STA321 Project Report

1. 问题描述

在股票市场中，主力资金意指集中了大量资金的交易者（机构），而主力资金的流向会对个股的涨跌产生重要影响，从直观上来看，跟随主力资金买卖赢面最大。因此计算股票市场中主力资金的流向极为重要。

虽然从交易所的数据并不能直接得到主力资金信息，但主力流入、主力流出、主力净流入等数据在一定程度上能够刻画个股大资金流入流出情况。因此本问题的核心在于根据深交所Level-2数据（逐笔委托、逐笔成交）提取主力净流入以及各单型交易量、交易额等数据。

2. 任务理解

在本任务中，共有两类原始数据：

逐笔委托数据表： 记录所有投资者在股票市场上委托的买入和卖出订单的详细信息。

逐笔成交数据表： 记录每一笔订单成交的详细信息，包括交易价格、交易数量、买卖方的帐户信息以及交易时间等等。

为了完成主力资金流向的分析和计算，任务可以拆解如下：

1. 逐笔成交数据的筛选与分类

- 筛选出成交记录（ExecType=F）。
- 依据买卖双方的委托索引（BidApplSeqNum 和 OfferApplSeqNum）在逐笔委托表中找到相应的交易时间（TransactTime）。
- 根据 TransactTime 判断每笔成交记录的主动性（主动买单或主动卖单）。

2. 根据时间窗口参数进行数据的去重与合并

- 根据输入的时间窗口参数 T_window，按时间段对逐笔成交数据进行分组统计。
- 将相同委托索引的主动成交记录合并，计算总成交量和总成交额。
- 合并后的成交量为 TradeQty 的加和，成交额为 TradeQty * Price 的加和。

3. 订单类型的分类

- 根据成交量、成交额、流通盘占比的标准，对每笔主动成交记录进行分类，判断其为小单、中单、大单或超大单。
- 分类依据：取成交量、成交额、流通盘占比三者中的最高标准作为单型的判定依据。
- 具体判定标准如图所示：

判断标准	成交量		成交额		流通盘占比	
	中小创	主板	中小创	主板	中小创	主板
超大单	≥10万股	≥20万股	≥50万元	≥100万元	≥0.3%	
大单	6–10万股	6–20万股	20–50万元	30–100万元	0.1%–0.3%	
中单	1–6万股		5–20万元	5–30万元	0.017%–0.1%	
小单	≤1万股		≤5万元		≤0.017%	

4. 主力资金的计算

- 通过订单类型分类数据，分别该时段计算主力流入、主力流出和主力净流入：
  - 主力流入 = 所有超大买单金额 + 所有大买单金额。
  - 主力流出 = 所有超大卖单金额 + 所有大卖单金额。
  - 主力净流入 = 主力流入 - 主力流出。

5. 输出结果

- 针对输入的一个交易日的股票数据，输出每个时间窗口内的以下指标：
  - 主力净流入、主力流入、主力流出；
  - 各类型订单的成交量与成交额（如超大买单成交量、超大卖单成交额等）

3. 难点分析

1.逐笔成交与逐笔委托的关联处理：

- 每笔成交记录都需要通过委托索引（BidApplSeqNum 和 OfferApplSeqNum）在逐笔委托表中找到对应的 TransactTime，这涉及多表关联操作，计算效率会显著降低。

2.主动成交的判断：这需要多表关联后逐条比较买卖双方的 TransactTime，这种逐条比较在大规模数据下可能造成较大的计算负担。

3.输出结果的完整性和一致性

- 在并行计算后，需确保输出的各项指标（如成交量、成交额等）在逻辑上和数据上都保持一致，防止因数据重复或遗漏导致计算结果不准确。
- 时间窗口划分后的指标汇总需要保持精确，需要注意边界条件的判断和时间格式的正确输出。

4. 整体技术方案

4.1 数据分析

正如难点分析部分所说，主动成交的判断按照project公示的方案完成，需要涉及到两个表的合并工作。沿着这条路分析，便是先通过hashmap存储每个委托单中单号和时间的关系，然后在接收成交记录时，通过hashmap寻回时间以判断委托单的主动单类型。这里有着hash函数的严重耗时，也会占用大量内存。

我们小组在观察数据时，发现了一个耐人寻味的备注，如图

买方委托索引	BidApplSeqNum	Int64		买方委托索引　从 1 开始计数，0 表示无对应委托
卖方委托索引	OfferApplSeqNum	Int64		卖方委托索引　从 1 开始计数，0 表示无对应委托

于是我们猜测委托索引是否和委托时间有相关性。

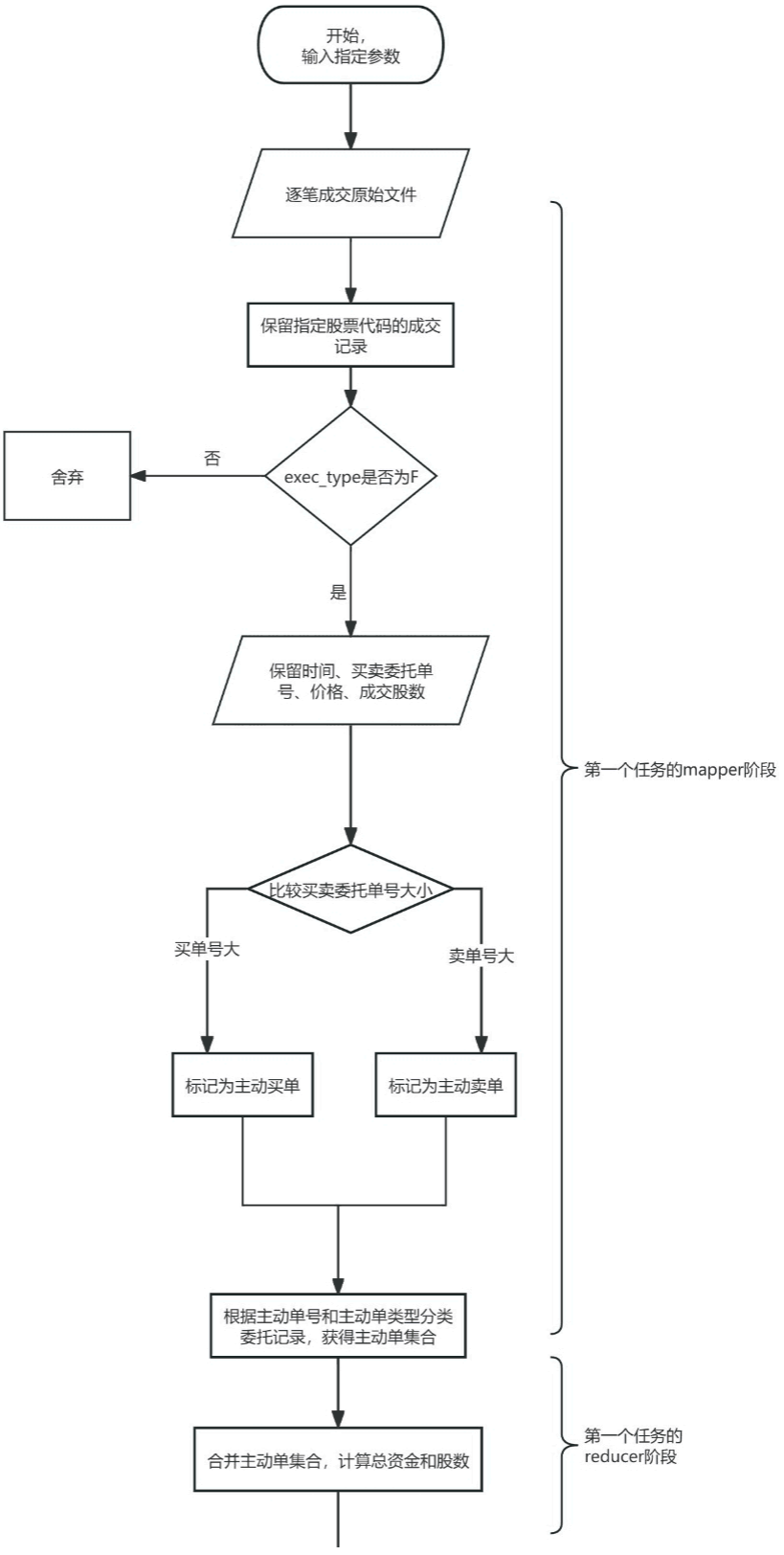
另一方面，主笔委托数据表中的主键，也被定义为证券集和委托索引共同决定。这意味着委托索引的生成逻辑是每个证券集下索引唯一自增的。这种特征很容易联想到这个委托索引和时间有相关。

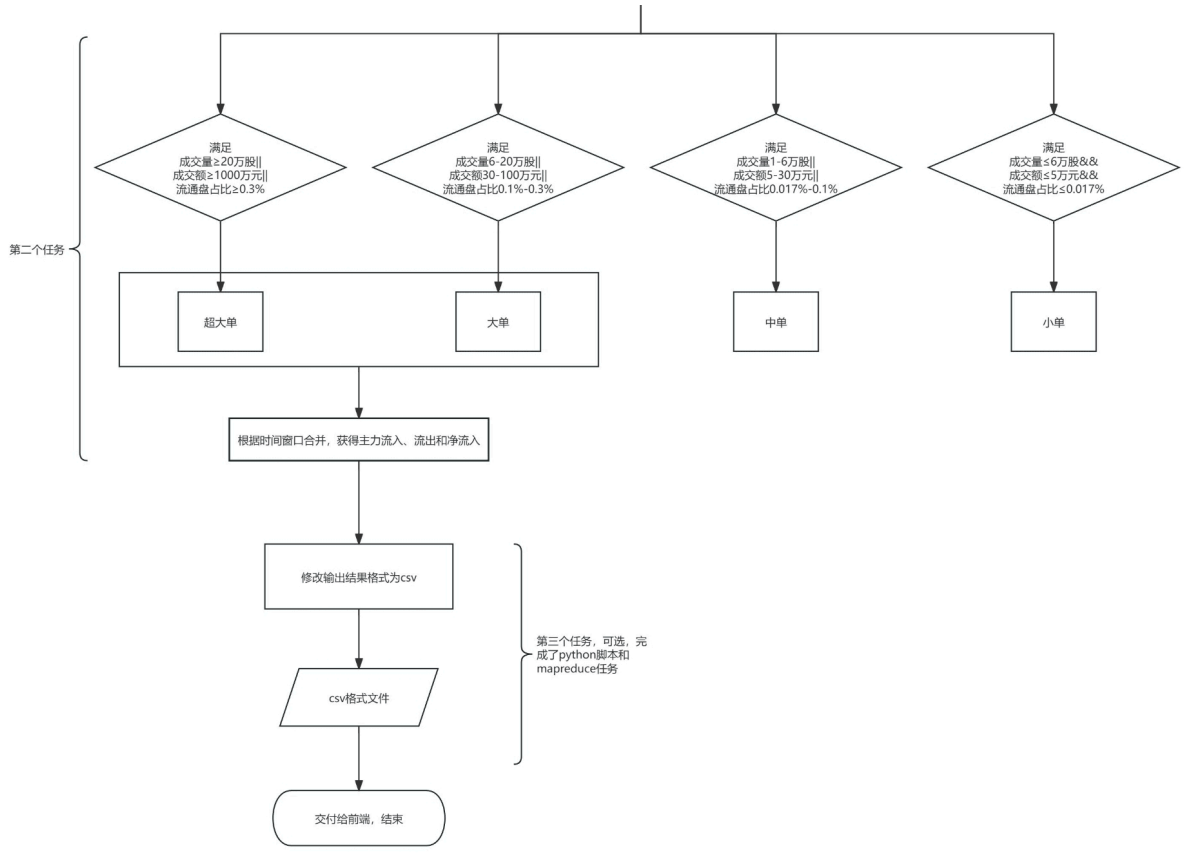
于是，我们完成了TEST.ipynb脚本进行验证。验证思路为每次截取100000条委托记录，根据股票代码进行分离。然后将委托记录依据索引值进行排序，再利用is_monotonic_increasing函数验证时间值是否保持单调递增。以下代码是进行验证的python函数。

```
# 遍历ChannelNo列表中的每个元素i（每个通道号）
for i in ChannelNo:

    # 判断当前通道号i的订单序列号ApplSeqNum是否是单调递增的
    # 使用DataFrame的is_monotonic_increasing属性来检查序列是否单调递增
    is_ascending = (easy_order_table[easy_order_table["ChannelNo"] == i]
["ApplSeqNum"].is_monotonic_increasing)

    # 如果订单序列号不是单调递增的
    if not is_ascending:
        # 打印出不满足单调递增条件的通道号
        print(i)
```

经过验证，本次project提供的数据维持了委托索引与委托时间的单调一致性。这为之后进行的流程优化提供了空间。

4.2 MapReduce任务流程设计





如图所示，本次mapreduce的任务分成了三个任务。

任务1

从逐笔成交原始文件数据出发，先过滤获得指定股票代码的成交记录，然后仅保留指定股票代码的成交记录，过滤留下exec_type为F的时间、买卖委托单号、价格和成交股数信息。
然后根据买卖委托单号大小，买单号大者为主动买单，反之为主动卖单，分类委托记录得到主动单集合。紧接着合并主动单集合，计算每个主动单的总资金和股数。

任务2

得到每个主动单的总资金和股数后，将每个主动单根据规则进行分类，得到超大单、大单、中单和小单四类。并根据成交时间分属于不同的时间窗口，再合并计算得到每个时间窗口的主力流入、流出和净流入。

任务3

修改输出结果格式为指定的csv格式。整理数据输出结构。以每个时间窗口为主键，按顺序输出每个类别的值。本任务如果由mapreduce框架完成耗时过多，故完成了另一python脚本做替代。同时jar包内默认执行第三个任务。

4.3前端实现

在前端网页中，可视化csv数据的结果。具体见第七部分。

5.代码模块化设计思路

5.1 数据分析

先完成验证代码（见4.1），然后完成自动分批处理进行验证的脚本。

5.2 MapReduce任务

5.2.1 TradeFilter

Mapper

核心处理逻辑：

`TradeFilterMapper` 任务的核心处理逻辑是从每行输入数据中提取关键信息，首先根据给定的证券 ID 过滤数据，然后根据买单和卖单的序列号确定订单类型（买单或卖单）。如果满足条件（执行类型为 `F` 且证券 ID 匹配），则生成一个唯一的键，包含订单类型、证券 ID、序列号和时间戳，并计算交易的数量和价格。最后，`Mapper` 将这些信息以键值对的形式写入上下文，供后续处理使用。

输出数据类型：

该任务的输出数据类型是键值对（`Text` 和 `TradeData`）。键（`keyOut`）是一个 `Text` 类型的字符串，包含了订单类型（买单或卖单）、证券 ID、序列号和时间戳。值（`valueOut`）是一个 `TradeData` 类型的对象，包含交易数量（`tradeQty`）和交易价格（`price`）。

Reducer

输出数据格式：

该 `Mapper` 的输出是一个键值对，键是 `Text` 类型，值是 `TradeData` 类型。`keyOut` 作为输出的键，由订单类型（买单或卖单）、证券 ID、序列号（买单的 `bidAppSeqNum` 或卖单的 `offerAppSeqNum`）和时间戳组成，格式为 `"Buy_<securityID>_<bidAppSeqNum>_<timeStamp>"` 或 `"Sell_<securityID>_<offerAppSeqNum>_<timeStamp>"`。`valueOut` 是一个 `TradeData` 对象，包含交易数量和价格，格式为 `TradeData(tradeQty, price)`。这些信息会被写入上下文，最终由 `Mapper` 输出到 `Reducer`。

Reducer

核心处理逻辑：

`TradeFilterReducer` 任务的核心处理逻辑是对每个来自 `Mapper` 的键值对进行聚合。对于相同的 `key`（即相同的订单类型、证券 ID、序列号和时间戳），`Reducer` 会迭代所有的 `TradeData` 对象，累计计算交易的总数量（`totalTradeQty`）和总成交额（`totalTradeAmount`）。成交额是通过将价格与交易数量相乘得到的。最后，`Reducer` 将总数量和总成交额拼接成一个字符串，并通过 `context.write` 输出该键值对。

输出数据类型：

该任务的输出数据类型是键值对（`Text` 和 `Text`）。键（`key`）是 `Text` 类型，表示订单类型、证券 ID、序列号和时间戳，值（`result`）也是 `Text` 类型，表示该订单的总交易数量和总成交额，格式为 `"totalTradeQty,totalTradeAmount"`。

5.2.2 TradeMerge

Mapper

核心处理逻辑：

`TradeMergeMapper` 任务的核心处理逻辑包括对每个输入的交易数据进行以下几个步骤的处理：

1. **提取信息**：从输入的每一行数据中提取交易的时间戳、订单类型、证券 ID、交易数量和交易金额等信息。
2. **分类交易类型**：根据交易数量、交易金额以及交易数量与股票总量的比例，将交易分为 "ExtraLarge"（超大单）、"Large"（大单）、"Medium"（中单）和 "Small"（小单）。
3. **时间窗口划分**：根据交易时间戳，使用 `getTradingTimeSegment` 方法将交易分配到相应的时间段（如连续竞价时段），并根据配置的时间窗口大小（`k`）进一步细分时间段。
4. **输出键值对**：将订单类型、证券 ID、时间段和交易类型作为键，交易数量和交易金额作为值，写入上下文。

输出数据类型：

该任务的输出数据类型是键值对（`Text` 和 `Text`）。

- 键（`keyOut`）是一个 `Text` 类型的字符串，包含订单类型、证券 ID、交易时间段和交易类型（"ExtraLarge"、"Large"、"Medium"、"Small"）。
- 值（`valueOut`）也是 `Text` 类型，表示交易数量和交易金额，格式为 `"tradeQty,tradeAmount"`。

Reducer

核心处理逻辑：

`TradeMergeReducer` 任务的核心处理逻辑是对相同键（`key`）的所有交易数据进行聚合计算：

1. **累计交易数量和成交额**：对于每个相同 `key` 的值（`values`），即同一时间段、证券 ID 和订单类型的所有交易记录，`Reducer` 会遍历这些交易数据，将交易数量（`tradeQty`）和交易金额（`tradeAmount`）累加。
2. **输出结果**：计算出该键对应的总交易数量和总成交额后，将结果以 `totalTradeQty,totalTradeAmount` 的格式保存在 `result` 中，并通过 `context.write` 输出到下游。

输出数据类型：

该任务的输出数据类型是键值对（`Text` 和 `Text`）。

- 键（`key`）是 `Text` 类型，表示订单类型、证券 ID、时间段和交易类型（如 "Large" 或 "Small"）。
- 值（`result`）是 `Text` 类型，表示该时间段、证券 ID 和交易类型下的总交易数量和总成交额，格式为 `"totalTradeQty,totalTradeAmount"`。

5.2.3 Convert

Mapper

核心处理逻辑：

`ConvertMapper` 任务的核心处理逻辑是将输入的交易数据进行格式化处理并输出：

1. **解析每一行数据**：首先，`Mapper` 会读取每一行输入数据，按照制表符（`\t`）将其分割为两个部分：键（包含时间和类别等信息）和值（包含成交量和成交额）。
2. **提取时间和类别**：从输入的键中提取时间（如 `09:25`）和交易类别（如 `Large`、`Medium` 等），并从值部分提取成交量和成交额。
3. **格式化输出**：将处理后的数据按照 `<时间，类别_成交量_成交额>` 的格式输出，键是时间，值是由买卖类型、类别、成交量和成交额组成的字符串。

输出数据类型：

该任务的输出数据类型是键值对（`Text` 和 `Text`）。

- **键（key）**：`Text` 类型，表示交易的时间（例如 `09:25`）。
- **值（value）**：`Text` 类型，格式为 `"Buy/Sell_类别_成交量_成交额"`，即包含买卖类型、类别、成交量和成交额的字符串。

Reducer

核心逻辑：

- **表头输出**：在第一次处理时输出表头，之后不再输出。
- **数据聚合**：通过遍历 `values`，累积每个类别（`ExtraLarge`、`Large`、`Medium`、`Small`）的买单和卖单的成交量与成交额。
- **指标计算**：
  - 计算主力流入（大单和超大单买入的成交额总和）。
  - 计算主力流出（大单和超大单卖出的成交额总和）。
  - 计算主力净流入（主力流入减去主力流出）。
- **结果输出**：将计算出的结果格式化为一个 CSV 格式的字符串，并输出。

输出数据类型：

- **输出键**：`Text` 类型（时间段）。
- **输出值**：`Text` 类型（包含各类成交量和成交额的 CSV 字符串）。

data_convert(可选)

这是一个用于转换数据的可选python脚本，由于分布式系统的mapreduce框架有大量耗时，对于转换数据的工作，使用python会更方便。故完成此脚本作为备选。
这个脚本的工作是从文件中提取每个时间段的买卖订单信息（包括成交量和成交额），并根据这些数据得到主力流入、流出及净流入等指标。最终，处理后的数据被整理成一个表格（`price_table`），并按照时间排序后输出为一个 CSV 文件（`price_table_sorted.csv`）

5.3 前端任务

详见第七部分

5.4辅助函数介绍

`getTradingTimeSegment`

时间窗口划分的方法通过交易的时间戳将交易数据划分为不同的时间段，并根据指定的时间窗口大小（`k`）将每个时段进一步细分。具体步骤如下：

1. **提取时间戳**：从输入数据中提取交易的时间戳，格式为 `yyyyMMddHHmmssSSS`（如：`20240101123045000`）。
2. **时间段分类**：
   - **开盘集合竞价（09:15 - 09:25）**：如果交易发生在这个时间段，直接跳过该交易（`skip`），不进行后续处理。
   - **上午连续竞价（09:30 - 11:30）**：将该时段的时间分成多个较小的时间窗口（如每 `k` 分钟一个窗口）。通过计算交易时间距离上午9:30的分钟数，再除以窗口大小 `k` 来确定交易的时间段。
   - **下午连续竞价（13:00 - 15:00）**：与上午时段类似，将下午的交易时间划分为多个窗口，并根据交易时间分配到相应的时间段。
3. **时间窗口划分**：每个时间段内的交易按照时间戳被分配到相应的时间窗口。例如，如果窗口大小 `k` 为10分钟，09:30到09:40的交易属于第一个时间段，09:40到09:50属于第二个时间段，依此类推。
4. **生成时间段标识**：根据计算的时间段，生成一个字符串标识，例如 `"0900to0910"` 表示从09:00到09:10的时间窗口。

`TradeData`

这是一个用于存储 tradeQty 和 price 的数据结构。继承了抽象类Writable。实现这个数据结构的目的是为了减少Text转String再转int的时间损耗。在第一个任务中，mapper传递的结果里有着大量需要计算的tradeQty和price，两者又分属不同的数据类型long和double。无法用单一的已有的数据结构实现。故创建了TradeData数据结构，来存储这两个数据。便于reducer进行计算。

6. 运行结果评估

6.1 运行时间

采用三个任务连续工作的计算时间约为1min40s，而如果采用两个任务连续工作，然后使用python脚本进行数据转换，则为1min20s，其中python的数据转换不足1s。大大减少了运行所用的耗时。

6.2 准确性

经过老师给出的评估代码，两种方式得到的正确率均为百分之百。

7. 数据可视化

7.1 代码设计思路

本项目基于 Spring Boot 构建，通过集成 Maven、OpenCSV、ECharts、Thymeleaf 等相关技术，旨在实现对 CSV 文件中数据的解析和可视化工作，项目主要模块如下：

模块名称	主要实现功能
CsvVisualizationApplication.java	Spring Boot 应用的入口类，启动应用，初始化 Spring 容器，扫描并加载所有的组件
MyData.java	数据模型类，将从 CSV 文件中解析的数据封装成对象
CsvService.java	服务层类，负责实际的 CSV 文件解析和数据处理
ChartController.java	处理前端请求，并返回相应的视图或数据
chart.html	前端数据展示与交互，用于动态展示数据可视化得到的图表
pom.xml	Maven 构建工具的配置文件，用于管理项目的依赖关系、插件配置等基本信息

实现主要技术的关键代码如下：

OpenCSV读取数据

```
public List<MyData> parseCsv(String filePath) throws IOException,
CsvValidationException {
        List<MyData> dataList = new ArrayList<>();
        try (CSVReader reader = new CSVReader(new
InputStreamReader(getClass().getClassLoader().getResourceAsStream(filePath)))) {
            String[] nextLine;// 跳过 CSV 文件的表头
            reader.readNext();
            //运用CSVReader和CsvValidationException读取csv文件数据
            while ((nextLine = reader.readNext()) != null) {//判断读取是否完成
                MyData data = new MyData(
                        Double.parseDouble(nextLine[0]),  // 主力净流入
                        Double.parseDouble(nextLine[1]),  // 主力流入
                        Double.parseDouble(nextLine[2]),  // 主力流出
                        Double.parseDouble(nextLine[3]),  // 超大买单成交量
                        Double.parseDouble(nextLine[4]),  // 超大买单成交额
                        Double.parseDouble(nextLine[5]),  // 超大卖单成交量
                        Double.parseDouble(nextLine[6]),  // 超大卖单成交额
                        Double.parseDouble(nextLine[7]),  // 大买单成交量
                        Double.parseDouble(nextLine[8]),  // 大买单成交额
                        Double.parseDouble(nextLine[9]), // 大卖单成交量
                        Double.parseDouble(nextLine[10]), // 大卖单成交额
                        Double.parseDouble(nextLine[11]), // 中买单成交量
                        Double.parseDouble(nextLine[12]), // 中买单成交额
                        Double.parseDouble(nextLine[13]), // 中卖单成交量
                        Double.parseDouble(nextLine[14]), // 中卖单成交额
                        Double.parseDouble(nextLine[15]), // 小买单成交量
                        Double.parseDouble(nextLine[16]), // 小买单成交额
                        Double.parseDouble(nextLine[17]), // 小卖单成交量
                        Double.parseDouble(nextLine[18]), // 小卖单成交额
                        nextLine[19].trim() // 时间范围
                );
                dataList.add(data);//添加数据到dataList
            }
        }
```

```
        return dataList;//返回读取到的数据
    }
```

将前端输入时间传递至后端更新数据

```
<form method="get" action="/showChart" style="grid-column: 1 / span 2;">
    //使用 method="get" 和 action="/showChart"，当用户点击按钮时，表单会通过 GET 请求提交到 /showChart 路径
    <label for="startTime">起始时间:</label>
    //添加标签显示
    <input type="number" id="startTime" name="startTime" step="1">//获取startTime
    <label for="endTime">结束时间:</label>
    //添加标签显示
    <input type="number" id="endTime" name="endTime" step="1">//获取endTime
    <button type="submit">筛选</button>
    //通过 <input> 元素的 name 属性将时间数据传递给后端
</form>
```

```
@GetMapping("/showChart")
//设置mapping的路径
public String showChart(@RequestParam(value = "startTime", required = false)
Double startTime,
                        @RequestParam(value = "endTime", required = false)
Double endTime,
                        Model model) throws IOException,
CsvValidationException
    //通过@RequestParam 注解用于接收前端提交的参数。这些参数将通过请求URL的查询字符串传递给控制器方法
```

```
if (startTime != null && endTime != null) {
    //判断传入的时间参数是否为空，若为空值则使用原始数据
    filteredDataList = dataList.stream()
            .filter(data -> {
                double time =
Double.parseDouble(data.getTimeRange().substring(data.getTimeRange().length() -
9, data.getTimeRange().length() - 5));//从时间字符串提取需要比较的时间数据，转换为double类型便于比较
                return time >= startTime && time <= endTime;//筛选符合要求的数据行
            })
            .collect(Collectors.toList());//将dataList的数据经过时间筛选更新到filteredDataList中
}
```

将后端更新数据传递至前端进行网页显示

```
model.addAttribute("timeRangeJson", timeRangeJson);//对于list类型数据，转换为json文件再进行传输
model.addAttribute("superBuyAmount", transactionAmounts[0]);//其他数据直接通过model.addAttribute方法传递到前端模板
```

```
const mainInflow = JSON.parse('[[${mainInflowJson}]]');
//前端HTML中使用Thymeleaf表达式，将后端传递的数据嵌入到JavaScript中，通过${}来引用传递的数据
console.log("Main Inflow:", mainInflow);//检查数据传递是否正确
```

使用ECharts显示图表并实现动态交互

```
<head>
    <!-- 定义文档的字符编码为UTF-8，支持多种字符 -->
    <meta charset="UTF-8">
    <!-- 设置网页为响应式布局，适应不同屏幕尺寸 -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- 设置网页的标题，将显示在浏览器的标签页上 -->
    <title>Stock Data Visualization</title>
    <!-- 从CDN引入ECharts库，用于生成图表 -->
    <script src="https://cdn.jsdelivr.net/npm/echarts@5.4.0/dist/echarts.min.js">
</script>
    <style>
        /* 使用CSS网格布局样式设置容器 */
        .container {
            display: grid; /* 启用网格布局 */
            grid-template-columns: 1fr 1fr; /* 定义两列等宽的布局 */
            grid-template-rows: 1fr 1fr; /* 定义两行等高的布局 */
            gap: 20px; /* 设置网格项之间的间距 */
            height: 100vh; /* 容器高度为视口高度 */
            padding: 20px; /* 容器的内边距 */
        }
        /* 图表容器的样式 */
        .chart-container {
            width: 100%; /* 宽度为100% */
            min-height: 300px; /* 设置图表的最小高度 */
            border: 1px solid #ddd; /* 设置浅灰色的边框 */
            padding: 10px; /* 图表容器的内边距 */
            box-sizing: border-box; /* 使内边距包含在宽度和高度计算中 */
        }
        /* 图表标题的样式 */
        .chart-title {
            text-align: center; /* 将标题居中对齐 */
            font-size: 20px; /* 设置标题的字体大小 */
            margin-bottom: 10px; /* 标题底部留出空间 */
        }
    </style>
</head>
```

```
// 以图表A为例介绍
const chartA = echarts.init(document.getElementById('chartA'));//初始化表A
const optionA = {
    tooltip: { trigger: 'item', formatter: '{a} <br/>{b}: {c} 元 ({d}%)' },
    //添加光标移动显示的动态标签，显示类别、金额、百分比等数据
    legend: {
        orient: 'vertical',//确定排版方向
        left: 'left',//确定标签位置
        data: ['超大买单', '大买单', '中买单', '小买单', '超大卖单', '大卖单', '中卖单',
'小卖单'],
```

```
    },//设置分类标签格式
    series: [
        {
            name: '成交额分布',
            type: 'pie',//设置图表类型为饼图
            radius: '60%',//设置饼图半径大小
            center: ['50%', '45%'],//设置饼图圆心位置
            data: [
                { value: data1.superBuyAmount, name: '超大买单', itemStyle: {
color: '#8B0000' } },
                { value: data1.bigBuyAmount, name: '大买单', itemStyle: { color:
'#B22222' } },
                { value: data1.midBuyAmount, name: '中买单', itemStyle: { color:
'#CD5C5C' } },
                { value: data1.smallBuyAmount, name: '小买单', itemStyle: { color:
'#FA8072' } },
                { value: data1.superSellAmount, name: '超大卖单', itemStyle: {
color: '#006400' } },
                { value: data1.bigSellAmount, name: '大卖单', itemStyle: { color:
'#228B22' } },
                { value: data1.midSellAmount, name: '中卖单', itemStyle: { color:
'#32CD32' } },
                { value: data1.smallSellAmount, name: '小卖单', itemStyle: {
color: '#90EE90' } },
                //设置不同数据对应名称及色号，颜色基于买卖单不同为深红到浅红、深绿到浅绿
            ],
        },
    ],
};
```

7.2 网页展示介绍

运行csv-visualization后，打开网页：http://localhost:8080/showChart

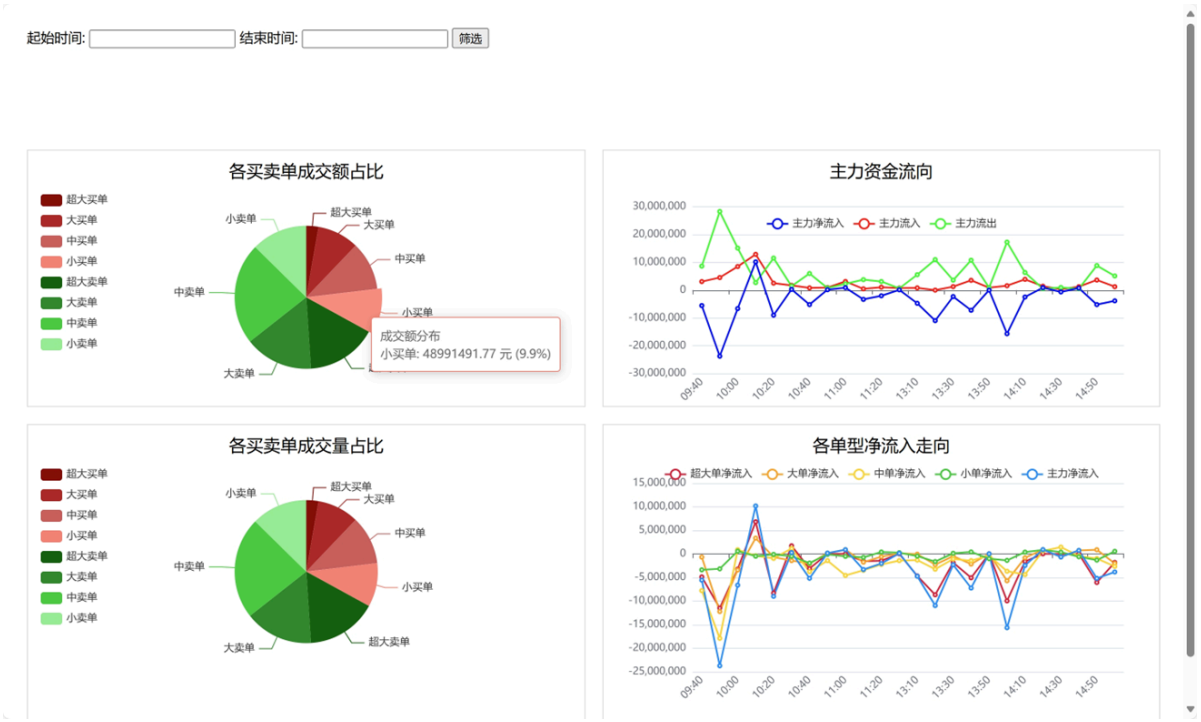
网页提供三种动态显示功能：

1.输入起始时间与结束时间后点击“筛选”，调整可视化数据的时间范围

2.动态显示光标所在处具体数据情况

3.点击标签，选择需要展示的数据类别

网页显示如下：


起始时间:
结束时间:
筛选
各买卖单成交额占比
超大买单
大买单
中买单
小买单
超大卖单
大卖单
中卖单
小卖单
成交额分布
小买单: 48991491.77 元 (9.9%)
主力资金流向
主力净流入
主力流入
主力流出
各买卖单成交量占比
各单型净流入走向
超大单净流入
大单净流入
中单净流入
小单净流入
主力净流入