# 1 设计思路

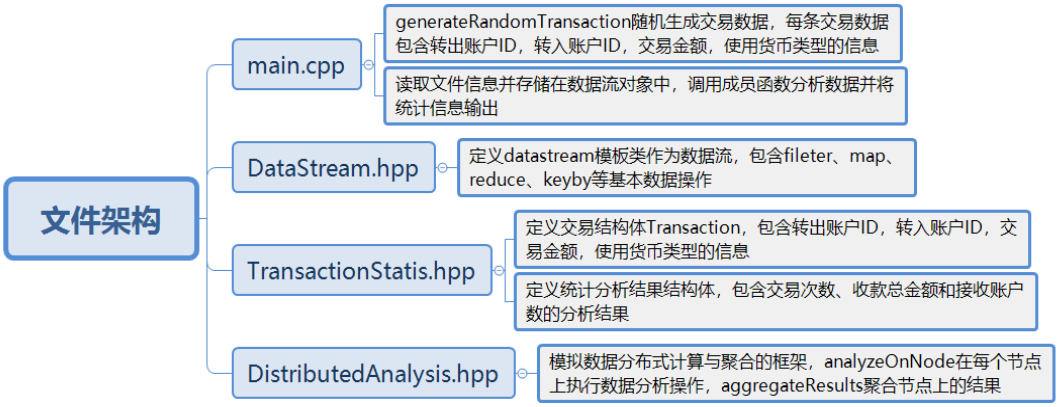
## 1.1 程序简介

本程序设计背景基于转账交易数据的分析需求，旨在针对一组包含转出账户 ID，转入账户 ID，交易金额，使用货币类型信息的数据，计算出每个账户的交易次数、每个账户的总收款金额、以及交易接收账户的数量，同时输出相关信息。系统的设计逻辑基于以下模块化目标：

1. 随机生成转账交易数据，并保存为文本文档。
2. 读取并解析数据，创建对应的交易数据流。
3. 对数据流进行筛选和格式映射，再通过结构化数据分析，生成结果统计。
4. 将分析结果以用户友好的格式输出。

同时，考虑到大数据处理的实际场景，我们在设计中实现了模拟分布式计算和聚合的方式处理。

## 1.2 总体架构




# 2 代码剖析

## 2.1 随机交易数据生成与解析

1) 使用当前时间的计数作为随机种子，并创建了 gen 作为随机数引擎；

```cpp
// 使用当前时间的计数作为种子
unsigned seed = static_cast<unsigned>(std::chrono::system_clock::now().time_since_epoch().count());

// 创建一个 Mersenne Twister 引擎
std::mt19937 gen(seed);
std::uniform_int_distribution<> totalTransactionAmount(50, 200);
int transactionAmount = totalTransactionAmount(gen);
```

2) 函数 generateRandomTransaction 分别随机生成单条交易数据的转出账户 ID、转入账户 ID、交易金额和货币类型，其中交易金额根据对应汇率转化。

```cpp
Transaction generateRandomTransaction(std::mt19937 &gen)
{
    std::uniform_int_distribution<> accountDistribution(1, 30);       // 随机生成的账户ID
    std::uniform_real_distribution<> amountDistribution(1.0, 1500.0); // 随机生成的交易金额
    std::uniform_int_distribution<> monetaryTypeDistribution(0, 2);   // 随机生成的货币类型

    Transaction transaction;
    transaction.fromAccountID = accountDistribution(gen);
    transaction.toAccountID = accountDistribution(gen);

    transaction.monetaryType = monetaryTypeDistribution(gen);
    switch (transaction.monetaryType)
    {
    case 0:
        transaction.amount = amountDistribution(gen);
        break;
    case 1:
        transaction.amount = amountDistribution(gen) / 7.8;
        break;
    case 2:
        transaction.amount = amountDistribution(gen) / 7.1;
        break;
    }

    return transaction;
}
```

3) parseAndCreateDataStream 函数从输入文件中载入交易信息，解析后以结构体的形式储存在向量列表中，最后返回一个包含所有交易的 DataStream 对象。

```cpp
// 解析并创建数据流函数
template <typename T>
DataStream<T> parseAndCreateDataStream(std::ifstream &inputFile)
{
    std::vector<T> transactions;
    std::string line;

    // 从文件中解析交易记录并创建数据流
    while (std::getline(inputFile, line))
    {
        transactions.push_back(parseTransaction(line));
    }

    return DataStream<T>(transactions);
}
```

## 2.2 交易数据&统计分析结果结构体

1) 定义交易数据结构体 Transaction，包含转出账户 ID，转入账户 ID，交易金额，使用货币类型的信息：

```
// 定义交易结构体
struct Transaction
{
    int fromAccountID; // 起始账户ID
    int toAccountID;   // 目标账户ID
    double amount;     // 交易金额
    int monetaryType;  // 货币类型,0代表人民币，1代表欧元，2代表美元
};
```

2) 定义统计分析结果结构体，包含交易次数、收款总金额和接收账户数的分析结果：

```
// 定义统计分析结果结构体，包含交易次数、收款总金额和接收账户数
struct TransactionStats
{
    std::unordered_map<int, int> transactionCounts;
    std::unordered_map<int, double> totalReceivedAmounts;
    std::unordered_map<int, int> uniqueReceivingAccountCounts;
};
```

## 2.3 Datastream 数据流对象

以模板类形式实现，模板参数 T 代表单条交易数据的结构体，该数据流包含了包含 fileter、map、reduce、keyby 等基本数据操作。

### 2.3.1 filter 函数

实现数据过滤功能，这里只以筛选掉两个交易对象相同的异常数据为例，实际上只需要更改 Lambda 表达式便可实现更多不同的筛选过滤功能。

```
// 筛选函数
DataStream<T> filter() const
{
    // 使用 Lambda 表达式定义判断条件
    auto isNotSameAccount = [](const T &transaction)
    {
        return transaction.fromAccountID != transaction.toAccountID;
    };
    // 筛选数据
    std::vector<T> filtered;
    std::copy_if(transactions_.begin(), transactions_.end(), std::back_inserter(filtered), isNotSameAccount);
    return DataStream<T>(filtered);
}
```

### 2.3.2 map 函数

实现数据映射转化功能，这里以汇率转化为例，映射返回汇率转化后的数据流，同样，只需要更改 mapper 这个 lambda 表达式便可以实现更多的映射可能。

```
DataStream<T> map() const
{
    // 定义汇率映射表
    std::unordered_map<int, double> exchangeRates;
    exchangeRates[0] = 1.0; // 人民币对人民币的汇率
    exchangeRates[1] = 7.8; // 欧元对人民币的汇率
    exchangeRates[2] = 7.1; // 美元对人民币的汇率
    auto mapper = [&exchangeRates](const T &transaction)
    {
        T transactionMap = transaction;
        transactionMap.amount *= exchangeRates[transaction.monetaryType];
        transactionMap.monetaryType = 0;
        return transactionMap;
    };
    std::vector<T> mapped;
    std::transform(transactions_.begin(), transactions_.end(), std::back_inserter(mapped), mapper);
    return DataStream<T>(mapped);
}
```

### 2.3.3 reduce 函数

实现归约功能，这里实现了对每个账户的交易次数，每个账户的收款总金额，以及转出的接收账户数的统计功能。

```
// 归约函数
TransactionStats reduce() const
{
    TransactionStats stats;
    stats.transactionCounts = std::accumulate(transactions_.begin(), transactions_.end(), std::unordered_map<int, int>{}, [](auto &result, const T &transaction)
                                              {
                                              result[transaction.fromAccountID]++;
                                              result[transaction.toAccountID]++;
                                              return result; });

    stats.totalReceivedAmounts = std::accumulate(transactions_.begin(), transactions_.end(), std::unordered_map<int, double>{}, [](auto &result, const T &transaction)
                                                 {
                                                 result[transaction.toAccountID] += transaction.amount;
                                                 return result; });
    stats.uniqueReceivingAccountCounts = std::accumulate(transactions_.begin(), transactions_.end(), std::unordered_map<int, int>{}, [](auto &result, const T &transaction)
                                                         {
                                                         result[transaction.fromAccountID]++;
                                                         return result; });
    return stats;
}
```

### 2.3.4 keyBy 函数

实现按照某个标识符将交易数据分组的功能，这里以“交易货币类型”作为分组依据。

```
// 按键分组函数
auto keyBy() const
{
    // 使用货币类型作为键进行分组
    std::unordered_map<int, std::vector<T>> grouped;
    for (const auto &item : transactions_)
    {
        grouped[item.monetaryType].push_back(item);
    }
    return grouped;
}
```

## 2.4 模拟数据分布式计算与聚合的框架

1) 将数据流分割为多个数据块

```
// 分割数据流并创建块
std::vector<DataStream<T>> splitIntoBlocks(std::size_t numBlocks) const
{
    std::vector<DataStream<T>> dataBlocks(numBlocks);

    // 计算块的大小
    std::size_t blockSize = transactions_.size() / numBlocks;
    std::size_t remainder = transactions_.size() % numBlocks;
    auto begin = transactions_.begin();

    for (std::size_t i = 0; i < numBlocks; ++i)
    {
        auto end = begin + blockSize + (i < remainder ? 1 : 0);
        dataBlocks[i] = DataStream<T>(std::vector<T>(begin, end));
        begin = end;
    }

    return dataBlocks;
}
```

2) 定义 DistributedAnalysis 模板类作为数据分布式计算与聚合处理的框架，analyzeOnNode 函数在每个节点上执行数据分析操作，aggregateResults 函数聚合节点上的结果

```cpp
// 模拟数据分布式计算与聚合的框架
template <typename T>
class DistributedAnalysis
{
public:
    DistributedAnalysis(std::vector<DataStream<T>> &dataBlocks) : dataBlocks_(dataBlocks) {}

    // 在每个节点上执行数据分析操作
    std::vector<TransactionStats> analyzeOnNode()
    {
        std::vector<TransactionStats> results;
        for (const auto &dataBlock : dataBlocks_)
        {
            TransactionStats result = dataBlock.reduce();
            results.push_back(result);
        }
        return results;
    }

    // 聚合节点上的结果
    TransactionStats aggregateResults(const std::vector<TransactionStats> &results)
    {
        TransactionStats aggregatedResult;

        for (const auto &result : results)
        {
            // 对各个数据块的结果进行累加
            for (const auto &entry : result.transactionCounts)
            {
                aggregatedResult.transactionCounts[entry.first] += entry.second;
            }

            for (const auto &entry : result.totalReceivedAmounts)
            {
                aggregatedResult.totalReceivedAmounts[entry.first] += entry.second;
            }

            for (const auto &entry : result.uniqueReceivingAccountCounts)
            {
                aggregatedResult.uniqueReceivingAccountCounts[entry.first] += entry.second;
            }
        }

        return aggregatedResult;
    }

private:
    std::vector<DataStream<T>> dataBlocks_;
};
```

# 3 实验结果

通过随机生成的交易数据来验证代码。系统根据所定义的条件分别找出符合以下三个情况的账户：总交易次数大于 5 次；总接收金额大于 5000 元；接收账户数量大于 2 个。

然后，我们打印这些账户以及对应的详细信息。这些信息会根据每次运行时随机生成的数据有所不同。另外，程序中还利用了 keyBy 中按货币分组所得到的统计出使用不同货币交易的交易数并输出。

示例一：统计分析结果输出

```
[Running] cd "e:\cpp\" && g++ main.cpp
交易次数超过5次的:
账户 3: 7 次交易
账户 20: 6 次交易
账户 22: 9 次交易
账户 26: 11 次交易
账户 14: 7 次交易
账户 7: 12 次交易
账户 24: 9 次交易
账户 19: 7 次交易
账户 29: 12 次交易
账户 12: 6 次交易
账户 18: 14 次交易
账户 9: 10 次交易
账户 16: 8 次交易
账户 13: 8 次交易
账户 30: 7 次交易
账户 2: 7 次交易
账户 17: 14 次交易
账户 11: 7 次交易
账户 27: 6 次交易
账户 10: 13 次交易
账户 23: 8 次交易

收款总金额超过5000元的:
账户 29: ￥6286.01
账户 10: ￥7088.53
账户 9: ￥6912.17

接收账户数超过两个的:
账户 1: 3 个接收账户
账户 13: 4 个接收账户
账户 16: 3 个接收账户
账户 8: 3 个接收账户
账户 9: 4 个接收账户
账户 18: 9 个接收账户
账户 29: 4 个接收账户
账户 11: 3 个接收账户
账户 2: 4 个接收账户
账户 22: 4 个接收账户
账户 12: 4 个接收账户
账户 24: 5 个接收账户
账户 7: 9 个接收账户
账户 17: 8 个接收账户
账户 23: 6 个接收账户
账户 30: 3 个接收账户
账户 27: 5 个接收账户
账户 10: 4 个接收账户
账户 26: 5 个接收账户
账户 20: 4 个接收账户
账户 3: 3 个接收账户

欧元交易数: 35
人民币交易数: 44
美元交易数: 36
```

随机生成的交易数据文本（只截了部分）

```
30 19 1457.32 0
27 11 210.398 2
13 13 28.227 0
17 23 118.347 1
23 24 416.637 0
14 26 1357.55 0
14 11 84.2537 0
5 19 57.3302 1
26 17 1027.32 0
20 6 40.4887 1
27 22 61.6827 2
3 17 192.359 2
7 11 572.344 0
19 22 81.0961 1
10 14 1232.63 0
2 8 169.029 1
3 13 52.6975 2
27 29 350.935 0
11 29 149.22 1
29 17 76.5598 1
18 27 174.772 0
12 24 36.1899 2
22 9 161.013 2
27 16 47.9707 1
7 19 117.757 2
22 16 36.0558 1
23 29 149.022 1
24 8 116.876 2
15 15 24.4014 1
25 10 72.697 2
26 18 37.2502 2
29 6 130.239 1
1 1 92.7541 1
8 19 101.645 0
20 22 1324.77 0
11 10 437.856 0
2 22 138.11 2
17 28 152.971 2
16 6 36.1677 1
6 10 95.3123 2
11 29 41.0593 2
9 9 1489.44 0
17 9 1284.75 0
20 14 104.278 0
24 17 106.636 0
21 10 56.7342 2
13 28 169.862 1
1 13 420.486 0
18 20 2.51291 2
7 18 7.69844 0
24 9 190.751 2
23 2 27.9885 1
29 18 131.475 2
18 10 93.8519 1
9 2 237.899 0
11 11 179.336 1
2 26 1407.84 0
26 10 114.274 2
```

示例二：统计分析结果输出

```
[Running] cd "e:\课程资料\大
交易次数超过5次的:
账户 10: 6 次交易
账户 15: 6 次交易
账户 6: 7 次交易
账户 11: 6 次交易
账户 13: 10 次交易
账户 29: 10 次交易
账户 12: 6 次交易
账户 24: 8 次交易
账户 28: 7 次交易
账户 27: 6 次交易
账户 16: 11 次交易
账户 9: 12 次交易
账户 4: 6 次交易
账户 21: 6 次交易
账户 18: 7 次交易

收款总金额超过5000元的:
账户 16: ￥6328.82

接收账户数超过两个的:
账户 10: 4 个接收账户
账户 8: 3 个接收账户
账户 15: 4 个接收账户
账户 13: 5 个接收账户
账户 30: 4 个接收账户
账户 7: 3 个接收账户
账户 29: 7 个接收账户
账户 12: 4 个接收账户
账户 11: 4 个接收账户
账户 16: 4 个接收账户
账户 9: 7 个接收账户
账户 6: 5 个接收账户
账户 17: 5 个接收账户
账户 18: 3 个接收账户

美元交易数: 33
人民币交易数: 30
欧元交易数: 26
```

随机生成的交易数据文本（只截了部分）

```
13 18 183.543 1
29 23 103.643 1
1 10 42.5991 1
10 18 318.252 0
24 16 47.295 0
12 18 70.5479 1
17 13 28.3557 2
21 22 187.847 2
20 21 1379.69 0
19 4 449.094 0
15 5 188.711 1
21 6 164.63 2
15 12 140.143 2
28 9 418.941 0
23 18 75.5221 1
12 26 139.396 1
8 13 336.365 0
5 1 30.495 1
18 17 49.2324 1
23 26 59.0815 1
1 29 116.645 2
7 18 126.657 1
11 22 181.996 2
26 2 389.093 0
17 7 32.5327 2
10 1 186.563 1
19 2 70.8666 2
22 26 192.346 2
18 30 67.6923 2
30 1 42.9786 2
30 9 153.147 1
4 6 73.6481 0
4 4 169.878 1
14 21 207.988 2
28 20 183.826 2
19 3 89.7739 2
26 8 160.495 1
```

# 4 总结

本程序成功实现了对一组转账交易的分析处理，体现了 C++面向对象的程序设计语言在开发时的优势。与 C 语言面向过程的特性不同，在 C++中我们将数据流视为对象，通过对象的分类、过滤、映射和聚合等操作，程序能够高效、简洁地对数据进行分析与统计。

同时，考虑到大数据处理的实际场景，程序还实施了一种模拟的分布式计算和聚合方式来处理数据，完成了分布式计算中的任务划分、执行和结果合并等过程。当然，在实际的分布式计算框架中，这些任务可能涉及更复杂的通信和同步机制。

然而，程序也有一些潜在的不足，比如，程序中缺乏对于文件读取错误的处理机制，对于大量实际操作中的错误情况缺乏对应处理方案。为了提高系统的可靠性，应当添加更多的错误处理代码以确保系统能够适应更复杂的使用环境。

此外，在学习完继承、多态的相关知识后，对于支持多种类型数据流来源的实时输入与解析这一可选要求，我发现可以定义一个数据流接口作为抽象类，并使用继承完成不同数据流类型的处理，这也符合实际开发的思路，以下给出一个大致的代码框架与思路：

```
// 数据流的基类,定义为抽象类作为数据流接口
template <typename T>
class DataStreamInterface
{
public:
    virtual ~DataStreamInterface() = default;

    virtual DataStreamInterface *filter() const = 0;
    virtual DataStreamInterface *map() const = 0;
    virtual TransactionStats reduce() const = 0;
    virtual std::unordered_map<int, std::vector<T>> keyBy() const = 0;
};

// DataStream1 派生类表示和之前一样的数据流
template <typename T>
```

```
class DataStream1 : public DataStreamInterface
{
public:
    DataStream1() = default;
    DataStream1(std::vector<Transaction>            data)            :
transactions_(std::move(data)) {}

    DataStreamInterface *filter() const override
    {
        // 与之前的实现一样，只是返回了一个派生类的指针
        return new DataStream1<T>(filtered);
    }

    DataStreamInterface *map() const override
    {
        // 与之前的实现一样，只是返回了一个派生类的指针
        return new DataStream1<T>(mapped);
    }

    TransactionStats reduce() const override
    {
        // 与之前的实现一样
        return stats;
    }

    virtual  std::unordered_map<int,  std::vector<T>>  keyBy()  const
override
    {
        // 与之前的实现一样
```

```
        return grouped;
    }

private:
    std::vector<Transaction> transactions_;
};

//DataStream2 派生类表示另外一种类型的数据流
class DataStream2 : public DataStreamInterface
{
public:
    DataStreamInterface *filter() const override
    {
        // 新的过滤操作实现
        return new DataStream2<T>(filtered);
    }

    DataStreamInterface *map() const override
    {
        // 新的映射操作实现
        return new DataStream2<T>(mapped);
    }

    TransactionStats reduce() const override
    {
        // 新的归约操作实现
        return stats;
    }
```

```
    auto keyBy() const override
    {
        // 新的筛选操作实现
        return grouped;
    }

}

// 解析并创建数据流函数
DataStreamInterface<T>        *parseAndCreateDataStream(std::ifstream
&inputFile)
{
    //需要根据不同的输入源类型，实现相应的判断逻辑并创建不同的数据流
    // 这里以原先的案例为例子，只需要修改一下返回值即可
    return new DataStream1<T>(transactions);
}

int main()
{
    std::ifstream inputFile("transaction_data");
    if (!inputFile.is_open())
    {
        std::cerr << "无法打开文件!\n";
        return 1;
    }

    // 从文件中解析并创建数据流，父类对象指针指向派生类对象
    DataStreamInterface                  *dataStream                  =
```

```
parseAndCreateDataStream(inputFile);

    // 利用多态处理分析不同类型的数据流
    auto filteredStream = dataStream->filter();
    auto mappedStream = dataStream->map();
    auto reducedStats = dataStream->reduce();
    auto groupedData = dataStream->keyBy();

    // 根据统计结果打印信息
    //...

    // 清理内存
    delete dataStream;

    return 0;
}
```