扫一扫二维码关注公众号

微信公众号福利：芋艿的后端小屋

2. 您对于源码的疑问每条留言都将得到认真回复

3. 新的源码解析文章实时收到通知，每周六十点更新

4. 认真的源码交流微信群

分类

Sharding-JDBC 源码分析 —— SQL 路由改写

2017-08-10 更新日期:2017-08-03 总阅读量:18次

文章目录

分享：

分享

> □□□关注**微信公众号：【芋艿的后端小屋】**有福利：
>
> 1. RocketMQ / MyCAT / Sharding-JDBC **所有**源码分析文章列表
> 2. RocketMQ / MyCAT / Sharding-JDBC **中文注释源码 GitHub 地址**
> 3. 您对于源码的疑问每条留言**都**将得到**认真**回复。**甚至不知道如何读源码也可以请教噢。**
> 4. **新的**源码解析文章**实时**收到通知。**每周更新一篇左右。**
> 5. **认真的**源码交流微信群。

1. 概述

前置阅读：《SQL 解析 (三) 之查询SQL》

本文分享**SQL 改写**的源码实现。主要涉及两方面：

1. SQL 改写：改写 SQL，解决分库分表后，查询结果需要聚合，需要对 SQL 进行调整，例如分页
2. SQL 生成：生成分表分库的执行 SQL

SQLRewriteEngine，SQL重写引擎，实现 SQL 改写、生成功能。从 Sharding-JDBC 1.5.0 版本，SQL 改写进行了调整和大量优化。

> 1.4.x及之前版本，SQL改写是在SQL路由之前完成的，在1.5.x中调整为SQL路由之后，因为SQL改写可以根据路由至单库表还是多库表而进行进一步优化。

😄 很多同学看完《SQL 解析-系列》可能是一脸懵逼，特别对**“SQL 半理解”**。

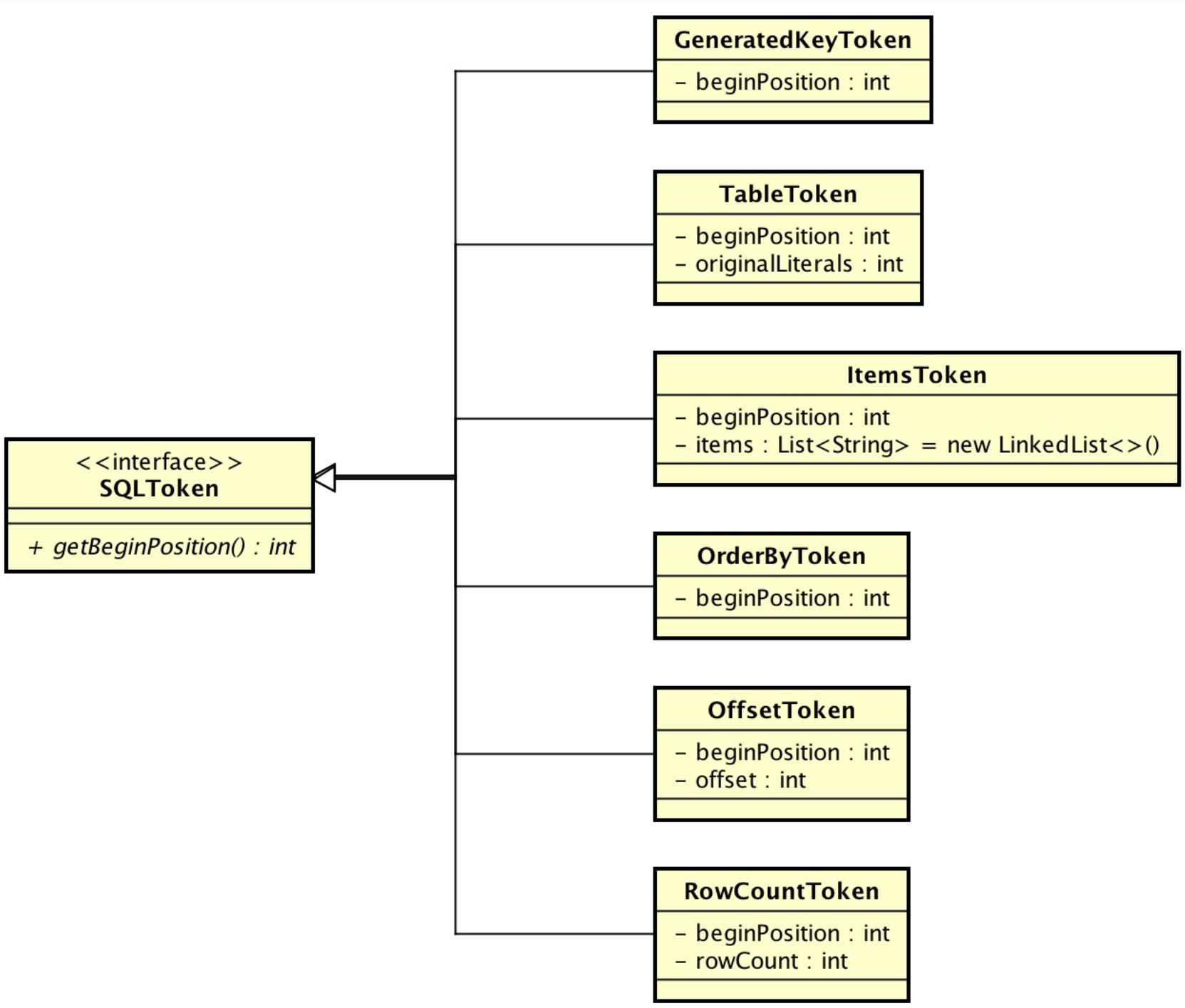
希望本文能给你一些启发。

> **Sharding-JDBC 正在收集使用公司名单：传送门。**
> **你的登记，会让更多人参与和使用 Sharding-JDBC。传送门**
> **Sharding-JDBC 也会因此，能够覆盖更多的业务场景。传送门**
> **登记吧，骚年！传送门**

2. SQLToken

😄 SQLToken 在本文中很重要，所以即使在《SQL 解析-系列》已经分享过，我们也换个姿势，再来一次。

SQLToken，SQL标记对象**接口**。SQLRewriteEngine 基于 SQLToken 实现 **SQL改写**。SQL解析器在 SQL解析过程中，很重要的一个目的是**标记需要SQL改写的部分**，也就是 SQLToken。


<<interface>>
SQLToken
+ getBeginPosition() : int
GeneratedKeyToken
- beginPosition : int
TableToken
- beginPosition : int
- originalLiterals : int
ItemsToken
- beginPosition : int
- items : List<String> = new LinkedList<>()
OrderByToken
- beginPosition : int
OffsetToken
- beginPosition : int
- offset : int
RowCountToken
- beginPosition : int
- rowCount : int

各 SQLToken 生成条件如下(*悲伤，做成表格形式排版是乱的*)：

1. GeneratedKeyToken 自增主键标记对象
    1. 插入SQL自增列不存在：`INSERT INTO t_order(nickname) VALUES ...` 中没有自增列 `order_id`
2. TableToken 表标记对象
    1. 查询列的表别名：`SELECT o.order_id` 的 `o`
    2. 查询的表名：`SELECT * FROM t_order` 的 `t_order`
3. ItemsToken 选择项标记对象
    1. AVG查询列：`SELECT AVG(price) FROM t_order` 的 `AVG(price)`
    2. ORDER BY 字段不在查询列：`SELECT order_id FROM t_order ORDER BY create_time` 的 `create_time`
    3. GROUP BY 字段不在查询列：`SELECT COUNT(order_id) FROM t_order GROUP BY user_id` 的 `user_id`
    4. 自增主键未在插入列中：`INSERT INTO t_order(nickname) VALUES ...` 中没有自增列 `order_id`
4. OffsetToken 分页偏移量标记对象
    1. 分页有偏移量，但**不是**占位符 `?`
5. RowCountToken 分页长度标记对象
    1. 分页有长度，但**不是**占位符 `?`
6. OrderByToken 排序标记对象
    1. 有 GROUP BY 条件，无 ORDER BY 条件：`SELECT COUNT(*) FROM t_order GROUP BY order_id` 的 `order_id`

3.SQL 改写

`SQLRewriteEngine#rewrite()` 实现了 **SQL改写** 功能。

```
// SQLRewriteEngine.java
/**
* SQL改写.
* @param isRewriteLimit 是否重写Limit
* @return SQL构建器
*/
public SQLBuilder rewrite(final boolean isRewriteLimit) {
   SQLBuilder result = new SQLBuilder();
   if (sqlTokens.isEmpty()) {
       result.appendLiterals(originalSQL);
```

```
            return result;
        }
        int count = 0;
        // 排序SQLToken，按照 beginPosition 递增
        sortByBeginPosition();
        for (SQLToken each : sqlTokens) {
            if (0 == count) { // 拼接第一个 SQLToken 前的字符串
                result.appendLiterals(originalSQL.substring(0, each.getBeginPosition()));
            }
            // 拼接每个SQLToken
            if (each instanceof TableToken) {
                appendTableToken(result, (TableToken) each, count, sqlTokens);
            } else if (each instanceof ItemsToken) {
                appendItemsToken(result, (ItemsToken) each, count, sqlTokens);
            } else if (each instanceof RowCountToken) {
                appendLimitRowCount(result, (RowCountToken) each, count, sqlTokens, isRewriteLimit);
            } else if (each instanceof OffsetToken) {
                appendLimitOffsetToken(result, (OffsetToken) each, count, sqlTokens, isRewriteLimit);
            } else if (each instanceof OrderByToken) {
                appendOrderByToken(result);
            }
            count++;
        }
        return result;
    }
```

- SQL改写以 SQLToken 为**间隔**，**顺序**改写。
  - 顺序：调用 `#sortByBeginPosition()` 将 SQLToken 按照 `beginPosition` **升序**。
  - 间隔：遍历 SQLToken，逐个拼接。

例如：

SELECT	o3	.* FROM (SELECT * FROM (SELECT * FROM	t_order	o) o2) o3 JOIN t_order_item i ON o3.order_id = i.order_id;
	TableToken		TableToken	

SQLBuilder，SQL构建器。下文会大量用到，我们看下实现代码。

```
public final class SQLBuilder {
    /**
     * 段集合
     */
    private final List<Object> segments;
```

```
    /**
     * 当前段
     */
    private StringBuilder currentSegment;

    public SQLBuilder() {
        segments = new LinkedList<>();
        currentSegment = new StringBuilder();
        segments.add(currentSegment);
    }

    /**
     * 追加字面量.
     *
     * @param literals 字面量
     */
    public void appendLiterals(final String literals) {
        currentSegment.append(literals);
    }

    /**
     * 追加表占位符.
     *
     * @param tableName 表名称
     */
    public void appendTable(final String tableName) {
        // 添加 TableToken
        segments.add(new TableToken(tableName));
        // 新建当前段
        currentSegment = new StringBuilder();
        segments.add(currentSegment);
    }

    public String toSQL(final Map<String, String> tableTokens) {
        // ... 省略代码，【SQL生成】处分享
    }

    @RequiredArgsConstructor
    private class TableToken {
        /**
         * 表名
         */
        private final String tableName;
    }
```

```
}
```

现在我们来逐个分析每种 SQLToken 的**拼接**实现。

3.1 TableToken

调用 `#appendTableToken()` 方法拼接。

```
// SQLRewriteEngine.java
/**
* 拼接 TableToken
*
* @param sqlBuilder SQL构建器
* @param tableToken tableToken
* @param count tableToken 在 sqlTokens 的顺序
* @param sqlTokens sqlTokens
*/
private void appendTableToken(final SQLBuilder sqlBuilder, final TableToken tableToken, final int count, final List<SQLToken> sqlTokens) {
   // 拼接 TableToken
   String tableName = sqlStatement.getTables().getTableNames().contains(tableToken.getTableName()) ? tableToken.getTableName() : tableToken.getOriginalLi
   sqlBuilder.appendTable(tableName);
   // 拼接 SQLToken 后面的字符串
   int beginPosition = tableToken.getBeginPosition() + tableToken.getOriginalLiterals().length();
   int endPosition = sqlTokens.size() - 1 == count ? originalSQL.length() : sqlTokens.get(count + 1).getBeginPosition();
   sqlBuilder.appendLiterals(originalSQL.substring(beginPosition, endPosition));
}
```

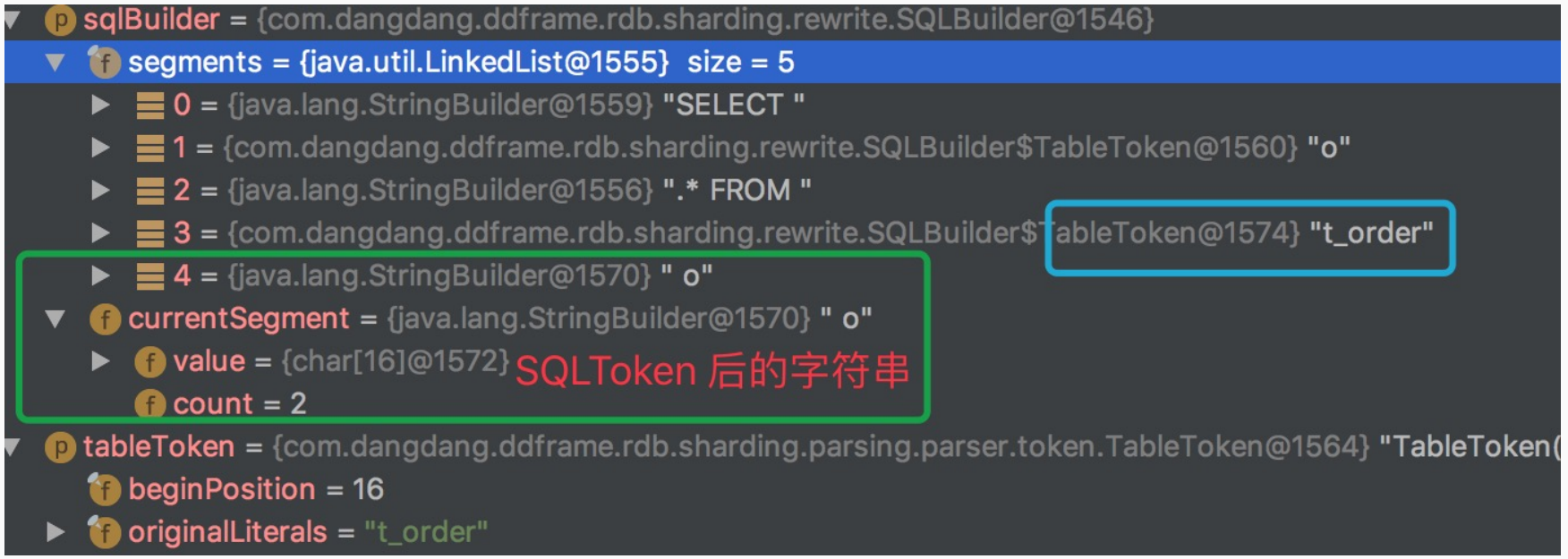
- 调用 `SQLBuilder#appendTable()` 拼接 TableToken。
- `sqlStatement.getTables().getTableNames().contains(tableToken.getTableName())` 目的是处理掉**表名前后有的特殊字符**，例如 `SELECT * FROM 't_order'` 中 `t_order` 前后有 `'` 符号。

```
// TableToken.java
/**
* 获取表名称.
*/
public String getTableName() {
   return SQLUtil.getExactlyValue(originalLiterals);
}
// SQLUtil.java
public static String getExactlyValue(final String value) {
   return null == value ? null : CharMatcher.anyOf("[]`'\"").removeFrom(value);
```

```
}
```

- 当 SQL 为 `SELECT o.* FROM t_order o`
  - TableToken 为查询列前的表别名 `o` 时返回结果：

  - TableToken 为表名 `t_order` 时返回结果：



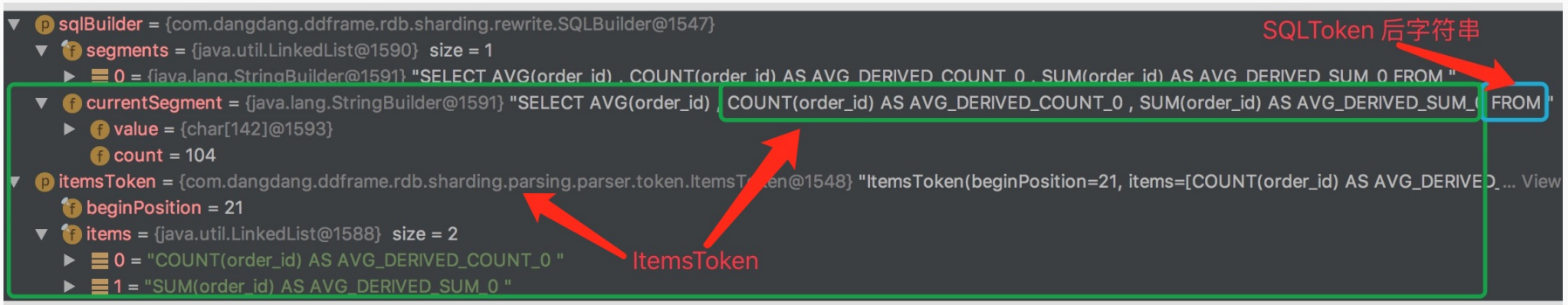
3.2 ItemsToken

调用 `#appendItemsToken()` 方法拼接。

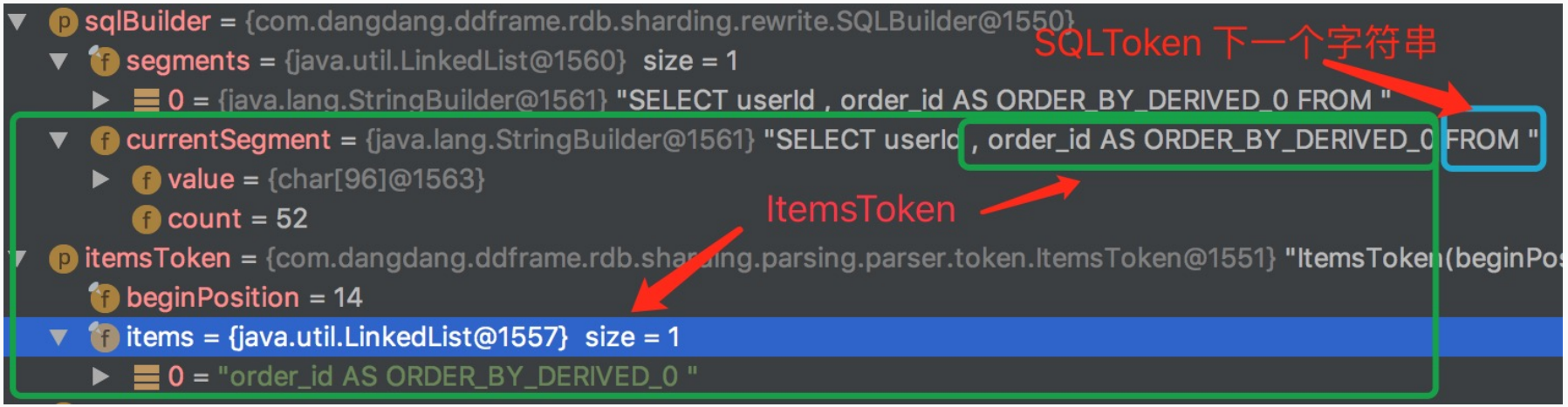
```
// SQLRewriteEngine.java
/**
* 拼接 TableToken
*
* @param sqlBuilder SQL构建器
* @param itemsToken itemsToken
* @param count itemsToken 在 sqlTokens 的顺序
* @param sqlTokens sqlTokens
*/
private void appendItemsToken(final SQLBuilder sqlBuilder, final ItemsToken itemsToken, final int count, final List<SQLToken> sqlTokens) {
   // 拼接 ItemsToken
   for (String item : itemsToken.getItems()) {

       sqlBuilder.appendLiterals(", ");
       sqlBuilder.appendLiterals(item);
   }
   // SQLToken 后面的字符串
   int beginPosition = itemsToken.getBeginPosition();
   int endPosition = sqlTokens.size() - 1 == count ? originalSQL.length() : sqlTokens.get(count + 1).getBeginPosition();
   sqlBuilder.appendLiterals(originalSQL.substring(beginPosition, endPosition));
}
```

- 第一种情况，**AVG查询列**，SQL 为 `SELECT AVG(order_id) FROM t_order o` 时返回结果：



- 第二种情况，**ORDER BY 字段不在查询列**，SQL 为 `SELECT userId FROM t_order o ORDER BY order_id` 时返回结果：



- 第三种情况，**GROUP BY 字段不在查询列**，类似第二种情况，就不举例子列。

3.3 OffsetToken

调用 `#appendLimitOffsetToken()` 方法拼接。

```
// SQLRewriteEngine.java
/**
* 拼接 OffsetToken
*
* @param sqlBuilder SQL构建器
* @param offsetToken offsetToken
* @param count offsetToken 在 sqlTokens 的顺序
* @param sqlTokens sqlTokens
* @param isRewrite 是否重写。当路由结果为单分片时无需重写
*/
private void appendLimitOffsetToken(final SQLBuilder sqlBuilder, final OffsetToken offsetToken, final int count, final List<SQLToken> sqlTokens, final bo
   // 拼接 OffsetToken
   sqlBuilder.appendLiterals(isRewrite ? "0" : String.valueOf(offsetToken.getOffset()));
   // SQLToken 后面的字符串
   int beginPosition = offsetToken.getBeginPosition() + String.valueOf(offsetToken.getOffset()).length();
   int endPosition = sqlTokens.size() - 1 == count ? originalSQL.length() : sqlTokens.get(count + 1).getBeginPosition();
   sqlBuilder.appendLiterals(originalSQL.substring(beginPosition, endPosition));
}
```

- 当分页**跨分片**时，需要每个分片都查询后在**内存**中进行聚合。此时 `isRewrite = true`。为什么是 `"0"` 开始呢？每个分片在 [0, offset) 的记录**可能**属于实际分页结果，因而查询每个分片需要从 0 开始。

- 当分页**单分片**时，则无需重写，该分片执行的结果即是最终结果。**SQL改写在SQL路由之后就有这个好处**。如果先改写，因为没办法知道最终是单分片还是跨分片，考虑正确性，只能统一使用跨分片。

3.4 RowCountToken

调用 `#appendLimitRowCount()` 方法拼接。

```
// SQLRewriteEngine.java
private void appendLimitRowCount(final SQLBuilder sqlBuilder, final RowCountToken rowCountToken, final int count, final List<SQLToken> sqlTokens, final b
   SelectStatement selectStatement = (SelectStatement) sqlStatement;
   Limit limit = selectStatement.getLimit();
   if (!isRewrite) { // 路由结果为单分片
       sqlBuilder.appendLiterals(String.valueOf(rowCountToken.getRowCount()));
   } else if ((!selectStatement.getGroupByItems().isEmpty() || // [1.1] 跨分片分组需要在内存计算，可能需要全部加载
           !selectStatement.getAggregationSelectItems().isEmpty()) // [1.2] 跨分片聚合列需要在内存计算，可能需要全部加载
           && !selectStatement.isSameGroupByAndOrderByItems()) { // [2] 如果排序一致，即各分片已经排序好结果，就不需要全部加载
       sqlBuilder.appendLiterals(String.valueOf(Integer.MAX_VALUE));
   } else { // 路由结果为多分片
       sqlBuilder.appendLiterals(String.valueOf(limit.isRowCountRewriteFlag() ? rowCountToken.getRowCount() + limit.getOffsetValue() : rowCountToken.getR
   }
   // SQLToken 后面的字符串
   int beginPosition = rowCountToken.getBeginPosition() + String.valueOf(rowCountToken.getRowCount()).length();
   int endPosition = sqlTokens.size() - 1 == count ? originalSQL.length() : sqlTokens.get(count + 1).getBeginPosition();
   sqlBuilder.appendLiterals(originalSQL.substring(beginPosition, endPosition));
}
```

- [1.1] `!selectStatement.getGroupByItems().isEmpty()` 跨分片**分组**需要在内存计算，**可能**需要全部加载。如果不全部加载，部分结果被分页条件错误结果，会导致结果不正确。
- [1.2] `!selectStatement.getAggregationSelectItems().isEmpty())` 跨分片**聚合列**需要在内存计算，**可能**需要全部加载。如果不全部加载，部分结果被分页条件错误结果，会导致结果不正确。
- [1.1][1.2]，**可能**变成必须的前提是 GROUP BY 和 ORDER BY 排序不一致。如果一致，各分片已经排序完成，无需内存中排序。

3.4.1 分页补充

OffsetToken、RowCountToken 只有在分页对应位置非占位符 `?` 才存在。当对应位置是占位符时，会对**分页条件对应的预编译 SQL 占位符参数**进行重写，**整体逻辑和 OffsetToken、RowCountToken 是一致的**。

```
// 🐜 ParsingSQLRouter#route() 调用 #processLimit()
// ParsingSQLRouter.java
/**
* 处理分页条件
```

```
     * @see SQLRewriteEngine#appendLimitRowCount(SQLBuilder, RowCountToken, int, List, boolean)
     * @param parameters 占位符对应参数列表
     * @param selectStatement Select SQL语句对象
     * @param isSingleRouting 是否单表路由
     */
    private void processLimit(final List<Object> parameters, final SelectStatement selectStatement, final boolean isSingleRouting) {
        boolean isNeedFetchAll = (!selectStatement.getGroupByItems().isEmpty() // // [1.1] 跨分片分组需要在内存计算，可能需要全部加载
                                  || !selectStatement.getAggregationSelectItems().isEmpty()) // [1.2] 跨分片聚合列需要在内存计算，可能需要全部加载
                                 && !selectStatement.isSameGroupByAndOrderByItems(); // [2] 如果排序一致，即各分片已经排序好结果，就不需要全部加载
        selectStatement.getLimit().processParameters(parameters, !isSingleRouting, isNeedFetchAll);
    }
    // Limit.java
    /**
     * 填充改写分页参数.
     * @param parameters 参数
     * @param isRewrite 是否重写参数
     * @param isFetchAll 是否获取所有数据
     */
    public void processParameters(final List<Object> parameters, final boolean isRewrite, final boolean isFetchAll) {
        fill(parameters);
        if (isRewrite) {
            rewrite(parameters, isFetchAll);
        }
    }
    /**
     * 将占位符参数里是分页的参数赋值给 offset 、rowCount
     * 赋值的前提条件是 offset、rowCount 是 占位符
     * @param parameters 占位符参数
     */
    private void fill(final List<Object> parameters) {
        int offset = 0;
        if (null != this.offset) {
            offset = -1 == this.offset.getIndex() ? getOffsetValue() : NumberUtil.roundHalfUp(parameters.get(this.offset.getIndex()));
            this.offset.setValue(offset);
        }
        int rowCount = 0;
        if (null != this.rowCount) {
            rowCount = -1 == this.rowCount.getIndex() ? getRowCountValue() : NumberUtil.roundHalfUp(parameters.get(this.rowCount.getIndex()));
            this.rowCount.setValue(rowCount);
        }
        if (offset < 0 || rowCount < 0) {
            throw new SQLParsingException("LIMIT offset and row count can not be a negative value.");
        }
    }
```

```
/**
* 重写分页条件对应的参数
* @param parameters 参数
* @param isFetchAll 是否拉取所有
*/
private void rewrite(final List<Object> parameters, final boolean isFetchAll) {
    int rewriteOffset = 0;
    int rewriteRowCount;
    // 重写
    if (isFetchAll) {
        rewriteRowCount = Integer.MAX_VALUE;
    } else if (rowCountRewriteFlag) {
        rewriteRowCount = null == rowCount ? -1 : getOffsetValue() + rowCount.getValue();
    } else {
        rewriteRowCount = rowCount.getValue();
    }
    // 参数设置
    if (null != offset && offset.getIndex() > -1) {
        parameters.set(offset.getIndex(), rewriteOffset);
    }
    if (null != rowCount && rowCount.getIndex() > -1) {
        parameters.set(rowCount.getIndex(), rewriteRowCount);
    }
}
```

3.5 OrderByToken

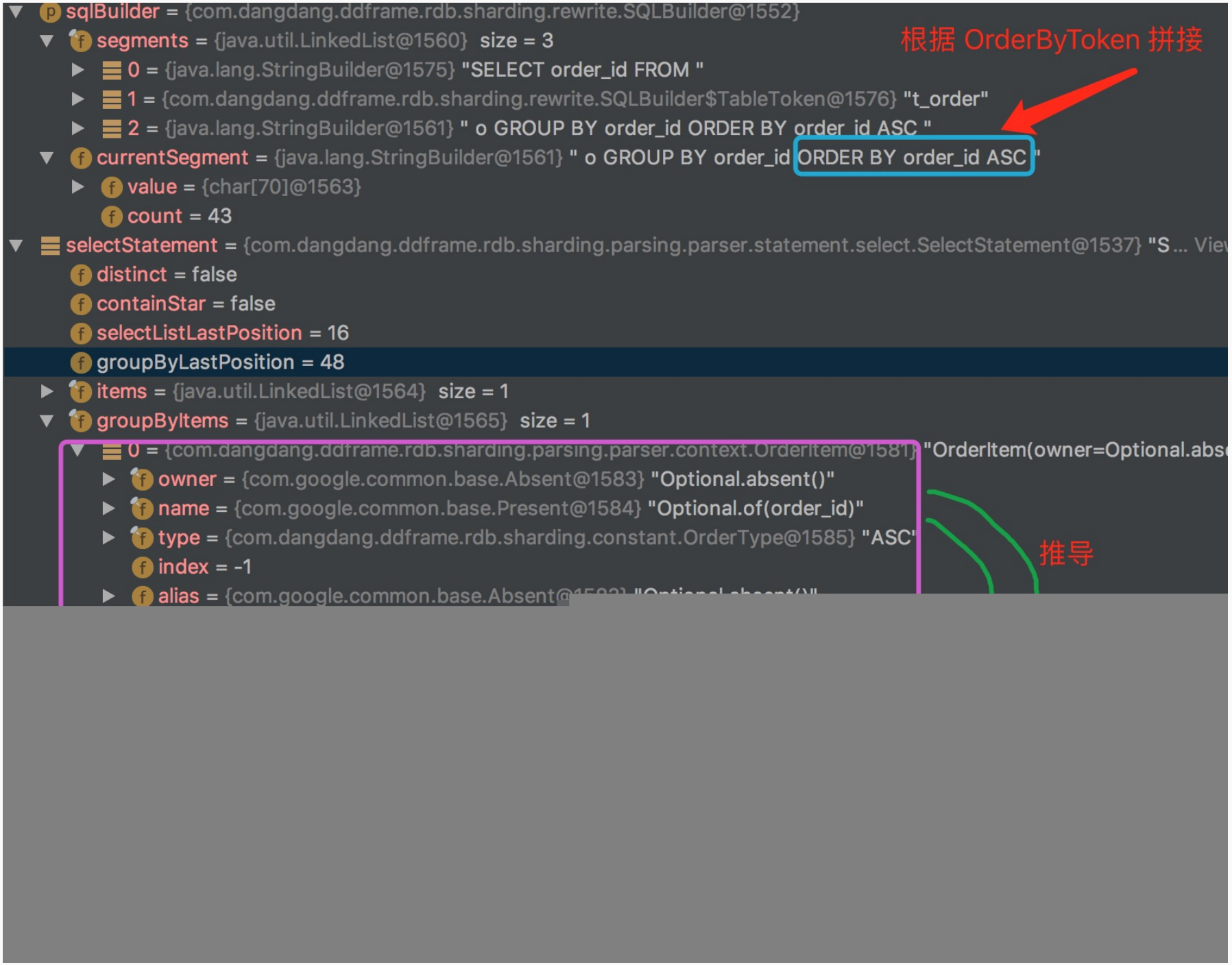
调用 `#appendOrderByToken()` 方法拼接。数据库里，当无 ORDER BY条件 而有 GROUP BY 条件时候，会使用 GROUP BY条件将结果升序排序：

- `SELECT order_id FROM t_order GROUP BY order_id` 等价于 `SELECT order_id FROM t_order GROUP BY order_id ORDER BY order_id ASC`
- `SELECT order_id FROM t_order GROUP BY order_id DESC` 等价于 `SELECT order_id FROM t_order GROUP BY order_id ORDER BY order_id DESC`

```
// ParsingSQLRouter.java
/**
* 拼接 OrderByToken
*
* @param sqlBuilder SQL构建器
*/
private void appendOrderByToken(final SQLBuilder sqlBuilder) {
    SelectStatement selectStatement = (SelectStatement) sqlStatement;
    // 拼接 OrderByToken
    StringBuilder orderByLiterals = new StringBuilder(" ORDER BY ");
    int i = 0;
```

```
    for (OrderItem each : selectStatement.getOrderByItems()) {
        if (0 == i) {
            orderByLiterals.append(each.getColumnLabel()).append(" ").append(each.getType().name());
        } else {
            orderByLiterals.append(",").append(each.getColumnLabel()).append(" ").append(each.getType().name());
        }
        i++;
    }
    orderByLiterals.append(" ");
    sqlBuilder.appendLiterals(orderByLiterals.toString());
}
```

- 当 SQL 为 `SELECT order_id FROM t_order o GROUP BY order_id` 返回结果：


sqlBuilder = {com.dangdang.ddframe.rdb.sharding.rewrite.SQLBuilder@1552}
segments = {java.util.LinkedList@1560} size = 3
0 = {java.lang.StringBuilder@1575} "SELECT order_id FROM "
1 = {com.dangdang.ddframe.rdb.sharding.rewrite.SQLBuilder$TableToken@1576} "t_order"
2 = {java.lang.StringBuilder@1561} " o GROUP BY order_id ORDER BY order_id ASC "
currentSegment = {java.lang.StringBuilder@1561} " o GROUP BY order_id ORDER BY order_id ASC "
value = {char[70]@1563}
count = 43
selectStatement = {com.dangdang.ddframe.rdb.sharding.parsing.parser.statement.select.SelectStatement@1537} "S... Vie
distinct = false
containStar = false
selectListLastPosition = 16
groupByLastPosition = 48
items = {java.util.LinkedList@1564} size = 1
groupByItems = {java.util.LinkedList@1565} size = 1
0 = {com.dangdang.ddframe.rdb.sharding.parsing.parser.context.OrderItem@1581} "OrderItem(owner=Optional.abs
owner = {com.google.common.base.Absent@1583} "Optional.absent()"
name = {com.google.common.base.Present@1584} "Optional.of(order_id)"
type = {com.dangdang.ddframe.rdb.sharding.constant.OrderType@1585} "ASC"
index = -1
alias = {com.google.common.base.Absent@
根据 OrderByToken 拼接
推导

3.6 GeneratedKeyToken

前置阅读：《SQL 解析（四）之插入SQL》

GeneratedKeyToken，和其它 SQLToken 不同，在 **SQL解析** 完进行处理。

```
// ParsingSQLRouter.java
@Override
public SQLStatement parse(final String logicSQL, final int parametersSize) {
    SQLParsingEngine parsingEngine = new SQLParsingEngine(databaseType, logicSQL, shardingRule);
    Context context = MetricsContext.start("Parse SQL");
    SQLStatement result = parsingEngine.parse();
    if (result instanceof InsertStatement) { // 处理 GenerateKeyToken
        ((InsertStatement) result).appendGenerateKeyToken(shardingRule, parametersSize);
    }
    MetricsContext.stop(context);
    return result;
}
// InsertStatement.java
/**
* 追加自增主键标记对象.
*
* @param shardingRule 分片规则
* @param parametersSize 参数个数
*/
public void appendGenerateKeyToken(final ShardingRule shardingRule, final int parametersSize) {
    // SQL 里有主键列
    if (null != generatedKey) {
        return;
    }
    // TableRule 存在
    Optional<TableRule> tableRule = shardingRule.tryFindTableRule(getTables().getSingleTableName());
    if (!tableRule.isPresent()) {
        return;
    }
    // GeneratedKeyToken 存在
    Optional<GeneratedKeyToken> generatedKeysToken = findGeneratedKeyToken();
    if (!generatedKeysToken.isPresent()) {
        return;
    }
    // 处理 GenerateKeyToken
    ItemsToken valuesToken = new ItemsToken(generatedKeysToken.get().getBeginPosition());
    if (0 == parametersSize) {
```

```
            appendGenerateKeyToken(shardingRule, tableRule.get(), valuesToken);
        } else {
            appendGenerateKeyToken(shardingRule, tableRule.get(), valuesToken, parametersSize);
        }
        // 移除 generatedKeysToken
        getSqlTokens().remove(generatedKeysToken.get());
        // 新增 ItemsToken
        getSqlTokens().add(valuesToken);
    }
```

- 根据**占位符参数**数量不同，调用的 `#appendGenerateKeyToken()` 是**不同**的：
- **占位符参数数量 = 0** 时，直接生成**分布式主键**，保持无占位符的做法。

```
// InsertStatement.java
private void appendGenerateKeyToken(final ShardingRule shardingRule, final TableRule tableRule, final ItemsToken valuesToken) {
    // 生成分布式主键
    Number generatedKey = shardingRule.generateKey(tableRule.getLogicTable());
    // 添加到 ItemsToken
    valuesToken.getItems().add(generatedKey.toString());
    // 增加 Condition，用于路由
    getConditions().add(new Condition(new Column(tableRule.getGenerateKeyColumn(), tableRule.getLogicTable()), new SQLNumberExpression(generatedKey)), sha
    // 生成 GeneratedKey
    this.generatedKey = new GeneratedKey(tableRule.getLogicTable(), -1, generatedKey);
}
```

- **占位符参数数量 > 0** 时，生成自增列的占位符，保持有占位符的做法。

```
private void appendGenerateKeyToken(final ShardingRule shardingRule, final TableRule tableRule, final ItemsToken valuesToken, final int parametersSize) {
    // 生成占位符
    valuesToken.getItems().add("?");
    // 增加 Condition，用于路由
    getConditions().add(new Condition(new Column(tableRule.getGenerateKeyColumn(), tableRule.getLogicTable()), new SQLPlaceholderExpression(parametersSize
    // 生成 GeneratedKey
    generatedKey = new GeneratedKey(tableRule.getGenerateKeyColumn(), parametersSize, null);
}
```

- 因为 GenerateKeyToken 已经处理完，所以移除，避免 `SQLRewriteEngine#rewrite()` 二次改写。另外，通过 ItemsToken 补充自增列。
- 生成 GeneratedKey 会在 ParsingSQLRouter 进一步处理。

```
// ParsingSQLRouter.java
```

```
public SQLRouteResult route(final String logicSQL, final List<Object> parameters, final SQLStatement sqlStatement) {
    final Context context = MetricsContext.start("Route SQL");
    SQLRouteResult result = new SQLRouteResult(sqlStatement);
    // 处理 插入SQL 主键字段
    if (sqlStatement instanceof InsertStatement && null != ((InsertStatement) sqlStatement).getGeneratedKey()) {
        processGeneratedKey(parameters, (InsertStatement) sqlStatement, result);
    }
    // ... 省略部分代码
}
/**
* 处理 插入SQL 主键字段
* 当 主键编号 未生成时，{@link ShardingRule#generateKey(String)} 进行生成
* @param parameters 占位符参数
* @param insertStatement Insert SQL语句对象
* @param sqlRouteResult SQL路由结果
*/
private void processGeneratedKey(final List<Object> parameters, final InsertStatement insertStatement, final SQLRouteResult sqlRouteResult) {
    GeneratedKey generatedKey = insertStatement.getGeneratedKey();
    if (parameters.isEmpty()) { // 已有主键，无占位符，INSERT INTO t_order(order_id, user_id) VALUES (1, 100);
        sqlRouteResult.getGeneratedKeys().add(generatedKey.getValue());
    } else if (parameters.size() == generatedKey.getIndex()) { // 主键字段不存在存在，INSERT INTO t_order(user_id) VALUES(?);
        Number key = shardingRule.generateKey(insertStatement.getTables().getSingleTableName()); // 生成主键编号
        parameters.add(key);
        setGeneratedKeys(sqlRouteResult, key);

    } else if (-1 != generatedKey.getIndex()) { // 主键字段存在，INSERT INTO t_order(order_id, user_id) VALUES(?, ?);
        setGeneratedKeys(sqlRouteResult, (Number) parameters.get(generatedKey.getIndex()));
    }
}
/**
* 设置 主键编号 到 SQL路由结果
* @param sqlRouteResult SQL路由结果
* @param generatedKey 主键编号
*/
private void setGeneratedKeys(final SQLRouteResult sqlRouteResult, final Number generatedKey) {
    generatedKeys.add(generatedKey);
    sqlRouteResult.getGeneratedKeys().clear();
    sqlRouteResult.getGeneratedKeys().addAll(generatedKeys);
}
```

- `parameters.size() == generatedKey.getIndex()` 处对应 `#appendGenerateKeyToken()` 的 **占位符参数数量 > 0** 情况，此时会生成**分布式主键**。😺 该处是不是可以考虑把生成**分布式主键**挪到 `#appendGenerateKeyToken()`，这样更加统一一些。

4. SQL 生成

SQL路由完后，会生成各数据分片的**执行SQL**。

```
// ParsingSQLRouter.java
@Override
public SQLRouteResult route(final String logicSQL, final List<Object> parameters, final SQLStatement sqlStatement) {
    SQLRouteResult result = new SQLRouteResult(sqlStatement);
    // 省略部分代码... 处理 插入SQL 主键字段

    // 路由
    RoutingResult routingResult = route(parameters, sqlStatement);

    // 省略部分代码... SQL重写引擎
    SQLRewriteEngine rewriteEngine = new SQLRewriteEngine(shardingRule, logicSQL, sqlStatement);
    boolean isSingleRouting = routingResult.isSingleRouting();
    // 省略部分代码... 处理分页
    // SQL 重写
    SQLBuilder sqlBuilder = rewriteEngine.rewrite(!isSingleRouting);
    // 生成 ExecutionUnit
    if (routingResult instanceof CartesianRoutingResult) {
        for (CartesianDataSource cartesianDataSource : ((CartesianRoutingResult) routingResult).getRoutingDataSources()) {
            for (CartesianTableReference cartesianTableReference : cartesianDataSource.getRoutingTableReferences()) {
                // 生成 SQL
                result.getExecutionUnits().add(new SQLExecutionUnit(cartesianDataSource.getDataSource(), rewriteEngine.generateSQL(cartesianTableReference
            }
        }
    } else {
        for (TableUnit each : routingResult.getTableUnits().getTableUnits()) {
            // 生成 SQL
            result.getExecutionUnits().add(new SQLExecutionUnit(each.getDataSourceName(), rewriteEngine.generateSQL(each, sqlBuilder)));
        }
    }
    return result;
}
```

- 调用 `RewriteEngine#generateSQL()` 生成**执行SQL**。对于笛卡尔积路由结果和简单路由结果传递的参数略有不同：前者使用 CartesianDataSource (CartesianTableReference)，后者使用路由表单元 (TableUnit)。对路由结果不是很了解的同学，建议看下 《SQL 路由 (二) 之分库分表路由》。

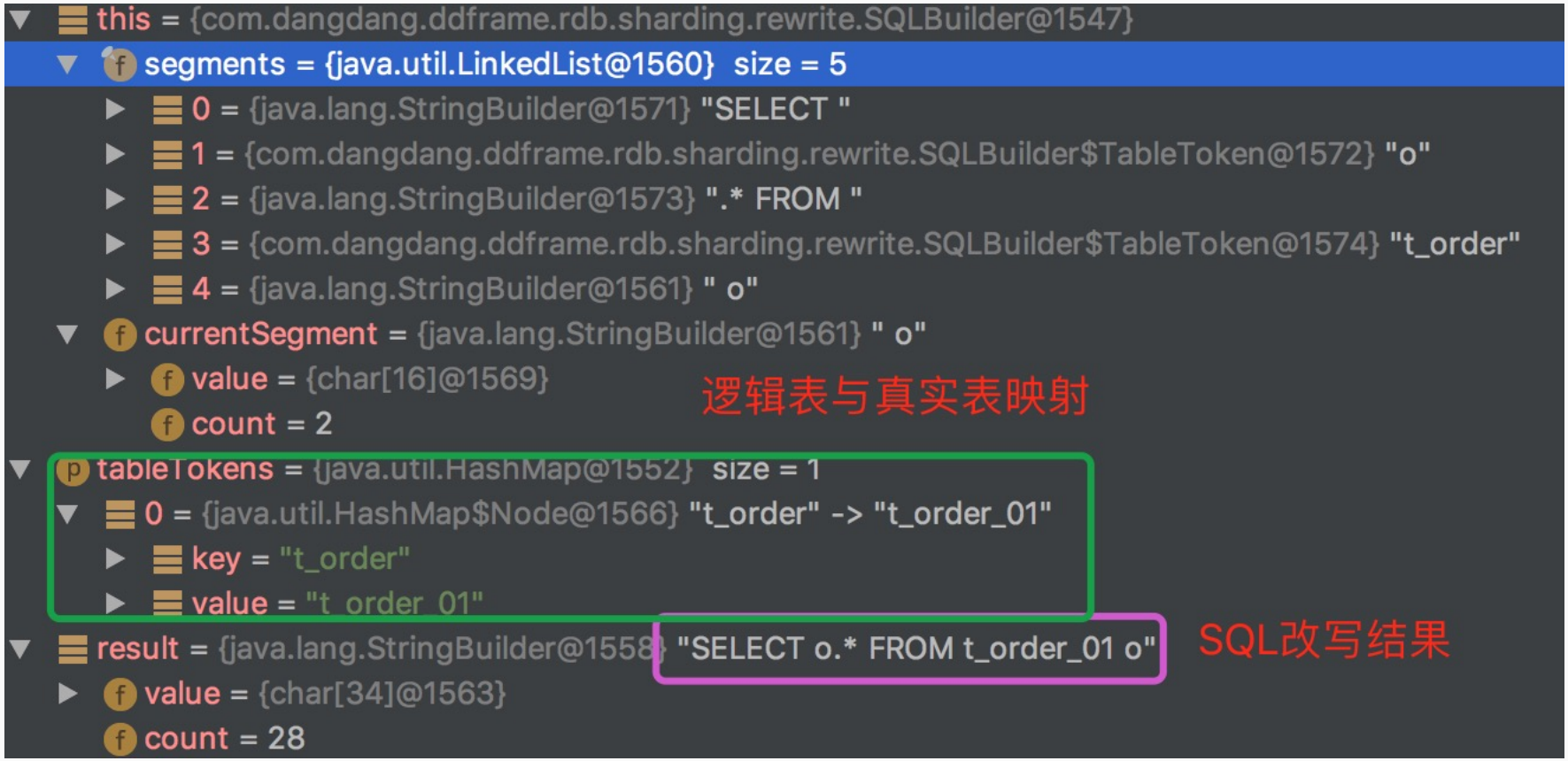
`RewriteEngine#generateSQL()` 对于笛卡尔积路由结果和简单路由结果两种情况，处理上大体是一致的：1. 获得 SQL 相关**逻辑表**对应的**真实表**映射，2. 根据映射改写 SQL 相关**逻辑表**为**真实表**。

```
// SQLRewriteEngine.java
/**
* 生成SQL语句.
```

```
 * @param tableUnit 路由表单元
 * @param sqlBuilder SQL构建器
 * @return SQL语句
 */
public String generateSQL(final TableUnit tableUnit, final SQLBuilder sqlBuilder) {
    return sqlBuilder.toSQL(getTableTokens(tableUnit));
}
/**
 * 生成SQL语句.
 * @param cartesianTableReference 笛卡尔积路由表单元
 * @param sqlBuilder SQL构建器
 * @return SQL语句
 */
public String generateSQL(final CartesianTableReference cartesianTableReference, final SQLBuilder sqlBuilder) {
    return sqlBuilder.toSQL(getTableTokens(cartesianTableReference));
}
// SQLRewriteEngine.java
// SQLBuilder.java
/**
 * 生成SQL语句.
 * @param tableTokens 占位符集合（逻辑表与真实表映射）
 * @return SQL语句
 */
public String toSQL(final Map<String, String> tableTokens) {
    StringBuilder result = new StringBuilder();

    for (Object each : segments) {
        if (each instanceof TableToken && tableTokens.containsKey(((TableToken) each).tableName)) {
            result.append(tableTokens.get(((TableToken) each).tableName));
        } else {
            result.append(each);
        }
    }
    return result.toString();
}
```

- `#toSQL()` 结果如图：



㊗ 对 **SQL改写** 是不是清晰很多了。

下面我们以**笛卡尔积路由结果**获得 SQL 相关**逻辑表**对应的**真实表**映射为例子(*简单路由结果基本类似而且简单*)。

```
// SQLRewriteEngine.java
/**
* 获得（笛卡尔积表路由组里的路由表单元逻辑表 和 与其互为BindingTable关系的逻辑表）对应的真实表映射（逻辑表需要在 SQL 中存在）
* @param cartesianTableReference 笛卡尔积表路由组

* @return 集合
*/
private Map<String, String> getTableTokens(final CartesianTableReference cartesianTableReference) {
   Map<String, String> tableTokens = new HashMap<>();
   for (TableUnit each : cartesianTableReference.getTableUnits()) {
       tableTokens.put(each.getLogicTableName(), each.getActualTableName());
       // 查找 BindingTableRule
       Optional<BindingTableRule> bindingTableRule = shardingRule.findBindingTableRule(each.getLogicTableName());
       if (bindingTableRule.isPresent()) {
```

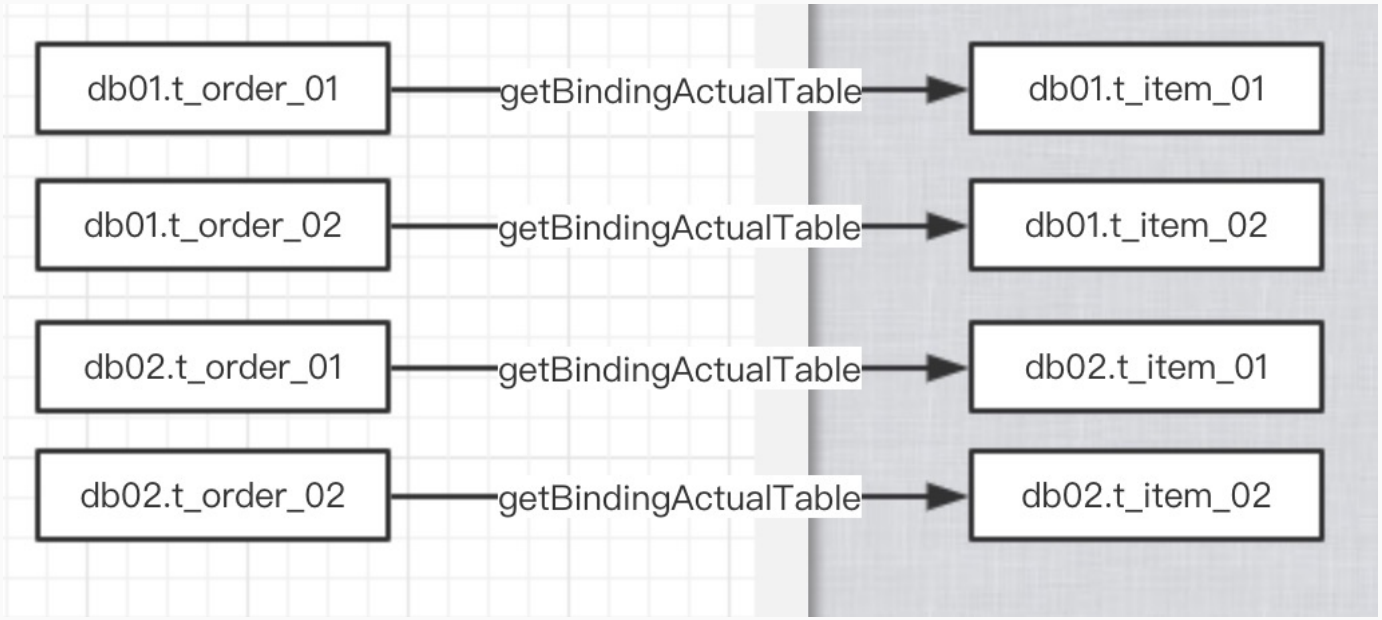
```
            tableTokens.putAll(getBindingTableTokens(each, bindingTableRule.get()));
        }
    }
    return tableTokens;
}
/**
* 获得 BindingTable 关系的逻辑表对应的真实表映射（逻辑表需要在 SQL 中存在）
* @param tableUnit 路由单元
* @param bindingTableRule Binding表规则配置对象
* @return 映射
*/
private Map<String, String> getBindingTableTokens(final TableUnit tableUnit, final BindingTableRule bindingTableRule) {
    Map<String, String> result = new HashMap<>();
    for (String eachTable : sqlStatement.getTables().getTableNames()) {
        if (!eachTable.equalsIgnoreCase(tableUnit.getLogicTableName()) && bindingTableRule.hasLogicTable(eachTable)) {
            result.put(eachTable, bindingTableRule.getBindingActualTable(tableUnit.getDataSourceName(), eachTable, tableUnit.getActualTableName()));
        }
    }
    return result;
}
```

- 笛卡尔积表路由组(CartesianTableReference)包含**多个**路由表单元(TableUnit)。每个路由表单元需要遍历。
- 路由表单元本身包含逻辑表和真实表，直接添加到映射即可。
- 互为 BindingTable 关系的表只计算一次路由分片，因此**未计算**的真实表需要以其对应的**已计算**的真实表去查找，即 `bindingTableRule.getBindingActualTable(tableUnit.getDataSourceName(), eachTable, tableUnit.getActualTableName())` 处逻辑。

```
// BindingTableRule.java
/**
* 根据其他Binding表真实表名称获取相应的真实Binding表名称.
*
* @param dataSource 数据源名称
* @param logicTable 逻辑表名称
* @param otherActualTable 其他真实Binding表名称
* @return 真实Binding表名称
*/
public String getBindingActualTable(final String dataSource, final String logicTable, final String otherActualTable) {
    // 计算 otherActualTable 在其 TableRule 的 actualTable 是第几个
    int index = -1;
    for (TableRule each : tableRules) {
        if (each.isDynamic()) {
            throw new UnsupportedOperationException("Dynamic table cannot support Binding table.");
        }
```

```
            index = each.findActualTableIndex(dataSource, otherActualTable);
            if (-1 != index) {
                break;
            }
        }
        Preconditions.checkState(-1 != index, String.format("Actual table [%s].[%s] is not in table config", dataSource, otherActualTable));
        // 计算 logicTable 在其 TableRule 的 第index 的 真实表
        for (TableRule each : tableRules) {
            if (each.getLogicTable().equalsIgnoreCase(logicTable)) {
                return each.getActualTables().get(index).getTableName();
            }
        }
        throw new IllegalStateException(String.format("Cannot find binding actual table, data source: %s, logic table: %s, other actual table: %s", dataSource
    }
```

可能看起来有些绕，我们看张图：



友情提示：这里不嫌啰嗦在提一句，互为 BindingTable 的表，配置 TableRule 时，`actualTables` 数量一定要一致，否则多出来的表，可能会无法被路由到。

666. 彩蛋

哈哈哈，看完**SQL改写**后，**SQL解析**是不是清晰多了！嘿嘿嘿，反正我现在有点嗨。恩，蛮嗨的。

当然，如果**SQL解析**理解上有点疑惑的你，**欢迎**加我的微信，咱 **1对1** 搞基。关注我的微信公众号：【芋艿的后端小屋】 即可获得。

道友，转发一波朋友圈可好？

Let's Go! 《分布式主键》、《SQL 执行》、《结果聚合》 继续。

感谢技术牛逼如你耐心的阅读本文。

Sharding-JDBC

PREVIOUS:
« Sharding-JDBC 源码分析 —— 分布式主键
NEXT:
» Sharding-JDBC 源码分析 —— SQL 路由（三）之Spring与YAML配置