芋艿v的博客

愿编码半生，如老友相伴！


扫一扫二维码关注公众号
关注后，可以看到
「RocketMQ」
「MyCAT」
所有源码解析文章
— 近期更新「Sharding-JDBC」中 —
你有233个小伙伴已经关注

分享

分类

Docker 2

MyCAT 9

Nginx 1

RocketMQ 14

Sharding-JDBC 17

技术杂文 2

分享

Sharding-JDBC 源码分析 —— SQL 路由（二）之分库分表路由

2017-08-06 更新日期:2017-08-02 总阅读量:18次

文章目录

> □□□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

分享

本文分享分表分库**路由**相关的实现。涉及内容如下：

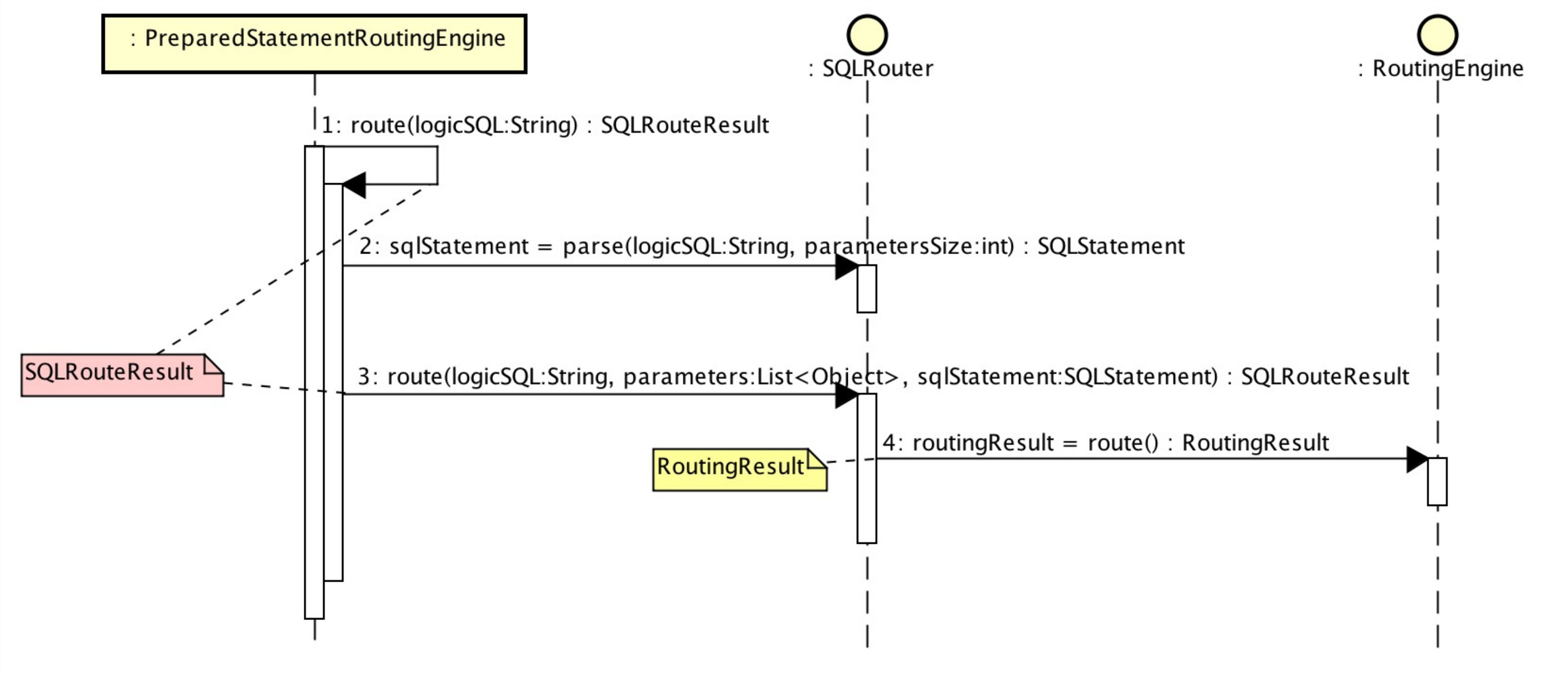
1. SQL 路由结果
2. 路由策略 x 算法
3. SQL 路由器

内容顺序如编号。

> **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**

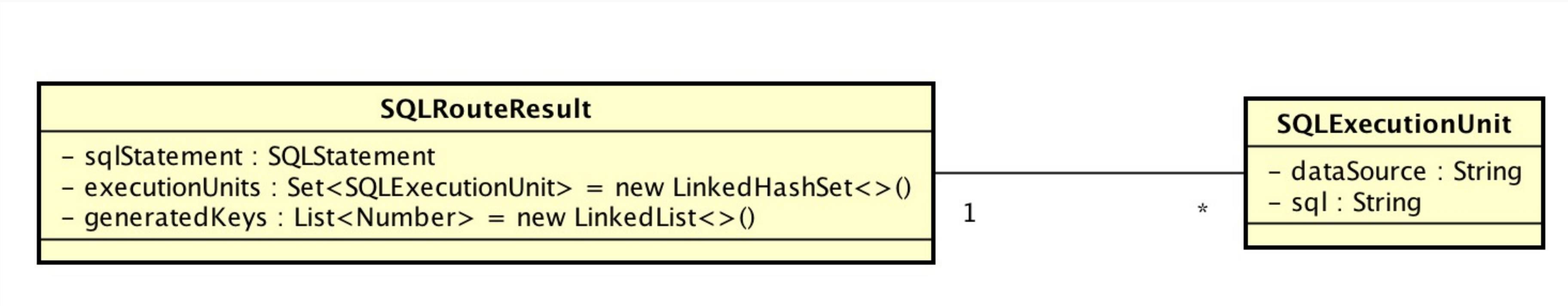
SQL 路由大体流程如下：

分
享



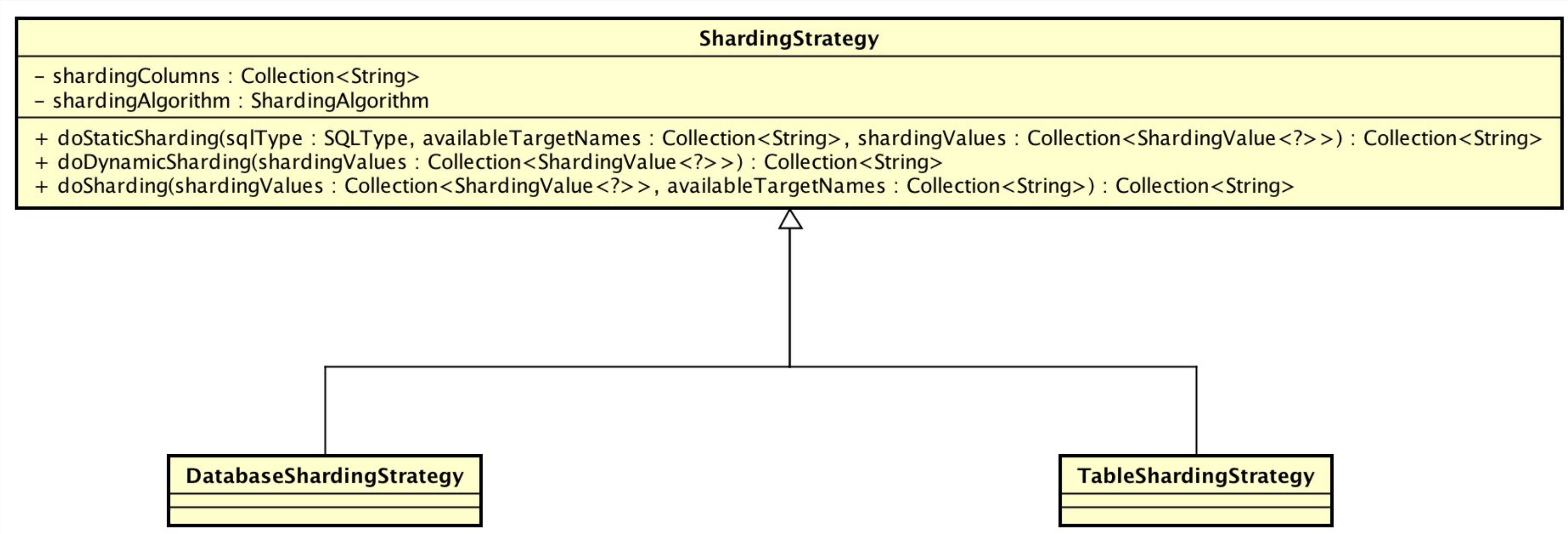
2. SQLRouteResult

经过 **SQL解析**、**SQL路由**后，产生**SQL路由结果**，即 SQLRouteResult。根据路由结果，**生成SQL**，**执行SQL**。



- `sqlStatement` ：SQL语句对象，经过**SQL解析**的结果对象。
- `executionUnits` ：SQL最小执行单元集合。**SQL执行**时，执行每个单元。
- `generatedKeys` ：**插入**SQL语句生成的主键编号集合。目前不支持批量插入而使用集合的原因，猜测是为了未来支持批量插入做准备。

3. 路由策略 x 算法



ShardingStrategy，分片策略。目前支持两种分片：

分片资源：在分库策略里指的是库，在分表策略里指的是表。

【1】 计算**静态**分片（常用）

```
// ShardingStrategy.java
```

```
/**
* 计算静态分片.
* @param sqlType SQL语句的类型
* @param availableTargetNames 所有的可用分片资源集合
* @param shardingValues 分片值集合
* @return 分库后指向的数据源名称集合
*/
public Collection<String> doStaticSharding(final SQLType sqlType, final Collection<String> availableTa
    Collection<String> result = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
    if (shardingValues.isEmpty()) {
        Preconditions.checkState(!isInsertMultiple(sqlType, availableTargetNames), "INSERT statement sh
        result.addAll(availableTargetNames);
    } else {
        result.addAll(doSharding(shardingValues, availableTargetNames));
    }
    return result;
}
/**
* 插入SQL 是否插入多个分片
* @param sqlType SQL类型
* @param availableTargetNames 所有的可用分片资源集合
* @return 是否
*/
private boolean isInsertMultiple(final SQLType sqlType, final Collection<String> availableTargetNames)
    return SQLType.INSERT == sqlType && availableTargetNames.size() > 1;
}
```

- 插入SQL 需要有片键值，否则无法判断单个分片资源。*（Sharding-JDBC 目前仅支持单条记录插入）*

分享

【2】计算**动态**分片

```
// ShardingStrategy.java
/**
* 计算动态分片.
* @param shardingValues 分片值集合
* @return 分库后指向的分片资源集合
*/
public Collection<String> doDynamicSharding(final Collection<ShardingValue<?>> shardingValues) {
   Preconditions.checkState(!shardingValues.isEmpty(), "Dynamic table should contain sharding value.")
   Collection<String> availableTargetNames = Collections.emptyList();
   Collection<String> result = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
   result.addAll(doSharding(shardingValues, availableTargetNames));
   return result;
}
```

- 动态分片对应 `TableRule.dynamic=true`
- 动态分片必须有分片值

😼 闷了，看起来两者没啥区别？答案在**分片算法**上。我们先看 `#doSharding()` 方法的实现。

```
// ShardingStrategy.java
/**
* 计算分片
* @param shardingValues 分片值集合
* @param availableTargetNames 所有的可用分片资源集合
* @return 分库后指向的分片资源集合
```

分享

```
*/
private Collection<String> doSharding(final Collection<ShardingValue<?>> shardingValues, final Collect
    // 无片键
    if (shardingAlgorithm instanceof NoneKeyShardingAlgorithm) {
        return Collections.singletonList(((NoneKeyShardingAlgorithm) shardingAlgorithm).doSharding(avai
    }
    // 单片键
    if (shardingAlgorithm instanceof SingleKeyShardingAlgorithm) {
        SingleKeyShardingAlgorithm<?> singleKeyShardingAlgorithm = (SingleKeyShardingAlgorithm<?>) shar
        ShardingValue shardingValue = shardingValues.iterator().next();
        switch (shardingValue.getType()) {
            case SINGLE:
                return Collections.singletonList(singleKeyShardingAlgorithm.doEqualSharding(availableTa
            case LIST:
                return singleKeyShardingAlgorithm.doInSharding(availableTargetNames, shardingValue);
            case RANGE:
                return singleKeyShardingAlgorithm.doBetweenSharding(availableTargetNames, shardingValue
            default:
                throw new UnsupportedOperationException(shardingValue.getType().getClass().getName());
        }
    }
    // 多片键
    if (shardingAlgorithm instanceof MultipleKeysShardingAlgorithm) {
        return ((MultipleKeysShardingAlgorithm) shardingAlgorithm).doSharding(availableTargetNames, sha
    }
    throw new UnsupportedOperationException(shardingAlgorithm.getClass().getName());
}
```

分
享

- 无分片键算法：对应 NoneKeyShardingAlgorithm 分片算法接口。

```
public interface NoneKeyShardingAlgorithm<T extends Comparable<?>> extends ShardingAlgorithm {
    String doSharding(Collection<String> availableTargetNames, ShardingValue<T> shardingValue);
}
```

- 单片键算法：对应 SingleKeyShardingAlgorithm 分片算法接口。

```
public interface SingleKeyShardingAlgorithm<T extends Comparable<?>> extends ShardingAlgorithm {
    String doEqualSharding(Collection<String> availableTargetNames, ShardingValue<T> shardingValue);
    Collection<String> doInSharding(Collection<String> availableTargetNames, ShardingValue<T> sharding
    Collection<String> doBetweenSharding(Collection<String> availableTargetNames, ShardingValue<T> sha
}
```

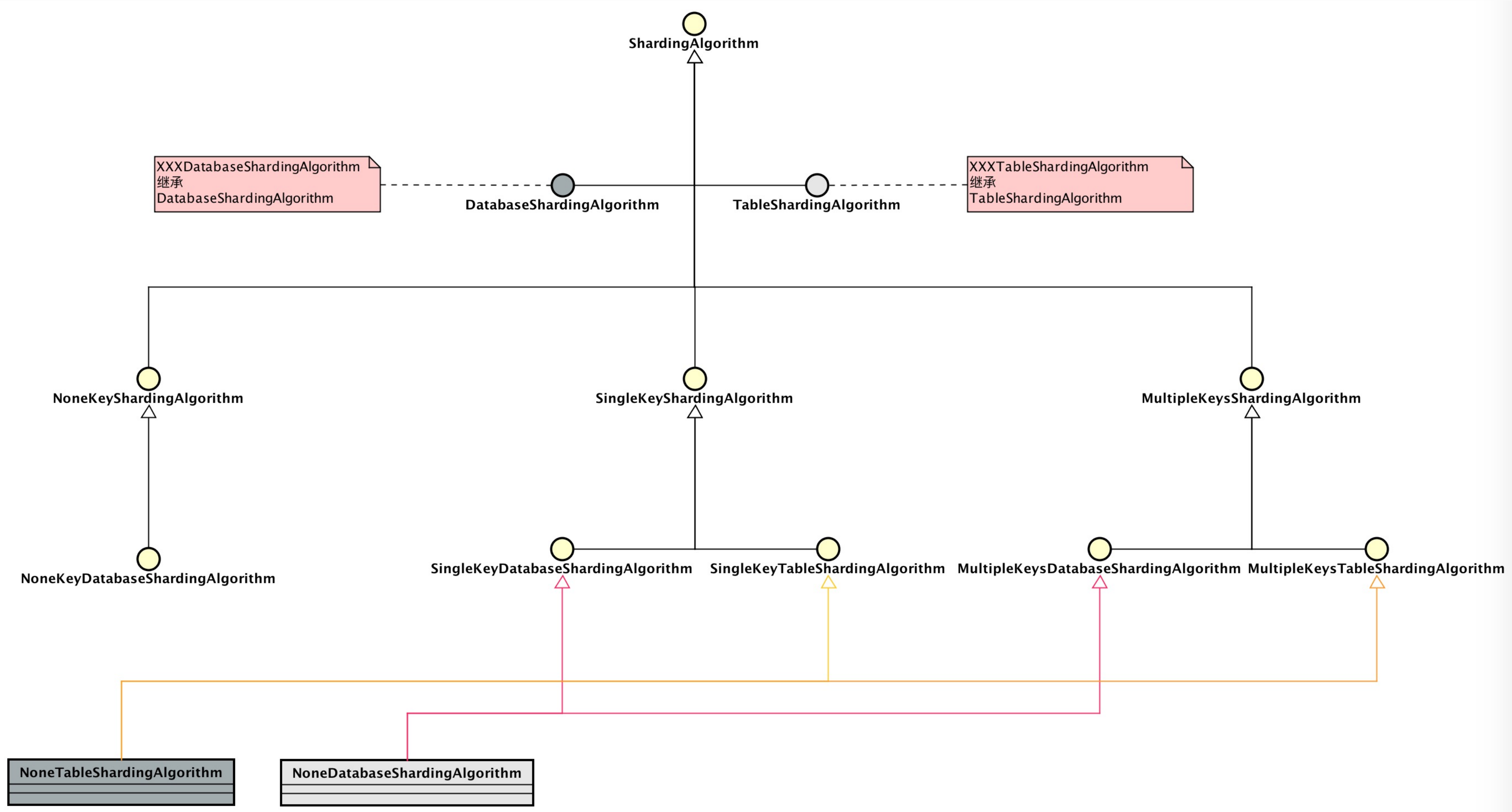
ShardingValueType	SQL 操作符	接口方法
SINGLE	=	#doEqualSharding()
LIST	IN	#doInSharding()
RANGE	BETWEEN	#doBetweenSharding()

- 多片键算法：对应 MultipleKeysShardingAlgorithm 分片算法接口。

```
public interface MultipleKeysShardingAlgorithm extends ShardingAlgorithm {
    Collection<String> doSharding(Collection<String> availableTargetNames, Collection<ShardingValue<?>
}
```

分享

分片算法类结构如下：



来看看 Sharding-JDBC 实现的无需分库的分片算法 NoneDatabaseShardingAlgorithm (NoneTableShardingAlgorithm 基本一模一样)：

```
public final class NoneDatabaseShardingAlgorithm implements SingleKeyDatabaseShardingAlgorithm<String>
    @Override
```

```
    public Collection<String> doSharding(final Collection<String> availableTargetNames, final Collecti
        return availableTargetNames;
    }
    @Override
    public String doEqualSharding(final Collection<String> availableTargetNames, final ShardingValue<S
        return availableTargetNames.isEmpty() ? null : availableTargetNames.iterator().next();
    }
    @Override
    public Collection<String> doInSharding(final Collection<String> availableTargetNames, final Shardi
        return availableTargetNames;
    }
    @Override
    public Collection<String> doBetweenSharding(final Collection<String> availableTargetNames, final S
        return availableTargetNames;
    }
}
```

- **一定要注意，NoneXXXXShardingAlgorithm 只适用于无分库/表的需求，否则会是错误的路由结果。**例如，`#doEqualSharding()` 返回的是第一个分片资源。

再来看测试目录下实现的**余数基偶分表算法** ModuloTableShardingAlgorithm 的实现：

```
// com.dangdang.ddframe.rdb.integrate.fixture.ModuloTableShardingAlgorithm.java
public final class ModuloTableShardingAlgorithm implements SingleKeyTableShardingAlgorithm<Integer> {
    @Override
    public String doEqualSharding(final Collection<String> tableNames, final ShardingValue<Integer> sh
        for (String each : tableNames) {
            if (each.endsWith(shardingValue.getValue() % 2 + "")) {
```

```
                return each;
            }
        }
        throw new UnsupportedOperationException();
    }
    @Override
    public Collection<String> doInSharding(final Collection<String> tableNames, final ShardingValue<In
        Collection<String> result = new LinkedHashSet<>(tableNames.size());
        for (Integer value : shardingValue.getValues()) {
            for (String tableName : tableNames) {
                if (tableName.endsWith(value % 2 + "")) {
                    result.add(tableName);
                }
            }
        }
        return result;
    }
    @Override
    public Collection<String> doBetweenSharding(final Collection<String> tableNames, final ShardingVal
        Collection<String> result = new LinkedHashSet<>(tableNames.size());
        Range<Integer> range = shardingValue.getValueRange();
        for (Integer i = range.lowerEndpoint(); i <= range.upperEndpoint(); i++) {
            for (String each : tableNames) {
                if (each.endsWith(i % 2 + "")) {
                    result.add(each);
                }
            }
        }
        return result;
```

分享

```
    }
}
```

- 我们可以参考这个例子编写自己的分片算哟 。
- 多片键分库算法接口实现例子：MultipleKeysModuloDatabaseShardingAlgorithm.java

---

😈 来看看**动态计算分片**需要怎么实现分片算法。

```
// com.dangdang.ddframe.rdb.integrate.fixture.SingleKeyDynamicModuloTableShardingAlgorithm.java
public final class SingleKeyDynamicModuloTableShardingAlgorithm implements SingleKeyTableShardingAlgor
    /**
    * 表前缀
    */
    private final String tablePrefix;
    @Override
    public String doEqualSharding(final Collection<String> availableTargetNames, final ShardingValue<I
        return tablePrefix + shardingValue.getValue() % 10;
    }
    @Override
    public Collection<String> doInSharding(final Collection<String> availableTargetNames, final Shardi
        Collection<String> result = new LinkedHashSet<>(shardingValue.getValues().size());
        for (Integer value : shardingValue.getValues()) {
            result.add(tablePrefix + value % 10);
        }
        return result;
    }
    @Override
```

分享

```
    public Collection<String> doBetweenSharding(final Collection<String> availableTargetNames, final S
        Collection<String> result = new LinkedHashSet<>(availableTargetNames.size());
        Range<Integer> range = shardingValue.getValueRange();
        for (Integer i = range.lowerEndpoint(); i <= range.upperEndpoint(); i++) {
            result.add(tablePrefix + i % 10);
        }
        return result;
    }
}
```

- 骚年，是不是明白了一些？**动态表**无需把真实表配置到 TableRule，而是通过**分片算法**计算出**真实表**。

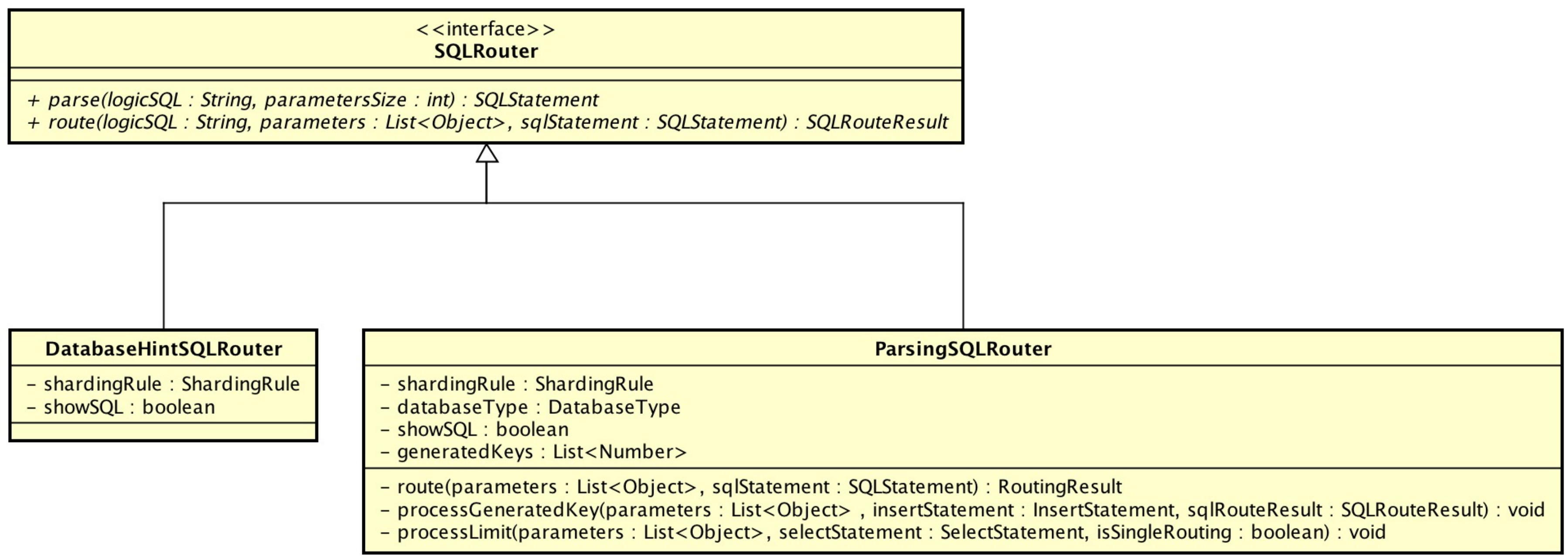
4. SQL 路由

SQLRouter，SQL 路由器接口，共有两种实现：

- DatabaseHintSQLRouter：通过提示且仅路由至数据库的SQL路由器
- ParsingSQLRouter：需要解析的SQL路由器

它们实现 `#parse()` 进行**SQL解析**，`#route()` 进行**SQL路由**。

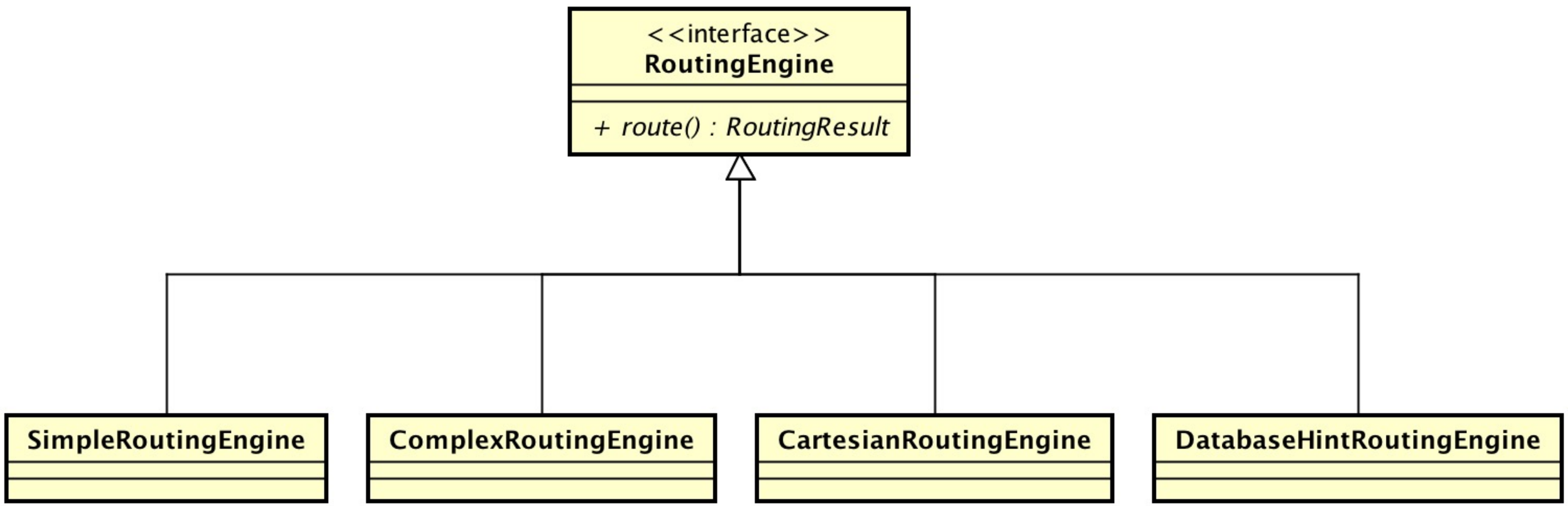
分享



RoutingEngine，路由引擎接口，共有四种实现：

- DatabaseHintRoutingEngine：基于数据库提示的路由引擎
- SimpleRoutingEngine：简单路由引擎
- CartesianRoutingEngine：笛卡尔积的库表路由
- ComplexRoutingEngine：混合多库表路由引擎

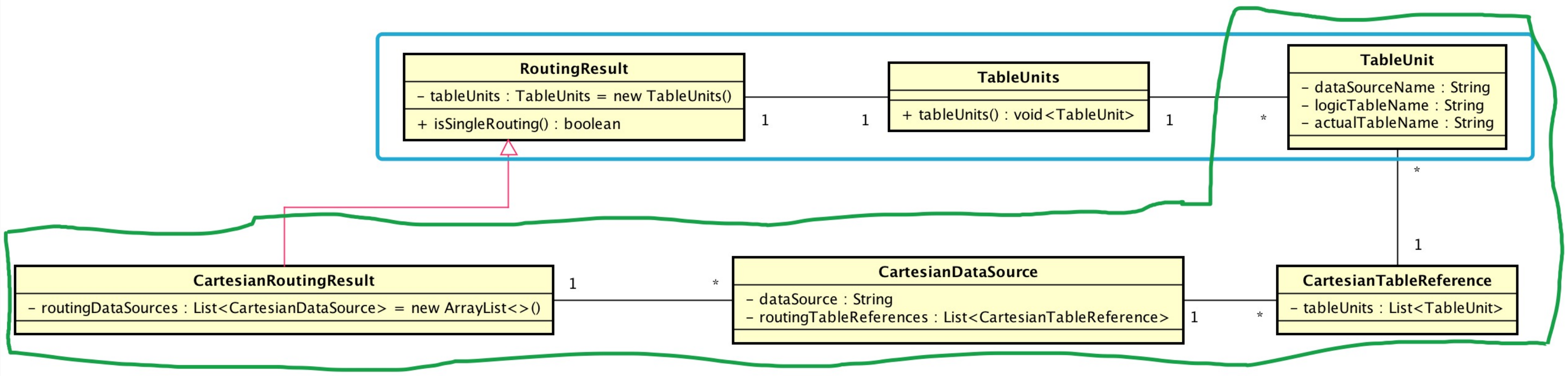
ComplexRoutingEngine 根据路由结果会转化成 SimpleRoutingEngine 或 ComplexRoutingEngine。下文会看相应源码。



路由结果有两种：

- RoutingResult：简单路由结果
- CartesianRoutingResult：笛卡尔积路由结果

分享

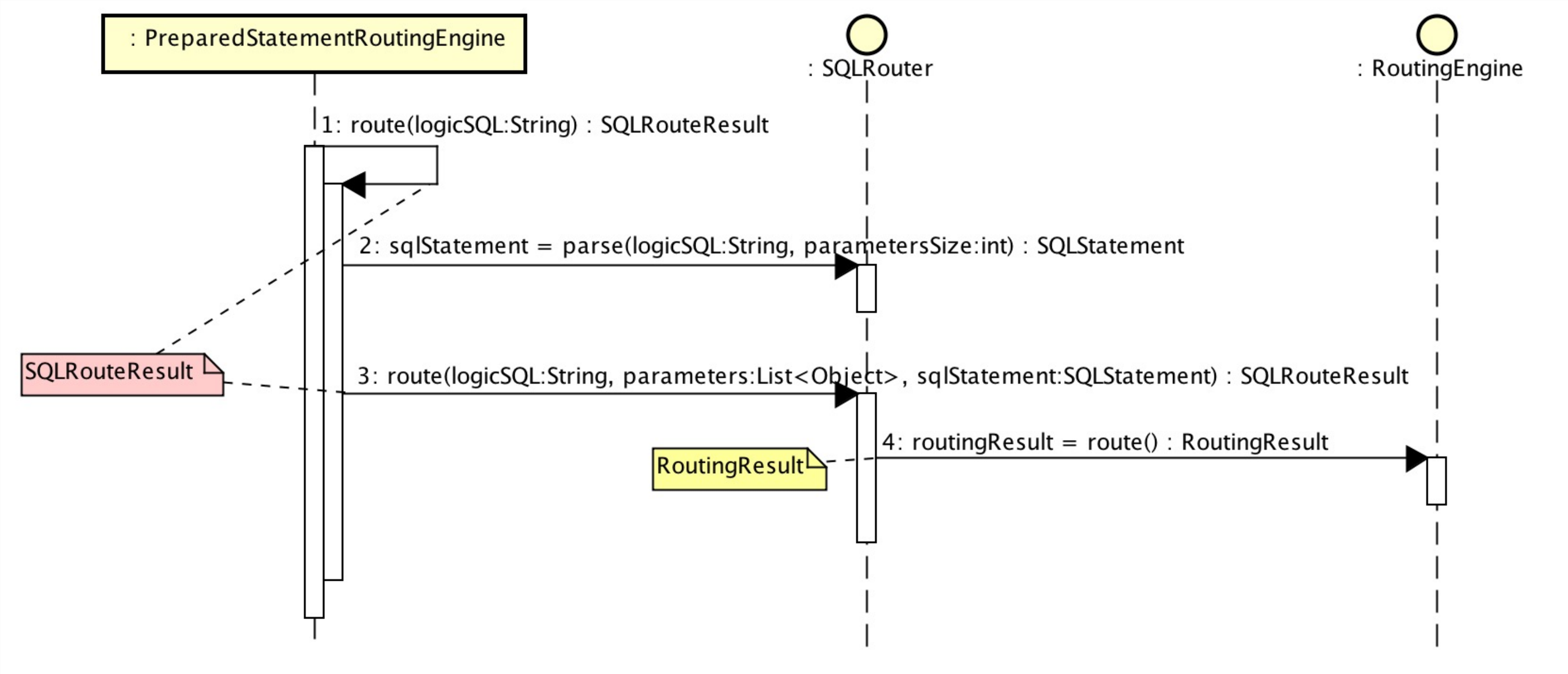


从图中，我们已经能大概看到两者有什么区别，更具体的下文随源码一起分享。

😈 SQLRouteResult 和 RoutingResult 有什么区别？

- SQLRouteResult：**整个SQL路由**返回的路由结果
- RoutingResult：**RoutingEngine**返回路由结果

分享



一下子看到这么多**"对象"，**可能有点**紧张**。不要紧张，我们一起在整理下。

路由器	路由引擎	路由结果
DatabaseHintSQLRouter	DatabaseHintRoutingEngine	RoutingResult
ParsingSQLRouter	SimpleRoutingEngine	RoutingResult
ParsingSQLRouter	CartesianRoutingEngine	CartesianRoutingResult

😈 逗比博主给大家解决了**"对象"，**是不是应该**分享朋友圈**。

分享

5. DatabaseHintSQLRouter

DatabaseHintSQLRouter，基于数据库提示的路由引擎。路由器工厂 SQLRouterFactory 创建路由器时，判断到使用数据库提示(Hint) 时，创建 DatabaseHintSQLRouter。

```
// DatabaseHintRoutingEngine.java
public static SQLRouter createSQLRouter(final ShardingContext shardingContext) {
   return HintManagerHolder.isDatabaseShardingOnly() ? new DatabaseHintSQLRouter(shardingContext) : ne
}
```

先来看下 HintManagerHolder、HintManager **部分相关**的代码：

```
// HintManagerHolder.java
public final class HintManagerHolder {
    /**
     * HintManager 线程变量
     */
    private static final ThreadLocal<HintManager> HINT_MANAGER_HOLDER = new ThreadLocal<>();
    /**
     * 判断是否当前只分库.
     *
     * @return 是否当前只分库.
     */
    public static boolean isDatabaseShardingOnly() {
        return null != HINT_MANAGER_HOLDER.get() && HINT_MANAGER_HOLDER.get().isDatabaseShardingOnly()
    }
    /**
```

```
     * 清理线索分片管理器的本地线程持有者.
     */
    public static void clear() {
        HINT_MANAGER_HOLDER.remove();
    }
}
// HintManager.java
public final class HintManager implements AutoCloseable {
    /**
     * 库分片值集合
     */
    private final Map<ShardingKey, ShardingValue<?>> databaseShardingValues = new HashMap<>();
    /**
     * 只做库分片
     * {@link DatabaseHintRoutingEngine}
     */
    @Getter
    private boolean databaseShardingOnly;
    /**
     * 获取线索分片管理器实例.
     *
     * @return 线索分片管理器实例
     */
    public static HintManager getInstance() {
        HintManager result = new HintManager();
        HintManagerHolder.setHintManager(result);
        return result;
    }
    /**
```

分享

```
     * 设置分库分片值.
     *
     * <p>分片操作符为等号.该方法适用于只分库的场景</p>
     *
     * @param value 分片值
     */
    public void setDatabaseShardingValue(final Comparable<?> value) {
        databaseShardingOnly = true;
        addDatabaseShardingValue(HintManagerHolder.DB_TABLE_NAME, HintManagerHolder.DB_COLUMN_NAME, va
    }
}
```

那么如果要使用 DatabaseHintSQLRouter，我们只需要 `HintManager.getInstance().setDatabaseShardingValue(库分片值)` 即可。这里有两点要注意下：

- `HintManager#getInstance()`，每次获取到的都是**新**的 HintManager，多次赋值需要小心。
- `HintManager#close()`，使用完需要去清理，避免下个请求读到遗漏的线程变量。

看看 DatabaseHintSQLRouter 的实现：

```
// DatabaseHintSQLRouter.java
@Override
public SQLStatement parse(final String logicSQL, final int parametersSize) {
   return new SQLJudgeEngine(logicSQL).judge(); // 只解析 SQL 类型
}
@Override
// TODO insert的SQL仍然需要解析自增主键
public SQLRouteResult route(final String logicSQL, final List<Object> parameters, final SQLStatement s
```

分享

```
    Context context = MetricsContext.start("Route SQL");
    SQLRouteResult result = new SQLRouteResult(sqlStatement);
    // 路由
    RoutingResult routingResult = new DatabaseHintRoutingEngine(shardingRule.getDataSourceRule(), shard
            .route();
    // SQL最小执行单元
    for (TableUnit each : routingResult.getTableUnits().getTableUnits()) {
        result.getExecutionUnits().add(new SQLExecutionUnit(each.getDataSourceName(), logicSQL));
    }
    MetricsContext.stop(context);
    if (showSQL) {
        SQLLogger.logSQL(logicSQL, sqlStatement, result.getExecutionUnits(), parameters);
    }
    return result;
}
```

- `#parse()` 只解析了 SQL 类型，即 SELECT / UPDATE / DELETE / INSERT 。
- **使用的分库策略来自 ShardingRule，不是 TableRule，这个一定要留心。？因为 SQL 未解析表名**。因此，即使在 TableRule 设置了 `actualTables` 属性也是没有效果的。
- 目前不支持 Sharding-JDBC 的主键自增。？因为 SQL 未解析**自增主键**。从代码上的 `TODO` 应该会支持。
- `HintManager.getInstance().setDatabaseShardingValue(库分片值)` 设置的库分片值使用的是 EQUALS，因而分库策略计算出来的只有**一个库分片**，即 TableUnit 只有一个，SQLExecutionUnit 只有一个。

看看 DatabaseHintSQLRouter 的实现：

```
// DatabaseHintRoutingEngine.java
```

```
@Override
public RoutingResult route() {
    // 从 Hint 获得 分片键值
    Optional<ShardingValue<?>> shardingValue = HintManagerHolder.getDatabaseShardingValue(new ShardingK
    Preconditions.checkState(shardingValue.isPresent());
    log.debug("Before database sharding only db:{} sharding values: {}", dataSourceRule.getDataSourceNa
    // 路由。表分片规则使用的是 ShardingRule 里的。因为没 SQL 解析。
    Collection<String> routingDataSources = databaseShardingStrategy.doStaticSharding(sqlType, dataSour
    Preconditions.checkState(!routingDataSources.isEmpty(), "no database route info");
    log.debug("After database sharding only result: {}", routingDataSources);
    // 路由结果
    RoutingResult result = new RoutingResult();
    for (String each : routingDataSources) {
        result.getTableUnits().getTableUnits().add(new TableUnit(each, "", ""));
    }
    return result;
}
```

- **只**调用 `databaseShardingStrategy.doStaticSharding()` 方法计算**库**分片。
- `new TableUnit(each, "", "")` 的 `logicTableName`，`actualTableName` 都是空串，相信原因你已经知道。

分享

6. ParsingSQLRouter

ParsingSQLRouter，需要解析的SQL路由器。

ParsingSQLRouter 使用 SQLParsingEngine **解析SQL**。对**SQL解析**有兴趣的同学可以看看拙作《Sharding-JDBC 源码分析 —— SQL 解析》。

```
// ParsingSQLRouter.java
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) {
        ((InsertStatement) result).appendGenerateKeyToken(shardingRule, parametersSize);
    }
    MetricsContext.stop(context);
    return result;
}
```

- `#appendGenerateKeyToken()` 会在《SQL 改写》分享

ParsingSQLRouter 在路由时，会根据**表情况**使用 SimpleRoutingEngine 或 CartesianRoutingEngine 进行路由。

```
private RoutingResult route(final List<Object> parameters, final SQLStatement sqlStatement) {
    Collection<String> tableNames = sqlStatement.getTables().getTableNames();
    RoutingEngine routingEngine;
    if (1 == tableNames.size() || shardingRule.isAllBindingTables(tableNames)) {
        routingEngine = new SimpleRoutingEngine(shardingRule, parameters, tableNames.iterator().next(),
    } else {
        // TODO 可配置是否执行笛卡尔积
        routingEngine = new ComplexRoutingEngine(shardingRule, parameters, tableNames, sqlStatement);
    }
    return routingEngine.route();
}
```

分享

- 当只进行**一张表**或者**多表互为BindingTable关系**时，使用 SimpleRoutingEngine 简单路由引擎。**多表互为BindingTable关系**时，每张表的路由结果是相同的，所以只要计算第一张表的分片即可。
- `tableNames.iterator().next()` 注意下，`tableNames` 变量是 `new TreeMap<>(String.CASE_INSENSITIVE_ORDER)`。所以 `SELECT * FROM t_order o join t_order_item i ON o.order_id = i.order_id` 即使 `t_order_item` 排在 `t_order` 前面，`tableNames.iterator().next()` 返回的是 `t_order`。当 `t_order` 和 `t_order_item` 为 **BindingTable关系** 时，计算的是 `t_order` 路由分片。
- BindingTable关系在 ShardingRule 的 `tableRules` 配置。配置该关系 TableRule 有如下需要遵守的规则：
  - 分片策略与算法相同
  - 数据源配置对象相同
  - 真实表**数量**相同

举个例子：

- SQL ：`SELECT * FROM t_order o join t_order_item i ON o.order_id = i.order_id`
- 分库分表情况：

```
multi_db_multi_table_01
  ├── t_order_0                        ├── t_order_item_01
  └── t_order_1                        ├── t_order_item_02
                                       ├── t_order_item_03
                                       ├── t_order_item_04
multi_db_multi_table_02
  ├── t_order_0                        ├── t_order_item_01
  └── t_order_1                        ├── t_order_item_02
                                       ├── t_order_item_03
```

分享

```
├── t_order_item_04
```

最终执行的SQL如下：

```
SELECT * FROM t_order_item_01 i JOIN t_order_01 o ON o.order_id = i.order_id
SELECT * FROM t_order_item_01 i JOIN t_order_01 o ON o.order_id = i.order_id
SELECT * FROM t_order_item_02 i JOIN t_order_02 o ON o.order_id = i.order_id
SELECT * FROM t_order_item_02 i JOIN t_order_02 o ON o.order_id = i.order_id
```

- `t_order_item_03`、`t_order_item_04` 无法被查询到。

下面我们看看 `#isAllBindingTables()` 如何实现**多表互为BindingTable关系**。

```
// ShardingRule.java
// 调用顺序 #isAllBindingTables()=>#filterAllBindingTables()=>#findBindingTableRule()=>#findBindingTabl
/**
* 判断逻辑表名称集合是否全部属于Binding表.
* @param logicTables 逻辑表名称集合
*/
public boolean isAllBindingTables(final Collection<String> logicTables) {
   Collection<String> bindingTables = filterAllBindingTables(logicTables);
   return !bindingTables.isEmpty() && bindingTables.containsAll(logicTables);
}
/**
* 过滤出所有的Binding表名称.
*/
public Collection<String> filterAllBindingTables(final Collection<String> logicTables) {
   if (logicTables.isEmpty()) {
       return Collections.emptyList();
```

```
    }
    Optional<BindingTableRule> bindingTableRule = findBindingTableRule(logicTables);
    if (!bindingTableRule.isPresent()) {
        return Collections.emptyList();
    }
    // 交集
    Collection<String> result = new ArrayList<>(bindingTableRule.get().getAllLogicTables());
    result.retainAll(logicTables);
    return result;
}
/**
* 获得包含<strong>任一</strong>在逻辑表名称集合的binding表配置的逻辑表名称集合
*/
private Optional<BindingTableRule> findBindingTableRule(final Collection<String> logicTables) {
    for (String each : logicTables) {
        Optional<BindingTableRule> result = findBindingTableRule(each);
        if (result.isPresent()) {
            return result;
        }
    }
    return Optional.absent();
}
/**
* 根据逻辑表名称获取binding表配置的逻辑表名称集合.
*/
public Optional<BindingTableRule> findBindingTableRule(final String logicTable) {
    for (BindingTableRule each : bindingTableRules) {
        if (each.hasLogicTable(logicTable)) {
            return Optional.of(each);
```

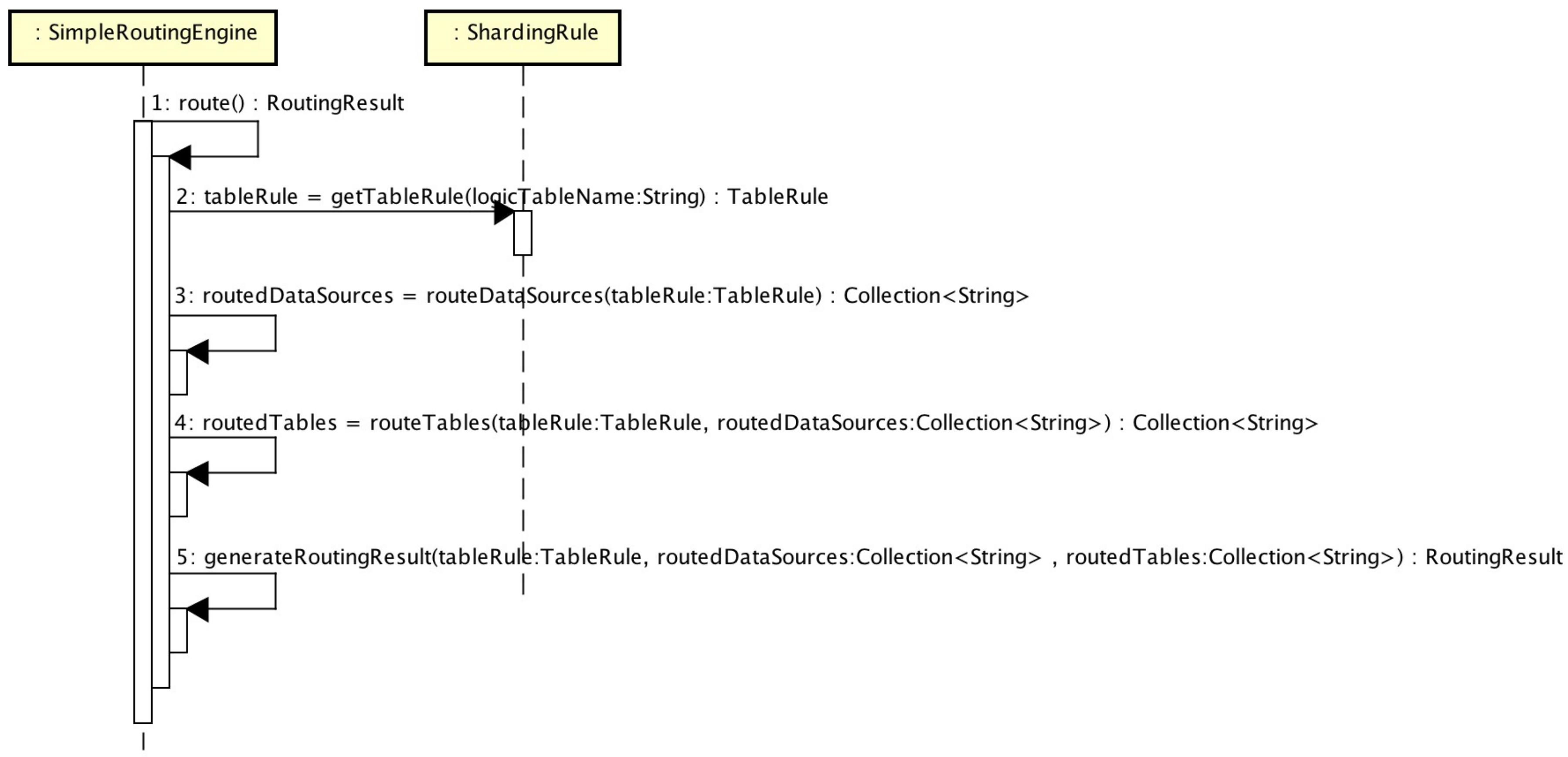
分享

```
        }
    }
    return Optional.absent();
}
```

- 逻辑看起来比较长，目的是找到一条 BindingTableRule 包含**所有**逻辑表集合
- 不支持《传递关系》：配置 BindingTableRule 时，**相同绑定关系一定要配置在一条**，必须是 `[a, b, c]`，而不能是 `[a, b], [b, c]`。

6.1 SimpleRoutingEngine

SimpleRoutingEngine，简单路由引擎。




```
// SimpleRoutingEngine.java
private Collection<String> routeDataSources(final TableRule tableRule) {
   DatabaseShardingStrategy strategy = shardingRule.getDatabaseShardingStrategy(tableRule);
   List<ShardingValue<?>> shardingValues = HintManagerHolder.isUseShardingHint() ? getDatabaseSharding
           : getShardingValues(strategy.getShardingColumns());
   Collection<String> result = strategy.doStaticSharding(sqlStatement.getType(), tableRule.getActualDa
   Preconditions.checkState(!result.isEmpty(), "no database route info");
   return result;
}
```

分享

```
private List<ShardingValue<?>> getShardingValues(final Collection<String> shardingColumns) {
   List<ShardingValue<?>> result = new ArrayList<>(shardingColumns.size());
   for (String each : shardingColumns) {
       Optional<Condition> condition = sqlStatement.getConditions().find(new Column(each, logicTableNa
       if (condition.isPresent()) {
           result.add(condition.get().getShardingValue(parameters));
       }
   }
   return result;
}
```

- 可以使用 HintManager 设置**库**分片值进行**强制路由**。
- `#getShardingValues()` 我们看到了《SQL 解析（二）之SQL解析》分享的 Condition 对象。之前我们提到过**Parser 半理解SQL的目的之一是：提炼分片上下文**，此处即是该目的的体现。Condition 里只放**明确**影响路由的条件，例如：`order_id = 1`，`order_id IN (1, 2)`，`order_id BETWEEN (1, 3)`，不放**无法计算**的条件，例如：`o.order_id = i.order_id`。该方法里，使用**分片键**从 Condition 查找 **分片值**。 是不是对 Condition 的认识更加清晰一丢丢落。

分享

```
// SimpleRoutingEngine.java
private Collection<String> routeTables(final TableRule tableRule, final Collection<String> routedDataS
   TableShardingStrategy strategy = shardingRule.getTableShardingStrategy(tableRule);
   List<ShardingValue<?>> shardingValues = HintManagerHolder.isUseShardingHint() ? getTableShardingVal
           : getShardingValues(strategy.getShardingColumns());
   Collection<String> result = tableRule.isDynamic() ? strategy.doDynamicSharding(shardingValues)
           : strategy.doStaticSharding(sqlStatement.getType(), tableRule.getActualTableNames(routedDat
   Preconditions.checkState(!result.isEmpty(), "no table route info");
```

```
    return result;
}
```

- 可以使用 HintManager 设置**表**分片值进行**强制路由**。
- 根据 `dynamic` 属性来判断调用 `#doDynamicSharding()` 还是 `#doStaticSharding()` 计算分片。

```
// SimpleRoutingEngine.java
private RoutingResult generateRoutingResult(final TableRule tableRule, final Collection<String> routed
    RoutingResult result = new RoutingResult();
    for (DataNode each : tableRule.getActualDataNodes(routedDataSources, routedTables)) {
        result.getTableUnits().getTableUnits().add(new TableUnit(each.getDataSourceName(), logicTableNa
    }
    return result;
}
// TableRule.java
/**
* 根据数据源名称过滤获取真实数据单元.
* @param targetDataSources 数据源名称集合
* @param targetTables 真实表名称集合
* @return 真实数据单元
*/
public Collection<DataNode> getActualDataNodes(final Collection<String> targetDataSources, final Colle
    return dynamic ? getDynamicDataNodes(targetDataSources, targetTables) : getStaticDataNodes(targetDa
}
private Collection<DataNode> getDynamicDataNodes(final Collection<String> targetDataSources, final Col
    Collection<DataNode> result = new LinkedHashSet<>(targetDataSources.size() * targetTables.size());
    for (String targetDataSource : targetDataSources) {
```

分享

```
        for (String targetTable : targetTables) {
            result.add(new DataNode(targetDataSource, targetTable));
        }
    }
    return result;
}
private Collection<DataNode> getStaticDataNodes(final Collection<String> targetDataSources, final Coll
    Collection<DataNode> result = new LinkedHashSet<>(actualTables.size());
    for (DataNode each : actualTables) {
        if (targetDataSources.contains(each.getDataSourceName()) && targetTables.contains(each.getTable
            result.add(each);
        }
    }
    return result;
}
```

- 在 SimpleRoutingEngine 只生成了当前表的 TableUnits。如果存在**与其互为BindingTable关系**的表的 TableUnits 怎么获得？你可以想想噢，当然在后文《SQL 改写》也会给出答案，看看和你想的是否一样。

6.2 ComplexRoutingEngine

ComplexRoutingEngine，混合多库表路由引擎。

```
// ComplexRoutingEngine.java
@Override
public RoutingResult route() {
    Collection<RoutingResult> result = new ArrayList<>(logicTables.size());
```

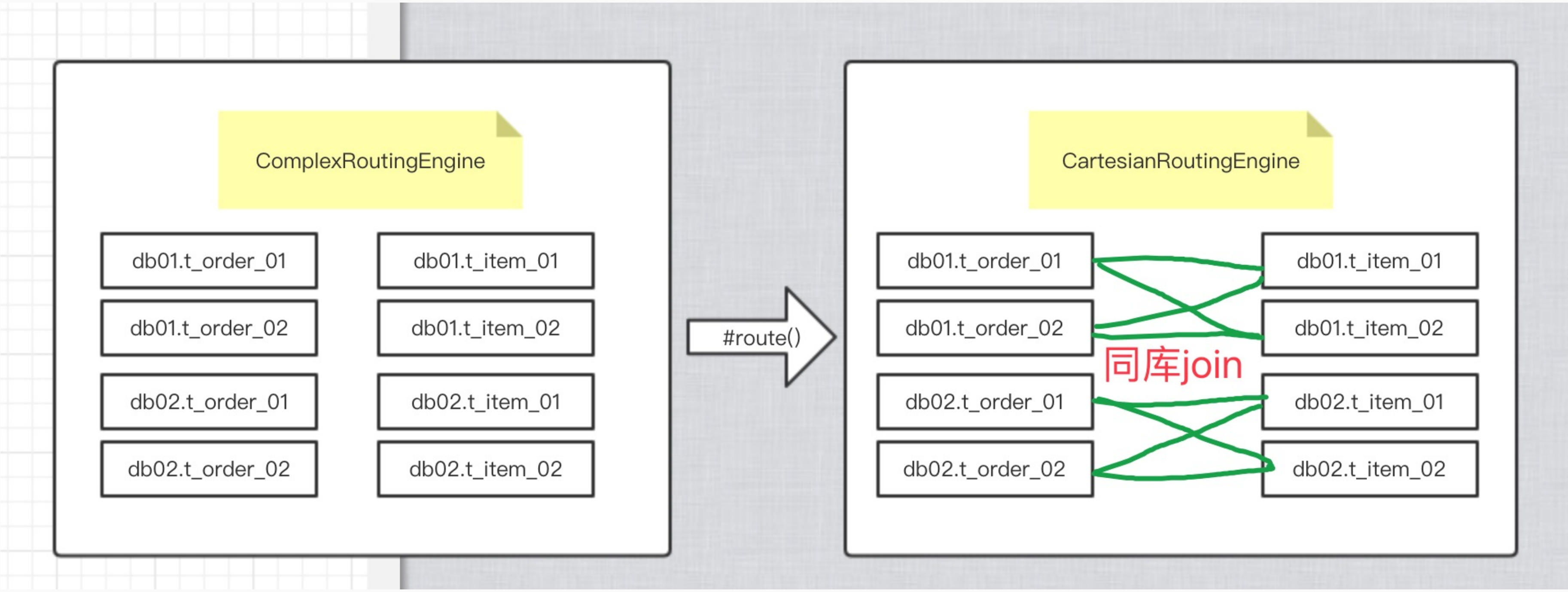
分享

```
    Collection<String> bindingTableNames = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
    // 计算每个逻辑表的简单路由分片
    for (String each : logicTables) {
        Optional<TableRule> tableRule = shardingRule.tryFindTableRule(each);
        if (tableRule.isPresent()) {
            if (!bindingTableNames.contains(each)) {
                result.add(new SimpleRoutingEngine(shardingRule, parameters, tableRule.get().getLogicTa
            }
            // 互为 BindingTable 关系的表加到 bindingTableNames 里，不重复计算分片
            Optional<BindingTableRule> bindingTableRule = shardingRule.findBindingTableRule(each);
            if (bindingTableRule.isPresent()) {
                bindingTableNames.addAll(Lists.transform(bindingTableRule.get().getTableRules(), new Fu

                    @Override
                    public String apply(final TableRule input) {
                        return input.getLogicTable();
                    }
                }));
            }
        }
    }
    log.trace("mixed tables sharding result: {}", result);
    if (result.isEmpty()) {
        throw new ShardingJdbcException("Cannot find table rule and default data source with logic tabl
    }
    // 防御性编程。shardingRule#isAllBindingTables() 已经过滤了这个情况。
    if (1 == result.size()) {
        return result.iterator().next();
    }
```

```
    // 交给 CartesianRoutingEngine 形成笛卡尔积结果
    return new CartesianRoutingEngine(result).route();
}
```

- ComplexRoutingEngine 计算每个逻辑表的简单路由分片，路由结果交给 CartesianRoutingEngine **继续**路由形成笛卡尔积结果。



- 由于目前 ComplexRoutingEngine 路由前已经判断**全部表互为 BindingTable 关系**，因而不会出现 `result.size == 1`，属于防御性编程。
- **部分表互为 BindingTable 关系**时，ComplexRoutingEngine 不重复计算分片。

6.3 CartesianRoutingEngine

CartesianRoutingEngine，笛卡尔积的库表路由。

实现逻辑上**相对**复杂，请保持耐心哟，😈 其实目的就是实现**连连看**的效果：

- RoutingResult[0] `x` RoutingResult[1] …… `x` RoutingResult[n- 1] `x` RoutingResult[n]
- **同库** 才可以进行笛卡尔积

```
// CartesianRoutingEngine.java
@Override
public CartesianRoutingResult route() {
   CartesianRoutingResult result = new CartesianRoutingResult();
   for (Entry<String, Set<String>> entry : getDataSourceLogicTablesMap().entrySet()) { // Entry<数据源
       // 获得当前数据源（库）的 路由表单元分组
       List<Set<String>> actualTableGroups = getActualTableGroups(entry.getKey(), entry.getValue()); /
       List<Set<TableUnit>> tableUnitGroups = toTableUnitGroups(entry.getKey(), actualTableGroups);
       // 笛卡尔积，并合并结果
       result.merge(entry.getKey(), getCartesianTableReferences(Sets.cartesianProduct(tableUnitGroups)
   }
   log.trace("cartesian tables sharding result: {}", result);
   return result;
}
```

- 第一步，获得**同库**对应的**逻辑表**集合，即 **Entry<数据源（库），Set<逻辑表>> entry**。
- 第二步，遍历**数据源（库）**，获得当前**数据源（库）的路由表单元分组。**
- 第三步，对**路由表单元分组**进行**笛卡尔积**，并合并到路由结果。

下面，我们一起逐步看看代码实现。

- SQL ： `SELECT * FROM t_order o join t_order_item i ON o.order_id = i.order_id`
- 分库分表情况：

```
multi_db_multi_table_01
  ├── t_order_0                        ├── t_order_item_01
  └── t_order_1                        ├── t_order_item_02
multi_db_multi_table_02
  ├── t_order_0                        ├── t_order_item_01
  └── t_order_1                        ├── t_order_item_02
```

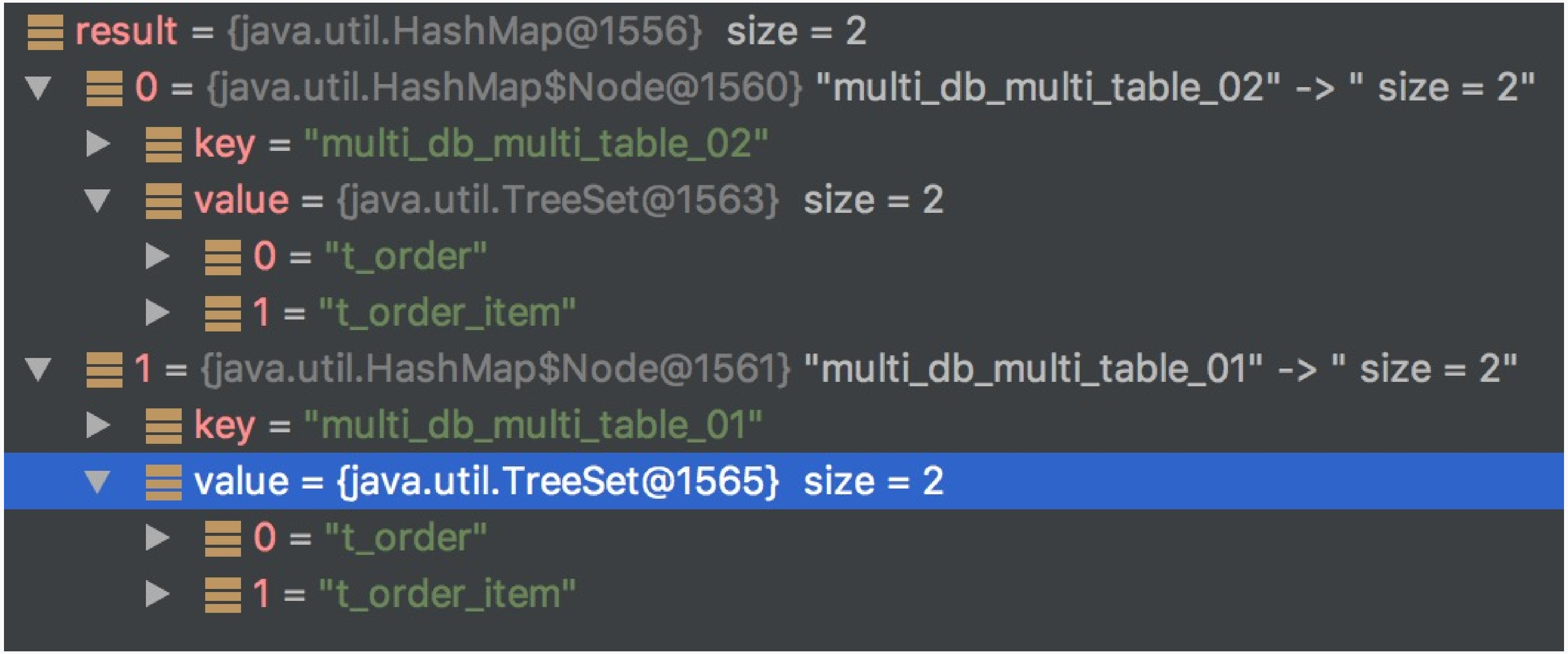
```
// 第一步
// CartesianRoutingEngine.java
/**
* 获得同库对应的逻辑表集合
*/
private Map<String, Set<String>> getDataSourceLogicTablesMap() {
   Collection<String> intersectionDataSources = getIntersectionDataSources();
   Map<String, Set<String>> result = new HashMap<>(routingResults.size());
   // 获得同库对应的逻辑表集合
   for (RoutingResult each : routingResults) {
       for (Entry<String, Set<String>> entry : each.getTableUnits().getDataSourceLogicTablesMap(inters
           if (result.containsKey(entry.getKey())) {
               result.get(entry.getKey()).addAll(entry.getValue());
           } else {
               result.put(entry.getKey(), entry.getValue());
           }
       }
   }
```

分享

```java
    return result;
}
/**
* 获得所有路由结果里的数据源（库）交集
*/
private Collection<String> getIntersectionDataSources() {
    Collection<String> result = new HashSet<>();
    for (RoutingResult each : routingResults) {
        if (result.isEmpty()) {
            result.addAll(each.getTableUnits().getDataSourceNames());
        }
        result.retainAll(each.getTableUnits().getDataSourceNames()); // 交集
    }
    return result;
}
```

- `#getDataSourceLogicTablesMap()` 返回如图：

分享

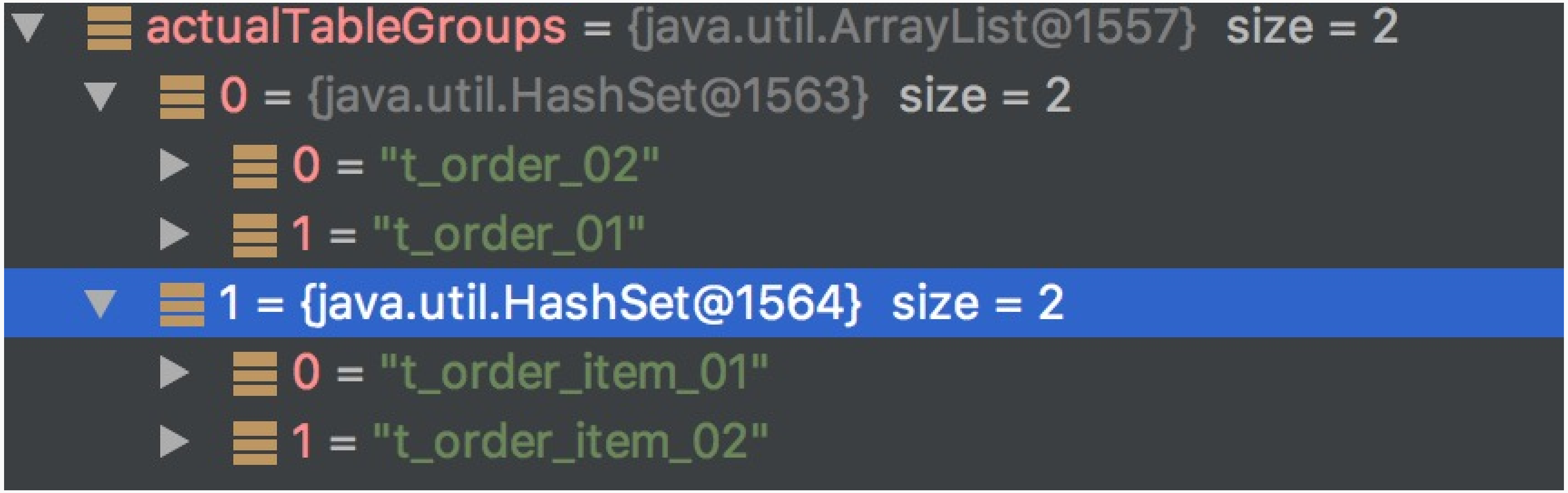


```
// 第二步
// CartesianRoutingEngine.java
private List<Set<String>> getActualTableGroups(final String dataSource, final Set<String> logicTables)
   List<Set<String>> result = new ArrayList<>(logicTables.size());
   for (RoutingResult each : routingResults) {
       result.addAll(each.getTableUnits().getActualTableNameGroups(dataSource, logicTables));
   }
   return result;
}
private List<Set<TableUnit>> toTableUnitGroups(final String dataSource, final List<Set<String>> actual
   List<Set<TableUnit>> result = new ArrayList<>(actualTableGroups.size());
   for (Set<String> each : actualTableGroups) {
```

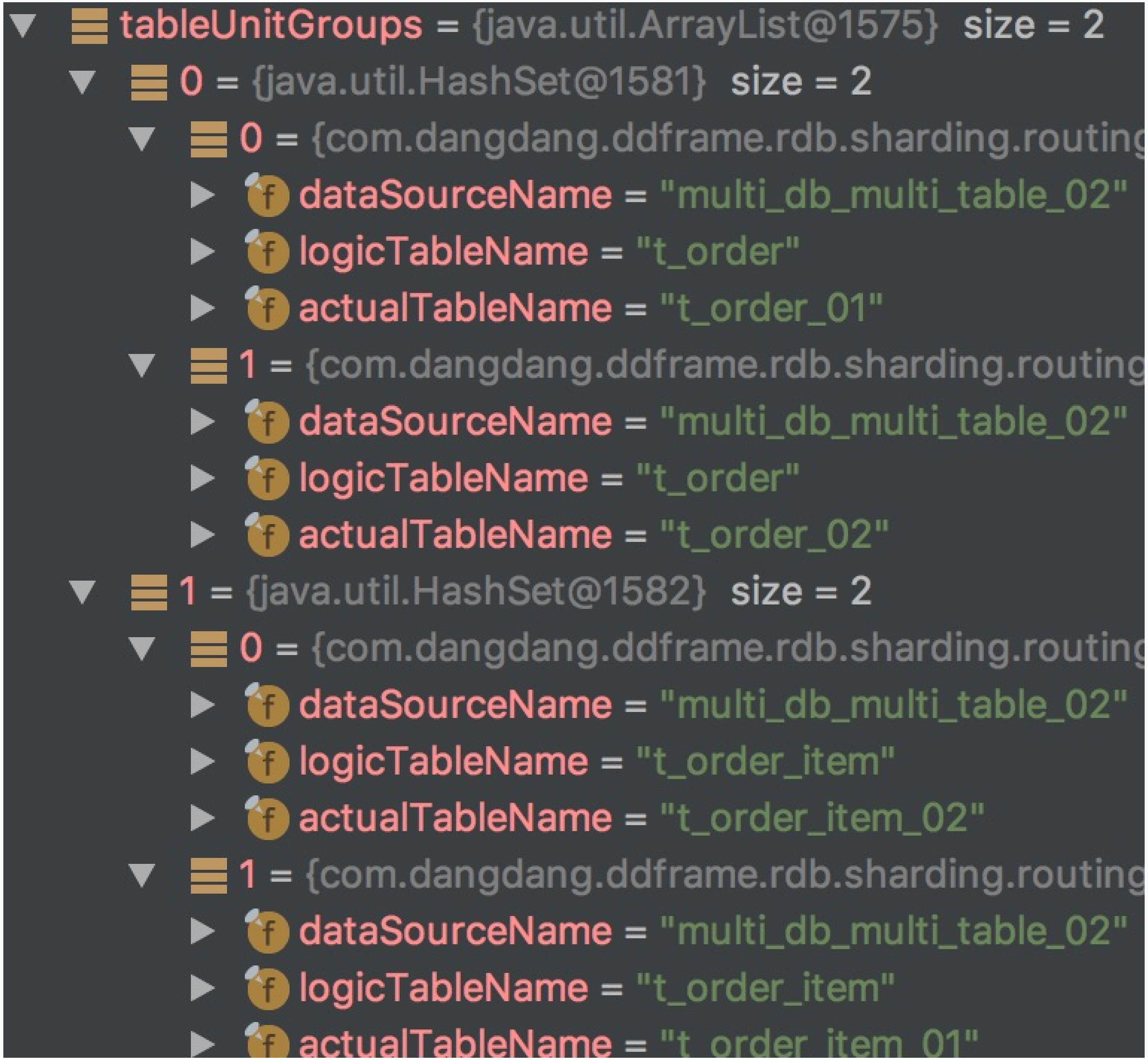
```
            result.add(new HashSet<>(Lists.transform(new ArrayList<>(each), new Function<String, TableUnit>

                @Override
                public TableUnit apply(final String input) {
                    return findTableUnit(dataSource, input);
                }
            })));
        }
        return result;
    }
```

- `#getActualTableGroups()` 返回如图：



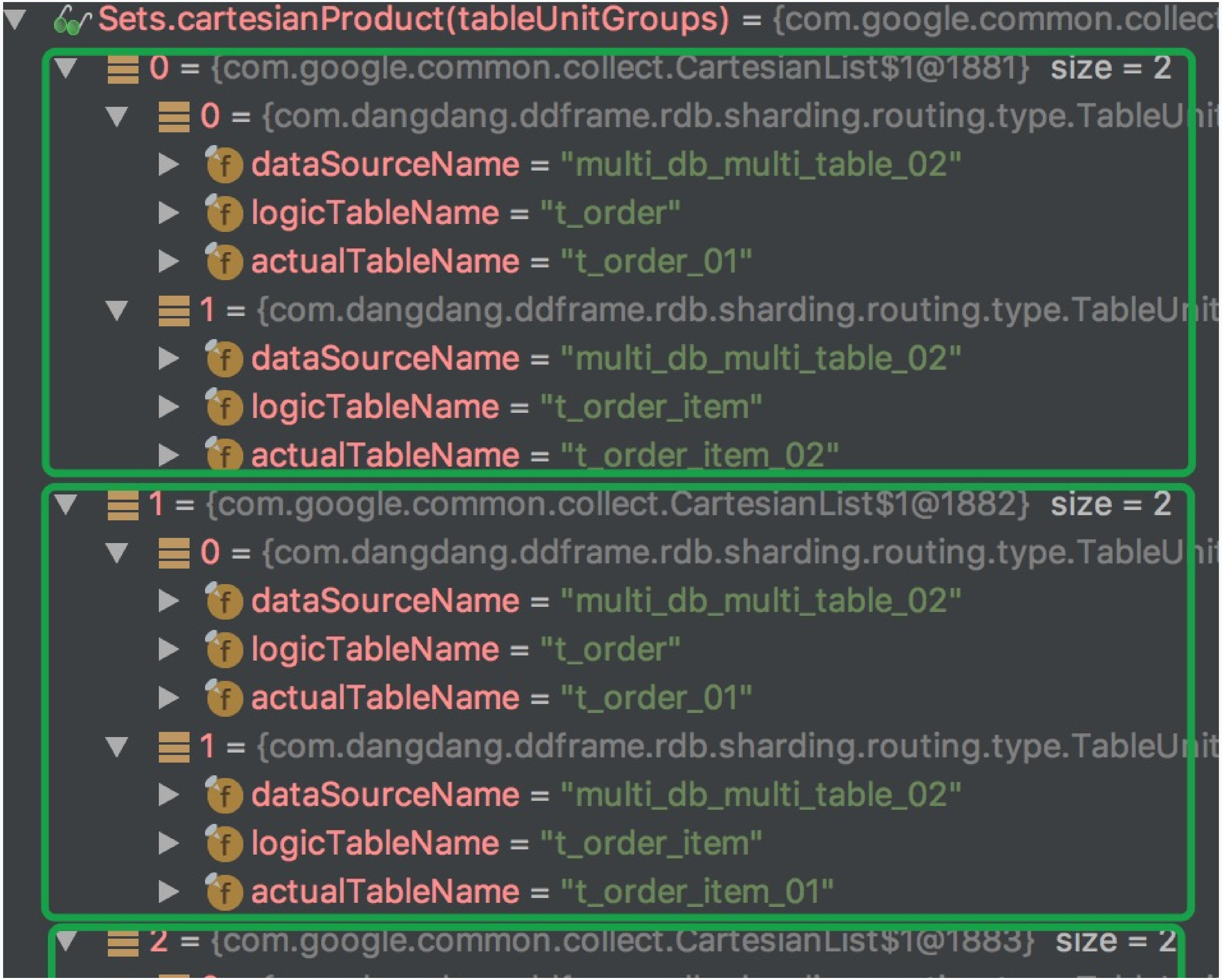
- `#toTableUnitGroups()` 返回如图：

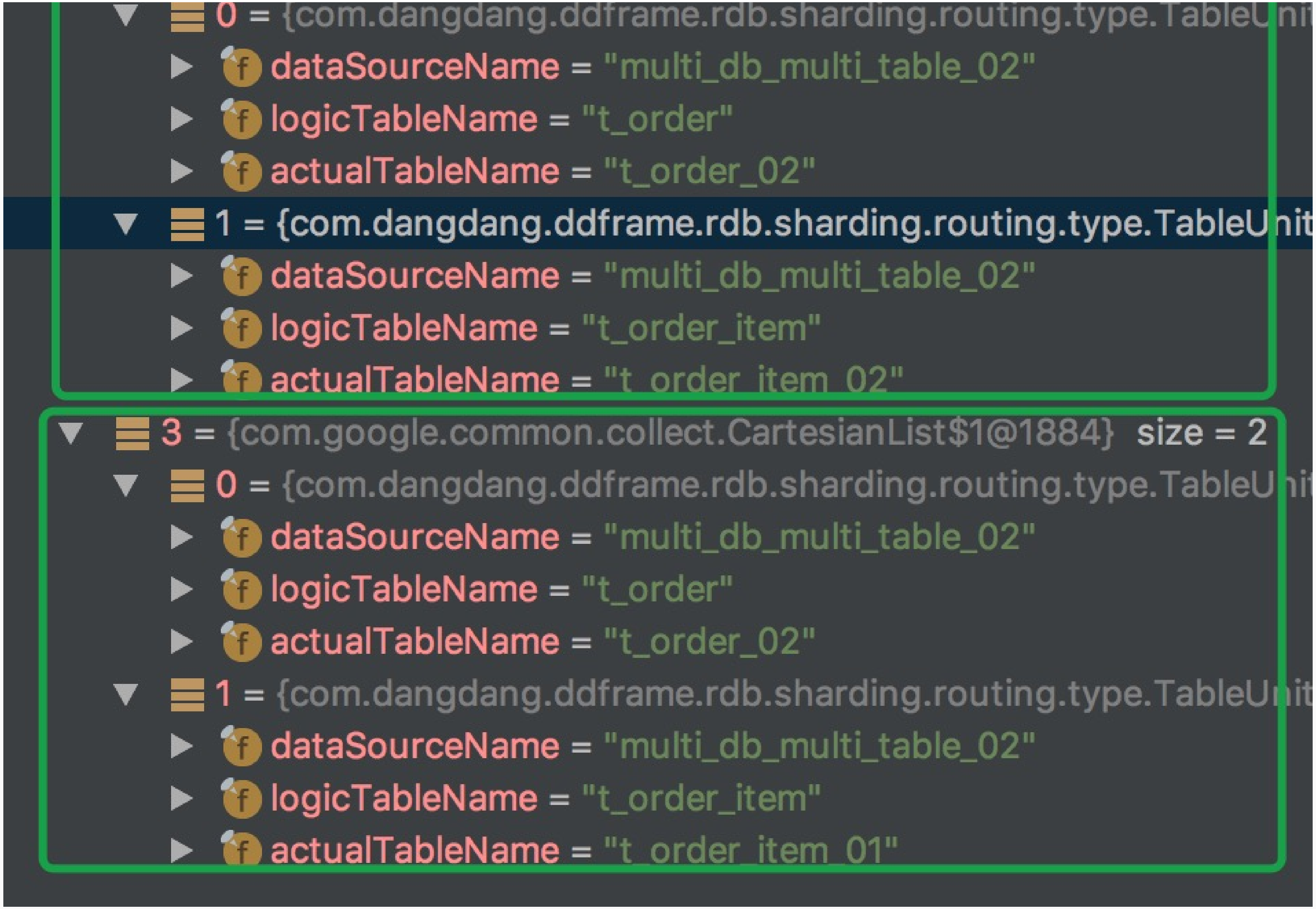

tableUnitGroups = {java.util.ArrayList@1575} size = 2
0 = {java.util.HashSet@1581} size = 2
0 = {com.dangdang.ddframe.rdb.sharding.routing
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order"
actualTableName = "t_order_01"
1 = {com.dangdang.ddframe.rdb.sharding.routing
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order"
actualTableName = "t_order_02"
1 = {java.util.HashSet@1582} size = 2
0 = {com.dangdang.ddframe.rdb.sharding.routing
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order_item"
actualTableName = "t_order_item_02"
1 = {com.dangdang.ddframe.rdb.sharding.routing
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order_item"
actualTableName = "t_order_item_01"

```
// CartesianRoutingEngine.java
private List<CartesianTableReference> getCartesianTableReferences(final Set<List<TableUnit>> cartesian
    List<CartesianTableReference> result = new ArrayList<>(cartesianTableUnitGroups.size());
    for (List<TableUnit> each : cartesianTableUnitGroups) {
        result.add(new CartesianTableReference(each));
    }
    return result;
}
// CartesianRoutingResult.java
@Getter
private final List<CartesianDataSource> routingDataSources = new ArrayList<>();
void merge(final String dataSource, final Collection<CartesianTableReference> routingTableReferences)
    for (CartesianTableReference each : routingTableReferences) {
        merge(dataSource, each);
    }
}
private void merge(final String dataSource, final CartesianTableReference routingTableReference) {
    for (CartesianDataSource each : routingDataSources) {
        if (each.getDataSource().equalsIgnoreCase(dataSource)) {
            each.getRoutingTableReferences().add(routingTableReference);
            return;
        }
    }
    routingDataSources.add(new CartesianDataSource(dataSource, routingTableReference));
}
```

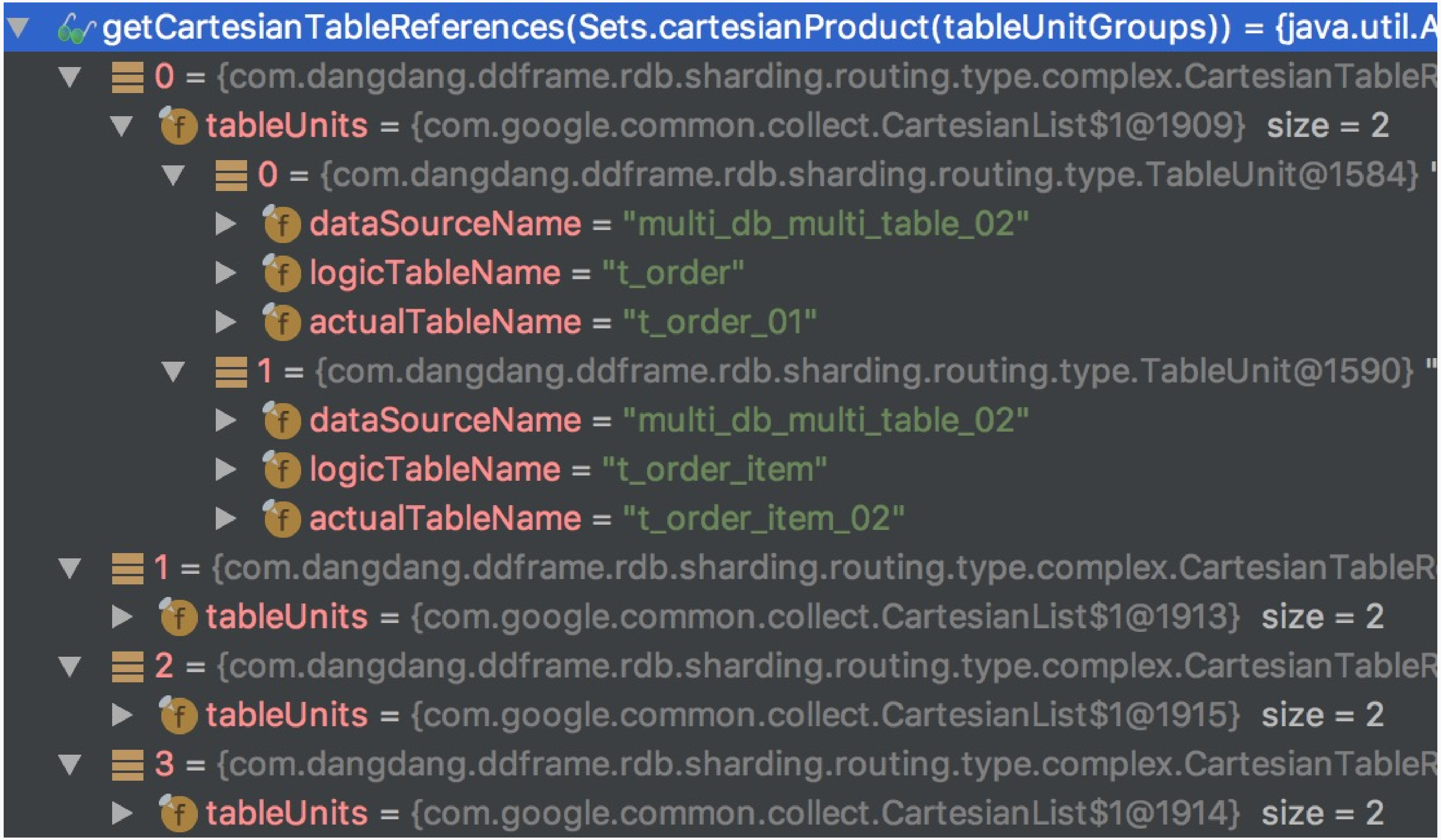
分享

- `Sets.cartesianProduct(tableUnitGroups)` 返回如图（Guava 工具库真强大）：


Sets.cartesianProduct(tableUnitGroups) = {com.google.common.collect
0 = {com.google.common.collect.CartesianList$1@1881} size = 2
0 = {com.dangdang.ddframe.rdb.sharding.routing.type.TableUnit
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order"
actualTableName = "t_order_01"
1 = {com.dangdang.ddframe.rdb.sharding.routing.type.TableUnit
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order_item"
actualTableName = "t_order_item_02"
1 = {com.google.common.collect.CartesianList$1@1882} size = 2
0 = {com.dangdang.ddframe.rdb.sharding.routing.type.TableUnit
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order"
actualTableName = "t_order_01"
1 = {com.dangdang.ddframe.rdb.sharding.routing.type.TableUnit
dataSourceName = "multi_db_multi_table_02"
logicTableName = "t_order_item"
actualTableName = "t_order_item_01"
2 = {com.google.common.collect.CartesianList$1@1883} size = 2



- `#getCartesianTableReferences()` 返回如图：



CartesianTableReference，笛卡尔积表**路由组**，包含**多条** TableUnit，即 TableUnit[0] `x` TableUnit[1] …… `x` TableUnit[n]。例如图中：`t_order_01 x t_order_item_02`，最终转换成 SQL 为 `SELECT * FROM t_order_01 o join t_order_item_02 i ON o.order_id = i.order_id`。

- `#merge()` 合并笛卡尔积路由结果。CartesianRoutingResult 包含多个 CartesianDataSource，因此需要将 CartesianTableReference 合并（添加）到对应的 CartesianDataSource。当然，目前在实现时已经是按照**数据

源（库）**生成对应的 CartesianTableReference。

6.4 ParsingSQLRouter 主#route()

```
// ParsingSQLRouter.java
@Override
public SQLRouteResult route(final String logicSQL, final List<Object> parameters, final SQLStatement s
    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).getGenerate
        processGeneratedKey(parameters, (InsertStatement) sqlStatement, result);
    }
    // 🐒🐒🐒 路由 🐒🐒🐒
    RoutingResult routingResult = route(parameters, sqlStatement);
    // SQL重写引擎
    SQLRewriteEngine rewriteEngine = new SQLRewriteEngine(shardingRule, logicSQL, sqlStatement);
    boolean isSingleRouting = routingResult.isSingleRouting();
    // 处理分页
    if (sqlStatement instanceof SelectStatement && null != ((SelectStatement) sqlStatement).getLimit())
        processLimit(parameters, (SelectStatement) sqlStatement, isSingleRouting);
    }
    // SQL 重写
    SQLBuilder sqlBuilder = rewriteEngine.rewrite(!isSingleRouting);
    // 生成 ExecutionUnit
    if (routingResult instanceof CartesianRoutingResult) {
        for (CartesianDataSource cartesianDataSource : ((CartesianRoutingResult) routingResult).getRout
            for (CartesianTableReference cartesianTableReference : cartesianDataSource.getRoutingTableR
```

分享

```
                result.getExecutionUnits().add(new SQLExecutionUnit(cartesianDataSource.getDataSource()
            }
        }
    } else {
        for (TableUnit each : routingResult.getTableUnits().getTableUnits()) {
            result.getExecutionUnits().add(new SQLExecutionUnit(each.getDataSourceName(), rewriteEngine
        }
    }
    MetricsContext.stop(context);
    // 打印 SQL
    if (showSQL) {
        SQLLogger.logSQL(logicSQL, sqlStatement, result.getExecutionUnits(), parameters);
    }
    return result;
}
```

- `RoutingResult routingResult = route(parameters, sqlStatement);` **调用**的就是上文分析的 SimpleRoutingEngine、ComplexRoutingEngine、CartesianRoutingEngine 的 `#route()` 方法。
- `#processGeneratedKey()`、`#processLimit()`、`#rewrite()`、`#generateSQL()` 等会放在《SQL 改写》 分享。

666. 彩蛋

篇幅有些长，希望能让大家对**路由**有比较完整的认识。
如果内容有错误，烦请您指正，我会**认真**修改。
如果表述不清晰，不太理解的，欢迎加我微信（wangwenbin-server）一起探讨。

谢谢你技术这么好，还**耐心**看完了本文。

强制路由 HintManager 讲的相对略过，可以看如下内容进一步了解：

1. 《官方文档-强制路由》
2. HintManager.java 源码

厚着脸皮，道友，辛苦**分享朋友圈**可好？！

Sharding-JDBC

PREVIOUS:
« Sharding-JDBC 源码分析 —— SQL 路由（三）之Spring与YAML配置
NEXT:
» Sharding-JDBC 源码分析 —— SQL 路由（一）之分库分表配置