT(CONCAT('%', 'ddd'), '%'))**

**AND (pwd=11 AND n1 IS NOT NULL AND n2 IS NOT NULL AND n3 IS NULL)**

**GROUP BY (x1,x2,x3)**

**HAVING (x1=11 OR x3=433)**

**ORDER BY (dd,d1,w2)**

**我去年开始接触mybatis-plus框架，每回写着 entityWrapper.where(“sex”, “’m’”).like(“name”, “lin”)这样的代码时，总会想起这么优雅代码背后的实现方式，当时对框架原理了解不多，总没有想通这个实现过程。今天理解了这个过程之后，特别兴奋，长时间搁置心头的一个问题算是被解开，再次说明学习这个框架的源码，能够开发脑洞思考，比学习数据结构，算法有趣。**

**对写出这个代码的Yanghu先生表示敬佩之意。我以前也写过一些巧妙的代码，解决了本会出现的代码冗余问题，我相信自己以后还能写出更有灵性的代码。每当遇见有设计性的代码，自己总会持欣赏态度，因为它们很多时候都能够刷新我对这块的认知。**

**2019年8月27日星期二**

**版本：8293b8675a55d5f5b6ce9d19576616ecd3c3fdde**

**commit：增加 mybatis 自动热加载插件 青苗 2016/8/25 21:39**

**Plus框架的插件：动态更新XML内容变化插件。**

**在典型SSM项目开发中，每当XML文件中有加入了新的sql，或者原有的sql语句发生变化，总要因为这样的变更而重新运行项目工程，以使更改生效。这样的情况偶尔来上一回两回，还是能够接受的，但是有一些问题不得不让我们花上很大功夫去调试，进而需要多次地重启工程，然后开发人员的大部分时间，就花费在重启项目中的等待了。虽然等待过程可以去忙活其它事情，但是这样处理事情总会被分心。我不知道别人的情况，反正我自己就容易被分心，忙着那边的事情而忘记回来处理这边的项目了。回过头来再处理，有可能还要找回切入点，找回这边的工作状态，也许我这上下文切换的成本有点高。**

**这个插件对此提供了热更新的功能，也就是说，当上面的情况再次发生了之后，开发者无需重启项目，插件本身可以检测到Mapper文件中的变更，然后将变更同步到项目系统中去。不得不佩服hubin大佬，想法真的很不错。Plus是SSM开发者的福音**

**在这里，它并非实现Mybatis框架的插件机制，而是和Spring框架结合，实现了一个可配置的插件bean，是当前版本中的MybatisXMLMapperLoader类，我了解到在后面的version中会更换为其它名称。**

**public class MybatisXMLMapperLoader implements DisposableBean, InitializingBean, ApplicationContextAware** {}

**热更新插件类，实现了Spring提供的三个接口，分别是DisposableBean,InitializingBean和ApplicationContextAware接口。**

**前面两个接口，提供了bean生命周期的中的钩子，分别是bean初始化后(afterPropertiesSet接口方法)， bean销毁之前(destroy接口方法)。我们就可以在这些钩子中做一些增强处理。**

**第三个接口，ApplicationContextAware接口，提供了将运行中的ApplicationContext实例注入到我们bean类属性的方式。**

**插件大概完成哪些工作？**

**插件实现了 bean初始化后 的生命钩子，在bean属性配置完成后，开始了插件的主要工作**

1. **检查bean属性值，是否开启插件热更新功能。若不开启，则工作停止，若开启，则看第2点；**
2. **创建一个ScheduledThreadPool线程池，核心线程数2，即将有两个工作线程。**
3. **线程池加入任务，每隔2秒，调用scanner.scanAllFileChange();方法，即扫描所有文件，处理可能需要热更新的情况。**
4. **再次加入任务，每隔100ms，调用scanner.scanHotspotFileChange();方法，即用更频繁的调度来扫描热点文件，处理可能需要热更新的情况。**

**热更新过程是如何完成的？**

**全部扫描：通过扫描找出有变更的Mapper.xml文件，将其添加至热点文件数组中。**

**热点文件扫描：同上，只不过是将范围缩小到记录中的热点文件。当发现热点文件有变更，最终会调用到reloadAll方法。通过对Configuration对象的反射，清除掉该对象中已加载过的Mapper.xml相关信息，然后将新parse的全部Mapper.xml信息重新写回configuration中。**

**什么条件下MapperXml文件将变成热点文件？**

**在全部扫描时，通过扫描找出有变更的Mapper.xml文件，将其添加至热点文件数组中。**

**Plus框架没有提供这个插件的测试代码，我也就不跟着测试了。这个类我了解过，在后期将会被删除，由nieqiurong同学贡献的MybatisMapperRefresh类代替。他的实现方式可能又有其它不同，到时继续了解。**

**了解到Spring框架中StopWatch类的用处，内部对System.currentTimilise() 进行封装，提供了更简洁的代码块耗时记录方法。**

**StopWatch sw = new StopWatch(); // 在创建实例时，会记录起始时间**

**sw.stop(); // 结束记录**

**System.out.println(sw.shortSummary());**

**2019年8月27日星期二**

**版本：8ae14b16b5b26185ff30a658e6a56ddb171559ea**

**commit：mapper动态加载 nieqiurong 2016/8/26 15:43**

**mapper动态加载，这是由nieqiurong同学贡献的另一个动态加载mapper的插件，这个工具特点是包含了 jarMapper 和工程Mapper的区分，但是两者的MapperXml文件都可能需要调用refresh()方法，这个刷新方法的逻辑和上面插件的刷新过程差别不多，均是清除原有的相关缓存，再重新解析MapperXml文件，将信息写回 Configuration 中。**

**这个类的方法代码看起来虽然很长，没有太多看下的去念头。好在其流程还是清晰的，且方法数量也不多。我从该类文件中的时间配置（延迟执行，周期执行）作为切入点去看代码，然后清晰许多了，一下子能够将延迟工作和周期工作的逻辑分离开。**

**在代码逻辑中，如果分支过多，让人感到心烦时候，也不要一个劲儿去看最具体实现，然后还给看得一头雾水。可以尝试将主要流程了解完，比如遇上一个if，可以先理解条件表达式，再大致浏览if体，有个大概了解，就可以看else if 或者 else 分支了。大致流程了解熟悉后，就可以来补充具体代码实现。哈哈，平常我写代码的套路也是这样的。**

**之前的心过于浮躁，心中想赶紧看完这部分的东西，就可以去看后续那些感觉更有趣的框架实现。但是眼前的关卡没通，且不感兴趣，鼠标上下滑动的频繁操作并不能让你快速去吸收理解，只是在做没有意义的挣扎。多次强调自己一定要沉下心来才能够有足够细致的理解体会。**

**再执行测试时候，失败了，很有挫败感。也不想要去死磕这部分的东西。暂时这样吧.等到后面具有稳定的版本，可以详实地对照一次，或者自己也能够实现一个出来，节省开发的时间。希望它能够支持的是：通过接口调用来实现开关后台插件。**

**2019年8月29日星期四**

**昨天晚上睡觉前，想着白天的插件测试失败情况，在测试一直失败的时候特别受挫，眼睛盯着屏幕，感觉都快要跳出来了。之后朋友让我陪他出去一趟，才中断了这个测试，不然我不知又得耗费多少精力在这个上面。我觉得不选择死磕才是正确的，我是个什么样的人啊，我根本不能用每个地方都去弄得明明白白，我只处理自己感兴趣的事情，这个才是taste过程中让人开心的地方。**

**今天，我要记录下mybatis框架中，有关反射包的使用。Mybatis框架封装的这个包很实用，值得学习。今天就要借助mybatis-plus上的使用方式，来熟悉其api。**

**在创建statementHandler实例后，interceptorChain调用所有的插件拦截器的plugin方法。**

**无比麻烦，基于问题驱动去debug调试mybatis框架的源代码。主要是对对mappedStatement实例的LanguageDriver部分的研究。 调试的切入点是cacheExcutor.doUpdate()开始。在mybatis框架中，默认的languageDriver本应为 XMLLanguageDriver，但是基于本次commit，这里的languageDriver必须为MybatisXMLLanguageDriver类型才行，它继承了XMLLanguageDriver类，覆盖了父类的createParameterHandler方法，这样才能够正常地引入plus框架自己写的MybatisDefaultParameterHandler类，该类扩展自DefaultParameterHandler，在保持原有功能情况下，增加对sql参数对象的处理，比如plus框架的主键自动生成，还有本次commit中对公共字段值统一补充的处理。**

**wrapToCollection(parameter)**

**CachingExecutor → SimpleExecutor**

**SimpleExecutor类的doUpdate方法：**

**@Override**

**public int** doUpdate**(MappedStatement** *ms***, Object** *parameter***) throws SQLException** {

**Statement** stmt = **null;**

**try** {

**Configuration** configuration = *ms***.**getConfiguration**();**

**StatementHandler** handler = configuration**.**newStatementHandler**(this,** *ms***,** *parameter***, RowBounds.*DEFAULT*, null, null);**

stmt = prepareStatement**(**handler**,** *ms***.**getStatementLog**());**

**return** handler**.**update**(**stmt**);**

} **finally** {

closeStatement**(**stmt**);**

}

}

**内部调用 PreparedStatementHandler 类的构造器**

**2020年1月6日星期一**

**开始了2020年新的征程。我的目标是继续学习mybatisplus框架，并从中彻底吃透mybatis框架。**

**Mybatis 调用sql，都经过了什么步骤**

**执行一次mapper方法的流程是：**

**调用代理类的invoke方法.**

**MapperRegistry:**

**缓存了不同Mapper接口的MapperProxyFactory(代理实例工厂)，工厂可以创建MapperProxy.**

**在开发中我们引用进来的各种 mapper 接口实例，它其实就是 MapperProxy 实例， 帮助我们完成 Mapper 接口要执行对应 xml 中sql的功能。**

protected **T** newInstance(MapperProxy<**T**> mapperProxy) {

return (**T**) Proxy.*newProxyInstance*(mapperInterface.getClassLoader()**,** new Class[] { mapperInterface }**,** mapperProxy)**;**

}

**MapperProxy:**

**它实现了InvocationHandler接口. 使得动态代理Proxy类能够构造该实例.**

**内部持有对象：**

|  |
| --- |
| **对应的 mapper 接口 Class对象** |
| **Mapper接口的Method哈希集合 methodCache 成员 ，他们每一个 method 映射到 对应的 MapperMethod对象** |
| **SqlSession对象** |

**关注该的invoke方法.**

**Invoke方法，首先获得mapper method对象，再根据 mapperMethod 对象的 execute 方法完成后面的sql调用。**

**传入参数是 sqlSession 和 mapper接口方法的方法参数.**

**获取mapper method 是通过一个缓存集合来获得，如果对应的MapperMethod对象已经缓存则从缓存中获得，否则需要构造一个MapperMethod.**

private MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = methodCache.get(method)**;**

if (mapperMethod == null) {

mapperMethod = new MapperMethod(mapperInterface**,** method**,** sqlSession.getConfiguration())**;**

methodCache.put(method**,** mapperMethod)**;**

}

return mapperMethod**;**

}

**如何构造一个 MapperMethod;**

public MapperMethod(Class<?> mapperInterface**,** Method method**,** Configuration config) {

this.command = new SqlCommand(config**,** mapperInterface**,** method)**;**

this.method = new MethodSignature(config**,** mapperInterface**,** method)**;**

}

构造一个 SqlCommand **,**

**通过拼接statementName，得到对应的 MappedStatement 对象，该对象包含其sql语句对应的sql command type，**

public SqlCommand(Configuration configuration**,** Class<?> mapperInterface**,** Method method) {

String statementName = mapperInterface.getName() + "." + method.getName()**;**

MappedStatement ms = null**;**

if (configuration.hasStatement(statementName)) {

ms = configuration.getMappedStatement(statementName)**;**

} else if (!mapperInterface.equals(method.getDeclaringClass())) { // issue #35

String parentStatementName = method.getDeclaringClass().getName() + "." + method.getName()**;**

if (configuration.hasStatement(parentStatementName)) {

ms = configuration.getMappedStatement(parentStatementName)**;**

}

}

if (ms == null) {

if(method.getAnnotation(Flush.class) != null){

name = null**;**

type = SqlCommandType.*FLUSH***;**

} else {

throw new BindingException("Invalid bound statement (not found): " + statementName)**;**

}

} else {

name = ms.getId()**;**

type = ms.getSqlCommandType()**;**

if (type == SqlCommandType.*UNKNOWN*) {

throw new BindingException("Unknown execution method for: " + name)**;**

}

}

}

**现在需要我们将焦点关注到mapper method 的execute方法中，就可以看出 switch 块的选择根据。**

**上游是：调用了mapper proxy 的invoke 方法，invoke方法调用了execute方法。**

**下面的代码中，请关注 Insert 和 select;**

**Insert 是我目前断点调试到的地方。（关注 sqlSession.insert(command.getName(), param)）**

**Select，是一个比较特殊值得说明的地方。（关注IF分支）**

**SELECT: 看到其中有多个if分支，是根据method的返回类型来调用对应的 execute\*方法，处理好查询返回结果集对应的数据结构。**

public Object execute(SqlSession sqlSession**,** Object[] args) {

Object result**;**

switch (command.getType()) {

case *INSERT*: {

Object param = method.convertArgsToSqlCommandParam(args)**;**

result = rowCountResult(sqlSession.insert(command.getName()**,** param))**;**

break**;**

}

case *UPDATE*: {

Object param = method.convertArgsToSqlCommandParam(args)**;**

result = rowCountResult(sqlSession.update(command.getName()**,** param))**;**

break**;**

}

case *DELETE*: {

Object param = method.convertArgsToSqlCommandParam(args)**;**

result = rowCountResult(sqlSession.delete(command.getName()**,** param))**;**

break**;**

}

case *SELECT*:

if (method.returnsVoid() && method.hasResultHandler()) {

executeWithResultHandler(sqlSession**,** args)**;**

result = null**;**

} else if (method.returnsMany()) {

result = executeForMany(sqlSession**,** args)**;**

} else if (method.returnsMap()) {

result = executeForMap(sqlSession**,** args)**;**

} else if (method.returnsCursor()) {

result = executeForCursor(sqlSession**,** args)**;**

} else {

Object param = method.convertArgsToSqlCommandParam(args)**;**

result = sqlSession.selectOne(command.getName()**,** param)**;**

}

break**;**

case *FLUSH*:

result = sqlSession.flushStatements()**;**

break**;**

default:

throw new BindingException("Unknown execution method for: " + command.getName())**;**

}

if (result == null && method.getReturnType().isPrimitive() && !method.returnsVoid()) {

throw new BindingException("Mapper method '" + command.getName()

+ " attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").")**;**

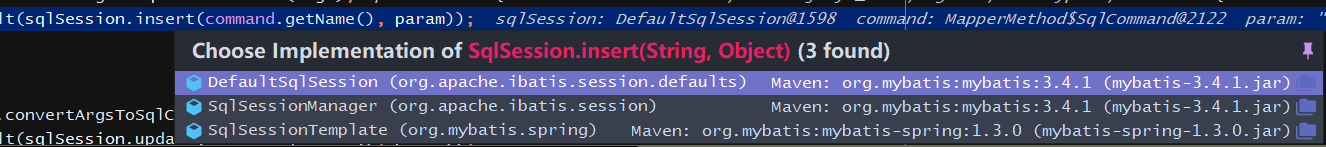
}

return result**;**

}

**SqlSession接口有3个实现类，当前断点查看的是 DefaultSqlSession，进去瞧瞧他的insert方法实现。**

**什么时候会用到其它实现类呢？**



@Override  
public int insert(String statement**,** Object parameter) {  
 return update(statement**,** parameter)**;**}

**Insert方法直接调用update方法. 想一想jdbc，jdbc中的statement是不是有 executeQuery 和 execut() 方法. 一个对应查询，另一个对应修改。 我们再看 update 方法.**

**参数：**

**statement : 映射SQL语句的标识符，他是唯一的( mapperInterface.method )**

**parameter：被映射过名称的参数.**

**方法内容： 获取相应的 MappedStatement对象，借助在SqlSession中注册的 executor 来继续执行 update。**

@Override  
public int update(String statement**,** Object parameter) {  
 try {  
 dirty = true**;** MappedStatement ms = configuration.getMappedStatement(statement)**;**

**// 关注点** return executor.update(ms**,** wrapCollection(parameter))**;** } catch (Exception e) {  
 throw ExceptionFactory.*wrapException*("Error updating database. Cause: " + e**,** e)**;** } finally {  
 ErrorContext.*instance*().reset()**;** }  
}

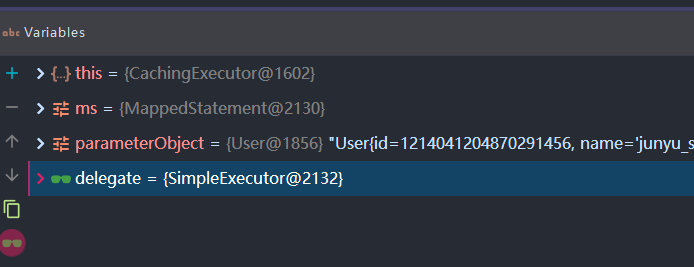
**当我们通过SqlSessionFactory创建一个SqlSession时，执行openSessionFromDatabase()方法时，会通过newExecutor()创建执行器。在后面不远处会贴上代码。**

**这里executor也是由一个接口，多个实现类的方式组成，当前使用的是 CacheingExecutor.**

@Override  
public int update(MappedStatement ms**,** Object parameterObject) throws SQLException {  
 flushCacheIfRequired(ms)**;**

**// 关注点** return delegate.update(ms**,** parameterObject)**;**}

**通过名字delegate ，明白这中间存在某一层代理关系。当前的调试面板是SimpleExecutor。**



**SimpleExecutor是MyBatis提供的默认的执行器,他里面封装了MyBatis对JDBC的操作, 虽然他叫XXXExecutor,但是真正去CRUD的还真不是SimpleExecutor**

**存在这一层代理关系，可以通过下面的方法看出来：使用CachingExecutor是因为开启了缓存配置（即 cacheEnabled == true）,他为executor代理相关的缓存操作。在cacheExecutor看来，他并不关心他的构造参数是具体哪一个Executor对象。**

**方法位置：**

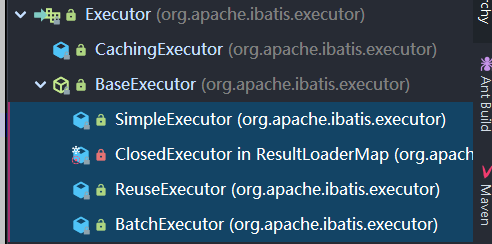
**org.apache.ibatis.session.Configuration#newExecutor(org.apache.ibatis.transaction.Transaction, org.apache.ibatis.session.ExecutorType)**

public Executor newExecutor(Transaction transaction**,** ExecutorType executorType) {  
 executorType = executorType == null ? defaultExecutorType : executorType**;** executorType = executorType == null ? ExecutorType.*SIMPLE* : executorType**;** Executor executor**;** if (ExecutorType.*BATCH* == executorType) {  
 executor = new BatchExecutor(this**,** transaction)**;** } else if (ExecutorType.*REUSE* == executorType) {  
 executor = new ReuseExecutor(this**,** transaction)**;** } else {  
 executor = new SimpleExecutor(this**,** transaction)**;** }

**// 关注点**  
 if (cacheEnabled) {  
 executor = new CachingExecutor(executor)**;** }  
 executor = (Executor) interceptorChain.pluginAll(executor)**;** return executor**;**}

**现在继续关注SimpleExecutor的实现。**

**SimpleExecutor实现了 BaseExecutor 抽象类. 他并没有重写父类的 update 方法.但是重写了其中的 doUpdate 方法，这种抽象方式挺优雅的。我父类实现一部分，而另外一部分我不提供实现，由子类提供。果然厉害啊。**



**Executor的关系图**

**方法位置：org.apache.ibatis.executor.BaseExecutor#update**

@Override  
public int update(MappedStatement ms**,** Object parameter) throws SQLException {  
 ErrorContext.*instance*().resource(ms.getResource()).activity("executing an update").object(ms.getId())**;** if (closed) {  
 throw new ExecutorException("Executor was closed.")**;** }  
 clearLocalCache()**;**

**// focus：要跳到具体子类的实现逻辑** return doUpdate(ms**,** parameter)**;**}

**SimpleExecutor#doUpdate.**

@Override  
public int doUpdate(MappedStatement ms**,** Object parameter) throws SQLException {  
 Statement stmt = null**;** try {  
 Configuration configuration = ms.getConfiguration()**;**

**// focus 1: handler 对象.** StatementHandler handler = configuration.newStatementHandler(this**,** ms**,** parameter**,** RowBounds.*DEFAULT***,** null**,** null)**;  
 // focus 2: stmt 对象是**Statement类型，已经开始涉及jdbc相关的api了

stmt = prepareStatement(handler**,** ms.getStatementLog())**;  
 // focus 3: update 方法**

return handler.update(stmt)**;** } finally {  
 closeStatement(stmt)**;** }  
}

**这一次的关注点应该是什么了呢？我想应该有3个地方:**

**Focus 1: handler对象**

**Focus2: stmt 对象是Statement类型，已经开始涉及jdbc相关的api了。 实际上要产生的是一个PreparedStatement对象。**

**Focus3: update方法;**

**方法本意应该是获得handler对象，由该对象的handler方法去代理statement对象执行sql的方法。**

**方法位置：org.apache.ibatis.executor.statement.PreparedStatementHandler#update**

**通过jdbc调用执行sql， 然后通过 keyGenerator.processAfter方法，处理了什么逻辑？**

@Override  
public int update(Statement statement) throws SQLException {  
 PreparedStatement ps = (PreparedStatement) statement**;** ps.execute()**;** int rows = ps.getUpdateCount()**;** Object parameterObject = boundSql.getParameterObject()**;** KeyGenerator keyGenerator = mappedStatement.getKeyGenerator()**;** keyGenerator.processAfter(executor**,** mappedStatement**,** ps**,** parameterObject)**;** return rows**;**}

**2020年1月9日星期四**

**继续上面的话题.**

**org.apache.ibatis.executor.SimpleExecutor#doUpdate**

**我们这次的焦点在此方法里面的 获取 StatementHandler对象逻辑中。**

StatementHandler handler = configuration.newStatementHandler(this**,** ms**,** parameter**,** RowBounds.*DEFAULT***,** null**,** null)**;**

**查看 newStatementHandler 方法的实现:**

**org.apache.ibatis.session.Configuration#newStatementHandler**

public StatementHandler newStatementHandler(Executor executor**,** MappedStatement mappedStatement**,** Object parameterObject**,** RowBounds rowBounds**,** ResultHandler resultHandler**,** BoundSql boundSql) {

**// 关注点**  
 StatementHandler statementHandler = new **RoutingStatementHandler**(executor**,** mappedStatement**,** parameterObject**,** rowBounds**,** resultHandler**,** boundSql)**;** statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler)**;** return statementHandler**;**}

**RoutingStatementHandler ， 他的工作是中间代理。这里仔细品，他还是一个多态的表现。**

**在 RoutingStatementHandler 的构造方法中，根据参数 mappedStatement的 type，路由到对应StatementHandler的实现类中。**

**所以这里的 statementHandler对象，当然就可以理解为是StatementHandler的具体实现类，比如 PreparedStatementHandler.**

**这是RoutingStatementHandler的构造方法.**

**org.apache.ibatis.executor.statement.RoutingStatementHandler#RoutingStatementHandler**

public RoutingStatementHandler(Executor executor**,** MappedStatement ms**,** Object parameter**,** RowBounds rowBounds**,** ResultHandler resultHandler**,** BoundSql boundSql) {  
  
 switch (ms.getStatementType()) {  
 case *STATEMENT*:  
 delegate = new SimpleStatementHandler(executor**,** ms**,** parameter**,** rowBounds**,** resultHandler**,** boundSql)**;** break**;** case *PREPARED*:  
 delegate = new PreparedStatementHandler(executor**,** ms**,** parameter**,** rowBounds**,** resultHandler**,** boundSql)**;** break**;** case *CALLABLE*:  
 delegate = new CallableStatementHandler(executor**,** ms**,** parameter**,** rowBounds**,** resultHandler**,** boundSql)**;** break**;** default:  
 throw new ExecutorException("Unknown statement type: " + ms.getStatementType())**;** }

}

**首先是声明父类接口引用，然后进行switch语句判断，选择构造具体的子类实例，最后调用子类实例的方法。**

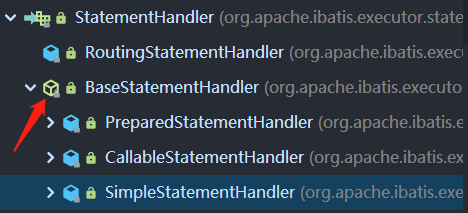
**但是按照自己的写法，完全是一个整段的代码，最多是分开为两个代码。**

**在来看看框架这段，典型的另外一种方式，借助中间代理来完成实际的子类调用，更具面向对象的特点。**

**继续关注如何构造PreparedStatementHandler代码，它引用了父类的实现。**

public PreparedStatementHandler(Executor executor**,** MappedStatement mappedStatement**,** Object parameter**,** RowBounds rowBounds**,** ResultHandler resultHandler**,** BoundSql boundSql) {  
 super(executor**,** mappedStatement**,** parameter**,** rowBounds**,** resultHandler**,** boundSql)**;**}

**PreparedStatementHandler的父类是：BaseStatementHandler**



**org.apache.ibatis.executor.statement.BaseStatementHandler#BaseStatementHandler**

**这是初始化 PreparementStatementHandler 的过程，该对象准备的数据将要如何使用，暂时还不清楚，可以在后面发现到了，回来这里填坑。**

**将每一个成员的情况标注出来。**

protected BaseStatementHandler(Executor executor**,** MappedStatement mappedStatement**,** Object parameterObject**,** RowBounds rowBounds**,** ResultHandler resultHandler**,** BoundSql boundSql) {  
 this.configuration = mappedStatement.getConfiguration()**;** this.executor = executor**;** this.mappedStatement = mappedStatement**;** this.rowBounds = rowBounds**;** this.typeHandlerRegistry = configuration.getTypeHandlerRegistry()**;** this.objectFactory = configuration.getObjectFactory()**;** if (boundSql == null) { // issue #435, get the key before calculating the statement  
 generateKeys(parameterObject)**;** boundSql = mappedStatement.getBoundSql(parameterObject)**;** }  
  
 this.boundSql = boundSql**;** this.parameterHandler = configuration.newParameterHandler(mappedStatement**,** parameterObject**,** boundSql)**;** this.resultSetHandler = configuration.newResultSetHandler(executor**,** mappedStatement**,** rowBounds**,** parameterHandler**,** resultHandler**,** boundSql)**;**}

**关注mappedStatement.getBound()方法**

**org.apache.ibatis.mapping.MappedStatement#getBoundSql**

public BoundSql getBoundSql(Object parameterObject) {

**// 关注点1**  
 BoundSql boundSql = sqlSource.getBoundSql(parameterObject)**;** List<ParameterMapping> parameterMappings = boundSql.getParameterMappings()**;** if (parameterMappings == null || parameterMappings.isEmpty()) {  
 boundSql = new BoundSql(configuration**,** boundSql.getSql()**,** parameterMap.getParameterMappings()**,** parameterObject)**;** }  
  
 // check for nested result maps in parameter mappings (issue #30)  
 for (ParameterMapping pm : boundSql.getParameterMappings()) {  
 String rmId = pm.getResultMapId()**;** if (rmId != null) {  
 ResultMap rm = configuration.getResultMap(rmId)**;** if (rm != null) {  
 hasNestedResultMaps |= rm.hasNestedResultMaps()**;** }  
 }  
 }  
  
 return boundSql**;**}

**这里的关注点1是为了解决代码运行过程出现的错误。它调用了sqlSource.getBoundSql();**

**这里的sqlSource对应的实现类为： DynamicSqlSource.**

**org.apache.ibatis.scripting.xmltags.DynamicSqlSource#getBoundSql**

@Override  
public BoundSql getBoundSql(Object parameterObject) {  
 DynamicContext context = new DynamicContext(configuration**,** parameterObject)**;** rootSqlNode.apply(context)**;** SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration)**;** Class<?> parameterType = parameterObject == null ? Object.class : parameterObject.getClass()**;** SqlSource sqlSource = sqlSourceParser.parse(context.getSql()**,** parameterType**,** context.getBindings())**;** BoundSql boundSql = sqlSource.getBoundSql(parameterObject)**;** for (Map.Entry<String**,** Object> entry : context.getBindings().entrySet()) {  
 boundSql.setAdditionalParameter(entry.getKey()**,** entry.getValue())**;** }  
 return boundSql**;**}

**我的程序在上面的parse中停止运行..**

**org.apache.ibatis.builder.SqlSourceBuilder#parse**

public SqlSource parse(String originalSql**,** Class<?> parameterType**,** Map<String**,** Object> additionalParameters) {  
 ParameterMappingTokenHandler handler = new ParameterMappingTokenHandler(configuration**,** parameterType**,** additionalParameters)**;** GenericTokenParser parser = new GenericTokenParser("#{"**,** "}"**,** handler)**;** String sql = parser.parse(originalSql)**;** return new StaticSqlSource(configuration**,** sql**,** handler.getParameterMappings())**;**}

**程序在上面的parse中停止运行..这时候，要怎么处理，再进去看一看.**

**INSERT INTO User ( test\_id,name,age,test\_type,role,phone ) VALUES**

**(#{id},#{name},#{age},#{testType},#{role.id},#{phone,typeHandler=im.lincq.mybatisplus.test.mysql.typehandler.PhoneTypeHandler} )**

**这一看，发现是我的sql语句有毛病. 这个需要去找找相关代码对比一下。**

**其实并不是，sql这么写，肯定就是希望能够将手机对象映射到user表中对应的 phone 字段去。只是这个映射过程出现了错误，找不到我这个自定义的TypeHandler类了。**

**关注GenericTokenParser # parse方法;**

public String parse(String text) {  
 final StringBuilder builder = new StringBuilder()**;** final StringBuilder expression = new StringBuilder()**;** if (text != null && text.length() > **0**) {  
 char[] src = text.toCharArray()**;** int offset = **0;** // search open token  
 int start = text.indexOf(openToken**,** offset)**;** while (start > -**1**) {  
 if (start > **0** && src[start - **1**] == '\\') {  
 // this open token is escaped. remove the backslash and continue.  
 builder.append(src**,** offset**,** start - offset - **1**).append(openToken)**;** offset = start + openToken.length()**;** } else {  
 // found open token. let's search close token.  
 expression.setLength(**0**)**;** builder.append(src**,** offset**,** start - offset)**;** offset = start + openToken.length()**;** int end = text.indexOf(closeToken**,** offset)**;** while (end > -**1**) {  
 if (end > offset && src[end - **1**] == '\\') {  
 // this close token is escaped. remove the backslash and continue.  
 expression.append(src**,** offset**,** end - offset - **1**).append(closeToken)**;** offset = end + closeToken.length()**;** end = text.indexOf(closeToken**,** offset)**;** } else {  
 expression.append(src**,** offset**,** end - offset)**;** offset = end + closeToken.length()**;** break**;** }  
 }  
 if (end == -**1**) {  
 // close token was not found.  
 builder.append(src**,** start**,** src.length - start)**;** offset = src.length**;** } else {

**// 在这里调用了TokenHandler实例的 handlerToken 方法**  
 builder.append(handler.handleToken(expression.toString()))**;** offset = end + closeToken.length()**;** }  
 }  
 start = text.indexOf(openToken**,** offset)**;** }  
 if (offset < src.length) {  
 builder.append(src**,** offset**,** src.length - offset)**;** }  
 }  
 return builder.toString()**;**}

**应该是每解析好一个占位符号，对应一次handlerToken方法的处理。**

**加入现在有一点衔接不上，则需要细品，仔细品。**

**TokenHandler 的实现类有多个，ParameterMappingTokenHandler类的实现，是我们这里出现报错问题的根本原因.**

public interface TokenHandler {  
 String handleToken(String content)**;**}

**对 #{xxxx} 内部 xxxx 的解析，**

**xxxx的表现形式可以为 : “id”(普通形式)，**

**“phone,typeHandler=com.xx.xx.PhoneTypeHandler”， （包含类型转换的规则类.）**

**这个方法有一些长.**

**org.apache.ibatis.builder.SqlSourceBuilder.ParameterMappingTokenHandler#buildParameterMapping**

private ParameterMapping buildParameterMapping(String content) {  
 Map<String**,** String> propertiesMap = parseParameterMapping(content)**;** String property = propertiesMap.get("property")**;** Class<?> propertyType**;** if (metaParameters.hasGetter(property)) { // issue #448 get type from additional params  
 propertyType = metaParameters.getGetterType(property)**;** } else if (typeHandlerRegistry.hasTypeHandler(parameterType)) {  
 propertyType = parameterType**;** } else if (JdbcType.*CURSOR*.name().equals(propertiesMap.get("jdbcType"))) {  
 propertyType = java.sql.ResultSet.class**;** } else if (property != null) {  
 MetaClass metaClass = MetaClass.*forClass*(parameterType**,** configuration.getReflectorFactory())**;** if (metaClass.hasGetter(property)) {  
 propertyType = metaClass.getGetterType(property)**;** } else {  
 propertyType = Object.class**;** }  
 } else {  
 propertyType = Object.class**;** }  
 ParameterMapping.Builder builder = new ParameterMapping.Builder(configuration**,** property**,** propertyType)**;** Class<?> javaType = propertyType**;** String typeHandlerAlias = null**;** for (Map.Entry<String**,** String> entry : propertiesMap.entrySet()) {  
 String name = entry.getKey()**;** String value = entry.getValue()**;** if ("javaType".equals(name)) {  
 javaType = resolveClass(value)**;** builder.javaType(javaType)**;** } else if ("jdbcType".equals(name)) {  
 builder.jdbcType(resolveJdbcType(value))**;** } else if ("mode".equals(name)) {  
 builder.mode(resolveParameterMode(value))**;** } else if ("numericScale".equals(name)) {  
 builder.numericScale(Integer.*valueOf*(value))**;** } else if ("resultMap".equals(name)) {  
 builder.resultMapId(value)**;** } else if ("typeHandler".equals(name)) {  
 typeHandlerAlias = value**;** } else if ("jdbcTypeName".equals(name)) {  
 builder.jdbcTypeName(value)**;** } else if ("property".equals(name)) {  
 // Do Nothing  
 } else if ("expression".equals(name)) {  
 throw new BuilderException("Expression based parameters are not supported yet")**;** } else {  
 throw new BuilderException("An invalid property '" + name + "' was found in mapping #{" + content + "}. Valid properties are " + *parameterProperties*)**;** }  
 }  
 if (typeHandlerAlias != null) {  
 builder.typeHandler(resolveTypeHandler(javaType**,** typeHandlerAlias))**;** }  
 return builder.build()**;**}

**SqlSource对象**

**Represents the content of a mapped statement read from an XML file or an annotation. It creates the SQL that will be passed to the database out of the input parameter received from the user.**

**代表了 mapped statement从xml文件或者注解中读取到的SQL。**

|  |  |
| --- | --- |
| **ProviderSqlSource** | **在注解中使用** |
| **StaticSqlSource** | **比DynamicSqlSource要快得多，它在注解sql解析中出现，在xmltags包出现，猜测，它可能是要将动态sql转换为StaticSqlSource，未验证。** |
| **DynamicSqlSource** | **在XML中使用，动态的** |
| **RawSqlSource** | **在XML中使用** |