
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

微信公众号福利：芋艿的后端小屋

0. 阅读源码葵花宝典

1. RocketMQ / MyCAT / Sharding-JDBC 详细中文注释源码

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

Docker [2]

MyCAT [9]

Nginx [1]

RocketMQ [14]

Sharding-JDBC [17]

技术杂文 [2]

分享

Sharding-JDBC 源码分析 —— 结果归并

2017-08-16 更新日期:2017-08-07 总阅读量:16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。

TODO 目录

---

1. 概述

本文分享**查询结果归并**的源码实现。

正如前文《SQL 执行》提到的**“分表分库，需要执行的 SQL 数量从单条变成了多条”，**多个**SQL执行**结果必然需要进行合并，例如：

```
SELECT * FROM t_order ORDER BY create_time
```

在各分片排序完后，Sharding-JDBC 获取到结果后，仍然需要再进一步排序。目前有 **分页**、**分组**、**排序**、**AVG聚合计算**、**迭代** 五种场景需要做进一步处理。当然，如果单分片**SQL执行**结果是无需合并的。在《SQL 执行》不知不觉已经分享了插入、更新、删除操作的结果合并，所以下面我们一起看看**查询结果归并**的实现。

> **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. MergeEngine

MergeEngine，分片结果集归并引擎。

```
// MergeEngine.java
/**
* 数据库类型
*/
private final DatabaseType databaseType;
/**
* 结果集集合
*/
private final List<ResultSet> resultSets;
/**
* Select SQL语句对象
*/
private final SelectStatement selectStatement;
/**
* 查询列名与位置映射
*/
private final Map<String, Integer> columnLabelIndexMap;

public MergeEngine(final DatabaseType databaseType, final List<ResultSet> resultSets, final SelectStat
    this.databaseType = databaseType;
    this.resultSets = resultSets;
```

```
        this.selectStatement = selectStatement;
        // 获得 查询列名与位置映射
        columnLabelIndexMap = getColumnLabelIndexMap(resultSets.get(0));
    }
    /**
    * 获得 查询列名与位置映射
    *
    * @param resultSet 结果集
    * @return 查询列名与位置映射
    * @throws SQLException 当结果集已经关闭
    */
    private Map<String, Integer> getColumnLabelIndexMap(final ResultSet resultSet) throws SQLException {
        ResultSetMetaData resultSetMetaData = resultSet.getMetaData(); // 元数据（包含查询列信息）
        Map<String, Integer> result = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);
        for (int i = 1; i <= resultSetMetaData.getColumnCount(); i++) {
            result.put(SQLUtil.getExactlyValue(resultSetMetaData.getColumnLabel(i)), i);
        }
        return result;
    }
```

- 当 MergeEngine 被创建时，会传入 `resultSets` 结果集集合，并根据其获得 `columnLabelIndexMap` 查询列名与位置映射。通过 `columnLabelIndexMap` ，可以很方便的使用查询列名获得在返回结果记录列(header)的第几列。

MergeEngine 的 `#merge()` 方法作为入口提供**查询结果归并**功能。

```
    /**
    * 合并结果集.
    *
```

```
     * @return 归并完毕后的结果集
     * @throws SQLException SQL异常
     */
    public ResultSetMerger merge() throws SQLException {
        selectStatement.setIndexForItems(columnLabelIndexMap);
        return decorate(build());
    }
```

- `#merge()` 主体逻辑就两行代码，设置查询列位置信息，并返回**合适**的归并结果集接口(ResultSetMerger) 实现。

2.1 SelectStatement#setIndexForItems()

```
// SelectStatement.java
/**
* 为选择项设置索引.
*
* @param columnLabelIndexMap 列标签索引字典
*/
public void setIndexForItems(final Map<String, Integer> columnLabelIndexMap) {
    setIndexForAggregationItem(columnLabelIndexMap);
    setIndexForOrderItem(columnLabelIndexMap, orderByItems);
    setIndexForOrderItem(columnLabelIndexMap, groupByItems);
}
```

- 部分**查询列**是经过**推到**出来，在 **SQL解析** 过程中，未获得到查询列位置，需要通过该方法进行初始化。对这块不了解的同学，回头可以看下《SQL 解析（三）之查询SQL》。 现在不用回头，皇冠会掉。
- `#setIndexForAggregationItem()` 处理 **AVG聚合计算列** 推导出其对应的 **SUM/COUNT 聚合计算列**的位置：

```
private void setIndexForAggregationItem(final Map<String, Integer> columnLabelIndexMap) {
    for (AggregationSelectItem each : getAggregationSelectItems()) {
        Preconditions.checkState(columnLabelIndexMap.containsKey(each.getColumnLabel()), String
        each.setIndex(columnLabelIndexMap.get(each.getColumnLabel()));
        for (AggregationSelectItem derived : each.getDerivedAggregationSelectItems()) {
            Preconditions.checkState(columnLabelIndexMap.containsKey(derived.getColumnLabel()),
            derived.setIndex(columnLabelIndexMap.get(derived.getColumnLabel()));
        }
    }
}
```

- `#setIndexForOrderItem()` 处理 **ORDER BY / GROUP BY 列不在查询列** 推导出的**查询列**的位置：

```
private void setIndexForOrderItem(final Map<String, Integer> columnLabelIndexMap, final List<O
for (OrderItem each : orderItems) {
  if (-1 != each.getIndex()) {
      continue;
  }
  Preconditions.checkState(columnLabelIndexMap.containsKey(each.getColumnLabel()), String.form
  if (columnLabelIndexMap.containsKey(each.getColumnLabel())) {
      each.setIndex(columnLabelIndexMap.get(each.getColumnLabel()));
  }
}
}
```

2.2 ResultSetMerger

ResultSetMerger，归并结果集接口。

我们先来看看整体的类结构关系：

AbstractStreamResultSetMerger：next时加载 AbstractMemoryResultSetMerger：加载完所有记录

3. OrderByStreamResultSetMerger

OrderByStreamResultSetMerger，基于 **Stream** 方式排序归并结果集实现。

3.1 归并算法

因为**各个分片结果集已经排序完成**，使用**《归并算法》**能够充分利用这个优势。

> 归并操作（merge），也叫归并算法，指的是将两个已经排序的序列合并成一个序列的操作。归并排序算法依赖归并操作。
>
> 【迭代法】
>
> 1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
> 2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
> 3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
> 4. 重复步骤3直到某一指针到达序列尾
> 5. 将另一序列剩下的所有元素直接复制到合并序列尾

从定义上看，是不是超级符合我们这个场景。😈 此时此刻，你是不是捂着胸口，感叹：“大学怎么没好好学数据结构与算法呢”？反正我是捂着了，都是眼泪。


```
public class OrderByStreamResultSetMerger extends AbstractStreamResultSetMerger {
    /**
     * 排序列
     */
    @Getter(AccessLevel.NONE)
    private final List<OrderItem> orderByItems;
    /**
     * 排序值对象队列
     */
    private final Queue<OrderByValue> orderByValuesQueue;
    /**
     * 默认排序类型
     */
    private final OrderType nullOrderType;
    /**
     * 是否第一个 ResultSet 已经调用 #next()
     */
    private boolean isFirstNext;

    public OrderByStreamResultSetMerger(final List<ResultSet> resultSets, final List<OrderItem> orderB
        this.orderByItems = orderByItems;
        this.orderByValuesQueue = new PriorityQueue<>(resultSets.size());
        this.nullOrderType = nullOrderType;
        orderResultSetsToQueue(resultSets);
        isFirstNext = true;
    }
```

```
private void orderResultSetsToQueue(final List<ResultSet> resultSets) throws SQLException {
    for (ResultSet each : resultSets) {
        OrderByValue orderByValue = new OrderByValue(each, orderByItems, nullOrderType);
        if (orderByValue.next()) {
            orderByValuesQueue.offer(orderByValue);
        }
    }
    // 设置当前 ResultSet，这样 #getValue() 能拿到记录
    setCurrentResultSet(orderByValuesQueue.isEmpty() ? resultSets.get(0) : orderByValuesQueue.peek
}
```

- 属性 `orderByValuesQueue` 使用的队列实现是**优先级**队列(PriorityQueue)。有兴趣的同学可以看看《JDK源码研究PriorityQueue》，本文不展开讲，不是主角戏份不多。我们记住几个方法的用途：
  - `#offer()` ：增加元素。增加时，会将该元素和已有元素们按照**优先级**进行排序
  - `#peek()` ：获得优先级第一的元素
  - `#pool()` ：获得优先级第一的元素**并移除**
- 一个 ResultSet 构建一个 OrderByValue 用于排序，即上文**归并算法**提到的**“空间”**。

```
public final class OrderByValue implements Comparable<OrderByValue> {
    /**
     * 已排序结果集
     */
    @Getter
    private final ResultSet resultSet;
```

```
    /**
     * 排序列
     */
    private final List<OrderItem> orderByItems;
    /**
     * 默认排序类型
     */
    private final OrderType nullOrderType;
    /**
     * 排序列对应的值数组
     * 因为一条记录可能有多个排序列，所以是数组
     */
    private List<Comparable<?>> orderValues;

    /**
     * 遍历下一个结果集游标.
     *
     * @return 是否有下一个结果集
     * @throws SQLException SQL异常
     */
    public boolean next() throws SQLException {
        boolean result = resultSet.next();
        orderValues = result ? getOrderValues() : Collections.<Comparable<?>>emptyList();
        return result;
    }
    /**
     * 获得 排序列对应的值数组
     *
```

```
     * @return 排序列对应的值数组
     * @throws SQLException 当结果集关闭时
     */
    private List<Comparable<?>> getOrderValues() throws SQLException {
        List<Comparable<?>> result = new ArrayList<>(orderByItems.size());
        for (OrderItem each : orderByItems) {
            Object value = resultSet.getObject(each.getIndex());
            Preconditions.checkState(null == value || value instanceof Comparable, "Order by v
            result.add((Comparable<?>) value);
        }
        return result;
    }
    /**
     * 对比 {@link #orderValues}，即两者的第一条记录
     *
     * @param o 对比 OrderByValue
     * @return -1 0 1
     */
    @Override
    public int compareTo(final OrderByValue o) {
        for (int i = 0; i < orderByItems.size(); i++) {
            OrderItem thisOrderBy = orderByItems.get(i);
            int result = ResultSetUtil.compareTo(orderValues.get(i), o.orderValues.get(i), thi
            if (0 != result) {
                return result;
            }
        }
        return 0;
```

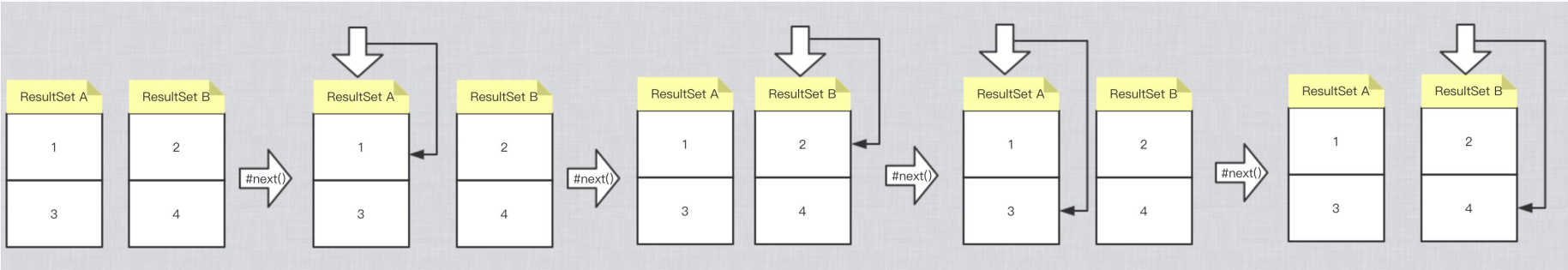
```
        }
    }
```

- 调用 `OrderByValue#next()` 方法时，获得其对应结果集**排在第一条**的记录，通过 `#getOrderValues()` 计算该记录的排序字段值。这样**两个OrderByValue** 通过 `#compareTo()` 方法可以比较**两个结果集**的第一条记录。

- `if (orderByValue.next()) {` 处，调用 `OrderByValue#next()` 后，添加到 PriorityQueue。因此，`orderByValuesQueue.peek().getResultSet()` 能够获得多个 ResultSet 中排在第一的。

3.2 #next()

通过调用 `OrderByStreamResultSetMerger#next()` 不断获得当前排在第一的记录。`#next()` 每次调用后，实际做的是当前 ResultSet 的替换，以及当前的 ResultSet 的记录指向下一条。这样说起来可能比较绕，我们来看一张图：



- 白色向下箭头：OrderByStreamResultSetMerger 对 ResultSet 的指向。
- 黑色箭头：ResultSet 对当前记录的指向。

- ps：这块如果分享的不清晰让您费劲，十分抱歉。欢迎加我微信（wangwenbin-server）交流下，这样我也可以优化表述。

```
// OrderByStreamResultSetMerger.java
@Override
public boolean next() throws SQLException {
    if (orderByValuesQueue.isEmpty()) {
        return false;
    }
    if (isFirstNext) {
        isFirstNext = false;
        return true;
    }
    // 移除上一次获得的 ResultSet
    OrderByValue firstOrderByValue = orderByValuesQueue.poll();
    // 如果上一次获得的 ResultSet还有下一条记录，继续添加到 排序值对象队列
    if (firstOrderByValue.next()) {
        orderByValuesQueue.offer(firstOrderByValue);
    }
    if (orderByValuesQueue.isEmpty()) {
        return false;
    }
    // 设置当前 ResultSet
    setCurrentResultSet(orderByValuesQueue.peek().getResultSet());
    return true;
}
```

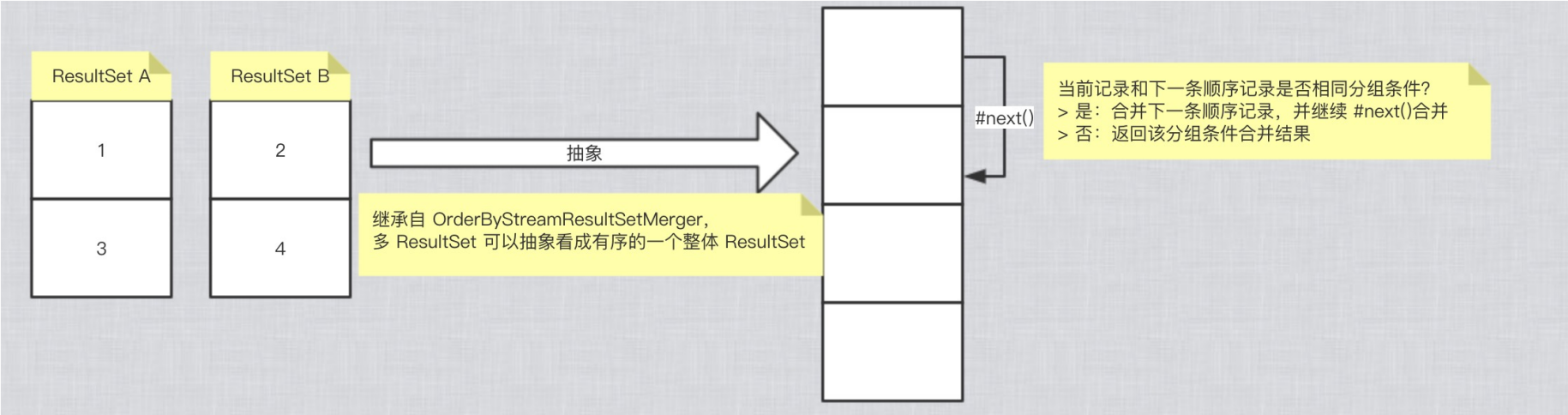
- `orderByValuesQueue.poll()` 移除上一次获得的 ResultSet。为什么不能 `#setCurrentResultSet()` 就移除呢？如果该 ResultSet 里面还存在下一条记录，需要继续参加**排序**。而判断是否有下一条，需要调用 `ResultSet#next()` 方法，这会导致 ResultSet 指向了下一条记录。因而 `orderByValuesQueue.poll()` 调用是**后置**的。
- `isFirstNext` 变量那的判断看着是不是很“灵异”？因为 `#orderResultSetsToQueue()` 处设置了第一次的 ResultSet。如果不加这个标记，会导致第一条记录“不见”了。
- 通过不断的 `Queue#poll()`、`Queue#offset()` 实现排序。巧妙！仿佛 Get 新技能了：

```
// 移除上一次获得的 ResultSet
OrderByValue firstOrderByValue = orderByValuesQueue.poll();
// 如果上一次获得的 ResultSet还有下一条记录，继续添加到 排序值对象队列
if (firstOrderByValue.next()) {
  orderByValuesQueue.offer(firstOrderByValue);
}
```

TODO Stream

4. GroupByStreamResultSetMerger

GroupByStreamResultSetMerger，基于 **Stream** 方式分组归并结果集实现。 它继承自 OrderByStreamResultSetMerger，在**排序**的逻辑上，实现分组功能。实现原理也较为简单：




```
public final class GroupByStreamResultSetMerger extends OrderByStreamResultSetMerger {
    /**
     * 查询列名与位置映射
     */
    private final Map<String, Integer> labelAndIndexMap;
    /**
     * Select SQL语句对象
     */
    private final SelectStatement selectStatement;
    /**
     * 当前结果记录
     */
    private final List<Object> currentRow;
    /**
     * 下一条结果记录 GROUP BY 条件
     */
    private List<?> currentGroupByValues;
```

分享

```
    public GroupByStreamResultSetMerger(
            final Map<String, Integer> labelAndIndexMap, final List<ResultSet> resultSets, final Selec
        super(resultSets, selectStatement.getOrderByItems(), nullOrderType);
        this.labelAndIndexMap = labelAndIndexMap;
        this.selectStatement = selectStatement;
        currentRow = new ArrayList<>(labelAndIndexMap.size());
        // 初始化下一条结果记录 GROUP BY 条件
        currentGroupByValues = getOrderByValuesQueue().isEmpty() ? Collections.emptyList() : new Group
    }


    @Override
    public Object getValue(final int columnIndex, final Class<?> type) throws SQLException {
        return currentRow.get(columnIndex - 1);
    }
    @Override
    public Object getValue(final String columnLabel, final Class<?> type) throws SQLException {
        Preconditions.checkState(labelAndIndexMap.containsKey(columnLabel), String.format("Can't find
        return currentRow.get(labelAndIndexMap.get(columnLabel) - 1);
    }

    @Override
    public Object getCalendarValue(final int columnIndex, final Class<?> type, final Calendar calendar
        return currentRow.get(columnIndex - 1);
    }
    @Override
    public Object getCalendarValue(final String columnLabel, final Class<?> type, final Calendar calen
        Preconditions.checkState(labelAndIndexMap.containsKey(columnLabel), String.format("Can't find
        return currentRow.get(labelAndIndexMap.get(columnLabel) - 1);
```

```
    }
}
```
```
* `currentRow` 为当前结果记录，使用 `#getValue()`、`#getCalendarValue()` 方法获得当前结果记录的查询列值。
* `currentGroupByValues` 为**下一条**结果记录 GROUP BY 条件，通过 GroupByValue 生成：
    ```Java
    public final class GroupByValue {

        /**
         * 分组条件值数组
         */
        private final List<?> groupValues;

        public GroupByValue(final ResultSet resultSet, final List<OrderItem> groupByItems) throws SQLE
            groupValues = getGroupByValues(resultSet, groupByItems);
        }

        /**
         * 获得分组条件值数组
         *
         * @param resultSet 结果集（单分片）
         * @param groupByItems 分组列
         * @return 分组条件值数组
         * @throws SQLException 当结果集关闭
         */
        private List<?> getGroupByValues(final ResultSet resultSet, final List<OrderItem> groupByItems
            List<Object> result = new ArrayList<>(groupByItems.size());
            for (OrderItem each : groupByItems) {
                result.add(resultSet.getObject(each.getIndex())); // 从结果集获得每个分组条件的值
```
```

```
            }
            return result;
        }
    }
```

```
 * 例如，`GROUP BY user_id, order_status` 返回的某条记录结果为 `userId = 1, order_status = 3`，对应的 `gro
```

- GroupByStreamResultSetMerger 在创建时，当前结果记录**实际未合并**，需要先调用 `#next()` ，在使用 `#getValue()` 等方法获取值，这个和 OrderByStreamResultSetMerger 不同，可能是个 BUG。

4.1 AggregationUnit

AggregationUnit，归并计算单元接口，有两个接口方法：

- `#merge()` ：归并聚合值
- `#getResult()` ：获取计算结果

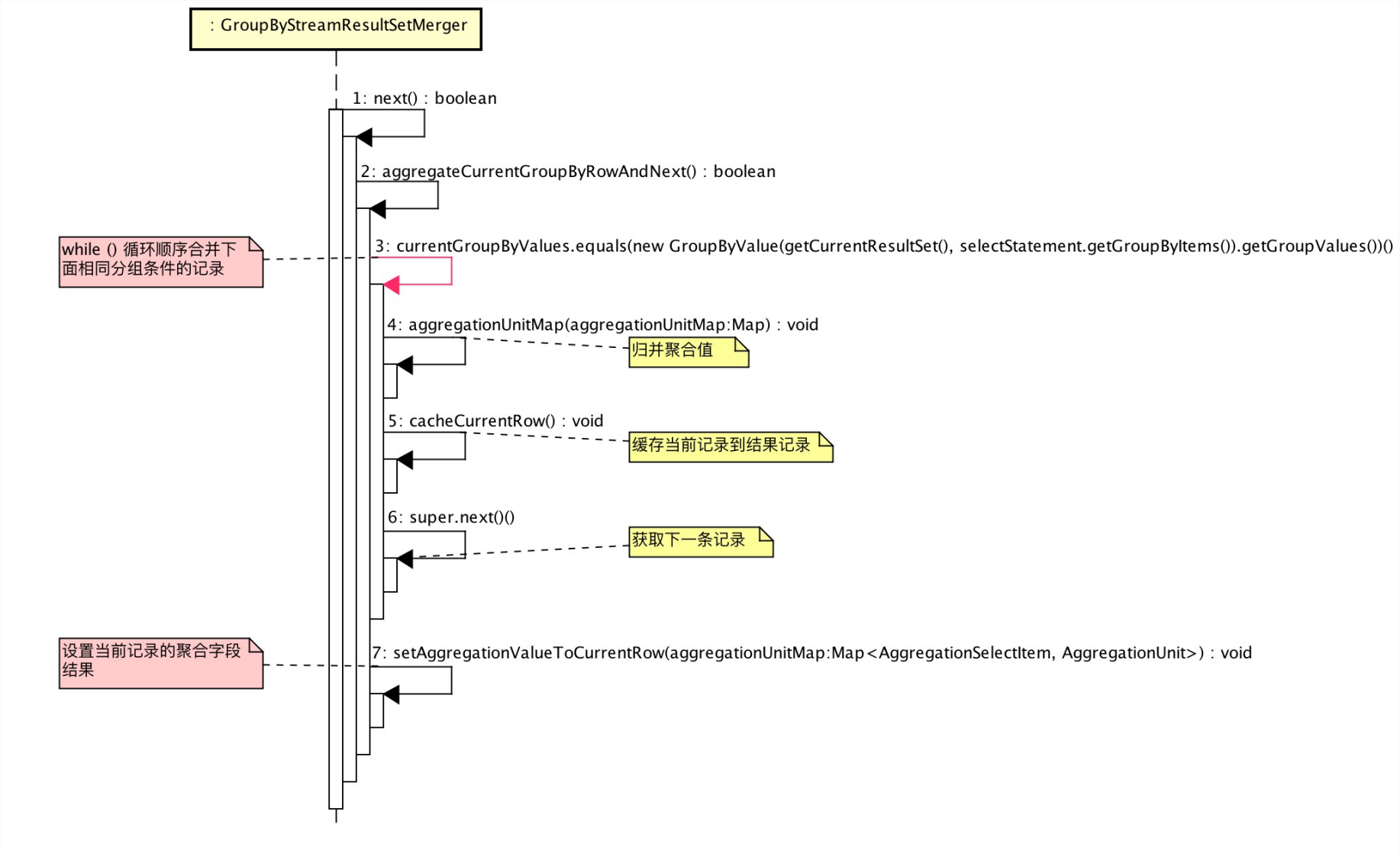
一共有三个实现类：

- AccumulationAggregationUnit：累加聚合单元，解决 COUNT、SUM 聚合列
- ComparableAggregationUnit：比较聚合单元，解决 MAX、MIN 聚合列
- AverageAggregationUnit：平均值聚合单元，解决 AVG 聚合列

实现都比较简单易懂，直接点击链接查看源码，我们就不浪费篇幅贴代码啦。

4.2 #next()

我们先看看大体的调用流程：



😺 看起来代码比较多，逻辑其实比较清晰，对照着顺序图顺序往下读即可。

```
// GroupByStreamResultSetMerger.java
@Override
public boolean next() throws SQLException {
    // 清除当前结果记录
    currentRow.clear();
    if (getOrderByValuesQueue().isEmpty()) {
        return false;
    }
    //
    if (isFirstNext()) {
        super.next();
    }
    // 顺序合并下面相同分组条件的记录
    if (aggregateCurrentGroupByRowAndNext()) {
        // 生成下一条结果记录 GROUP BY 条件
        currentGroupByValues = new GroupByValue(getCurrentResultSet(), selectStatement.getGroupByItems(
    }
    return true;
}
private boolean aggregateCurrentGroupByRowAndNext() throws SQLException {
    boolean result = false;
    // 生成计算单元
    Map<AggregationSelectItem, AggregationUnit> aggregationUnitMap = Maps.toMap(selectStatement.getAggr

        @Override
        public AggregationUnit apply(final AggregationSelectItem input) {
            return AggregationUnitFactory.create(input.getType());
        }
    });
```

```
        // 循环顺序合并下面相同分组条件的记录
        while (currentGroupByValues.equals(new GroupByValue(getCurrentResultSet(), selectStatement.getGroup
            // 归并聚合值
            aggregate(aggregationUnitMap);
            // 缓存当前记录到结果记录
            cacheCurrentRow();
            // 获取下一条记录
            result = super.next();
            if (!result) {
                break;
            }
        }
        // 设置当前记录的聚合字段结果
        setAggregationValueToCurrentRow(aggregationUnitMap);
        return result;
    }

    private void aggregate(final Map<AggregationSelectItem, AggregationUnit> aggregationUnitMap) throws SQ
        for (Entry<AggregationSelectItem, AggregationUnit> entry : aggregationUnitMap.entrySet()) {
            List<Comparable<?>> values = new ArrayList<>(2);
            if (entry.getKey().getDerivedAggregationSelectItems().isEmpty()) { // SUM/COUNT/MAX/MIN 聚合列
                values.add(getAggregationValue(entry.getKey()));
            } else {
                for (AggregationSelectItem each : entry.getKey().getDerivedAggregationSelectItems()) { // A
                    values.add(getAggregationValue(each));
                }
            }
            entry.getValue().merge(values);
        }
```

```
    }

    private void cacheCurrentRow() throws SQLException {
        for (int i = 0; i < getCurrentResultSet().getMetaData().getColumnCount(); i++) {
            currentRow.add(getCurrentResultSet().getObject(i + 1));
        }
    }

    private Comparable<?> getAggregationValue(final AggregationSelectItem aggregationSelectItem) throws SQ
        Object result = getCurrentResultSet().getObject(aggregationSelectItem.getIndex());
        Preconditions.checkState(null == result || result instanceof Comparable, "Aggregation value must im
        return (Comparable<?>) result;
    }

    private void setAggregationValueToCurrentRow(final Map<AggregationSelectItem, AggregationUnit> aggrega
        for (Entry<AggregationSelectItem, AggregationUnit> entry : aggregationUnitMap.entrySet()) {
            currentRow.set(entry.getKey().getIndex() - 1, entry.getValue().getResult()); // 获取计算结果
        }
    }
```

5. GroupByMemoryResultSetMerger

GroupByMemoryResultSetMerger，基于 **内存** 分组归并结果集实现。

6. IteratorStreamResultSetMerger

IteratorStreamResultSetMerger，基于 **Stream** 迭代归并结果集实现。

```
public final class IteratorStreamResultSetMerger extends AbstractStreamResultSetMerger {
    /**
     * ResultSet 数组迭代器
     */
    private final Iterator<ResultSet> resultSets;
    public IteratorStreamResultSetMerger(final List<ResultSet> resultSets) {
        this.resultSets = resultSets.iterator();
        // 设置当前 ResultSet，这样 #getValue() 能拿到记录
        setCurrentResultSet(this.resultSets.next());
    }
    @Override
    public boolean next() throws SQLException {
        // 当前 ResultSet 迭代下一条记录
        if (getCurrentResultSet().next()) {
            return true;
        }
        if (!resultSets.hasNext()) {
            return false;
        }
        // 获得下一个ResultSet， 设置当前 ResultSet
        setCurrentResultSet(resultSets.next());
        boolean hasNext = getCurrentResultSet().next();
        if (hasNext) {
            return true;
        }
        while (!hasNext && resultSets.hasNext()) {
            setCurrentResultSet(resultSets.next());
            hasNext = getCurrentResultSet().next();
        }
```

```
            return hasNext;
        }
    }
```

7. LimitDecoratorResultSetMerger

LimitDecoratorResultSetMerger，基于 **Decorator** 分页结果集归并实现。

666. 彩蛋

Sharding-JDBC

PREVIOUS:
« Sharding-JDBC 源码分析 —— JDBC 实现
NEXT:
» Sharding-JDBC 源码分析 —— SQL 执行