# 支付对账系统序章：千万级数据对账怎么这么难？

原创 楼下小黑哥 小黑十一点半 2021年12月21日 08:56

## 支付对账

很早 之前写过一篇支付对账相关文章，那时候负责对账系统日均处理数量比较小。

那最近正在接手现在的对账系统，由于当前系统日均数量都在千万级，所以对账系统架构与之前架构完全不一样。

那就这个话题，聊聊如何实现千万级数据支付的对账系统。

## 什么是对账？

我们先来回顾下什么是对账？

也许你对对账这个概念比较模糊，但是这个场景你肯定碰到过。

上班路上买了一个煎饼，加了根里脊与王中王，然后你扫了老板的二维码付了 10 元钱。

你跟老板说你已经付了 10 元钱，老板看了下手机，果然有一条 10 元支付记录，老板确认收到钱，然后就把煎饼给你。

这个过程，你说你付了 10 元，老板确认收到 10 元，这就是一只简单的对账过程。

回到我们支付场景，用户下单使用微信支付 100 元购买了一个狗头抱枕，这时我们这边会生成一条支付记录，同时微信支付也会生成记录。

那微信第二天就会生成一个账单记录，我们拿到之后把我们的交易记录跟微信记录逐笔核对，这就是支付对账。

## 为什么需要对账？

正常支付的情况下，两边（我们/第三方支付渠道）都会产生交易数据，那支付对账过程，两边数据一致，大家各自安好，不用处理什么。

但是有些异常情况下，可能由于网络问题，导致两边数据存在不一致的情况，支付对账就可以主动发现这些交易。

对账可以说支付系统最后一道安全防线，通过对账我们可及时的对之前支付进行纠错，避免订单差错越积越多，最后财务盘点变成一笔糊涂账。

## 支付对账系统

开篇先来一张图，先来看下整体对账系统架构图：


整个对账系统分为两个模块

- 对账模块
- 差错模块

对账模块，主要负责对账文件拉取，数据解析，数据核对，数据汇总等任务。

差错模块是对账模块后置任务，对账模块核对过程产生无法核对成功的数据，这类数据件将会推送给差错系统。

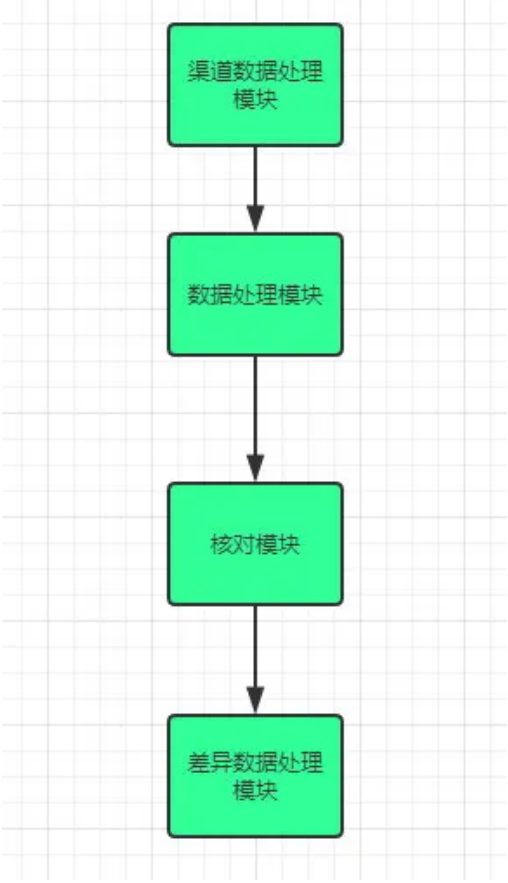
差错系统将会根据规则生成差错订单，运营人员可以在后台处理这列数据。

今天这篇文章先不聊具体的系统设计，先来回顾下之前的对账系统设计，简单了解下对账的整体流程。

## 对账系统设计

对账系统如果从流程上来讲，其实非常简单，引用一下之前文章流程图：

https://studyidea.cn/articles/2019/08/26/1566790305561.html

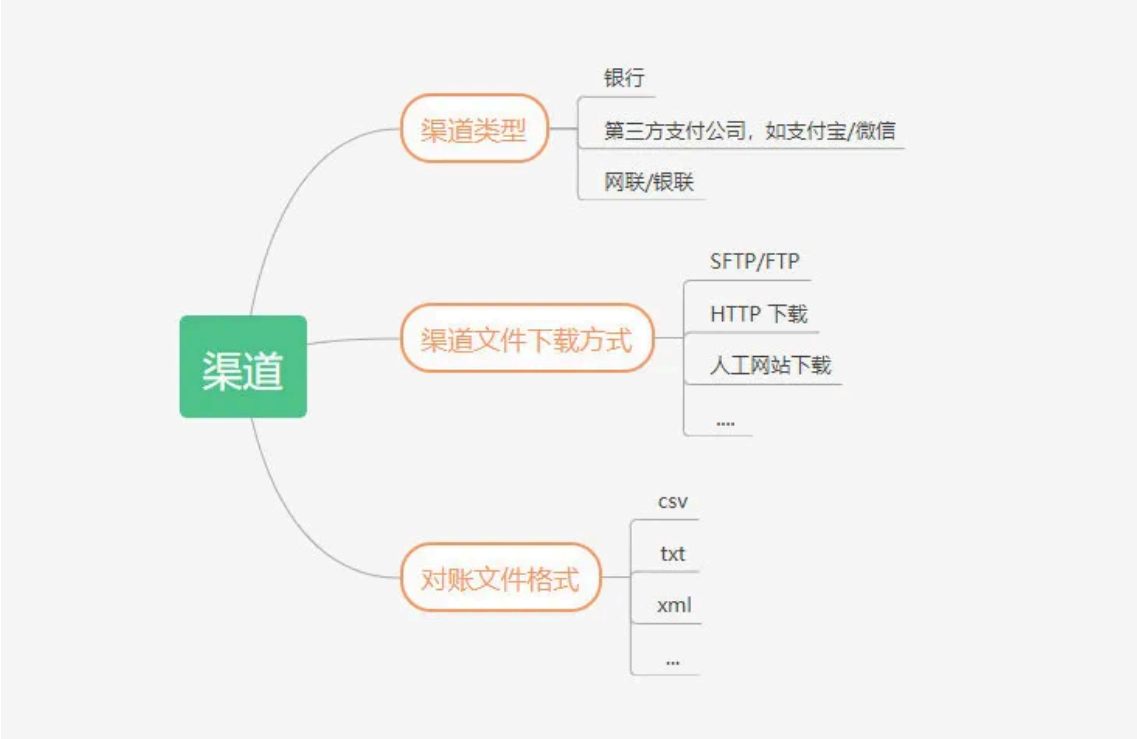



整体流程可以简单分为三个模块：

- 本端数据处理
- 对端数据处理
- 本端数据与渠道端数据核对

本端数据指的是我们应用产生的支付记录，这里根据账期（交易日期）与渠道编号获取单一渠道的所有支付记录。

对端数据指的是第三方支付渠道支付记录，一般通过下载对账文件获取。




由于每个渠道下载方式，文件格式都不太一样，对端数据处理的时候需要将其转化统一数据格式，标准化在入库存储。

网上找了一份通用账单，可以参考：

```java
// 获取某一渠道本地支付数据
List<LocalDataDTO> localDataList = getLocalPayData("支付渠道编号", "账期");
// 获取某一渠道对账单数据
List<ChannelDataDTO> channelDataList = getChannelPayData("支付渠道编号", "账期");

Map<String, LocalDataDTO> localDataMap =
        localDataList.stream().collect(Collectors.toMap(localData -> localData.getOrderNo() +
localData.getAmount(), o -> o));


Map<String, ChannelDataDTO> channelDataMap =
        channelDataList.stream()
                .collect(Collectors.toMap(channelData -> channelData.getOrderNo() +
channelData.getAmount(), o -> o));

List<LocalDataDTO> localDiffDetails = new ArrayList<>();

// 用本地账单数据的键值逐笔核对
localDataMap.forEach((key, localDataDTO) -> {
    if (channelDataMap.containsKey(key)) {
        channelDataMap.remove(key);
    } else {
        // 渠道账单数据不包含本地数据键值
        localDiffDetails.add(localDataDTO);
    }
});

// 若 channelDataMap 里面还有元素将其放入 channelDiffDataList

List<ChannelDataDTO> channelDiffDataList = new ArrayList<>(channelDataMap.values());
// 记录差异数据
recordDiffData(localDiffDetails,channelDiffDataList);
```

对端数据转化存储之后，对账流程中，对端数据也需要跟本端数据一样，获取当前账期下所有记录。

两端数据都获取成功之后，接下来就是本地数据逐笔核对。

核对流程可以参考之前写的流程：

| 字段名 | 字段含义 | 额外说明 |
| --- | --- | --- |
| ID | ID主键 | 账单表唯一主键，如果是Mysql可以用自增ID，如果是TIDB或Hive则建议用UUID |
| BILL_DATE | 账单日期 | 账单日期，即第三方账单的下发日期，对于国内渠道一般为结算日期 |
| FILE_ID | 账单文件编号 | 文件编号，下载的原始账单文件在完成标准格式转换后会生成一个唯一的文件ID便于检索 |
| CHANNEL | 渠道编码 | 渠道编码，可以根据支付平台对对接渠道的内部定义进行转换 |
| CHANNEL_NAME | 渠道名称 | 渠道名称，如支付宝、微信这样 |
| SUB_CHANNEL | 二级渠道编码 | 这里是为了适配在使用支付渠道事存在第四方支付的情况，或下级渠道的情况 |
| SUB_CHANNEL_NAME | 二级渠道名称 | 二级渠道名称，如通过Ping++对接了QQ支付，这里的二级渠道就是QQ支付 |
| ORDER_ID | 平台订单号 | 这里是指，支付平台与第三方渠道交互的系统唯一订单号 |
| TRADE_TYPE | 交易类型 | 支付平台根据交易类型进行的定义，如charge表示支付;refund表示退款 |
| PAY_TYPE | 支付类型 | 第三方渠道对其支付产品的定义在自己系统中的统一转换编码，如微信APP支付 |
| TRADE_NO | 渠道支付订单号 | 第三方支付渠道生成的渠道支付订单号 |
| TRADE_TIME | 交易时间 | 交易发生的事件，可以统一格式为 "YYYY-MM-DD HH:MM:SS" |
| STATUS | 交易状态 | 交易状态转换，可根据支付平台定义的交易转换对账单中的状态进行映射转换 |
| ORD_AMT | 支付金额 | 支付订单金额，为了统一处理，一律转换为最小单位，如 "分" |
| CURRENCY | 币种 | 支付币种，根据国籍支付币种进行统一转换，如cny,eur等 |
| FEE | 支付手续费 | 支付手续费总额 |
| FEE_DETAIL | 手续费明细 | 支付手续费明细，如海外渠道手续费可能分为各种费，这里可以记录费用明细 |
| FEE_RULE | 手续费规则 | 对手续费规则的定义，如果存在计费系统或者对渠道计费有管理，这里可以填充相应标示 |
| CH_MER_ID | 渠道商户号 | 对应的渠道商户号，如同一个渠道可能申请多个商户号，这里可以进行区分 |
| COUNTRY | 国家编码 | 根据交易订单中的收单国家情况进行定义，可以设计成统一的国际国家编码，如中国86这样 |
| CITY | 城市编码（新增） | 城市，这个字段对于国内渠道来说可能意义不大，但是有些海外渠道则可能会有区别 |
| USER_FLG | 渠道用户标识 | 第三方支付渠道用户公开表示，如微信OpenId这样，便于检索排查 |
| REFUND_ORIGIN_ORDER | 退款原订单号 | 如果为退款账单数据，这里可以记录其原始的支付订单号 |
| DESC_ | 支付信息描述 | 关于支付备注信息的描述 |
| CHANNEL_TRADE_TYPE | 渠道原始交易类型 | 第三方原始交易类型的表述，之所以加这个，在于有些渠道，如银联类型太多，便于排错 |
| META_DATA | 支付原数据 | 对于目前很多支付渠道，都具备传输原数据的功能，即你传什么数据，账单就给你返回什么，可以用于平台自己的一些特殊业务标记 |
| CREATE_TIME | 创建时间 | 数据入库时间 |
| UPDATE_TIME | 更新时间 | 数据最后被更改的时间 |
| EXTEND_1 | 扩展1 | 扩张字段1 |
| EXTEND_2 | 扩展2 | 扩张字段2 |
| EXTEND_3 | 扩展3 | 扩展字段3 |



上面流程其实也比较简单，翻译一下：

查找本端数据/对端数据，然后转化存储到 Map 中，其中 key 为订单号，value 为本端/对端订单对象。

然后遍历本端数据 Map 对象，依次去对端数据 Map查找。如果能查找到，说明对端数据也有这笔。这笔核对成功，对端数据集中移除这笔。

如果查找不到，说明这笔数据为差异数据，它在本端存在，对端不存在 ，将其移动到差异数据集中。

最后，本端数据遍历结束，如果对端数据集还存在数据，那就证明这些数据也是差异数据，他们在对端存在，本端不存在 ，将其也移动到差异数据集中。

> PS:上述流程存在瑕疵，只能核对出两边订单互有缺失的流程 ，但是实际情况下还会碰到两边订单都存在，但是订单金额却不一样的差异数据。这种情况有可能发现在系统Bug，比如渠道端上送金额单位为元，但是实际上送金额单位为分，这就导致对账两端金额不一致。

之前对账系统日均处理的支付数据峰值在几十万，所以上面的流程没什么问题，还可以抗住，正常处理。

但是目前的支付数据日均在千万级，如果还是用这种方式对账，当前系统可能会直接崩了。。。

## 千万数据级带来的挑战

第一个，查询效率。

本端/对端数据通过分页查询业务数据表获取当天所有的数据。随着每天支付数据累计，业务表中数据将会越来越多，这就会导致数据查询变慢。

实际过程我们发现，单个渠道数据量很大的情况下，对账完成需要一两个小时。

虽然说对账是一个离线流程，允许对账完成时间可以久一点。但是对账流程是后续其他任务的前置流程，整个对账流程还是需要在中午之前完成，这样运营同学就可以在下午处理。

第二个问题，OOM。

上面流程中，我们把把全部数据加载到内存中，小数据量下没什么问题。

但是在千万级数据情况下，数据都加载到内存中，并且还是加载了两份数据（本端、对端），这就很容易吃完整个应用内存，从而导致 Full GC，甚至还有可能导致应用 OOM。

而且这还会导致级联反应，一个任务引发 Full GC，导致其他渠道对账收到影响。

第三个问题，性能问题。

原先系统设计上，单一渠道对账处理流程只能在单个机器上处理，无法并行处理。

这就导致系统设计伸缩性很差，服务器资源也被大量的浪费。

## 千万数据级对账解决办法

上面系统代码，实际上还是存在优化空间，可以利用单机多线程并行处理，但是大数据下其实带来效果不是很好。

那主要原因是因为发生在系统架构上，当前系统使用底层使用 MySQL 处理的。

传统的 MySQL 是 OLTP （on-line transaction processing），这个结构决定它适合用于高并发，小事务 业务数据处理。

但是对账业务特性动辄就是百万级，千万级数据，数据量处理非常大。但是对账数据处理大多是一次性，不会频繁更新。

上面业务特性决定了，MySQL 这种 OLTP 系统不太适合大数据级对账业务。

那专业的事应该交给专业的人去做，对账业务也一样，这种大数据级业务比较适合由 Hive、Spark SQL 等 OLAP去做。

## 总结

今天本篇文章只是一个序曲，主要聊聊对账业务基本流程，聊聊之前系统架构在大数据下存在的问题。

后面文章再会介绍下大数据下对账系统如何设计，对账之后差错数据如何处理，尽请期待。

支付 3　系统架构设计 5　#对账系统 4

系统架构设计 · 目录

上一篇
进来偷学一招，数据归档二三事儿

下一篇
千万级支付对账系统怎么玩（上篇）？

阅读原文