SQL 构建器类

问题

Java 开发人员必须做的一件最令人讨厌的事情就是将 SQL 嵌入到 Java 代码中。通常这样做是因为 SQL 必须动态生成，否则你可以在文件中或存储过程中将其外部化。正如你已经看到的，MyBatis 在其 XML 映射功能中为动态 SQL 生成提供了一个强大的答案。但是，有时有必要在 Java 代码中构建一个 SQL 语句字符串。在这种情况下，MyBatis 还有一个功能可以帮助你，在你将自己简化为典型的加号、引号、换行符、格式问题和嵌套条件混乱以处理额外的逗号或 AND 连接之前。事实上，在 Java 中动态生成 SQL 代码可能是一个真正的噩梦。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";
```

解决方案

MyBatis 3 提供了一个方便的实用程序类来帮助解决此问题。使用 SQL 类，你只需创建一个实例，让你可以针对它调用方法来逐个步骤构建 SQL 语句。使用 SQL 类重新编写时，上述示例问题将如下所示

```
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();
}
```

该示例有什么特别之处？好吧，如果你仔细观察，它不必担心意外重复"AND"关键字，或在"WHERE"和"AND"之间进行选择，或者根本不选择。SQL 类负责理解"WHERE"需要放在哪里，"AND"应该用在哪里以及所有字符串连接。

SQL 类

这里有一些示例

```
// 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	说明
• `SELECT(String)` • `SELECT(String...)`	启动或追加到 `SELECT` 子句。可以多次调用，参数将追加到 `SELECT` 子句。参数通常是逗号分隔的列和别名列表，但可以是驱动程序可以接受的任何内容。
• `SELECT_DISTINCT(String)` • `SELECT_DISTINCT(String...)`	启动或追加到 `SELECT` 子句，还向生成的查询添加 `DISTINCT` 关键字。可以多次调用，参数将追加到 `SELECT` 子句。参数通常是逗号分隔的列和别名列表，但可以是驱动程序可以接受的任何内容。
• `FROM(String)` • `FROM(String...)`	启动或追加到 `FROM` 子句。可以多次调用，参数将追加到 `FROM` 子句。参数通常是表名和别名，或驱动程序可以接受的任何内容。
• `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...)`	添加一个新的 `JOIN` 子句，类型取决于调用的方法。参数可以包括一个标准连接，其中包含要连接的列和条件。
• `WHERE(String)` • `WHERE(String...)`	追加一个新的 `WHERE` 子句条件，用 `AND` 连接。可以多次调用，这将导致它每次都用 `AND` 连接新条件。使用 `OR()` 用 `OR` 分割。
`OR()`	用 `OR` 分割当前 `WHERE` 子句条件。可以多次调用，但连续调用多次将生成不稳定的 `SQL`。
`AND()`	用 `AND` 分割当前 `WHERE` 子句条件。可以多次调用，但连续调用多次将生成不稳定的 `SQL`。因为 `WHERE` 和 `HAVING` 都自动与 `AND` 连接，所以这是一个非常不常用的方法，只是为了完整性才包含在内。
• `GROUP_BY(String)` • `GROUP_BY(String...)`	追加一个新的 `GROUP BY` 子句元素，用逗号连接。可以多次调用，这将导致它每次都用逗号连接新条件。
• `HAVING(String)` • `HAVING(String...)`	追加一个新的 `HAVING` 子句条件，用 `AND` 连接。可以多次调用，这将导致它每次都用 `AND` 连接新条件。使用 `OR()` 用 `OR` 分割。
• `ORDER_BY(String)` • `ORDER_BY(String...)`	追加新的 `ORDER BY` 子句元素，用逗号连接。可以多次调用，每次都会用逗号连接新条件。
• `LIMIT(String)` • `LIMIT(int)`	追加 `LIMIT` 子句。此方法与 SELECT()、UPDATE() 和 DELETE() 一起使用时有效。此方法设计为在使用 SELECT() 时与 OFFSET() 一起使用。（自 3.5.2 起可用）
• `OFFSET(String)` • `OFFSET(long)`	追加 `OFFSET` 子句。此方法与 SELECT() 一起使用时有效。此方法设计为在使用 SELECT() 时与 LIMIT() 一起使用。（自 3.5.2 起可用）
• `OFFSET_ROWS(String)`	追加 `OFFSET n ROWS` 子句。此方法与 SELECT() 一起使用时有效。此方法设计为在使用 SELECT() 时与 FETCH_FIRST_ROWS_ONLY() 一起使用。（自 3.5.2 起可用）

• `OFFSET_ROWS(long)`	
• `FETCH_FIRST_ROWS_ONLY(String)` • `FETCH_FIRST_ROWS_ONLY(int)`	追加 `FETCH FIRST n ROWS ONLY` 子句。此方法与 SELECT() 一起使用时有效。此方法设计为在使用 SELECT() 时与 OFFSET_ROWS() 一起使用。（自 3.5.2 起可用）
`DELETE_FROM(String)`	开始删除语句并指定要从中删除的表。通常应后跟 WHERE 语句！
`INSERT_INTO(String)`	开始插入语句并指定要插入到的表。应后跟一个或多个 VALUES() 或 INTO_COLUMNS() 和 INTO_VALUES() 调用。
• `SET(String)` • `SET(String...)`	追加到更新语句的"set"列表。
`UPDATE(String)`	开始更新语句并指定要更新的表。应后跟一个或多个 SET() 调用，通常还应后跟一个 WHERE() 调用。
`VALUES(String, String)`	追加到插入语句。第一个参数是要插入的列，第二个参数是要插入的值。
`INTO_COLUMNS(String...)`	追加列矩语到插入语句。这应与 INTO_VALUES() 一起调用。
`INTO_VALUES(String...)`	追加值矩语到插入语句。这应与 INTO_COLUMNS() 一起调用。
`ADD_ROW()`	添加新行以进行批量插入。（自 3.5.2 起可用）

注意请务必注意，SQL 类会按原样将 `LIMIT`、`OFFSET`、`OFFSET n ROWS` 和 `FETCH FIRST n ROWS ONLY` 子句写入生成的语句。换句话说，该库不会尝试为不支持这些子句的数据库规范化这些值。因此，用户了解目标数据库是否支持这些子句非常重要。如果目标数据库不支持这些子句，则使用此支持很可能会创建运行时错误的 SQL。

从版本 3.4.2 开始，您可以按如下方式使用可变长度参数

```
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();
}
```

从版本 3.5.2 开始，您可以按如下方式创建用于批量插入的插入语句

```
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();
}
```

从 3.5.2 版本开始，您可以创建用于限制搜索结果行的选择语句子句，如下所示

```
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（已弃用）

在 3.2 版本之前，我们采用了一种略有不同的方法，利用 ThreadLocal 变量来掩盖一些语言限制，这些限制使得 Java DSL 变得有点繁琐。但是，这种方法现在已被弃用，因为现代框架已经让人们习惯了使用构建器类型模式和匿名内部类来处理此类事情。因此，SelectBuilder 和 SqlBuilder 类已被弃用。

以下方法仅适用于已弃用的 SqlBuilder 和 SelectBuilder 类。

方法	说明
`BEGIN()` / `RESET()`	这些方法清除 SelectBuilder 类的 ThreadLocal 状态，并准备构建新语句。在开始新语句时，`BEGIN()` 的可读性最好。在执行过程中由于某种原因（也许逻辑要求在某些条件下使用完全不同的语句）而清除语句时，`RESET()` 的可读性最好。
`SQL()`	这将返回生成的 `SQL()` 并重置 `SelectBuilder` 状态（就像调用了 `BEGIN()` 或 `RESET()` 一样）。因此，此方法只能调用一次！

SelectBuilder 和 SqlBuilder 类并不是神奇的，但了解它们的工作原理很重要。SelectBuilder 和 SqlBuilder 使用静态导入和 ThreadLocal 变量的组合来启用可以轻松与条件交织的简洁语法。要使用它们，您可以像这样从类中静态导入方法（一个或另一个，不能同时使用）

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

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

这允许我们创建如下方法

```
/* 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();
}
```