获取固定时间窗口内主力资金分析
——以平安银行为例

李思晴　12212964
杨月祺　12211056

一、 问题重述

在股市中，主力资金对股票的涨跌具有显著影响。主力资金的流向可以反映出市场的供求关系及投资者的倾向。然而，由于交易所未公开实时及盘后的个股资金流向数据，且每个订单并不包含交易者的唯一标识（ID），使得主力资金信息的直接获取变得困难。因此，现有的股票行情软件通过计算超大单、大单的指标，间接反映主力资金的情况。

本项目旨在处理逐笔成交数据，识别主动买卖情况，合并主动订单后获取成交订单单型。首先筛选成交订单，通过比较买卖双方的交易时间确定主动买单或卖单。然后，在给定时间范围内去重合并相同索引的主动单，计算成交量和成交额总和。最后，根据成交量、成交额及流通盘占比，我们将判断哪些成交属于大单或超大单。

通过以上步骤，我们将能够更准确地分析主力资金流向，从而帮助市场参与者更好地理解市场动态以及潜在的投资机会，提供有价值的决策支持。

二、 问题具体分析

本项目旨在对于我们感兴趣的股票（平安银行，证券代码为 000001），选取上午 9:30 到 11:30 以及下午 13:00 到 15:00 两个时段，共四个小时的逐笔委托以及逐笔成交数据。对每一支股票(`SecurityID`)的逐笔成交数据,筛选为成交的记录(`ExecType=F`)。并分别依据其 `BidApplSeqNum` 和 `OfferApplSeqNum`索引在逐笔委托中找到对应的`TransactTime`，判断该笔成交是主动买还是主动卖（`ActiveOrder = B`或`ActiveOrder = S`)。把给定时间范围内的所有 `BidApplSeqNum` 和 `OfferApplSeqNum` 相同的主动单进行去重合并，合并方式为将`BidApplSeqNum` 或 `OfferApplSeqNum` 相同的主动单成交量和成交额加和计算。最后根据合并后的成交量、成交额以及流通盘，按照规则判断是否为大单或超大单。

任务需要我们在给定单日逐笔委托和逐笔成交数据的情况下，按时间范围配置参数 T_{window}（该时间以 `tradetime` 字段为依据)。当用户输入时间范围 T_{window} 的值后，输出如下数据，包括每 T_{window} 时间内的：

主力净流入，主力流入，主力流出，超大买单成交量，超大买单成交额，超大卖单成交量，超大卖单成交额，大买单成交量，大买单成交额，大卖单成交量，大卖单成交

额，中买单成交量，中买单成交额，中卖单成交量，中卖单成交额，小买单成交量，小买单成交额，小卖单成交量，小卖单成交额。

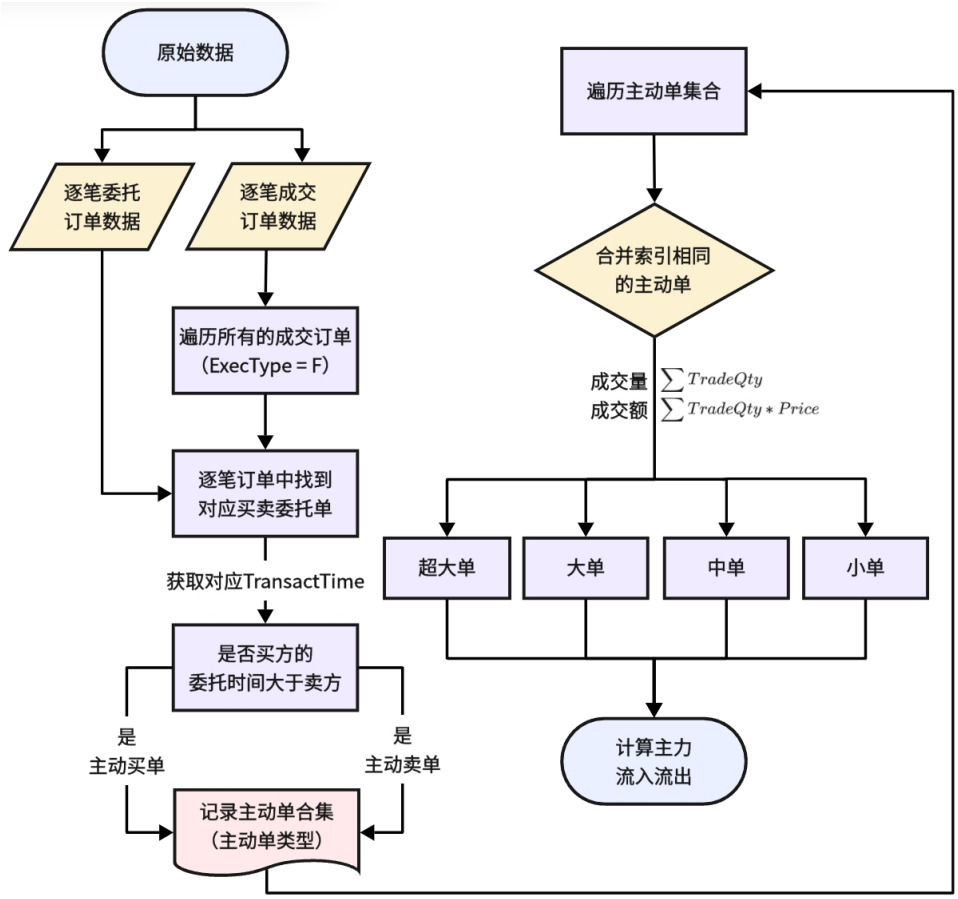

图 1　任务总流程图

三、 难点分析

在本项目的实施过程中，我们遇到了几个关键的技术难点，这些难点对于项目的顺利进行至关重要。以下是我们识别的主要难点及其分析：

1. 主动方判断的准确性

难点在于如何准确地根据逐笔委托订单数据判断交易的主动方。由于交易数据中没有直接标识交易主动方的信息，我们需要通过比较买卖双方的交易时间来间接判断。这种方法虽然可行，但对数据处理的准确性和实时性提出了较高要求。

2. 数据格式的标准化输出

另一个难点是按照统一的格式输出数据，并确保输出结果包含所有必要的表头信息。由于数据来源多样，格式不一，我们需要设计一个灵活且健壮的数据转换流程，以确保输出数据的一致性和可读性。

3. 时间窗口的精确划分

精确地根据给定的时间范围配置参数 T_{window}，并确保所有交易都能被正确地划分到相应的时间窗口内，是一个技术挑战。这不仅需要精确的时间计算，还需要高效的算法来处理可能的时间窗口重叠和边界情况。

与此同时，由于最终输出的时间时间窗口数据是字符串格式，而在 MapReduce 的数据传输过程中，字符串的序列化和反序列化比整数更复杂，通常会消耗更多的时间。同样的，字符串的比较涉及逐字符的比较，对于程序的效率提出了挑战。

4. 大数据集的处理效率

面对庞大的数据集，如何提高数据处理的效率是一个重要的难点。我们需要在数据筛选、过滤和合并的过程中，采用高效的算法和数据结构，以减少 I/O 操作和计算时间。

四、 整体技术方案

根据原本的任务流程，我们需要通过原数据集中的逐笔成交数据的 `ApplSeqNum` 以及 `BidApplSeqNum` 和 `OfferApplSeqNum` 索引，对应逐笔委托数据中的 `ApplSeqNum` 找到对应的 `TransactTime`。通过买卖方的委托时间对比判断是主动买单或主动卖单。

在完成任务的过程中，我们发现在逐笔委托数据中，相同股票（`SecurityID`）的订单索引是随时间单调增的。换句话说，我们可以通过比较一条成交订单的 `BidApplSeqNum` 和 `OfferApplSeqNum` 的大小关系来判断主动单的类型。具体而言，当 `OfferApplSeqNum` 大于 `BidApplSeqNum` 时，该笔订单被认定为主动卖单；反之，则为主动买单。这一判断依据简化了原有的复杂过程，使得我们可以更高效地识别主动单类型。

基于上述分析，在优化后的任务中，我们仅需关注逐笔成交数据。通过对成交数据中的 `BidApplSeqNum` 和 `OfferApplSeqNum` 进行比较，我们能够直接判断成交订单的主动性，从而去重合并给定时间范围内所有相同的主动单。这一方法显著提高了处理效率，并降低了对逐笔委托数据的依赖。通过优化后的方法，我们简化了主动买单与主动卖单的识别过程，使得数据处理更加高效。

4.1 原订单数据的筛选

总体任务的原数据如下：

- 逐笔委托数据：
  - 文件名：am_hq_order_spot.txt，大小：1.66GB
  - 文件名：pm_hq_order_spot.txt，大小：1.06GB
- 逐笔成交数据：
  - 文件名：am_hq_trade_spot.txt，大小：1.34GB
  - 文件名：pm_hq_trade_spot.txt，大小：986.6 MB

由此可见，源数据较大，需要对原数据进行筛选和过滤，以选出所需的行和列。根据后续任务的需要，我们对逐笔成交数据进行了以下筛选：

- 选择 `SecurityID = 000001`、`ExecType = F` 以及 `tradetime` 在 `20190102093000000` 到 `20190102113000000` 以及 `20190102130000000` 到 `20190102150000000` 之间的行。
- 选择的列包括 `BidApplSeqNum`、`OfferApplSeqNum`、`Price`、`TradeQty`、`ExecType` 和 `tradetime`。

4.2 主动买单和主动卖单的识别

在筛选出的订单中，代码的这一部分会判断每笔交易是买单还是卖单。这是通过比较买方索引 `BidApplSeqNum` 和卖方索引 `OfferApplSeqNum` 的大小来实现的。如果买方索引大于或等于卖方索引，则标记为买单 B；如果小于，则标记为卖单 S。这个判断基于交易索引的大小关系，反映了交易的主动方。

4.3 所属时间窗口的判断

时间窗口的大小记为 `windowSize`（以秒为单位）。例如，如果 `windowSize` 设置为 1200 秒（即 20 分钟），则每 20 分钟划分为一个时间窗口。我们计算交易时间与 9:30 或 13:00 的秒数差，并将其除以时间窗口大小，然后乘以时间窗口大小，得到该交易所在的时间窗口的起始时间（`windowStart`）。同样可以得到所在时间窗口的结束时间（`windowEnd`）。

4.4 按相同索引合并交易量和交易额

合并所属时间窗口和索引相同的交易记录的交易量和交易额，得到总交易量 `totalTradeQty`和总交易额`totalTradeValue`。再计算流通盘占比，依据其和总交易量和总交易额判断类型：

类别	交易量	交易额	流通盘占比
超大单 (super)	$\geq 200,000$	$\geq 1,000,000$	$\geq 0.3\%$
大单 (big)	$60,000 \leq x < 200,000$	$300,000 \leq y < 1,000,000$	$0.1\% \leq z < 0.3\%$
中单 (middle)	$10,000 \leq x < 60,000$	$50,000 \leq y < 300,000$	$0.017\% \leq z < 0.1\%$
小单 (small)	$< 10,000$	$< 50,000$	$< 0.017\%$

表 1　交易类别标准

4.5 给定时间窗口内订单信息输出

合并相同时间窗口内的各单型的交易量和交易额，记录数据后先输出表头，然后按顺序输出主力净流入`mainNetInflow`, 主力流入`mainInflow`, 主力流出`mainOutflow`, 超大买单成交量`totalSuperBuyQty`，超大买单成交额`totalSuperBuyValue`，超大卖单成交量`totalSuperSellQty`，超大卖单成交额`totalSuperSellValue`，大买单成交量`totalBigBuyQty`,大买单成交额`totalBigBuyValue`,大卖单成交量`totalBigSellQty`，大卖单成交额`totalBigSellValue`，中买单成交量`totalMidBuyQty`，中买单成交额`totalMidBuyValue`,中卖单成交量`totalMidSellQty`,中卖单成交额`totalMidSellValue`，小买单成交量`totalSmallBuyQty`，小买单成交额`totalSmallBuyValue`，小卖单成交量`totalSmallSellQty`,小卖单成交额`totalSmallSellValue`,时间区间`time_interval`

五、 代码模块化设计思路

5.1 代码模块建立过程

首先，我们将整体任务分为了四个不同的 Mapreduce 模块，如下所示：

- **job1** 完成逐笔成交订单的筛选。
- **job2** 区分买单卖单，按索引和时间窗口合并。
- **job3** 区分超大单、大单、中单和小单。
- **job4** 整理同一时间窗口里的数据，得到最终输出。

为了减少数据传输的时间，我们首先通过将 4 个 job 合并成 2 个，总耗时减少 40s 左右，总计用时达到 1 分 21 秒，如下所示：

- **job1** 完成逐笔成交订单的筛选，买卖单的区分，得到时间窗口，按索引合并交易量和交易额，区分单型
- **job2** 按单型合并交易量和交易额，得到最终输出

5.2 最终代码模块的建立

为了提高代码运行效率，减少不必要的数据传输时间，我们最终将任务简化为一个 Mapreduce 模块任务，总计用时减少到 51 秒左右流程图可见图 2。

5.2.1 Mapper 过程建立

为了高效地处理逐笔成交数据，我们对上午和下午的数据分别调用不同 mapper，分别是 AmTradeMapper 和 PMTradeMapper，而不是直接将上下午的逐笔成交订单数据合并为大表，是为了降低后续 mapper 过程中的计算复杂度。`AmTradeMapper`通过以下步骤处理股票交易数据：

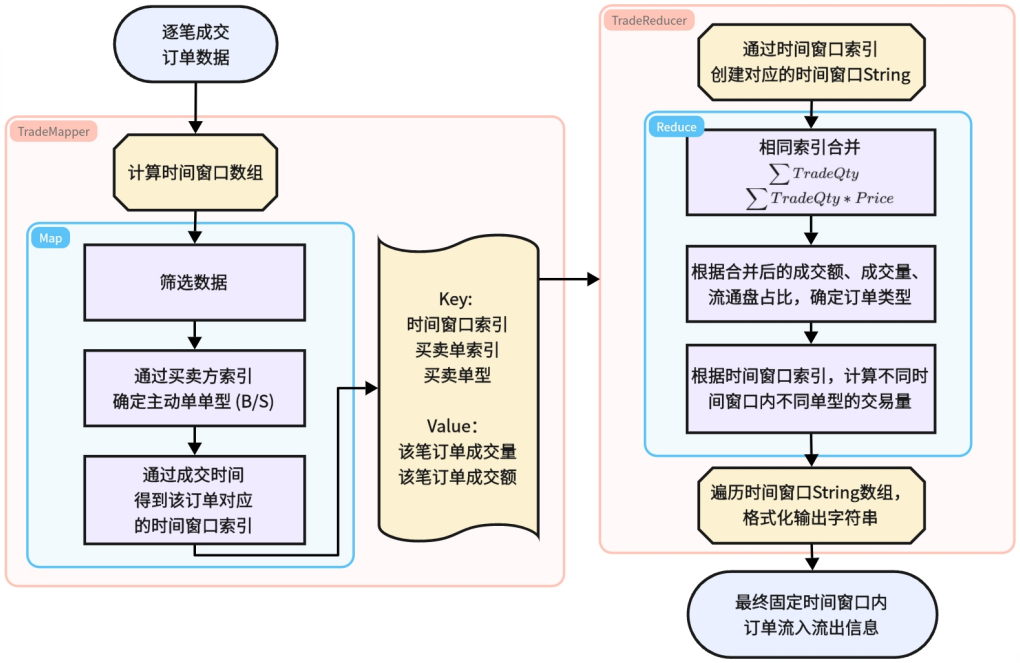

图 2　最终代码模块化思路

- 初始化时间窗口。
- 读取和过滤输入记录。
- 提取必要字段，并根据买卖方向进行分类。
- 计算并输出每个时间窗口的交易数据。

具体代码叙述如下：

1. 初始化和设置

- 常量定义：
  - `EXPECTED_FIELDS`：预期的字段数量（16 个字段）。
  - `START_TIME`和`END_TIME`：指定的交易时间范围（从 9:30 到 11:30）。
  - `TARGET_SECURITY_ID`：目标股票代码（例如“000001”）。
  - `WINDOW_SIZE`：每个时间窗口的大小（600 秒）。
  - `TRADEDATE`：处理的交易日期（例如“20190102”）。
- 时间窗口初始化：在`setup`方法中，创建时间窗口列表（`timeWindows`）。从 34200 秒（9:30 AM）开始，直到 41400 秒（11:30 AM），以`WINDOW_SIZE`为步长创建时间窗口，并将其格式化为字符串添加到列表中。处理剩余时间，如果不足一个窗口，也将其添加到列表中。

2. 数据映射

- `map`方法接收输入数据，处理每一行记录：
  - 将输入行按制表符分割为字段数组（`fields`）。

  - 获取第 15 个字段（execType），第 9 个字段（securityID），以及第 16 个字段（tradetime）。
- 过滤条件：如果execType不是“F”或securityID不是目标股票代码，或者tradetime不在指定的时间范围内，则跳过该记录。

3. 提取和输出数据

- 提取所需字段，包括：
  - 买方索引（bidApplSeqNum）。
  - 卖方索引（offerApplSeqNum）。
  - 成交价格（price）。
  - 成交量（tradeQty）。
- 确定买卖方向：通过比较买方和卖方索引来确定是买单（“B”）还是卖单（“S”）。
- 计算时间索引：将交易时间（tradetime）转换为对应的时间窗口索引（timeIndex），通过调用getTimeIndex方法。使用以上方法来获取时间窗口，而不是将每一条订单信息tradetime直接转换为时间窗口字符串，不仅可以减少处理时间，将计算每一条时间数据的时间窗口转换成通过数组查询访问到对应的时间窗口，而且可以减少后续 Reducer 里面通过时间窗口来合并订单的比对时间（对比整数所需的时间比对比字符串所需的时间要短得多），从而可以显著提高代码运行的效率。代码如附录1.1所示。
- 输出中间结果：创建Text类型的outputKey和outputValue。根据买单或卖单的类型，设置outputKey（包括时间索引和索引类型）和outputValue（包括成交量和成交额）。

4. 时间索引计算在getTimeIndex方法中：

- 从tradetime提取小时、分钟和秒，计算总秒数。
- 通过与基准时间（34200 秒，即 9:30 AM）计算差值，得到时间窗口的索引。

5.2.2 Reducer 过程建立

TradeReducer通过以下方式高效地聚合交易数据：

- 定义特定的时间窗口。
- 根据分类汇总交易。
- 以结构化的格式输出结果以供进一步分析。

具体代码叙述如下：

1. 初始化和设置

- 常量定义：

  - CIRCULATING_SHARES：流通股数。
  - WINDOW_SIZE：每个时间窗口的持续时间（单位：秒）。
  - TRADEDATE：处理的交易日期（例如“20190102”）。
- 数组初始化：在setup方法中，计算时间窗口的数量（am_num），这是通过将总时间（7200 秒）除以WINDOW_SIZE得到的。初始化一个TotalSumObject数组，用于存储每个时间窗口的聚合结果。

2. 创建时间窗口

使用两个循环创建交易数据的时间窗口：

- 第一个循环处理从 9:30 AM 到 11:00 AM 的时间段。
- 第二个循环处理从 1:30 PM 到 3:00 PM 的时间段。

对于每个窗口，设置窗口开始和结束时间的字符串表示。代码如附录1.2所示。

3. 数据归约

- reduce方法处理每个键（包括时间索引和交易类型）的输入值。
- 聚合：
  - totalTradeQty：窗口内总的成交量。
  - totalTradeValue：窗口内总的成交额。

4. 交易分类

根据总成交量、总成交额和计算的流通盘占比将交易分类：

- **超大单**：高阈值的成交量和成交额。
- **大单**：中等阈值。
- **中单**：较低的阈值。
- **小单**：其他交易。

根据交易是买入还是卖出，更新TotalSumObject数组中相应的字段。

5. 输出生成

在cleanup方法中，准备输出表头，并遍历TotalSumObject数组生成每个时间窗口的格式化输出字符串。计算：

- mainInflow：超大买单和大买单成交额的总和。
- mainOutflow：超大卖单和大卖单成交额的总和。
- mainNetInflow：流入与流出的差值。

最后，将聚合结果写入上下文。

六、 主力资金分布可视化

主力资金分布可视化的目的是为了帮助投资者和分析师更直观地理解市场资金流向的动态变化。通过可视化方法，我们可以有效地展示不同时间段内的主力资金流入、流出及其净流向。这对于评估市场趋势、判断投资机会具有重要意义。

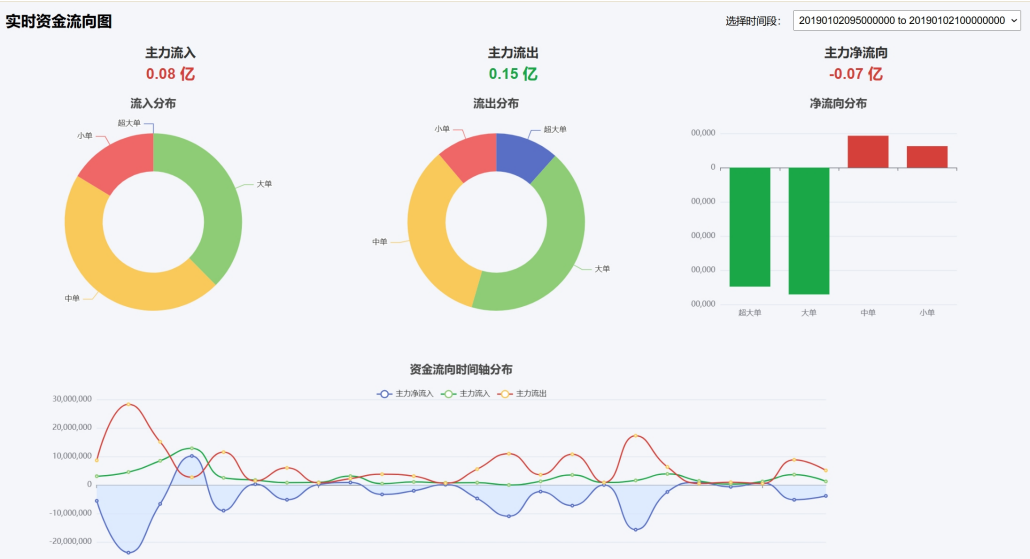

图 3　可视化界面

在本章节中，我们使用了 ECharts 库来创建交互式图表，展现主力资金的分布情况，界面如图 3所示。ECharts 是基于 JavaScript 的数据可视化图表库，提供直观，生动，可交互，可个性化定制的数据可视化图表。具体的可视化包括：

- **流入分布饼图：**该图表展示了在选定时间段内，超大单、大单、中单和小单的资金流入情况。通过该图，用户可以快速识别资金流入的主要来源和强度。
- **流出分布饼图：**与流入分布相对应，该图表展示了各类单的资金流出情况，帮助用户判断市场的卖出压力。
- **净流向柱状图：**该图通过柱状形式展示净流向的分布情况，直观地反映出不同类型单的资金流入与流出差值，为用户提供清晰的资金流向分析。
- **资金流向时间轴图：**该图表展示了在整个观察时间段内，各类资金的流入、流出及净流入的变化趋势。通过时间轴的形式，用户能够清晰地看到资金流向的变化，进而进行趋势分析。

通过这些可视化图表，用户可以实时监测市场状况，辅助决策过程，进而优化投资策略。这种数据驱动的分析方法，结合直观的图形展示，为用户提供了强大的工具，以应对复杂的市场动态。

七、 总结与讨论

在本项目的实施过程中，我们实现了一个高效的主力资金流向分析系统。以下是对项目成果的总结和讨论：

7.1 技术实现

我们通过深入分析逐笔成交数据和逐笔委托数据，开发了一套有效的算法来判断交易的主动方。此外，我们还设计了一个灵活的数据输出流程，确保了输出数据的标准化和一致性。

通过优化算法和采用并行处理技术，我们显著提高了数据处理的效率。项目最终实现了在较短时间内处理大规模数据集的目标，这对于实时分析股市资金流向具有重要意义。

7.2 项目意义

通过对于固定时间窗口主力资金信息的汇总，可以得到当如的主力资金分布，从而提高市场参与者对主力资金流向的理解，还为投资决策提供了有力的数据支持。通过分析主力资金的流入和流出，投资者可以更好地把握市场动态，发现潜在的投资机会。我们在这次的项目的完成过程中也在实际问题上实践了 hadoop、mobaxterm、JavaScript 的数据可视化等工具的使用，同时对 MapReduce 这一常用大数据处理流程有了更加深刻的理解。

尽管本项目取得了一定的成果，但仍有进一步改进和优化的空间。未来的工作可以集中在提高算法的准确性、优化数据处理流程、以及扩展系统的功能，例如增加更多的数据分析维度和提供更直观的数据可视化支持。

附录 A　文件列表

文件名	功能描述
TimeWindowProcessor.java	Mapper 中时间窗口处理相关方法
FormattingResults.java	Reducer 中格式化输出的 cleanup 方法

1.1　TimeWindowProcessor

```
1    /**
2     * 初始化时间窗口的方法，根据交易上下文设置时间窗口。
3     *
```

```
4       * @param context 调用该方法的上下文。
5       * @throws IOException 如果发生I/O错误。
6       * @throws InterruptedException 如果线程被中断。
7       */
8      protected void setup(Context context) throws IOException,
    InterruptedException {
9          super.setup(context);
10
11         long startSeconds = 34200; // 开始时间（单位：秒，09
    :30:00）
12         long endSeconds = 41400;   // 结束时间（单位：秒，11
    :30:00）
13
14         // 填充前半部分时间窗口
15         long currentStart = startSeconds;
16         while (currentStart + WINDOW_SIZE <= endSeconds) {
17             long windowStart = currentStart;
18             long windowEnd = currentStart + WINDOW_SIZE;
19
20             // 确保窗口结束时间不超出设定的结束时间
21             timeWindows.add(formatWindow(TRADEDATE,
    windowStart, windowEnd));
22             currentStart += WINDOW_SIZE; // 增加到下一个窗口开
    始时间
23         }
24
25         // 处理剩余时间（不足一个窗口）
26         if (currentStart < endSeconds) {
27             timeWindows.add(formatWindow(TRADEDATE,
    currentStart, endSeconds));
28         }
29     }
30
31     /**
32      * 计算给定交易时间的时间索引，基于窗口大小。
```

```
33      *
34      * @param tradetime 交易时间（单位：毫秒，自1970年1月1日起）。
35      * @param windowSize 每个时间窗口的大小（单位：秒）。
36      * @return 给定交易时间的时间窗口索引。
37      */
38     private long getTimeIndex(long tradetime, int windowSize) {
39         // 提取交易时间的HHMMSS部分
40         long timeStart = (tradetime / 1000) % 1000000; // 转换为秒并获取最后6位数字
41         long hour = timeStart / 10000; // 获取小时
42         long min = (timeStart / 100) % 100; // 获取分钟
43         long sec = timeStart % 100; // 获取秒
44         long totalSeconds = hour * 3600 + min * 60 + sec; // 计算总秒数
45
46         // 特殊处理时间点，例如15:00:00
47         if (totalSeconds == 41400) {
48             totalSeconds -= 1; // 调整以避免边界情况
49         }
50
51         long amSeconds = 34200; // 参考开始时间
52         long diff = totalSeconds - amSeconds; // 计算与开始时间的差值
53
54         return diff / windowSize; // 返回窗口索引
55     }
56
57     /**
58      * 将时间窗口格式化为字符串。
59      *
60      * @param tradeDate 交易日期（格式：YYYYMMDD）。
61      * @param startSeconds 窗口开始时间（单位：秒）。
62      * @param endSeconds 窗口结束时间（单位：秒）。
```

```
63      * @return 格式化后的时间窗口字符串（格式：
    YYYYMMDDHHMMSS_YYYYMMDDHHMMSS）。
64      */
65     private static String formatWindow(String tradeDate, long
    startSeconds, long endSeconds) {
66         return tradeDate + formatTime(startSeconds) + "_" +
    tradeDate + formatTime(endSeconds);
67     }
68
69     /**
70      * 将秒级时间转换为格式化时间字符串。
71      *
72      * @param timeInSeconds 秒级时间。
73      * @return 格式化后的时间字符串（格式：HHMMSS）。
74      */
75     private static String formatTime(long timeInSeconds) {
76         long hour = timeInSeconds / 3600; // 计算小时
77         long minute = (timeInSeconds % 3600) / 60; // 计算分钟
78         long second = timeInSeconds % 60; // 计算秒
79         return String.format("%02d%02d%02d", hour, minute,
    second); // 格式化为HHMMSS
80     }
```

1.2 FormattingResults

```
1      // cleanup方法在任务结束时执行，用于输出结果
2      protected void cleanup(Context context) throws IOException
    , InterruptedException {
3          // 输出表头
4          String header = "主力净流入,主力流入,主力流出,超大买单
    成交量,超大买单成交额,超大卖单成交量," +
5                  "超大卖单成交额,大买单成交量,大买单成交额,大卖
    单成交量,大卖单成交额,中买单成交量," +
6                  "中买单成交额,中卖单成交量,中卖单成交额,小买单
    成交量,小买单成交额,小卖单成交量," +
```

```
                "小卖单成交额,时间区间";
        // 遍历数组，输出每个时间窗口的统计结果
        context.write(new Text(header), NullWritable.get());
        for (int i = 0;i < array.length;i++) {
            String time_interval = array[i].getStr();
            long totalSuperBuyQty = array[i].
getTotalSuperBuyQty();
            double totalSuperBuyValue = array[i].
getTotalSuperBuyValue();
            long totalSuperSellQty = array[i].
getTotalSuperSellQty();
            double totalSuperSellValue = array[i].
getTotalSuperSellValue();
            long totalBigBuyQty = array[i].getTotalBigBuyQty()
;
            double totalBigBuyValue = array[i].
getTotalBigBuyValue();
            long totalBigSellQty = array[i].getTotalBigSellQty
();
            double totalBigSellValue = array[i].
getTotalBigSellValue();
            long totalMidBuyQty = array[i].getTotalMidBuyQty()
;
            double totalMidBuyValue = array[i].
getTotalMidBuyValue();
            long totalMidSellQty = array[i].getTotalMidSellQty
();
            double totalMidSellValue = array[i].
getTotalMidSellValue();
            long totalSmallBuyQty = array[i].
getTotalSmallBuyQty();
            double totalSmallBuyValue = array[i].
getTotalSmallBuyValue();
            long totalSmallSellQty = array[i].
getTotalSmallSellQty();
```

```
27          double totalSmallSellValue = array[i].
    getTotalSmallSellValue();
28          double mainInflow = totalSuperBuyValue +
    totalBigBuyValue; //主力流入
29          double mainOutflow = totalSuperSellValue +
    totalBigSellValue; //主力流出
30          double mainNetInflow = mainInflow - mainOutflow;
    //主力净流入
31
32          String outputLine = mainNetInflow + "," +
    mainInflow + "," + mainOutflow + ","
33                  + totalSuperBuyQty + "," +
    totalSuperBuyValue + "," + totalSuperSellQty +
34                  "," + totalSuperSellValue + "," +
    totalBigBuyQty + "," + totalBigBuyValue
35                  + "," + totalBigSellQty + "," +
    totalBigSellValue + "," + totalMidBuyQty
36                  + "," + totalMidBuyValue + "," +
    totalMidSellQty + "," + totalMidSellValue
37                  + "," + totalSmallBuyQty + "," +
    totalSmallBuyValue + "," + totalSmallSellQty
38                  + "," + totalSmallSellValue+ "," +
    time_interval;
39          context.write(new Text(outputLine), NullWritable.
    get());
40      }
41   }
```