**SQL语句构建器**

**问题**

Java程序员面对的最痛苦的事情之一就是在Java代码中嵌入SQL语句。这么来做通常是由于SQL语句需要动态来生成-否则可以将它们放到外部文件或者存储过程中。正如你已经看到的那样，MyBatis在它的XML映射特性中有一个强大的动态SQL生成方案。但有时在Java代码内部创建SQL语句也是必要的。此时，MyBatis有另外一个特性可以帮到你，在减少典型的加号,引号,新行,格式化问题和嵌入条件来处理多余的逗号或 AND 连接词之前。事实上，在Java代码中来动态生成SQL代码就是一场噩梦。例如：

String sql = "**SELECT** P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME, "

"P.LAST\_NAME,P.CREATED\_ON, P.UPDATED\_ON " +

"**FROM** PERSON P, **ACCOUNT** A " +

"**INNER** **JOIN** DEPARTMENT D **on** D.ID = P.DEPARTMENT\_ID " +

"**INNER** **JOIN** COMPANY C **on** D.COMPANY\_ID = C.ID " +

"**WHERE** (P.ID = A.ID **AND** P.FIRST\_NAME **like** ?) " +

"**OR** (P.LAST\_NAME **like** ?) " +

"**GROUP** **BY** P.ID " +

"**HAVING** (P.LAST\_NAME **like** ?) " +

"**OR** (P.FIRST\_NAME **like** ?) " +

"**ORDER** **BY** P.ID, P.FULL\_NAME";

**The Solution**

MyBatis 3提供了方便的工具类来帮助解决该问题。使用SQL类，简单地创建一个实例来调用方法生成SQL语句。上面示例中的问题就像重写SQL类那样：

**private** String selectPersonSql() {

**return** **new** SQL() {{

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME");

SELECT("P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON");

FROM("PERSON P");

FROM("ACCOUNT A");

INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID");

INNER\_JOIN("COMPANY C on D.COMPANY\_ID = C.ID");

WHERE("P.ID = A.ID");

WHERE("P.FIRST\_NAME like ?");

**OR**();

WHERE("P.LAST\_NAME like ?");

GROUP\_BY("P.ID");

HAVING("P.LAST\_NAME like ?");

**OR**();

HAVING("P.FIRST\_NAME like ?");

ORDER\_BY("P.ID");

ORDER\_BY("P.FULL\_NAME");

}}.toString();

}

该例中有什么特殊之处？当你仔细看时，那不用担心偶然间重复出现的"AND"关键字，或者在"WHERE"和"AND"之间的选择，抑或什么都不选。该SQL类非常注意"WHERE"应该出现在何处，哪里又应该使用"AND"，还有所有的字符串链接。

**SQL类**

这里给出一些示例：

// Anonymous inner class

**public** String **deletePersonSql**() {

**return** **new** SQL() {{

DELETE\_FROM("PERSON");

WHERE("ID = ${id}");

}}.toString();

}

// Builder / Fluent style

**public** String **insertPersonSql**() {

String sql = **new** SQL()

.INSERT\_INTO("PERSON")

.VALUES("ID, FIRST\_NAME", "${id}, ${firstName}")

.VALUES("LAST\_NAME", "${lastName}")

.toString();

**return** sql;

}

// With conditionals (note the final parameters, required for the anonymous inner class to access them)

**public** String **selectPersonLike**(**final** String id, **final** String firstName, **final** String lastName) {

**return** **new** SQL() {{

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST\_NAME, P.LAST\_NAME");

FROM("PERSON P");

**if** (id != **null**) {

WHERE("P.ID like ${id}");

}

**if** (firstName != **null**) {

WHERE("P.FIRST\_NAME like ${firstName}");

}

**if** (lastName != **null**) {

WHERE("P.LAST\_NAME like ${lastName}");

}

ORDER\_BY("P.LAST\_NAME");

}}.toString();

}

**public** String **deletePersonSql**() {

**return** **new** SQL() {{

DELETE\_FROM("PERSON");

WHERE("ID = ${id}");

}}.toString();

}

**public** String **insertPersonSql**() {

**return** **new** SQL() {{

INSERT\_INTO("PERSON");

VALUES("ID, FIRST\_NAME", "${id}, ${firstName}");

VALUES("LAST\_NAME", "${lastName}");

}}.toString();

}

**public** String **updatePersonSql**() {

**return** **new** SQL() {{

UPDATE("PERSON");

SET("FIRST\_NAME = ${firstName}");

WHERE("ID = ${id}");

}}.toString();

}

| **方法** | **描述** |
| --- | --- |
| SELECT(String) | 开始或插入到 SELECT子句。 可以被多次调用，参数也会添加到 SELECT子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。 |
| SELECT\_DISTINCT(String) | 开始或插入到 SELECT子句， 也可以插入 DISTINCT关键字到生成的查询语句中。 可以被多次调用，参数也会添加到 SELECT子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。 |
| FROM(String) | 开始或插入到 FROM子句。 可以被多次调用，参数也会添加到 FROM子句。 参数通常是表名或别名，也可以是数据库驱动程序接受的任意类型。 |
| JOIN(String),INNER\_JOIN(String),LEFT\_OUTER\_JOIN(String),RIGHT\_OUTER\_JOIN(String) | 基于调用的方法，添加新的合适类型的 JOIN子句。 参数可以包含由列命和join on条件组合成标准的join。 |
| WHERE(String) | 插入新的 WHERE子句条件， 由AND链接。可以多次被调用，每次都由AND来链接新条件。使用 OR() 来分隔OR。 |
| OR() | 使用OR来分隔当前的 WHERE子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的SQL。 |
| AND() | 使用AND来分隔当前的 WHERE子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的SQL。因为 WHERE 和 HAVING 二者都会自动链接 AND, 这是非常罕见的方法，只是为了完整性才被使用。 |
| GROUP\_BY(String) | 插入新的 GROUP BY子句元素，由逗号连接。 可以被多次调用，每次都由逗号连接新的条件。 |
| HAVING(String) | 插入新的 HAVING子句条件。 由AND连接。可以被多次调用，每次都由AND来连接新的条件。使用 OR() 来分隔OR. |
| ORDER\_BY(String) | 插入新的 ORDER BY子句元素， 由逗号连接。可以多次被调用，每次由逗号连接新的条件。 |
| DELETE\_FROM(String) | 开始一个delete语句并指定需要从哪个表删除的表名。通常它后面都会跟着WHERE语句！ |
| INSERT\_INTO(String) | 开始一个insert语句并指定需要插入数据的表名。后面都会跟着一个或者多个VALUES()。 |
| SET(String) | 针对update语句，插入到"set"列表中 |
| UPDATE(String) | 开始一个update语句并指定需要更新的表明。后面都会跟着一个或者多个SET()，通常也会有一个WHERE()。 |
| VALUES(String, String) | 插入到insert语句中。第一个参数是要插入的列名，第二个参数则是该列的值。 |

**SqlBuilder 和 SelectBuilder (已经废弃)**

在3.2版本之前，我们使用了一点不同的做法，通过实现ThreadLocal变量来掩盖一些导致Java DSL麻烦的语言限制。但这种方式已经废弃了，现代的框架都欢迎人们使用构建器类型和匿名内部类的想法。因此，SelectBuilder 和 SqlBuilder 类都被废弃了。

下面的方法仅仅适用于废弃的SqlBuilder 和 SelectBuilder 类。

| **方法** | **描述** |
| --- | --- |
| BEGIN() / RESET() | 这些方法清空SelectBuilder类的ThreadLocal状态，并且准备一个新的构建语句。开始新的语句时， BEGIN()读取得最好。 由于一些原因（在某些条件下，也许是逻辑需要一个完全不同的语句），在执行中清理语句 RESET()读取得最好。 |
| SQL() | 返回生成的 SQL() 并重置 SelectBuilder 状态 (好像 BEGIN() 或 RESET() 被调用了). 因此，该方法只能被调用一次！ |

SelectBuilder 和 SqlBuilder 类并不神奇，但是知道它们如何工作也是很重要的。 SelectBuilder 使用 SqlBuilder 使用了静态导入和ThreadLocal变量的组合来开启整洁语法，可以很容易地和条件交错。使用它们，静态导入类的方法即可，就像这样(一个或其它，并非两者):

**import** **static** **org**.apache.ibatis.jdbc.SelectBuilder.\*;

**import** **static** **org**.apache.ibatis.jdbc.SqlBuilder.\*;

这就允许像下面这样来创建方法：

/\* DEPRECATED \*/

**public** String **selectBlogsSql**() {

BEGIN(); // Clears ThreadLocal variable

SELECT("\*");

FROM("BLOG");

**return** SQL();

}

/\* DEPRECATED \*/

**private** String selectPersonSql() {

BEGIN(); // Clears ThreadLocal variable

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME");

SELECT("P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON");

FROM("PERSON P");

FROM("ACCOUNT A");

INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID");

INNER\_JOIN("COMPANY C on D.COMPANY\_ID = C.ID");

WHERE("P.ID = A.ID");

WHERE("P.FIRST\_NAME like ?");

**OR**();

WHERE("P.LAST\_NAME like ?");

GROUP\_BY("P.ID");

HAVING("P.LAST\_NAME like ?");

**OR**();

HAVING("P.FIRST\_NAME like ?");

ORDER\_BY("P.ID");

ORDER\_BY("P.FULL\_NAME");

**return** SQL();

}