基于ETL的地铁客运状况实时分析系统

摘要

近年来，地方政府越来越重视以地铁为代表的城市轨道交通建设。地铁的普及大大改善城市地面交通拥堵和城市交通环境，使人们出行更加方便。分析和准确预测地铁客流一直是城市轨道交通管理部门的主要任务之一。特别是在轨道交通运力快速增长、乘客需求变化，人口稠密的城市越来越高度依赖交通系统的背景下，对客流分析的需求以及实现可靠，高效的资源利用和交通管理更为迫切。幸运的是，近几年，实时 ETL 技术不断成熟，可以充分的为地铁交通客流分析赋能。实时ETL的提取、转换和加载过程包括： 从外部来源提取数据； 将其改造以适应分析需要；将其加载到最终的目标数据库或数据仓库中。本文基于 ETL 和 Flink 等大数据技术，利用深圳地铁刷卡数据集，模拟实时刷卡场景，完成了对深圳地铁客流的分析。此设计的完成对解决城市拥堵问题，优化交通网络，保障公共交通，具有十分重要的意义。本文的主要研究为，调查地铁发展现状，分析了地铁发展中遇到的问题；学习kafka 系统，将redis 处理后的海量数据，流式的消费到 kafka topic 中，为下一阶段做准备；研究flink技术，模拟实时的流式传输，构造实时的 ETL 系统；深入学习 OLAP 数据库 clickhouse, 研究数据仓库的建模，作为数据的最终流向，可提供快速的分析结果；调研可视化方案，展示出实时清晰的动态结果。kafka, flink, clickhouse 都被阿里巴巴，腾讯，字节跳动等大型互联网公司采用，并且有着活跃开源社区的大数据技术，所以研究具有切实的可行性。

**关键词**

地铁；大数据技术；ETL；FLink；实时分析；数据仓库；kakfka；clickhouse；数仓建模

### 

# 绪论

## 研究背景

图表, 条形图

描述已自动生成城市轨道交通的飞速发展，已经大幅度减轻了城市地面交通的拥挤情况，不仅让人们的出行更加便利，而且带动了城市经济的发展。地铁虽然故障率低，运输量大，稳定安全，但是在一线城市，以深圳为例，大批的年轻人，打工族涌入，城市交通的负载进一步加重。到 2021 年为止，深圳总共有 11 条线地铁线路在运营，而 2021 年一月，10条(不包含深港地铁四号线)运营线路月客运量为16316万人次，各线客运量如下图所示。

## 研究现状

一些研究基于地铁刷卡数据分析乘客的出行行为；特征提取：根据行人的出行记录，对行人到达时间进行聚类，得到乘客的固定出行天数，得到乘客的出行特征；旅客聚集：根据旅客的出行特点，进行乘客聚类。这个研究只是将乘客进行划分，可以对乘客进行定向推送特定的地铁信息，或者地铁站用来分析地铁高峰；但是这并不具有实时性，也没有将结果进行动态展示，而且分析的结果单一，只是对乘客进行了聚类，实际作用很局限。

另一项研究基于智能交通卡数据，从多空间尺度研究地铁客流的时空分布特征，为城市公共交通需求管理提供科学依据，只是利用基本数据分析方法，将搜集的数据通过一些数学计算和图表分析，**引用1: 从全天、早高峰和晚高峰时段、市辖区尺度、环 路尺度、街道尺度，对北京市地铁客流的时空分布特征进行了系统性分析**，并没有利用当前的先进的大数据技术，不具有实时性和动态性。

关于 ETL技术，传统上，ETL被安排为夜间批处理过程，以在非工作时间刷新数据。但是，随着科技的进步，24X7全天候业务运营模式，市场竞争以及引入现代数据源，已迫使人们采取以下行动：更频繁地刷新其数据仓库，以提供近乎-实时环境。传统的ETL流程难以应对具有近实时环境的要求，因为它没有为此目的进行设计和优化。结果，更先进的ETL版本才能满足近乎实时环境，例如低延迟，高可用性和高扩展性。本文将模拟搭建一个实时 ETL 系统，用来进行研究。

## 研究内容和意义

客流量不断增加，但是地铁建设不能盲目增加，要和城市环境，地下空间相协调，于是促使着我们寻求新的方案来进一步提高地铁的便利性，提高人们的乘车幸福感。互联网，大数据的爆发式发展，智慧城市，智慧生活的普及，促使着我们把目光转向物联网和大数据技术，物联网和大数据技术带来的工业 4.0 革命极大的影响了公共生活方式，刷卡进站逐渐演变为刷 APP 二维码即可，我们获取数据的方式越来越便捷，这是一个数据丰富和累积前所未有的时代。假如我们把这些数据利用起来，利用数据挖掘，实时计算，流式分析等大数据技术，实时提供高质量的交通运输信息，让人们实时的获取动态数据，方便规划行程或者采取其他交通方式，同时也方便交通部门分析民众的出行模式，提前规划城市交通的未来发展。研究发现，随着大数据时代的发展，人们逐渐意识到传统 ETL 技术已经不能满足业务的需求，因为其加载周期是固定的，比较长的，一般是天级，周级，不能及时的对数据变化作出响应，与之对应的是实时 ETL 系统，其作为传统 ETL 的延伸，可以捕捉数据的快速变化并对数据做出实时的分析。实时数仓的发展经历了三个过程：Lambda架构在离线数据仓库的基础上增加实时数据，通过流计算引擎处理实时业务；Kappa架构直接采用实时部分，删去了多余的离线处理；几年来，分析型数据库系统的发展越来越快，具有代表性的有阿里的 analyticDB 以及 俄罗斯的开源数据库Clickhouse，于是新的实时架构诞生了，这种体系结构的优点是部分计算从流计算引擎（如Flink和spark）转移到OLAP分析引擎。本文将采用实时 OLAP 架构的 ETL 技术对深圳地铁的130多万条数据进行模拟，分析和展示数据结果，对行人出行，交通状况提出建议，帮助交通部门正确的规划下一步发展。

## 其余部分安排

本文的其他部分的结构如下。第二节相关工作介绍了本文使用的具体技术，如何利用大数据技术将数据集模拟为实时动态流，从而进行实时分析。第三节中，对系统的架构进行了分析，并且对架构中每一个模块都进行了功能和实现的分析描述。在第四节中，讲述了数据仓库的建模，结合具体的场景，对ODS，DWD 等数据层的数据库表结构进行设计。第五节将详细的介绍系统的每一部分设计以及功能是如何实现的，通过流程图，类图等清晰的展现出来。在第六节中，我介绍了在系统设计和实现的过程中遇到的困难以及我是怎么解决的。最后一节，展示了系统的运行效果，并对系统进行展望，分析有哪些不足，未来该如果改进。

# 相关工作

获取133.7w条深圳地铁刷卡数据，将这些离线数据作为源头，通过redis、Kafka、Flink和Clickhouse的数据链，构造实时 ETL 系统，最终将分析结果实时的动态展现出来。

搭建 ETL 实时系统。顾名思义，ETL具有三个主要方面阶段，每个阶段都照顾整体的特定部分数据刷新过程。提取阶段从异构数据系统读取数据并将其存储到中间存储中。转换阶段读取提取的数据，开展业务规则和基本数据操作，例如清理，数据类型转换等。在加载阶段，转换后的数据最终加载到目标数据仓库中。通常，数据仓库是通过按预定义的时间执行ETL流程，每天晚上刷新。这样可以确保用户可以在工作时间和ETL流程中使用最新的数据，无需与用户查询竞争资源。少数其他因素，例如业务需求最近的数据，数据仓库的处理能力和源数据的性质也可能会产生影响关于数据刷新的频率。本系统属于实时范畴。

采用 redis 来存储源数据，Redis是一个高性能的K-V数据库，很多公司将其作为缓存层，在本项目中，将其作为持久化存储数据库。因为 Redis 具有自然去重的特性，使用 HSET 命令可以将 json 文件中的数据无重复的写入到 redis 中。其写入性能相当好，可以到达 8.1w次/s，节省了很多时间。

Kafka作为一个具有高吞吐量、持久性和支持数据流处理的分布式流计算平台，在这个项目中起着连接生产者和消费者的作用，即 Redis 和 Clickhouse。kafka 有三个角色：消息系统、存储系统和流处理平台。本项目采用了消息系统和流式处理平台两个角色，消息系统和流式处理平台，作为消息系统，其体系结构中引入了生产者和消费者，生产者是发送消息并负责创建消息的一方，通过 flink 的连接，redis 中的数据被送到 kafka 中，消费者是接收消息的一方，通过 flink 的连接，clickhouse 获取到了来自 kafka 的数据。Kafka 具有高吞吐，低延时和高可用的消息传输能力，这使得其成为流式系统中十分优秀的数据来源，例如，Apache Flink将其用作可靠的数据源，本项目也是这样来构造数据流式链路的。kafka 的生态十分丰富，在进行 ETL 的过程中，可以使用Kafka eagle监视数据流量。

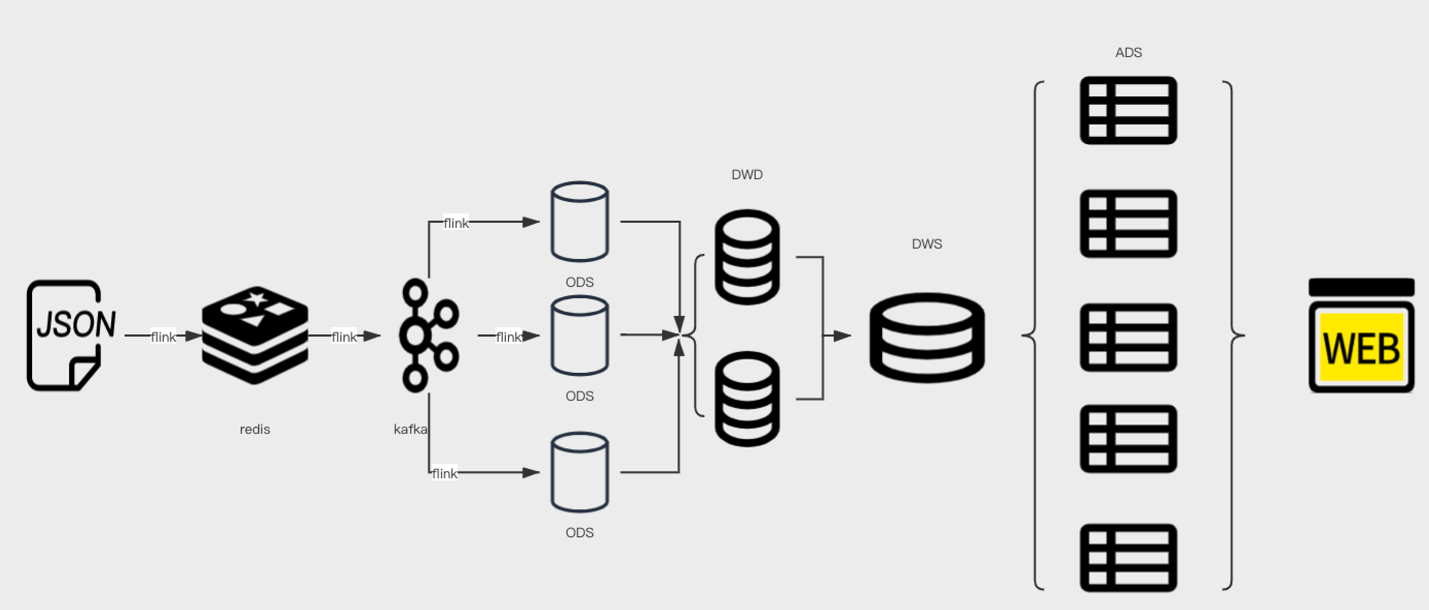
再谈谈 Flink 在该项目中发挥的作用，Flink是一个用于处理流和批数据的开源系统，在实时 ETL 中的充当着重要的角色，从头到尾都可以看到 Flink 的影子。Apache Flink遵循一个例子，该范例将流及批处理在编程模型和执行引擎中统一。流处理结合了持久消息队列（如 kafka），在实时处理最新事件、在大窗口中周期持续的聚合数据或处理数 TB 的历史数据之间不作区分。这些不同类型的计算只是在持久流的不同点开始处理，并在计算过程中保持不同形式的状态。通过高度灵活的窗口机制，**引用：Flink 可以用相同的操作执行 lambda 架构的多路计算。Flink 支持不同的时间概念（事件时间、进入时间、处理时间）**，以便让程序员在定义事件应如何关联方面具有高度的灵活性。本项目中大量使用了 Flink 提供的 API，完成 ETL 的三个阶段，并且模拟出了实时的效果。

Clickhouse是最流行的OLAP数据仓库之一，由俄罗斯yandex开发并开源，该数据库的特点包括，在读多写少的场景下具有极佳的性能，我的项目正好是这一场景，需要频繁的读取数据进行分析，又应为实时的原因，要求读数据的速度很快；在大宽度表中，读取大量行，但只读取少数列，不需要全部读取；采用列式存储，列存在分析性场景中有很多的优势，如果不需要参与计算的列被读取出来，势必会导致读放大，采用列存后，它只需要将计算中涉及的列读入内存，这大大减少了磁盘I/O，查询速度大大加快，更有利于实时场景。在分析前时，需要多某些列进行计算，clickhouse 在计算层做了很多优化，划分 partition, 充分发挥 CPU 的多核心实现数据的并行处理，这样一来，一条 query 就可以将大部分的 CPU 都利用起来，从根本上降低查询的延迟。Clickhouse 内部实现了向量执行引擎，并且在内存中的数据也按列存储，这样可以提高缓存命中率，内部计算按批执行，一批计算调用一次 SIMD 指令，可以最大化发挥 SIMD 并行能力。

前端展示采用了 EasyV 提供的可视化大屏展示服务，可以快速上手，通过查询 Clickhouse，将得到的结果按照显示在面板的对应的部分，实时的对结果进行更正，清晰的将分析结果展示给乘客。

# 系统架构设计与分析

## 项目架构设计



图示

描述已自动生成整条数据流链路分为四段，每一段都需要将数据完整的，实时的，可靠的，高效的输送的下一个阶段。鉴于此目的，整个过程采用Flink编程模型和 api，因为 Flink 的架构十分清晰，APIs 方便快捷。Flink 的架构氛围为软件栈和分布式系统。尽管Flink的API堆栈仍在增长，但可以区分四个主要层：部署、核心、APIs 和库。

将深圳地铁刷卡数据流式写入 redis 中，供 kafka 消费，Flink的 Java 数据流 API 可以将任何可序列化对象转换为流，将 json 数据集作为数据源，通过 flink streaming 流式的将数据传输到 redis，同时启动 Kafka，redis 作为生产者，源源不断的将过滤后的数据输送到 kafka 的 topic 中，线程采取随机睡眠的方式尽可能模拟出真实的流式场景。最后采用 OLAP 数据库 clickhouse 作为消费者，通过 flink 可以流式的消费 kafka topic 中的数据，作为 ODS 存储起来。json 文件到 redis, redis 到 kafka, kafka 到 clickhouse 这些环节都可以看作 flink 流式应用，下面讲解一个 flink 流式应用的执行过程，这个过程中我们需要做些什么，从而保证整条流的稳定高效。

因为项目的需求是实时和流式，所以采用进行流处理的 DataStream API。实现Flink流应用程序的第一步是设置执行环境，执行环境可以是集群或者本地机器，本项目直接在本地机器上执行；配置执行环境后，需要创建流数据源，这些数据源可以来自文件或者消息队列，也可以是实时产生的，比如传感器数据，web 页面点击产生的数据，创建数据源的最终目的是将其传到下一个应用中。拿到数据源后，需要对其进行转化，生成新的数据源。然后新生成的数据源会被发送到外部系统，在本项目中，外部系统包括，redis, kafka, clickhouse。flink 使用延迟执行的方法，上述环境配置，数据源生成和数据转化不会触发 flink 的计算，只有调用 execute() API 后，才会执行它。执行任务交给 JobManager, 最终到达外部系统。

我们最终的目的是根据 ODS 中的实时数据，分析出地铁的流量，高峰期，不同车站的人流等等，并实时的进行展示。所以需要严格的按照数据仓库的建模步骤进行建模。首先需要对 ODS 进行分层，分为 DWD(数据细节层) 和 DWS(数据汇总层)。DWD 是业务层与数据仓库之间的隔离层。主要对ODS数据层进行数据清洗和标准化操作。数据汇总层 DWS，是在 DWD 基础数据的基础上，集成并汇总成一个务数据层，用于分析一个主题领域，通常是一个宽表，用于提供后续业务查询，OLAP分析等。然后根据展示需要，从不同的维度构建维度表, 从而形成星型表结构，查询分析各维度表，得到最终的展示信息。

如何进行最终的结果展示，调研了 Datav, FindReport, echart, EasyV 等多种可视化工具后，发现EasyV 很好的支持了 clickhouse，并且使用起来相对容易上手，所以最终决定使用 EasyV 将从维度表中分析出来的数据结果展示在前端上。

非功能性需求包括，数据一致性，系统整体容错性，稳定性，流量控制。这些在 flink 中都能得到保障。Flink 通过严格的一次处理一致性保证提供可靠的执行，并通过检查点和部分重执行处理故障。系统为有效提供这些保证而作出的一般假设是，数据源是持久和可重播的。此类数据源的例子包括文件和持久消息队列（例如 Kafka）。在实践中，也可以通过在源数据算子的状态内保留WAL来合并非持久数据源。Apache Flink 的检查点机制以分布式一致性快照的概念为基础来实现一次处理保证。数据流可能无界的性质使得在恢复时重新计算不切实际，因为可能需要重播数月的计算，这是长期的工作。为了限制恢复时间，Flink定期保存操作符状态的快照，包括输入流的当前位置。

Flink 通过设置并行度的方法来控制流量的大小。将数据流提交给作业管理器后，生成的DAG就可以执行该算子了，每个算子可以产生一到多个任务，并行度就是执行任务的数目。

# 数据库设计与分析

## ODS table （原始数据库表）

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| deal\_date | String | null | 进站(交易)时间 |
| close\_date | String | null | 出战(结算)时间 |
| card\_no | String | null | 卡号 |
| deal\_value | String | null | 实收费用 |
| deal\_type | String | null | 交通类型 (进站/出站/巴士) |
| company\_name | String | null | 地铁线名 |
| car\_no | String | null | 车号 |
| station | String | null | 站名 |
| conn\_mark | String | null | 联程标记 |
| deal\_money | String | Null | 应收费用 |
| equ\_no | String | Null | 闸机号 |

## DWD table

DWD 是 ODS 清洗和降维后的结果。

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| deal\_date | String | null | 进站(交易)时间 |
| close\_date | String | null | 出战(结算)时间 |
| card\_no | String | null | 卡号 |
| deal\_value | String | null | 实收费用 |
| deal\_type | String | null | 交易类型 (进站/出站) |
| company\_name | String | null | 地铁线名 |
| car\_no | String | null | 车号 |
| station | String | null | 站名 |
| conn\_mark | String | null | 联程标记 |
| deal\_money | String | Null | 应收费用 |
| equ\_no | String | Null | 闸机号 |

然后进一步降维，分为进站表和出站表

进站表

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| deal\_date | String | null | 进站(交易)时间 |
| card\_no | String | null | 卡号 |
| deal\_type | String | null | 交易类型 (进站) |
| company\_name | String | null | 地铁线名 |
| car\_no | String | null | 车号 |
| station | String | null | 站名 |
| equ\_no | String | Null | 闸机号 |

出站表

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| deal\_date | String | null | 进站(交易)时间 |
| close\_date | String | null | 出战(结算)时间 |
| card\_no | String | null | 卡号 |
| deal\_value | String | null | 实收费用 |
| deal\_type | String | null | 交易类型 (出站) |
| company\_name | String | null | 地铁线名 |
| car\_no | String | null | 车号 |
| station | String | null | 站名 |
| conn\_mark | String | null | 联程标记 |
| deal\_money | String | Null | 应收费用 |
| equ\_no | String | Null | 闸机号 |

## DWS 表

根据具体的业务，并基于 DWD 表得到以下的 DWS 表

1.站点进站人数排行榜

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| station | String | Null | 站名 |
| deal\_dates | ARRAY | Null | 进站(交易)时间集合 |
| card\_nos | ARRAY | Null | 卡号集合 |
| company\_names | ARRAY | Null | 地铁战线集合 |
| equ\_nos | ARRAY | Null | 闸机号集合 |
| Rank | Int | Null | 排名 |

2.站点出站人数排行榜

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| station | String | Null | 站名 |
| deal\_dates | ARRAY | Null | 进站(交易)时间集合 |
| card\_nos | ARRAY | Null | 卡号集合 |
| deal\_values | ARRAY | Null | 实收费用集合 |
| company\_names | ARRAY | Null | 地铁战线集合 |
| conn\_marks | ARRAY | Null | 联乘标记集合 |
| deal\_moneys | ARRAY | Null | 应收费用集合 |
| equ\_nos | ARRAY | Null | 闸机号集合 |
| Rank | Int | Null | 排名 |

3.站点进出站人数排行榜

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| station | String | Null | 站名 |
| deal\_dates | ARRAY | Null | 进站(交易)时间集合 |
| card\_nos | ARRAY | Null | 卡号集合 |
| deal\_values | ARRAY | Null | 实收费用集合 |
| deal\_types | ARRAY | Null | 交易类型集合 |
| company\_names | ARRAY | Null | 地铁战线集合 |
| conn\_marks | ARRAY | Null | 联乘标记集合 |
| deal\_moneys | ARRAY | Null | 应收费用集合 |
| equ\_nos | ARRAY | Null | 闸机号集合 |
| Rank | Int | Null | 排名 |

4.站点盈利排行榜

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 备注 |
| station | String | Null | 站名 |
| company\_name | String | Null | 地铁战线 |
| deal*money*sum | Float64 | Null | 应收费用集合 |
| deal*value*sum | Float64 | Null | 实收费用集合 |

# 功能详细设计

## 功能设计

Redis是最流行的存储中间件，它以超高的性能，清晰的文档，简洁的源码被程序员一致好评。国内外大型互联网公司都在用。本项目中主要采用了 redis 的存储功能，离线数据通过 flink api 写入 redis。redis 有很多数据结构，项目中使用了哈希对象，创建 FlinkJedis 配置后，通过提供的 RedisSink将数据以 page 为 key 插入到 redis 中。目前 redis 的 HashMap 有两种实现方式：当HashMap的成员比较少时，Redis会使用类似一维数组的方式来节省内存，而不会使用真正的HashMap结构，此时，对应的 value 的 redisObject 被编码为 zipmap，当键值对的数目增加时，redis 会自动转换成真正的 HashMap, 编码也会变为 ht，hashmap 的 key 是不可重复的，可以对数据进行一次去重操作。RedisSink 是使用Jedis客户端将数据传递到Redis通道的 sink，此 sink 可以使用三种不同的方法与不同类型的Redis环境通信：1.单个Redis服务器 2. Redis集群 3. Redis哨兵。项目中使用了单个服务器完成通信。

1. public class JsonToRedis {
2. public static void main(String[] args) throws Exception {
3. StreamExecutionEnvironment streamExecutionEnvironment = StreamExecutionEnvironment
4. .getExecutionEnvironment();
5. *// get flink data stream*
6. DataStreamSource<String> datastream = streamExecutionEnvironment
7. .readTextFile("dataset/dataset.json").setParallelism(1);
8. ...
9. *//for sending data to Redis.*
10. parsed.addSink(new RedisSink<JSONObject>(conf, new RedisSinkFunc()));
11. streamExecutionEnvironment.execute("JSON to Redis");
12. }
13. }
14. class RedisSinkFunc implements RedisMapper<JSONObject> {
15. @Override
16. public RedisCommandDescription getCommandDescription() {
17. return new RedisCommandDescription(RedisCommand.HSET, "pageJSON");
18. }
19. @Override
20. public String getKeyFromData(JSONObject jsonObject) {
21. return jsonObject.getInteger("page").toString();
22. }
23. ...
24. }

由于使用多路复用 IO 阻塞机制，数据结构简单，操作便捷使得数据在写入 redis 过程中十分的高效。因为 redis 的瓶颈不在 CPU 上, 而主要在网络带宽和内存大小上，redis 是单线程的，单线程切换的成本非常小，可以避免不必要的竞争和加锁，进一步提高 Redis 的性能。通过 RDB 持久化，数据可以在某个时间写入磁盘，可以防止数据丢失，安全可靠。

kafka 已经成为互联网公司数据管道的关键部分。kafka 是一个分布式消息传递系统，它能够收集和传递大量低延迟的数据，适合在线消息的使用。kafka 的体系架构包含N个生产者， N个消费者, N 个 Broker 和一个 zookeeper 集群，Broker 负责将接受到的消息存到磁盘。生产者负责向 Broker 发送消息, 消费者负责从 Broker 获取消息。此外，Kafka 还提出了 topic 和 partition 的概念，生产者会把消息发送到指定的 topic, 然后消费者通过订阅 topic 拿到消息。生产逻辑由以下步骤组成

1. 参数配置：包含客户端参数和生产者实例
2. Properties properties = new Properties();
3. properties.setProperty("bootstrap.servers", "");
4. FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<String>(
5. "my-topic",
6. new ProducerStringSerializationSchema("my-topic"),
7. properties,
8. Semantic.EXACTLY\_ONCE);
9. 创建需要发送的消息，即 redis 中的数据源，这里要再进行一下过滤
10. class FilterRedisSource extends RichSourceFunction<String> {
11. @Override
12. public void run(SourceContext<String> sourceContext) throws Exception {
13. jedis\_ = getJedisClient();
14. for(int i = 1; i <= PAGECNT; ++i) {
15. String value = jedis\_.hget("pageJSON", i + "");
16. *// System.out.println(value);*
17. JSONObject jsonVal = JSON.parseObject(value);
18. *// {"data": [{}, {}, {}]}*
19. JSONArray dataArray = jsonVal.getJSONArray("data");
20. for (Object element: dataArray) {
21. String val = element.toString();
22. JSONObject jVal = JSON.parseObject(val);
23. if(jVal.size() == ELESIZE) {
24. sourceContext.collect(val);
25. }
26. }
27. }
28. }
29. ...
30. }
31. 消息构建完成后，需要借助 Flink 的数据流 API 发送，首先需要创建StreamExecutionEnvironment，然后把DataSource 加入到执行环境中，执行即可，这样消息就会被流式的发送。
32. public class RedisToKafka {
33. public static void main(String[] args) throws Exception {
34. StreamExecutionEnvironment streamExecutionEnvironment = StreamExecutionEnvironment
35. .getExecutionEnvironment().setParallelism(1);
36. Properties properties = new Properties();
37. properties.setProperty("bootstrap.servers", "");
38. SingleOutputStreamOperator<String> dataStreamSource = streamExecutionEnvironment.addSource(new FilterRedisSource()).map(
39. (MapFunction<String, String>) s -> {
40. *//Simulate real-time state*
41. Thread.sleep(ThreadLocalRandom.current().nextInt(0, 10));
42. return s;
43. });
44. FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<String>(
45. "my-topic",
46. new ProducerStringSerializationSchema("my-topic"),
47. properties,
48. Semantic.EXACTLY\_ONCE);
49. dataStreamSource.addSink(producer);
50. streamExecutionEnvironment.execute("Redis To Kafka");
51. }
52. }
53. 最后，如果已经没有消息需要发送了，就可以关闭生产者
54. public void cancel() {
55. jedis\_.close();
56. }

数据源源不断的从 redis 流入到 kafka 的 topic 中，此时需要消费者及时的从 kafka 的 topic 中取出数据，流入到 clickhouse 中。消费者的消费逻辑由以下步骤组成

1. 参数配置：包含消费者客户端参数和消费者实例，需要注意的是，kafka 的消费组成中有一层消费组，每个消费者跟一个消费组对应。实际上topic 是被消费组订阅的，消息一旦发布，会被投送到订阅这个消息的消费组里的消费者中。所以需要指定一下消费组的 id。消费者和消费组这种模式有一个好处就是可以提高消费能力的横向拓展性，通过增加消费者或者减少消费者来提高或者削弱消费能力。
2. Properties properties = new Properties();
3. properties.setProperty("bootstrap.servers", "10.227.89.202:9092");
4. properties.setProperty("group.id", "test-consumer-group");
5. 消费组订阅指定的 topic, 将 kafka topic 中的数据作为 DataStreamSource。
6. SingleOutputStreamOperator<String> dataStreamSource = streamExecutionEnvironment
7. .addSource(new FlinkKafkaConsumer<>(
8. "my-topic",
9. new SimpleStringSchema(),
10. properties).setStartFromEarliest()
11. ).map((MapFunction<String, String>) s -> {
12. Thread.sleep(ThreadLocalRandom.current().nextInt(0, 10));
13. return s;
14. });
15. 指定消息的去处，即Data Sink。这里是将数据消费到 clickhouse 中，为了保证数据的稳定传输，这里使用 jdbc 进行连接
16. dataStreamSource.addSink(new FlinkToCK("10.227.89.202", 8123 + "", "default", "123"));
17. public void open(Configuration param) throws SQLException {
18. this.connection\_ = DriverManager.getConnection("jdbc:clickhouse://" + host\_ + ":" + port\_, user\_, pwd\_);
19. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS test (" +
20. "first Nullable(String)) ENGINE = Memory()");
21. }
22. 一切就绪后，就可以在流式执行环境中拉取消息进行消费

1.streamExecutionEnvironment.execute("flink consume kafka topic");

5．消费完成后，关闭连接器

1. public void close() throws SQLException {
2. connection\_.close();
3. }

至此，数据被存到了OLAP 分析型数据库 clickhouse 中，数据仓库建模完成后，可以把原始数据进行多维度的细化，用于最后的分析场景中。

上述功能的实现，都离不开 flink, flink 贯穿整条功能链。Flink 是一个开源流式框架，它的高性能，高可用，以及易用性保证了实时场景的稳定模拟，Flink 的特点是计算是有状态的，可以实时的记录计算状态，Exactly-once 可以保证失败恢复可以正确的进行。状态在计算中无处不在，由于实时场景中的数据一般是无限的，所以为了避免内部状态的无限增长，算子一般保留的是一个摘要状态，可以节省计算资源。Flink 中蕴含着 DataFlow 的编程思想，其描述了不同操作之间的数据如何流动，DataFlow 表示为 DAG，即有向无环图。Flink 程序的执行过程是一个接一个的执行 DAG, 可以并行进行，从而更好的利用计算资源。

本项目无时不刻的在使用 Flink DataStream 的 API，包括 DataSource, Transformation, Sink 等等。DataSource 的含义是数据源，通过下面的方式就可以添加数据源，这时候可以指定一下并行度，默认情况下，算子的并行度与算子执行的应用环境并行度相关联。

1. StreamExecutionEnvironment
2. .getExecutionEnvironment().addSource();

Transformation 是对数据的具体操作，比如 Map, Filter等。项目中主要用到的是 map， 直接通过一个 lambda 算子就可以对数据进行过滤或者类型转换，lambda 算子提供了简明的方式实现具体的操作逻辑。

1. (MapFunction<String, String>) s -> {
2. Thread.sleep(ThreadLocalRandom.current().nextInt(0, 10));
3. return s;
4. }

Sink，故名思义，「沉入」，即数据将被存储到新的存储介质中，在本项目中包括，redis, kafka, clickhouse。

世间没有银弹，Hadoop 的生态有时候会很麻烦，需要大量的环境配置，在实时性和海量数据场景下，Hadoop 不太能够应付。Clickhouse 的诞生为大大的为 BI 赋能，虽然分析型数据库层出不穷，包括以Mysql 为代表的关系数据库，HIVE，SparkSql，但是它们都被具有丰富功能的 clickhouse 打败。其支持在线实时查询，采用列式存储，有完善的 SQL 支持，并且不依靠 Hadoop 数据生态，还有其最大的特色，查询性能非常高，在很多基准测试的比较重，Clickhouse 都位居榜首，所以我最终决定采用 Clickhouse 作为本项目的实时数仓。

Clickhouse 完全支持标准的 JDBC 协议，底层为 Http 通信接口。

1. class FlinkToCK extends RichSinkFunction<String> {
2. FlinkToCK(String host, String port, String user, String pwd) {
3. super();
4. host\_ = host;
5. port\_ = port;
6. user\_ = user;
7. pwd\_ = pwd;
8. }
9. @Override
10. public void invoke(String value, SinkFunction.Context context) throws SQLException {
11. String sql = "INSERT INTO test (first) VALUES (?)";
12. PreparedStatement statement = connection\_.prepareStatement(sql);
13. statement.setString(1, value);
14. statement.addBatch();
15. statement.executeBatch();
16. }
17. public void close() throws SQLException {
18. connection\_.close();
19. }
20. @Override
21. public void open(Configuration param) throws SQLException {
22. this.connection\_ = DriverManager.getConnection("jdbc:clickhouse://" + host\_ + ":" + port\_, user\_, pwd\_);
23. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS test (" +
24. "first Nullable(String)) ENGINE = Memory()"
25. );
26. }
27. ...
28. }

原始数据进入 clickhouse 后，就是数据仓库建模的第一步，得到 ODS，即原始数据表。

1. // ods table
2. this.connection\_.createStatement().execute("DROP TABLE IF EXISTS ods\_subway\_data");
3. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS ods\_subway\_data (" +
4. "deal\_date Nullable(String),"
5. + "close\_date Nullable(String),"
6. + "card\_no Nullable(String),"
7. + "deal\_value Nullable(String),"
8. + "deal\_type Nullable(String),"
9. + "company\_name Nullable(String),"
10. + "car\_no Nullable(String),"
11. + "station Nullable(String),"
12. + "conn\_mark Nullable(String),"
13. + "deal\_money Nullable(String),"
14. + "equ\_no Nullable(String)"
15. + ") ENGINE = Memory()"
16. );
17. String sql = "INSERT INTO ods\_subway\_data (deal\_date, close\_date, card\_no, deal\_value, deal\_type, company\_name, car\_no, station, conn\_mark, deal\_money, equ\_no) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
18. PreparedStatement statement = connection\_.prepareStatement(sql);
19. Map<String, String> mapVal = new Gson().fromJson(
20. value, new TypeToken<HashMap<String, String>>() {}.getType()
21. );
22. statement.setString(1, mapVal.get("deal\_date"));
23. statement.setString(2, mapVal.get("close\_date"));
24. statement.setString(3, mapVal.get("card\_no"));
25. statement.setString(4, mapVal.get("deal\_value"));
26. statement.setString(5, mapVal.get("deal\_type"));
27. statement.setString(6, mapVal.get("company\_name"));
28. statement.setString(7, mapVal.get("car\_no"));
29. statement.setString(8, mapVal.get("station"));
30. statement.setString(9, mapVal.get("conn\_mark"));
31. statement.setString(10, mapVal.get("deal\_money"));
32. statement.setString(11, mapVal.get("equ\_no"));
33. statement.addBatch();
34. statement.executeBatch();
35. }

维度建模是数据仓库领域的数据仓库建模的代表。数据仓库的核心目标就是为展示层提供清晰的结果。维度建模就是为 OLAP 的场景创造的，灵活分解分析的需求，在海量数据场景下能够快速响应。

数据仓库建模的第二步，对 DW 进行分层，DW 是数据仓库的核心，一般是三层结构。这种分层是为了更好的组织，管理和维护数据。所以数据仓库分层是数据仓库设计中的重要一环。分层后的业务表可以直接呈现给上层，还可以减少重复的计算和存储资源的浪费。DW 分为 DWD, DWB, DWS 三层，这里主要使用了 DWD (数据细节层) 和 DWS (数据汇总层)

在 DWD 层，做了一些清理数据的工作，清洗掉 bus 相关的数据 且不在地铁运营时间的数据。提高了数据的质量，会后续提供了了干净，一致的可靠数据

1. this.connection\_.createStatement().execute("DROP TABLE IF EXISTS dwd\_subway\_data");
2. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS dwd\_subway\_data (" +
3. "deal\_date Nullable(String),"
4. + "close\_date Nullable(String),"
5. + "card\_no Nullable(String),"
6. + "deal\_value Nullable(String),"
7. + "deal\_type Nullable(String),"
8. + "company\_name Nullable(String),"
9. + "car\_no Nullable(String),"
10. + "station Nullable(String),"
11. + "conn\_mark Nullable(String),"
12. + "deal\_money Nullable(String),"
13. + "equ\_no Nullable(String)"
14. + ") ENGINE = Memory()"
15. );
16. String dwd\_subway\_data\_sql = "INSERT INTO dwd\_subway\_data (deal\_date, close\_date, card\_no, deal\_value, deal\_type, company\_name, car\_no, station, conn\_mark, deal\_money, equ\_no) SELECT "
17. + "deal\_date,"
18. + "close\_date,"
19. + "card\_no,"
20. + "deal\_value,"
21. + "deal\_type,"
22. + "company\_name,"
23. + "car\_no,"
24. + "station,"
25. + "conn\_mark,"
26. + "deal\_money,"
27. + "equ\_no "
28. + "FROM ods\_subway\_data "
29. + "WHERE deal\_type != '巴士'";
30. statement = connection\_.prepareStatement(dwd\_subway\_data\_sql);

然后进一步进行降维操作，分为进站和出战两个层次，方便 DWS 层做汇聚

1. // 进站
2. this.connection\_.createStatement().execute("DROP TABLE IF EXISTS dwd\_in\_station\_data");
3. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS dwd\_in\_station\_data (" +
4. "deal\_date Nullable(String),"
5. + "card\_no Nullable(String),"
6. + "deal\_type Nullable(String),"
7. + "company\_name Nullable(String),"
8. + "car\_no Nullable(String),"
9. + "station Nullable(String),"
10. + "equ\_no Nullable(String)"
11. + ") ENGINE = Memory()"
12. );
13. String dwd\_in\_station\_data =
14. "INSERT INTO dwd\_in\_station\_data (deal\_date, card\_no, deal\_type, company\_name, car\_no, station, equ\_no) SELECT "
15. + "deal\_date,"
16. + "card\_no,"
17. + "deal\_type,"
18. + "company\_name,"
19. + "car\_no,"
20. + "station,"
21. + "equ\_no "
22. + "FROM dwd\_subway\_data "
23. + "WHERE deal\_type == '地铁入站'";
24. statement = connection\_.prepareStatement(dwd\_in\_station\_data);
25. // 出战
26. this.connection\_.createStatement().execute("DROP TABLE IF EXISTS dwd\_out\_station\_data");
27. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS dwd\_out\_station\_data (" +
28. "deal\_date Nullable(String),"
29. + "close\_date Nullable(String),"
30. + "card\_no Nullable(String),"
31. + "deal\_value Nullable(String),"
32. + "deal\_type Nullable(String),"
33. + "company\_name Nullable(String),"
34. + "car\_no Nullable(String),"
35. + "station Nullable(String),"
36. + "conn\_mark Nullable(String),"
37. + "deal\_money Nullable(String),"
38. + "equ\_no Nullable(String)"
39. + ") ENGINE = Memory()"
40. );
41. String dwd\_out\_station\_data\_sql =
42. "INSERT INTO dwd\_out\_station\_data (deal\_date, close\_date, card\_no, deal\_value, deal\_type, company\_name, car\_no, station, conn\_mark, deal\_money, equ\_no) SELECT "
43. + "deal\_date,"
44. + "close\_date,"
45. + "card\_no,"
46. + "deal\_value,"
47. + "deal\_type,"
48. + "company\_name,"
49. + "car\_no,"
50. + "station,"
51. + "conn\_mark,"
52. + "deal\_money,"
53. + "equ\_no "
54. + "FROM dwd\_subway\_data "
55. + "WHERE deal\_type == '地铁出站'";
56. statement = connection\_.prepareStatement(dwd\_out\_station\_data\_sql);
57. statement.execute();

在 DWS 层，基于 DWD 层的基本数据，将某个主题的服务数据进行集成和汇总，DWS 层将覆盖应用场景，可以快速响应数据需求。包含以下的主题

1.点进站人数排行榜

1. connection\_.createStatement().execute("DROP TABLE IF EXISTS  dwd\_in\_station\_rank");
2. connection\_.createStatement().execute("CREATE TABLE  IF NOT EXISTS dwd\_in\_station\_rank (" +
3. "station Nullable(String),"
4. + "deal\_dates Array(String),"
5. + "card\_nos Array(String),"
6. + "company\_names Array(String),"
7. + "equ\_nos Array(String),"
8. + "nums Int64"
9. + ") ENGINE  = MergeTree() order by nums");
10. String dwd\_in\_station\_rank\_sql =
11. "INSERT INTO dwd\_in\_station\_rank (station, deal\_dates, card\_nos, company\_names, equ\_nos, nums) SELECT "
12. + "station,"
13. + "groupArray(deal\_date),"
14. + "groupArray(card\_no),"
15. + "groupArray(company\_name),"
16. + "groupArray(equ\_no),"
17. + "count(\*) nums "
18. + "FROM dwd\_in\_station\_data "
19. + "GROUP BY station "
20. + "ORDER BY nums DESC";
21. statement = connection\_.prepareStatement(dwd\_in\_station\_rank\_sql);
22. statement.execute();

2.站点出站人数排行榜

1. connection\_.createStatement().execute("DROP TABLE IF EXISTS  dwd\_out\_station\_rank");
2. connection\_.createStatement().execute("CREATE TABLE  IF NOT EXISTS dwd\_out\_station\_rank (" +
3. "station Nullable(String),"
4. + "deal\_dates Array(String),"
5. + "card\_nos Array(String),"
6. + "deal\_values Array(String),"
7. + "company\_names Array(String),"
8. + "conn\_marks Array(String),"
9. + "deal\_moneys Array(String),"
10. + "equ\_nos Array(String),"
11. + "nums Int64"
12. + ") ENGINE  = MergeTree() ORDER BY nums"
13. );
14. String dwd\_out\_station\_rank\_sql =
15. "INSERT INTO dwd\_out\_station\_rank (station, deal\_dates, card\_nos, deal\_values, company\_names, conn\_marks, deal\_moneys, equ\_nos, nums) SELECT "
16. + "station,"
17. + "groupArray(deal\_date),"
18. + "groupArray(card\_no),"
19. + "groupArray(deal\_value),"
20. + "groupArray(company\_name),"
21. + "groupArray(conn\_mark),"
22. + "groupArray(deal\_money),"
23. + "groupArray(equ\_no),"
24. + "count(\*) nums "
25. + "FROM dwd\_out\_station\_data "
26. + "GROUP BY station "
27. + "ORDER BY nums DESC";
28. statement = connection\_.prepareStatement(dwd\_out\_station\_rank\_sql);
29. statement.execute();
30. connection\_.createStatement().execute("DROP TABLE IF EXISTS  dwd\_out\_station\_rank");
31. connection\_.createStatement().execute("CREATE TABLE  IF NOT EXISTS dwd\_out\_station\_rank (" +
32. "station Nullable(String),"
33. + "deal\_dates Array(String),"
34. + "card\_nos Array(String),"
35. + "deal\_values Array(String),"
36. + "company\_names Array(String),"
37. + "conn\_marks Array(String),"
38. + "deal\_moneys Array(String),"
39. + "equ\_nos Array(String),"
40. + "nums Int64"
41. + ") ENGINE  = MergeTree() ORDER BY nums"
42. );
43. String dwd\_out\_station\_rank\_sql =
44. "INSERT INTO dwd\_out\_station\_rank (station, deal\_dates, card\_nos, deal\_values, company\_names, conn\_marks, deal\_moneys, equ\_nos, nums) SELECT "
45. + "station,"
46. + "groupArray(deal\_date),"
47. + "groupArray(card\_no),"
48. + "groupArray(deal\_value),"
49. + "groupArray(company\_name),"
50. + "groupArray(conn\_mark),"
51. + "groupArray(deal\_money),"
52. + "groupArray(equ\_no),"
53. + "count(\*) nums "
54. + "FROM dwd\_out\_station\_data "
55. + "GROUP BY station "
56. + "ORDER BY nums DESC";
57. statement = connection\_.prepareStatement(dwd\_out\_station\_rank\_sql);
58. statement.execute();

3.站点进站出站人数排行榜

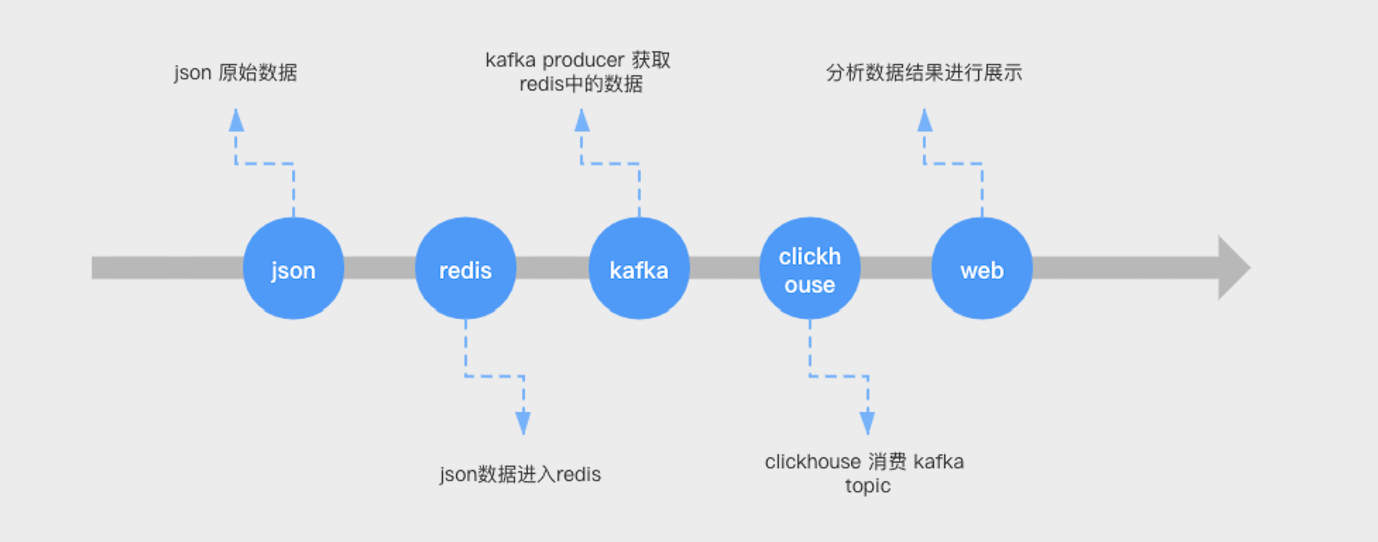
1. connection\_.createStatement().execute("DROP TABLE IF EXISTS  dwd\_in\_out\_station\_rank");
2. connection\_.createStatement().execute("CREATE TABLE  IF NOT EXISTS dwd\_in\_out\_station\_rank (" +
3. "station Nullable(String),"
4. + "deal\_dates Array(String),"
5. + "card\_nos Array(String),"
6. + "deal\_values Array(String),"
7. + "deal\_types Array(String),"
8. + "company\_names Array(String),"
9. + "conn\_marks Array(String),"
10. + "deal\_moneys Array(String),"
11. + "equ\_nos Array(String),"
12. + "nums Int64"
13. + ") ENGINE  = MergeTree() ORDER BY nums"
14. );
15. String dwd\_in\_out\_station\_rank\_sql =
16. "INSERT INTO dwd\_in\_out\_station\_rank (station, deal\_dates, card\_nos, deal\_values, deal\_types, company\_names, conn\_marks, deal\_moneys, equ\_nos, nums) SELECT "
17. + "station,"
18. + "groupArray(deal\_date),"
19. + "groupArray(card\_no),"
20. + "groupArray(deal\_value),"
21. + "groupArray(deal\_type),"
22. + "groupArray(company\_name),"
23. + "groupArray(conn\_mark),"
24. + "groupArray(deal\_money),"
25. + "groupArray(equ\_no),"
26. + "count(\*) nums "
27. + "FROM dwd\_subway\_data "
28. + "GROUP BY station "
29. + "ORDER BY nums DESC";
30. statement = connection\_.prepareStatement(dwd\_in\_out\_station\_rank\_sql);
31. statement.execute();

4.站点盈利排行榜

1. connection\_.createStatement().execute("DROP TABLE IF EXISTS  dwd\_station\_earning\_rank");
2. connection\_.createStatement().execute("CREATE TABLE IF NOT EXISTS dwd\_station\_earning\_rank (" +
3. "station String,"
4. + "company\_name String,"
5. + "deal\_value\_sum Float64,"
6. + "deal\_money\_sum Float64"
7. + ") ENGINE  = TinyLog()"
8. );
9. String dwd\_station\_earning\_rank\_sql =
10. "INSERT INTO dwd\_station\_earning\_rank (station, company\_name, deal\_value\_sum, deal\_money\_sum) SELECT "
11. + "station,"
12. + "company\_name,"
13. + "sum(toFloat64OrZero(deal\_value))/100 AS deal\_value\_sum,"
14. + "sum(toFloat64OrZero(deal\_money))/100 AS deal\_money\_sum "
15. + "FROM dwd\_out\_station\_data "
16. + "GROUP BY company\_name, station "
17. + "ORDER BY deal\_value\_sum desc";
18. statement = connection\_.prepareStatement(dwd\_station\_earning\_rank\_sql);
19. statement.execute();

至此，数据仓库的建模就完成了，前端页面通过循环的查询来获取 DWS 层主题的最新情况，展示在动态面板上。

## 项目的时序图



## 项目类图

# 难点和重点分析

## 获取数据

鉴于地铁刷卡数据有一定的隐私性，所以并没有多少城市将这些数据集公开，在寻找数据的过程中花费了很多时间。浏览了二十多个城市的地铁交通网站，包括北京，上海，成都，广州，深圳等。感谢深圳地铁局提供的宝贵数据，使得本项目可以进行。经过这次经历，我碰到不错的数据集都会收藏起来，以备不时之需。

## 模拟实时效果

深圳地铁网站之前有提供 API 去获取实时的刷卡数据，但是现在没有了。只拿到了一个离线数据集，所以需要考虑如果模拟成实时的效果，让项目场景更加真实。不同于之前的批处理系统，实时一般要求的是秒级别的延迟，而批处理往往是天级别。在实时技术的选型过程中，多个子系统需要相互协作、相互依赖，形成一个数据处理环节。这不同于之前的 hadoop 时代，目前的实时技术并没有一家独大的局面，备选方案很多，开源界百花齐放。分析项目的需求，首先考虑到的就是 flink, 作为实时处理的黑马，其可以作为一条无形的流，将项目整个串起来，同时提供实时效果。然后使用 kafka 作为链路中承前启后的模块，连接了生产者和消费者，确保了数据在链路中的顺利流通。最后一步，就是实时分析，传统的OLTP数据库不能满足实时场景的要求，如何选择一个合适的分析型数据库对项目的成功与否至关重要。现阶段可选的 OLAP 数据库有，analyticDB、clickhouse、DorisDB、Presto、Druid等。项目的场景是读请求在请求中占比很高，不会修改已经写入的数据，查询会读取大量的行但是只需要少许的列，一般查询允许的延时大约为100ms，不需要事务。选择 OLAP 数据库的时候主要从这三方面考虑， 性能，数据量，灵活性。以 Presto 为代表的 MPP 架构的 OLAP 数据库可以支持大量的数据并且灵活性也不错，但是一旦查询的复杂度提高或者数据量增大后，其响应就会变慢，不能再提供实时的分析。再看看 Clickhouse, 简直就是分析型数据库的黑马，其性能优化做的非常细节，它的性能超过了市面上 90% 的数据库；它具有丰富的功能，并提供类似SQL的查询，支持嵌套的数据结构和数组；是一款开源的数据库，在开源界十分火爆，成熟度和稳定度优于大多数开源的OLAP数据库，在一定程度上减少了学习和使用的困难。

## 涉及到的组件很多

在模拟实时效果的难点分析中，提到了很多的存储和中间件系统， 如 redis, clickhouse, flink, kafka。这些系统都是成熟和功能丰富的软件。并且学习成本都是比较高，只有熟悉了这些系统，项目才可以推进。而且在决定使用这些系统前，都做了丰富的调研，横向和纵向对比这些系统的竞争对手，所以前前后后了解和学习了十几种系统，最终才选择了它们，并且对它们进行进一步学习，这个过程是十分耗时和消耗精力的。

## 前端实时展示

不同于普通的各种管理系统的前端，本项目要求的前端类似于一种数据可视化工具，能够实时的动态大屏更新展示；经过调研 DataV, EasyV, echart, FineReport 等都属于这个范畴。DataV 只支持访问阿里云相关数据库，如 analyticDB，不支持 clickhouse 的接入。echart 兼容性差，灵活度也不高。最后综合考察决定使用 EasyV, 其特别的支持 Clickhouse 接入，且有视频教程，一定程度可以降低使用曲线，并且灵活，支持自定义，并且提供了样例参考。

# 总结和展望

本文收集了深圳市刷卡离线数据集，利用进站和出站乘客的刷卡数据，借助大数据时代的各种存储和流式系统，把离线的数据集模拟成实时的数据场景，构建 ETL 系统，建立数据仓库模型，存储数据通过分析型数据库 Clickhouse 分析站点峰值，分析地铁站时空客流规律和乘客出行特点，最后利用可视化技术把结果呈现出来。挖掘这些信息，这对于解决城市交通拥堵问题，优化现有交通网络，进一步规划新的交通线路，保障城市公共交通安全具有重要意义。相信随着大数据技术的发展，实时系统将更加成熟，搭建一个实时流式系统的成本将大大降低，更有利于此系统的普及。

# 参考文献