进击的 Flink：网易云音乐实时数仓建设实践

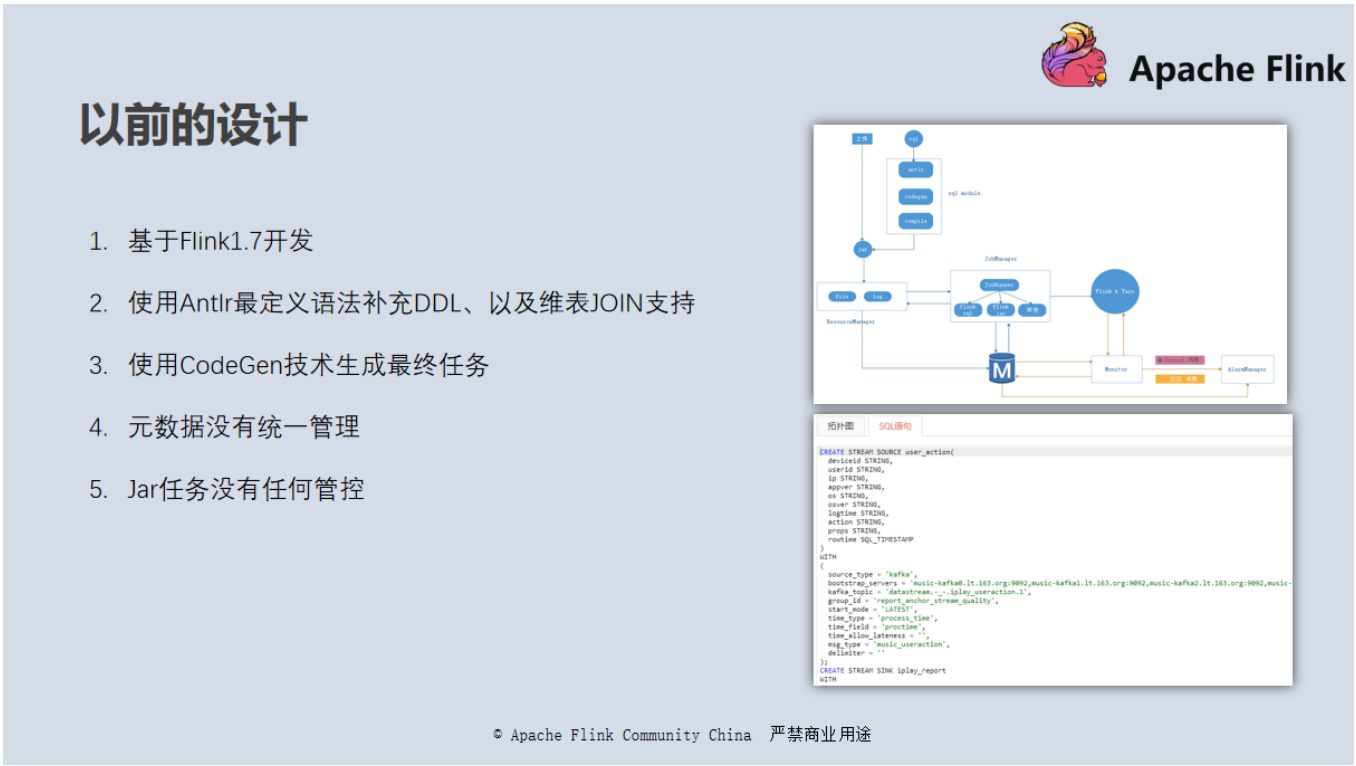
汪磊 蔡芳芳

发布于：2020 年 7 月 16 日 09:00

背景介绍

网易云音乐从 2018 年开始搭建实时计算平台，到目前为止已经发展至如下规模：

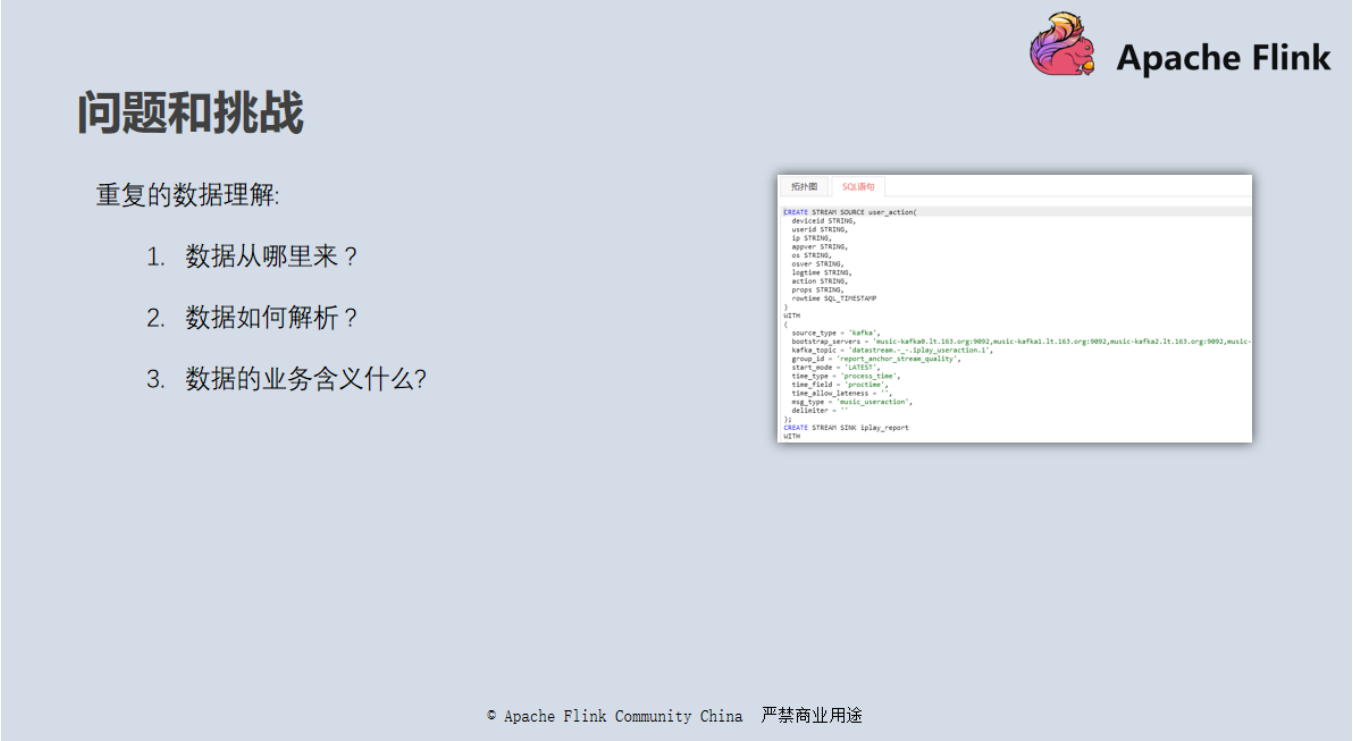
1. 机器数量：130+
2. 单 Kafka 峰值 QPS：400W+
3. 在线运行任务数：500+
4. 开发者：160+
5. 业务覆盖：在线业务支持，实时报表统计，实时特征处理，实时索引支持
6. 2020 年 Q1 任务数增长 100%，处于高速发展中



这是网易云音乐实时数仓 18 年的版本，基于 Flink 1.7 版本开发，当时 Flink SQL 的整体架构也还不是很完善。我们使用了 Antlr （通用的编程语言解析器，它只需编写名为 G4 的语法文件，即可自动生成解析的代码，并且以统一的格式输出，处理起来非常简单。由于 G4 文件是通过开发者自行

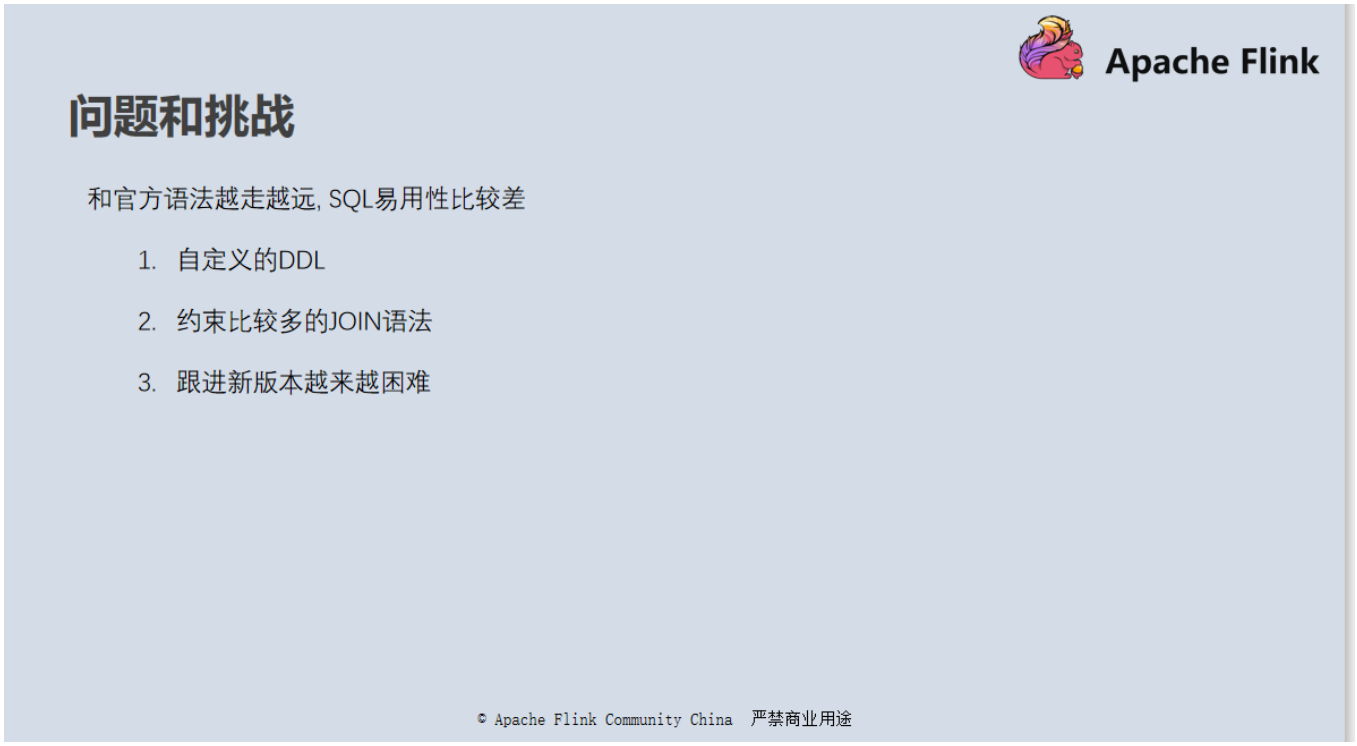
定制的，因此由 Antlr 生成的代码也更加简洁和个性化）自定义了一些 DDL 完善了维表 join 的语法。通过 Antlr 完成语法树的解析以后，再通过 CodeGen（根据接口文档生成代码）技术去将整个 SQL 代码生成一个 Jar 包，然后部署到 Flink 集群上去。

此时还没有统一的元数据管理系统。在 JAR 包任务的开发上, 我们也没有任何框架的约束，平台也很难知道 JAR 的任务上下游以及相关业务的重要性和优先级。这套架构我们跑了将近一年的时间，随着任务越来越多，我们发现了以下几个问题:



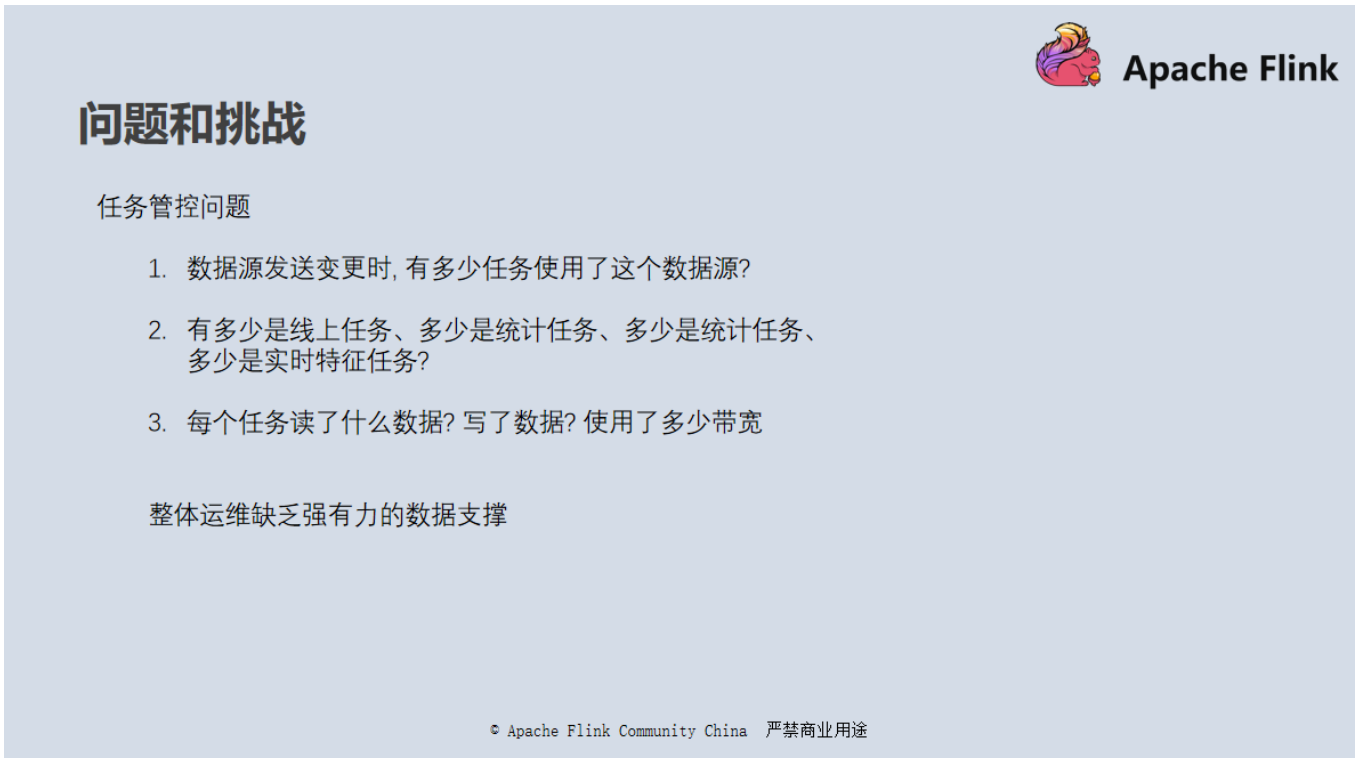
重复的数据理解

由于没有进行统一的元数据管理，每个任务的代码里面都需要预先定义 DDL 语句，然后再进行 Select 等业务逻辑的开发；消息的元数据不能复用，每个开发都需要进行重复的数据理解，需要了解数据从哪里来、数据如何解析、数据的业务含义是什么；整个过程需要多方沟通，整体还存在理解错误的风险；也缺乏统一的管理系统去查找自己想要的数据。



和官方版本越走越远

由于早期版本很多 SQL 的语法都是我们自己自定义的，随着 Flink 本身版本的完善，语法和官方版本差别越来越大，功能完善性上也渐渐跟不上官方的版本，易用性自然也越来越差。如果你本身就是一名熟知 Flink SQL 的开发人员，可能还需要重新学习我们平台自己的语法，整体不是很统一，有些问题也很难在互联网上找到相关的资料，只能靠运维来解决。



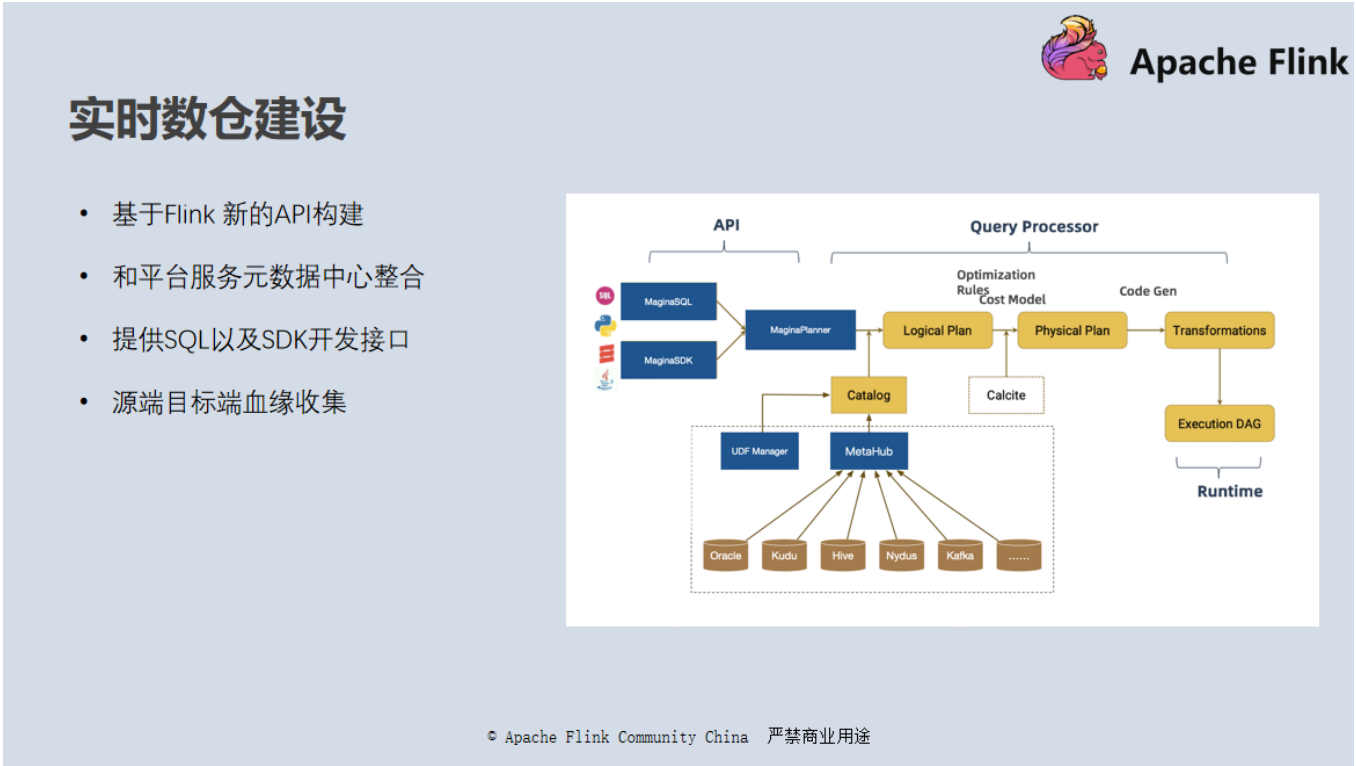
任务运维问题

SQL 任务没有统一的元数据管理、上下游的数据源没有统一的登记、JAR 包任务没有统一的框架约束、平台方很难跟踪整个平台数据流的走向，我们不知道平台上运行的几百个任务分别是干什么的，哪些任务读了哪个数据源？输出了什么数据源？任务的种类是什么？是线上的，测试的，重要的还是不重要的。没有这些数据的支撑，导致整个运维工作非常局限。

网易云音乐的业务发展非常快，数据量越来越大，线上库和一些其它的库变更十分频繁，相关的实时任务也要跟着业务架构的调整，变更相关数据源的地址。此时我们就需要知道哪些任务用到了相关的数据源，如果平台没有能力很快筛选出相关任务，整个流程处理起来就十分繁杂了。

首先，需要联系平台所有的开发者确认是否有相关任务的数据源，整个流程非常浪费时间，而且还有可能产生疏漏；其次，假设出现平台流量激增，做运维工作时，如果我们不知道任务在干什么，自然也不能知道的任务的重要性，不知道哪些任务可以限流，哪些任务可以做暂时性的停止，哪些任务要重点保障。

实时数仓建设



带着这些问题，我们开始进行新版本的构建工作。

- 在 Flink 1.9 版本以后，Flink 有了重大变化，重构了 Catalog 的 API，这和之前我们做的离线方向的工作有一定的契合。在离线的生态上，网易云音乐有着一套非常完整的服务体系，打通元数

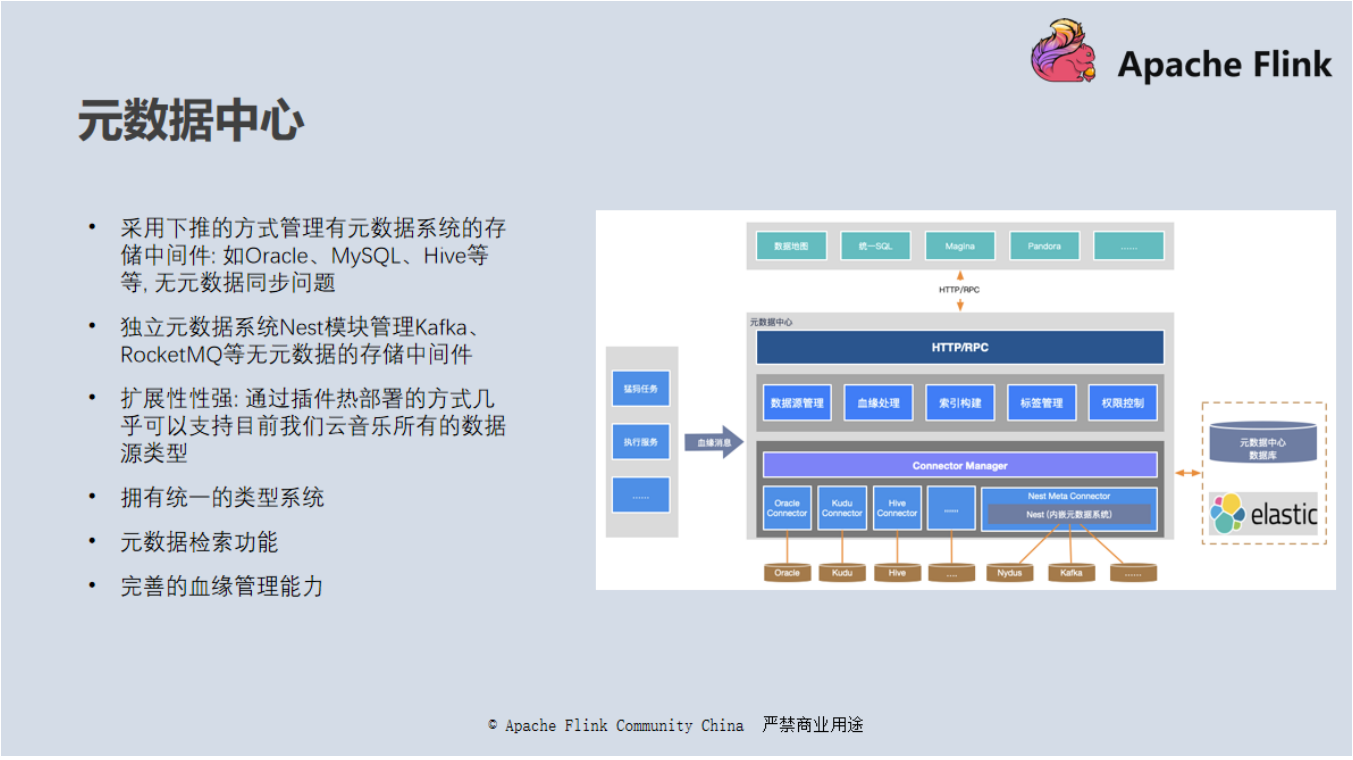
据中心和 Spark SQL，可以通过 Spark SQL 连接元数据中心的元数据，进行异构数据源的联邦查询以及数据传输工作；

- 同样基于 Flink 1.10，我们利用新的 Catalog 的 API 实现了一个元数据中心的 Catalog。将元数据中心作为 Flink SQL 的底层元数据组件，实现了 Kafka 到元数据中心任一数据源的实时的数据传输，以及 Redis、HBase、Kudu 等数据源的维表 JOIN 的实现；

- 除了纯 SQL 的开发方式外，我们还提供了一套 SDK，让用户可以通过 SQL 加代码混合使用的方式来实现自己的业务逻辑，提升整个 Flink API 的易用性，大大降低用户的开发门槛，提升了平台对任务的管控能力；

- 有了统一的元数据的管理以及 SDK 的开发方式，血缘收集也变得水到渠成，有了上下游数据的走向信息，平台也很容易通过数据源的业务属性来判断任务的重要性。

元数据中心



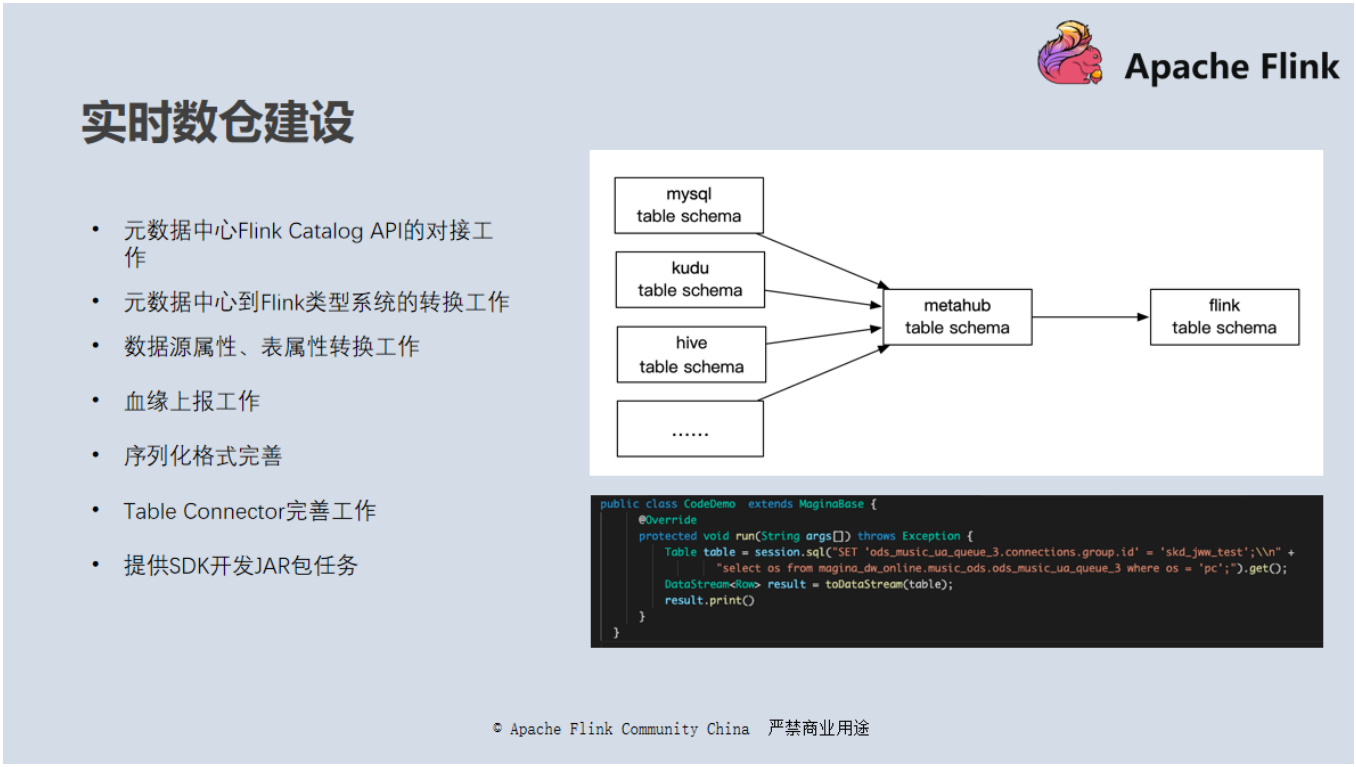
不知道大家有没有用过 Apache Atlas、Netflix 的 Metacat 等工具，网易云音乐的元数据中心顾名思义就是一个元数据管理的程序，用于管理网易云音乐所有数据源的元数据。你有可能在实际的开发中用到 Oracle、Kudu、Hive 等工具，也有可能是自研的分布式数据库。如果没有统一的元数据管理，我们很难知道我们有哪些数据，数据是如何流转的，也很难快速找到自己想要的数据。

将它们统一管理的好处是，可以通过元数据中心快速找到自己想要的数据，了解数据表的连接信息、schema 信息，字段的业务含义，以及所有表的数据来源和走向。

我们的元数据中心系统有以下几个特点:

1. **扩展性强**：元数据中心系统理论上是可以管理所有的数据存储中间件的，每个存储中间件都可以通过插件的方式热部署扩展上去，目前我们已经支持了云音乐内部几乎所有的存储中间件；
2. **下推查询**：对于自身有元数据系统的存储中间件，如刚刚提到的 Oracle、Kudu 、Hive 等，我们采用的是下推查询的方式，直接去查询它们的元数据的数据库，获取到相应的元数据信息，这样就不会存在元数据不一致的问题；
3. **Nest 元数据登记**：对于像 Kafka、RocketMQ 这种自身并不存在元数据体系的，元数据中心内部有一个内嵌的元数据模块 Nest，Nest 参考了 Hive 元数据的实现，用户可以手动登记相关数据的 Schema 信息；
4. **统一的类型系统**：为了更好的管理不同类型的的数据源，方便外部查询引擎对接，元数据中心有一套完善的类型系统，用户在实现不同数据源的插件时需要实现自身类型体系到元数据类型的映射关系；
5. **元数据检索**：我们会定期用全量数和增量的方式将元数据同步到 ES 当中，方便用户快速查找自己想要的数据；
6. **完善的血缘功能**：只要将任务的上下游按照指定的格式上报到元数据中心，就可以通过它提供的血缘接口去拿到整个数据流的血缘链路。

建设流程



需要进行的工作包括：

1. 使用元数据中心的 API 实现 Flink Catalog API。
2. 元数据中心到 Flink 系统的数据类型转换，因为元数据中有一套统一的类型系统，只需要处理 Flink 的类型系统到元数据类型系统的映射即可，不需要关心具体数据源的类型的转换。
3. 数据源属性和表属性的转换，Flink 中表的属性决定了它的源头、序列化方式等，但是元数据中心也有自己的一套属性，所以需要手动转换一些属性信息，主要是一些属性 key 的对齐问题。
4. 血缘解析上报。
5. 序列化格式完善。
6. Table Connector 的完善，完善常用的存储中间件的 Table Connector，如 Kudu、网易内部的 DDB 以及云音乐自研的 Nydus 等。
7. 提供 SDK 的开发方式：SDK 开发类似于 Spark SQL 的开发方式，通过 SQL 读取数据，做一些简单的逻辑处理，然后转换成 DataStream，利用底层 API 实现一些复杂的数据转换逻辑，最后再通过 SQL 的方式 sink 出去。简单来说就是，SQL 加代码混编的方式，提升开发效率，让开发专注于业务逻辑实现，同时保证血缘的完整性和便利性，且充分利用了元数据。

完成以上工作后，整体基本就能实现我们的预期。

在一个 Flink 任务的开发中，涉及的数据源主要有三类：

- 流式数据：来自 Kafka 或者 Nydus，可以作为源端和目标端；
- 维表 JOIN 数据：来自 HBase 、Redis、JDBC 等，这个取决于我们自己实现了哪些；
- 落地数据源：一般为 MySQL、HBase、Kudu、JDBC 等，在流处理模式下通常作为目标端。

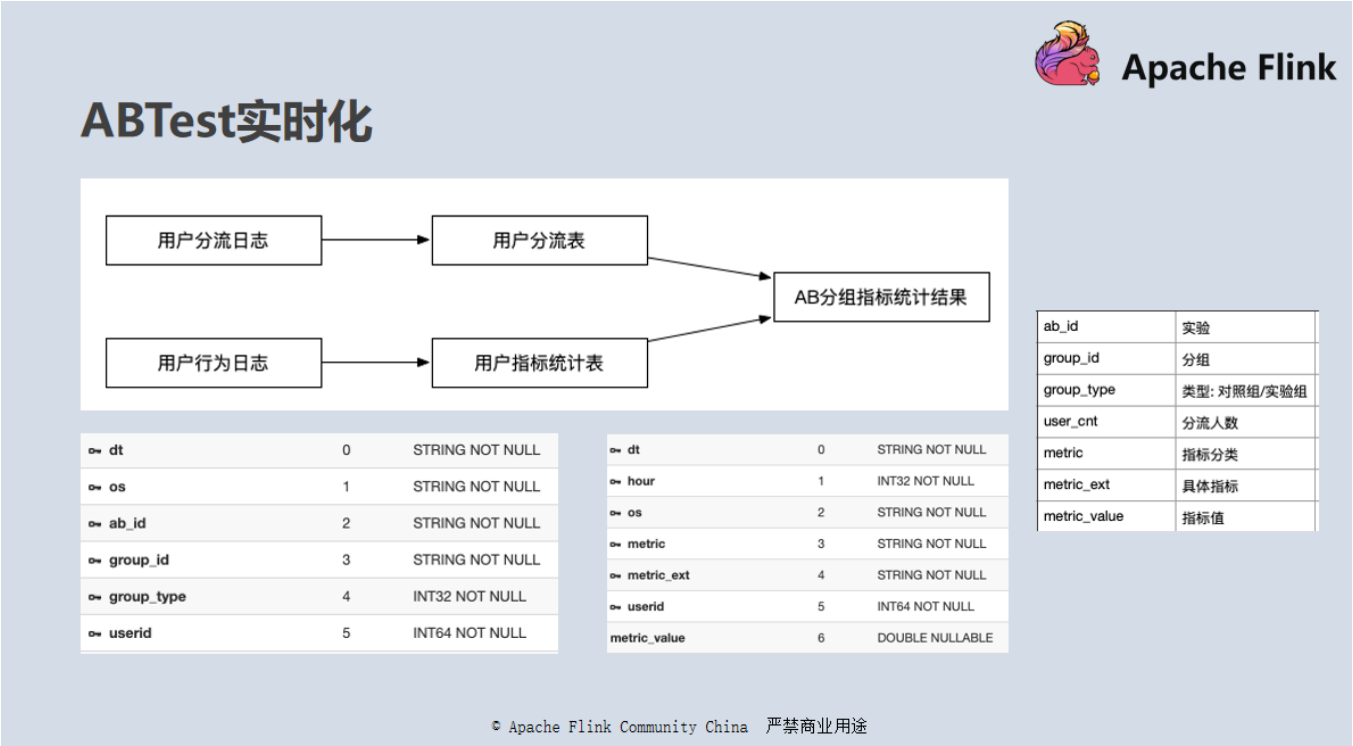
对于流式数据，我们使用元数据中心自带的元数据系统 Nest 登记管理（参考右上角的图）；对于维表以及落地数据源等，可以直接通过元数据中心获取库表 Schema 信息，无需额外的 Schema 登记，只需要一次性登记下数据源连接信息即可（参考右下角的图）。整体对应我们系统中数仓模块的元数据管理、数据源登记两个页面。

完成登记工作以后，我们可以通过`[catalog.][db.][table]`等方式访问任一元数据中心中登记的表，进行 SQL 开发工作。其中 Catalog 是在数据源登记时登记的名字；db 和 table 是相应数据源自身的 DB 和 Table，如果是 MySQL 就是 MySQL 自身元数据中的 DB 和 Table。

最终效果可以参考左下角读取实时表数据写入 Kudu 的的例子，其中红框部分是一个 Kudu 数据表，在使用前只需要登记相关连接信息即可，无需登记表信息，平台会从元数据中心获取。

ABTest 项目实践

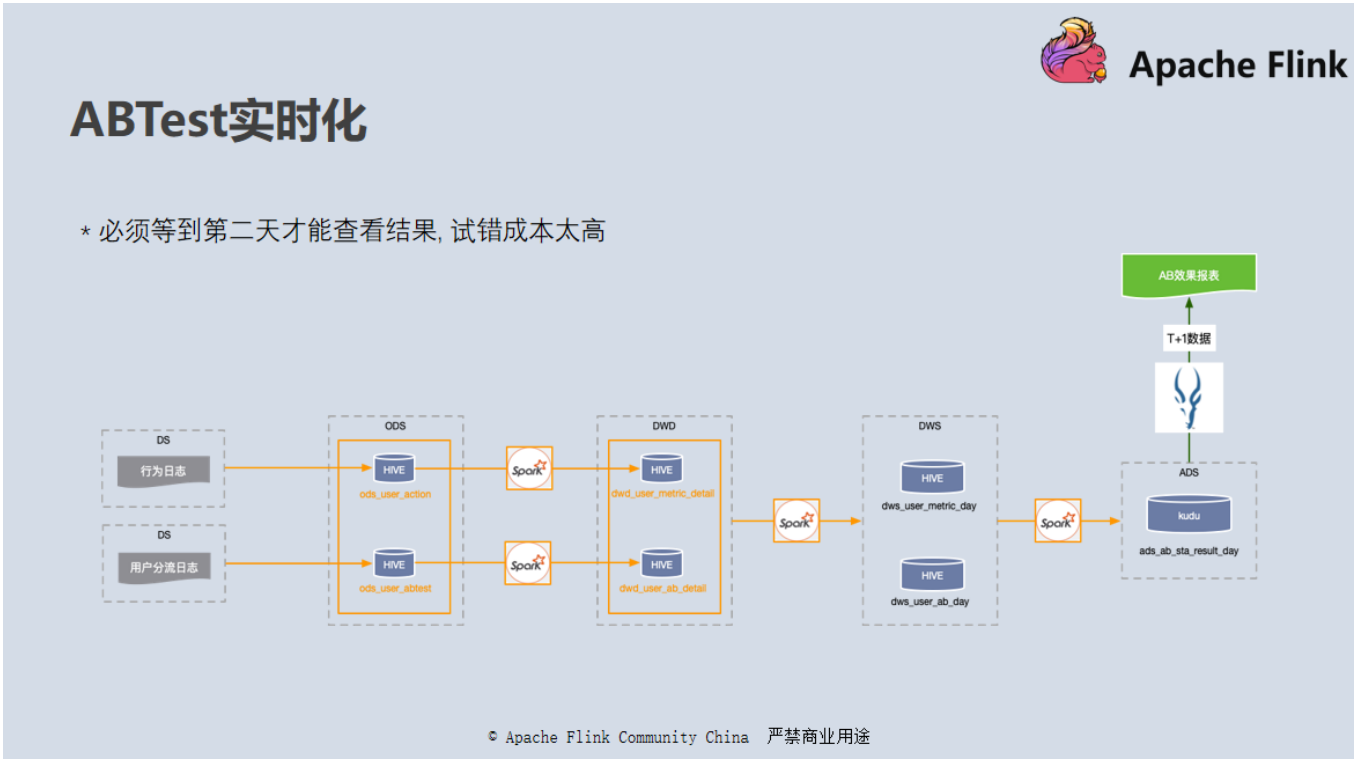
项目说明



ABTest 是目前各大互联网公司用来评估前端改动或模型上线效果的一种有效手段，它主要涉及了两类数据：第一个是用户分流数据，一个 AB 实验中用户会被分成很多组；然后就是相关指标统计数据，我们通过统计不同分组的用户在相应场景下指标的好坏，来判断相关策略的好坏。这两类数据被分为两张表：

- 用户分流表：dt 表示时间，os 表示操作系统。 ab_id 是某个 ABTest 的 id 号，group_id 就是分组 id ，group_type 分为两种，对照组指的是 ABTest 里面的基准，而实验组即是这次 ABTest 需要去评估的这批数据。userId 就是用户 id 了。

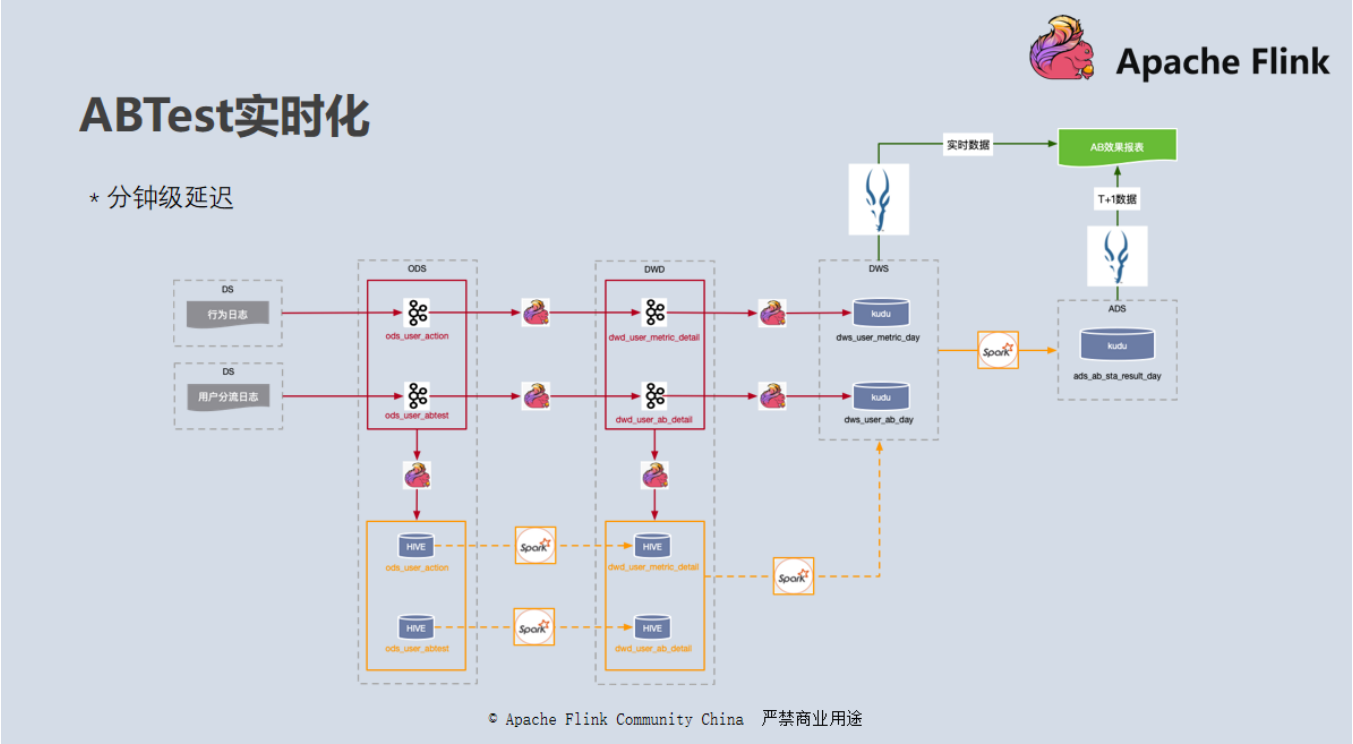
- 指标统计表：根据 dt、os 等不同维度来统计每个用户的有效播放，曝光，点击率等指标，metric、metric_ext 组合成一个具体含义。



在早期版本中，我们使用 Spark 按照小时粒度完成从 ODS 到 DWD 层数据清洗工作，生成用户分流表和指标统计表。然后再使用 Spark 关联这两张表的数据将结果写入到 Kudu 当中，再使用 Impala 系统对接，供用户进行查询。

这套方案的最大的问题是延迟太高，往往需要延迟一到两个小时，有些甚至到第二天才能看到结果。对于延迟归档的数据也不能及时对结果进行修正。

这个方案对我们的业务方比如算法来说，上线一个模型需要等到两个小时甚至第二天才能看到线上的效果，试错成本太高，所以后来使用新版的实时仓开发了一套实时版本。



如上图所示，是我们实时版本 ABTest 的数据走向，我们整体采用了 Lambda 架构：

- **第一步**：使用 Flink 订阅 ODS 原始的数据日志，处理成 DWD 层的数据分流表和指标统计表，同时也将实时的 DWD 层数据同步到相同结构的 Hive 表当中。DWD 层处理的目的是将业务数据清洗处理成业务能看懂的数据，没有聚合操作，实现比较简单。但是流数据归档到 Hive 的过程中需要注意小文件问题，文件落地的频率越高，延迟越低，同时落地的小文件也会越多，所以需要在技术和需求上权衡这个问题。同时在下方，我们也会有一条离线的数据流来处理同样的过程，这个离线不是必须的，如果业务方对数据的准确性要求非常高，我们需要用离线处理做一次修正，解决数据重复问题。这一步还涉及到一个埋点的复杂问题，如果一个指标的埋点非常复杂，比如需要依赖时间顺序路径的归因，而且本身客户端日志的延迟程度也非常不可靠的话，离线的修复策略就更加有必要了。

- **第二步**：DWS 层处理，读取第一步生成的 DWD 的流表数据使用 Flink 按照天和小时的维度做全局聚合，这一步利用了 Flink 状态计算的特点将中间结果维护在 RocksDB 的状态当中。然后使用 RetractionSink 将结果数据不断写入到 Kudu ，生成一个不断修正的 DWS 层聚合数据。同样我们也会使用 Spark 做一套同样逻辑的计算历史数据来做数据的修正。

- 这个步骤涉及到几个问题:

  1. Flink 大状态的运维和性能问题：为了解决了这个问题，我们使用 SSD 的机器专门用来运行这种大状态的任务，保障 RocksDB 状态的吞吐性能；

  2. Kudu 的 Update 性能问题：这里做了一些 minibatch 的的优化降低 Kudu 写入的压力；

  3. Lambda 架构的运维成本：实时离线两套代码运维成本比较高。

- **第三步：**结果数据对接

  - 对于实时的结果数据我们使用 Impala 直接关联用户分流表和指标数据表，实时计算出结果反馈给用户；

  - 对于 T+1 的历史数据，因为数据已经落地，并且不会再变了，所以为了降低 Impala 的压力，我们使用 Spark 将结果提前计算好存在 Kudu 的结果表中，然后使用 Impala 直接查询出计算好的结果数据。

批流一体

前面介绍的 ABTest 实时化整个实现过程就是一套完整的批流一体 Lambda 架构的实现。ODS 和 DWD 层既可以订阅访问，也可以批量读取。DWD 层落地在支持更新操作的 Kudu 当中，和上层

OLAP 引擎对接，为用户提供实时的结果。目前实现上还有一些不足，但是未来批流一体的努力方向应该能看得比较清楚了。

我们认为批流一体主要分以下三个方面：

1. 结果的批流一体

使用数据的人不需要关心数据是批处理还是流处理，在提交查询的那一刻，拿到的结果就应该是截止到目前这一刻最新的统计结果，对于最上层用户来说没有批和流的概念。

2. 存储的批流一体

上面的 ABTest 例子中我们已经看到 DWD、DWS 层数据的存储上还有很多不足，业界也有一些相应解决方案等待去尝试，我们希望的批流一体存储需要以下几个特性：

1. 同时提供增量订阅读取以及批量读取的能力，如 Apache Pulsar，我们可以批量读取它里面的归档数据，也可以通过 Flink 订阅它的流式数据，解决 DWD 层两套存储的问题。

2. 高性能的实时 / 批量 append 和 update 能力，读写互不影响，提供类似于 MVCC 的机制，类似于 Kudu 这种，但是性能需要更加强悍来解决 DWS 层存储的问题。

3. 和 OLAP 引擎的对接能力，比如 Impala、Presto 等，并且如果想要提升查询效率可能还要考虑到列式存储，具备较强的 scan 或者 filter 能力，来满足上层用户对业务结果数据查询效率的诉求

3. 计算引擎的批流一体

做到一套代码解决批流统一场景，降低开发运维成本，这个也是 Flink 正在努力的方向，未来我们也会在上面做一些尝试。