

SQL语句构建器类

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
<ul style="list-style-type: none"> SELECT(String) 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。
<ul style="list-style-type: none"> SELECT_DISTINCT(String) 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。
<ul style="list-style-type: none"> FROM(String) 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。
<ul style="list-style-type: none"> JOIN(String) JOIN(String...) INNER_JOIN(String) INNER_JOIN(String...) LEFT_OUTER_JOIN(String) LEFT_OUTER_JOIN(String...) RIGHT_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。参数可以包含由列名和 join on 条件组合成标准的 join。
<ul style="list-style-type: none"> WHERE(String) 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。
<ul style="list-style-type: none"> GROUP_BY(String) 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。
<ul style="list-style-type: none"> HAVING(String) 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。
<ul style="list-style-type: none"> ORDER_BY(String) 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，每次都由逗号连接新的条件。

方法	描述
<ul style="list-style-type: none"> <code>LIMIT(String)</code> <code>LIMIT(int)</code> 	Appends a <code>LIMIT</code> clause. This method valid when use together with <code>SELECT()</code> , <code>UPDATE()</code> and <code>DELETE()</code> . And this method is designed to use together with <code>OFFSET()</code> when use <code>SELECT()</code> . (Available since 3.5.2)
<ul style="list-style-type: none"> <code>OFFSET(String)</code> <code>OFFSET(long)</code> 	Appends a <code>OFFSET</code> clause. This method valid when use together with <code>SELECT()</code> . And this method is designed to use together with <code>LIMIT()</code> . (Available since 3.5.2)
<ul style="list-style-type: none"> <code>OFFSET_ROWS(String)</code> <code>OFFSET_ROWS(long)</code> 	Appends a <code>OFFSET n ROWS</code> clause. This method valid when use together with <code>SELECT()</code> . And this method is designed to use together with <code>FETCH_FIRST_ROWS_ONLY()</code> . (Available since 3.5.2)
<ul style="list-style-type: none"> <code>FETCH_FIRST_ROWS_ONLY(String)</code> <code>FETCH_FIRST_ROWS_ONLY(int)</code> 	Appends a <code>FETCH FIRST n ROWS ONLY</code> clause. This method valid when use together with <code>SELECT()</code> . And this method is designed to use together with <code>OFFSET_ROWS()</code> . (Available since 3.5.2)
<code>DELETE_FROM(String)</code>	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！
<code>INSERT_INTO(String)</code>	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() or INTO_COLUMNS() and INTO_VALUES()。
<ul style="list-style-type: none"> <code>SET(String)</code> <code>SET(String...)</code> 	针对update语句，插入到"set"列表中
<code>UPDATE(String)</code>	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()。
<code>VALUES(String, String)</code>	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。
<code>INTO_COLUMNS(String...)</code>	Appends columns phrase to an insert statement. This should be call <code>INTO_VALUES()</code> with together.
<code>INTO_VALUES(String...)</code>	Appends values phrase to an insert statement. This should be call <code>INTO_COLUMNS()</code> with together.
<code>ADD_ROW()</code>	Add new row for bulk insert. (Available since 3.5.2)

NOTE It is important to note that SQL class writes `LIMIT`, `OFFSET`, `OFFSET n ROWS` and `FETCH FIRST n ROWS ONLY` clauses into the generated statement as is. In other words, the library does not attempt to normalize those values for databases that don't support these clauses directly. Therefore, it is very important for users to understand whether or not the target database supports these clauses. If the target database does not support these clauses, then it is likely that using this support will create SQL that has runtime errors.

Since version 3.4.2, you can use variable-length arguments as follows:

```

public String selectPersonSql() {
    return new SQL()
        .SELECT("P.ID", "A.USERNAME", "A.PASSWORD", "P.FULL_NAME", "D.DEPARTMENT_NAME", "C.COMPANY_NAME")
        .FROM("PERSON P", "ACCOUNT A")
        .INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID", "COMPANY C on D.COMPANY_ID = C.ID")
        .WHERE("P.ID = A.ID", "P.FULL_NAME like #{name}")
        .ORDER_BY("P.ID", "P.FULL_NAME")
        .toString();
}

public String insertPersonSql() {
    return new SQL()
        .INSERT_INTO("PERSON")
        .INTO_COLUMNS("ID", "FULL_NAME")
        .INTO_VALUES("#{id}", "#{fullName}")
        .toString();
}

public String updatePersonSql() {
    return new SQL()
        .UPDATE("PERSON")
        .SET("FULL_NAME = #{fullName}", "DATE_OF_BIRTH = #{dateOfBirth}")
        .WHERE("ID = #{id}")
        .toString();
}

```

Since version 3.5.2, you can create insert statement for bulk insert as follow:

```

public String insertPersonsSql() {
    // INSERT INTO PERSON (ID, FULL_NAME)
    //     VALUES (#{mainPerson.id}, #{mainPerson.fullName}) , (#{subPerson.id}, #{subPerson.fullName})
    return new SQL()
        .INSERT_INTO("PERSON")
        .INTO_COLUMNS("ID", "FULL_NAME")
        .INTO_VALUES("#{mainPerson.id}", "#{mainPerson.fullName}")
        .ADD_ROW()
        .INTO_VALUES("#{subPerson.id}", "#{subPerson.fullName}")
        .toString();
}

```

Since version 3.5.2, you can create select statement for limiting search result rows clause as follow:

```

public String selectPersonsWithOffsetLimitSql() {
    // SELECT id, name FROM PERSON
    //      LIMIT #{limit} OFFSET #{offset}
    return new SQL()
        .SELECT("id", "name")
        .FROM("PERSON")
        .LIMIT("#{limit}")
        .OFFSET("#{offset}")
        .toString();
}

public String selectPersonsWithFetchFirstSql() {
    // SELECT id, name FROM PERSON
    //      OFFSET #{offset} ROWS FETCH FIRST #{limit} ROWS ONLY
    return new SQL()
        .SELECT("id", "name")
        .FROM("PERSON")
        .OFFSET_ROWS("#{offset}")
        .FETCH_FIRST_ROWS_ONLY("#{limit}")
        .toString();
}

```

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();  
}
```