原生 SQL 语句的编写

```
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";
```

使用 SQL 语句构建器编写 SQL 语句

```
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();

}
```

SQL 语句构建器的两种使用风格

1. 匿名内部类风格

```
// 匿名内部类风格

public String deletePersonSql() {

  return new SQL() {{

    DELETE_FROM("PERSON");

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

  }}.toString();

}
```

2. Builder/Fluent 风格

```
public String insertPersonSql() {

  String sql = new SQL()

    .INSERT_INTO("PERSON")

    .VALUES("ID, FIRST_NAME", "#{id}, #{firstName}")

    .VALUES("LAST_NAME", "#{lastName}")
```

```
    .toString();

  return sql;

}
```

3.动态条件（参数需要使用 final 修饰，以便返回值中的匿名内部类使用）

```
public String selectPersonLike(final String id, final String firstName, fina
l 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();

}
```

```
    SET("FIRST_NAME = #{firstName}");
    WHERE("ID = #{id}");
  }}.toString();
}
```

方法	描述
• `SELECT(String)` • `SELECT(String...)`	开始新的或追加到已有的 `SELECT` 子句。可以被多次调用，参数会被追加到 `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参数。
• `SELECT_DISTINCT(String)` • `SELECT_DISTINCT(String...)`	开始新的或追加到已有的 `SELECT` 子句，并添加 `DISTINCT` 关键字到生成的查询中。可以被多次调用，参数会被追加到 `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参数。
• `FROM(String)` • `FROM(String...)`	开始新的或追加到已有的 `FROM` 子句。可以被多次调用，参数会被追加到 `FROM` 子句。 参数通常是一个表名或别名，也可以是数据库驱动程序接受的任意参数。
• `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。 参数可以包含一个由列和连接条件构成的标准连接。
• `WHERE(String)` • `WHERE(String...)`	插入新的 `WHERE` 子句条件，并使用 `AND` 拼接。可以被多次调用，对于每一次调用产生的新条件，会使用 `AND` 拼接起来。要使用 `OR` 分隔，请使用 `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会生成错误的 `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会生成错误的 `SQL`。由于 `WHERE` 和 `HAVING` 都会自动使用 `AND` 拼接, 因此这个方法并不常用，只是为了完整性才被定义出来。
• `GROUP_BY(String)` • `GROUP_BY(String...)`	追加新的 `GROUP BY` 子句，使用逗号拼接。可以被多次调用，每次调用都会使用逗号将新的条件拼接起来。
• `HAVING(String)` • `HAVING(String...)`	追加新的 `HAVING` 子句。使用 AND 拼接。可以被多次调用，每次调用都使用 `AND` 来拼接新的条件。要使用 `OR` 分隔，请使用 `OR()`。
• `ORDER_BY(String)` • `ORDER_BY(String...)`	追加新的 `ORDER BY` 子句，使用逗号拼接。可以多次被调用，每次调用会使用逗号拼接新的条件。

方法	描述
• `LIMIT(String)` • `LIMIT(int)`	追加新的 `LIMIT` 子句。 仅在 SELECT()、UPDATE()、DELETE() 时有效。 当在 SELECT() 中使用时，应该配合 OFFSET() 使用。（于 3.5.2 引入）
• `OFFSET(String)` • `OFFSET(long)`	追加新的 `OFFSET` 子句。 仅在 SELECT() 时有效。 当在 SELECT() 时使用时，应该配合 LIMIT() 使用。（于 3.5.2 引入）
• `OFFSET_ROWS(String)` • `OFFSET_ROWS(long)`	追加新的 `OFFSET n ROWS` 子句。 仅在 SELECT() 时有效。 该方法应该配合 FETCH_FIRST_ROWS_ONLY() 使用。（于 3.5.2 加入）
• `FETCH_FIRST_ROWS_ONLY(String)` • `FETCH_FIRST_ROWS_ONLY(int)`	追加新的 `FETCH FIRST n ROWS ONLY` 子句。 仅在 SELECT() 时有效。 该方法应该配合 OFFSET_ROWS() 使用。（于 3.5.2 加入）
`DELETE_FROM(String)`	开始新的 delete 语句，并指定删除表的表名。通常它后面都会跟着一个 WHERE 子句！
`INSERT_INTO(String)`	开始新的 insert 语句，并指定插入数据表的表名。后面应该会跟着一个或多个 VALUES() 调用，或 INTO_COLUMNS() 和 INTO_VALUES() 调用。
• `SET(String)` • `SET(String...)`	对 update 语句追加 "set" 属性的列表
`UPDATE(String)`	开始新的 update 语句，并指定更新表的表名。后面都会跟着一个或多个 SET() 调用，通常也会有一个 WHERE() 调用。
`VALUES(String, String)`	追加数据值到 insert 语句中。第一个参数是数据插入的列名，第二个参数则是数据值。
`INTO_COLUMNS(String...)`	追加插入列子句到 insert 语句中。应与 INTO_VALUES() 一同使用。
`INTO_VALUES(String...)`	追加插入值子句到 insert 语句中。应与 INTO_COLUMNS() 一同使用。
`ADD_ROW()`	添加新的一行数据，以便执行批量插入。（于 3.5.2 引入）

提示 注意，SQL 类将原样插入 `LIMIT`、`OFFSET`、`OFFSET n ROWS` 以及 `FETCH FIRST n ROWS ONLY` 子句。换句话说，类库不会为不支持这些子句的数据库执行任何转换。 因此，用户应该要了解目标数据库是否支持这些子句。如果目标数据库不支持这些子句，产生的 SQL 可能会引起运行错误。

从版本 3.4.2 开始，你可以像下面这样使用可变长度参数：

```
public String selectPersonSql() {
  return new SQL()
    .SELECT("P.ID", "A.USERNAME", "A.PASSWORD", "P.FULL_NAME", "D.DEPARTMENT_NAME", "C.COMPANY_NAM
E")
    .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，你可以像下面这样构建批量插入语句：

```
public String insertPersonsSql() {
  // INSERT INTO PERSON (ID, FULL_NAME)
  //     VALUES (#{mainPerson.id}, #{mainPerson.fullName}) , (#{subPerson.id}, #{subPerson.fullNam
e})
  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();
}
```

从版本 3.5.2 开始，你可以像下面这样构建限制返回结果数的 SELECT 语句,：

```
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)

在版本 3.2 之前，我们的实现方式不太一样，我们利用 ThreadLocal 变量来掩盖一些对 Java DSL 不太友好的语言限制。现在，现代 SQL 构建框架使用的构建器和匿名内部类思想已被人们所熟知。因此，我们废弃了基于这种实现方式的 SelectBuilder 和 SqlBuilder 类。

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，并准备好构建一个新的语句。开始新的语句时，`BEGIN()` 是最名副其实的（可读性最好的）。但如果由于一些原因（比如程序逻辑在某些条件下需要一个完全不同的语句），在执行过程中要重置语句构建状态，就很适合使用 `RESET()`。
`SQL()`	该方法返回生成的 `SQL()` 并重置 `SelectBuilder` 状态（等价于调用了 `BEGIN()` 或 `RESET()`）。因此，该方法只能被调用一次！

SelectBuilder 和 SqlBuilder 类并不神奇，但最好还是知道它们的工作原理。 SelectBuilder 以及 SqlBuilder 借助静态导入和 ThreadLocal 变量实现了对插入条件友好的简洁语法。要使用它们，只需要静态导入这个类的方法即可，就像这样（只能使用其中的一条，不能同时使用）：

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

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

然后就可以像下面这样创建一些方法：

```
/* 已被废弃 */
public String selectBlogsSql() {
  BEGIN(); // 重置 ThreadLocal 状态变量
  SELECT("*");
  FROM("BLOG");
  return SQL();
}
```

```
/* 已被废弃 */
private String selectPersonSql() {
  BEGIN(); // 重置 ThreadLocal 状态变量
  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();
}
```
